The CVS-Server Case Study: *A Formalized Security Architecture*

Extended Abstract

Achim D. Brucker, Frank Rittinger, and Burkhart Wolff
{brucker,rittinge,wolff}@informatik.uni-freiburg.de

1 Introduction

These days, the *Concurrent Versions System* (CVS) is a widely used tool for version management in many industrial software development projects, and plays a key role in open source projects usually performed by highly distributed teams [4]. CVS provides a central database (the *repository*) and means to synchronize local partial copies (the *working copies*) and their local modifications with the repository. CVS can be accessed via a network; this requires a security architecture establishing authentication, access control and non-repudiation. A further complication of the CVS security architecture stems from the fact that the administration of authentication and access control is done via CVS itself; i.e. the relevant data is accessed and modified via standard CVS operations and, thus, access to objects may change dynamically.

The current standard "out-of-the-box" CVS-server has a number of shortcomings with respect to security. To overcome some of these problems, we propose a number of improvements of the standard CVS-server [1], either on the level of its implementation (via patches), its configuration (i.e. the file system, including the initial state of the repository) or its architecture (i.e. the particular setup of a CVS-server and its configuration in a network).

The first aim of our configuration of CVS-server is to enforce a particular access control model, namely role-based access control [6]. Our second aim is an *open* CVS-server architecture, where the repository is part of the shared file system. While this "open CVS-server architecture" has a number of technical advantages, the correctness of the security mechanisms become a major concern.

The purpose of this paper is to give an *overview* over an ongoing case study [3] that provides a formal model of the "open CVS-server architecture" and (the begin of) a formal analysis performed with HOL-Z 2.0 [2]. We believe that this application is quite typical for client-server applications, where a particular security model must be mapped on the concrete security technology of POSIX/UNIX.

We will proceed as follows: After a discussion on the architecture notion and its refinement, we will first outline the CVS-server *system architecture*, that incorporates a *role-based access control model*. Second we will describe the key concepts of the *implementation architecture* based on the security mechanism of the Unix file system. Both layers will be connected via a refinement. Third, security properties were stated over state transition sequences on both levels.

2 Formalizing Architectures

Architecture models were used in the early design phases of a software development; they are composed by *components* and *connectors*. Components are computational units that interact via connectors with each other; connectors can be remote procedure calls, communication protocols or access to shared variables. An informal diagram showing the open architecture can be seen in Fig. 1.

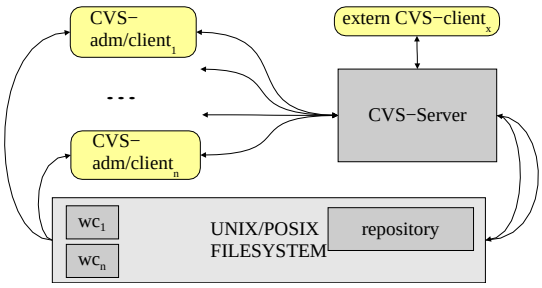

Fig. 1. The Open CVS-server Architecture

The boxes represent the components of the system, internal boxes parts of their state, and the connectors represent operations components may engage in. In particular, CVS client programs may engage in cvs commands like "cvs update" or in standard UNIX filesystem commands like "cp" or "chmod".

Following the approaches of Garlan and Shaw [7], architecture models are biased towards event-oriented or behavioral modeling; it is quite natural to use a process algebra such as CSP for its formalization and analysis. In this setting, components are just processes, while connectors are also processes or particular parallel operators of CSP.

However, for CVS-server, a data model is much more appropriate: the complexity of the model does not consist in the interaction, but in the invariants of a highly structured state. Consequently, we model the states of clients and server individually as well as their operations; the parallel composition of them is just the conjunction of the corresponding operation schemas.

3 Refining Architectures

The problem is adequately represented by an abstract *system architecture*, that models the repository, the working copies and the access operations incorporating the desired security model, similar to the $RBAC_1$-model described in the formal framework presented in [6]. This security model must be *mapped* on a concrete, widely used security technology, namely standard Discretionary Access Control (DAC) as implemented by the Unix/POSIX.1 file system layer. This description is the concrete *implementation architecture*. This mapping involves the

representation of "roles" in form of unique UNIX owners and groups and appropriate settings of permissions in regular files and directories in the working copies and the repository.

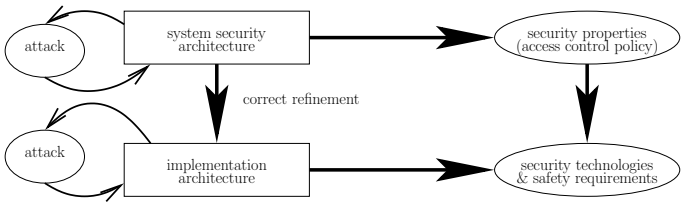

Fig. 2. Refining Security Architectures

Such a connection between abstract and more concrete views on a system and their semantic underpinning is well–known under the term *refinement*, and security technology mappings can be understood as a special case; we chose to apply data refinement following [8].

4 The CVS-Server Case Study

4.1 The System Architecture

The CVS-server provides the following commands that modify the repository and the working copies:

login: authenticates a user (in a specific role) to the server,
add: put files or directories under version control,
cd: set a filter on the files affected CVS operations,
commit: transfer local changes to the repository, and
update: update working copy with newer information from the repository.

Note that the functionality provided by CVS for conflict resolution (*merges*), for accessing the history, for branching, and for logging information, is not covered by our security (!) model.

4.2 The Implementation Architecture

The implementation is based on a standard CVS-server [4] embedded into a Unix/POSIX.1 filesystem. Therefore, the implementation has to cope with the full range of Unix/POSIX.1 methods for accessing files and chancing their access attributes and thus attacker on the implementation level can use the standard cvs commands (commit, etc) *and* the Unix commands for copying files (cp), removing (rm), chancing attributes (chmod, chown, setumask), etc, which can be applied to the repository and the working copies directly.

4.3 Establishing the Refinement

In order to prove that the concrete architecture correctly implements the abstract one, we have to define an abstraction schema which relates the components of the abstract state schemas to the components of the concrete implementation state schemas. In particular, we must map abstract names and abstract data to paths and files in the sense of the Unix file system.

Importing the abstract and concrete state schemas introduces all components that have to be set into relation. Note that some components of the abstract state also appear in the concrete state, having the same meaning in both states, they don't need to be related in this schema. We restrict this schema such that the permission tables in the abstract and in the concrete state are the same and that the roles and passwords that are assigned to each file in the working copy of the abstract state have corresponding attributes in the concrete state.

Following [8], we must prove two refinement conditions for each operation on the abstract state and its corresponding operation on the concrete state: Condition (a) ensures that a concrete operation terminates whenever its corresponding abstract operation is guaranteed to terminate, and condition (b) ensures that the state after the concrete operation represents one of those abstract states in which the abstract operation could terminate.

4.4 Security Properties of Both Architecture Layers

By their nature, security properties as well as functional properties are *safety properties*, i.e. it is necessary to consider the set of possible *sequences* of operations the system may engage in and state requirements on the possible states the system reaches after arbitrarily mixed commands. Hence, the specification of the safety properties in the system architecture and the implementation architecture motivate two Z sections that contain classical *behavioral* specifications.

As *functional properties*, we describe requirements what the system should do: giving access to the objects in the repository the user has permission to. In contrast, *security properties* in our setting state that the user *must not* access data that he is not allowed to, whatever combination of cvs-commands he applies.

Methodically, we need an interface between the operation schemas of the two architecture layers the behavioral part allowing to specify the safety properties. The trick is done by converting the suitably restricted operation schemas of both system layers into relations over the underlying state:

$$
\begin{aligned}
op_1 R &= op_1 \wedge R_1 & \qquad \mathrm{step} &= op_1 R \vee \ldots \vee op_n R \\
&\ldots & \mathrm{trans} &= \{step \mid (\theta state, \theta state')\}* \\
op_n R &= op_n \wedge R_n & \mathrm{SecProp} &= \forall\, \mathrm{trans}(|\ \mathrm{init}\ |) \bullet P
\end{aligned}
$$

In the above scheme, $op_i R$ represent the restricted operation schemas, their schema disjunction *step* the overall step relation of the system, that is converted into a transitively closed relation *trans*. The security property *SecProp* can be stated over the set of a state reachable via *trans* from an initial state.

We instantiate this scheme as follows: We assume that a user "knows" a set of pairs of roles and passwd, and that the user "invents" only files from a

given set of pairs from names to data in the add-operation. Further, we assume logins to be restricted to role/passwd's the user "knows" and adds restricted to name/data the user "invents").

Now we can postulate the following list of security properties $SecProp_P$:

1. Any sequence of CVS-operations starting from an empty working copy does not lead to a working copy with data the user has no permission to (except he was able to "invent" it),
2. any sequence of CVS-operations does not lead to a repository with "invented" data, except the user "knows" the corresponding role/passwd.
3. A CVS-administrator can withdraw all role/passwd for a user (after a suitable update of the authentication table); i.e. after a withdraw, the user does not get data into his working copy except by "inventing" it.

Moreover, there is the obvious property that the user can access "his" data.

This specification pattern is repeated on the level of the implementation architecture. Unfortunately, *implementing* one security architecture by another opens the door to *new* types of attacks on the implementation architecture, that can be completely overlooked on the abstract level: on the implementation level, we have more operations available (the schema disjunction *step* comprises additionally the UNIX commands "cp" or "setumask").

4.5 Summing Up

At present, the overall specification in [3] consists of ca. 80 pages. Its overall organization in Z-sections follows directly the overall scheme presented in Sec. 2.

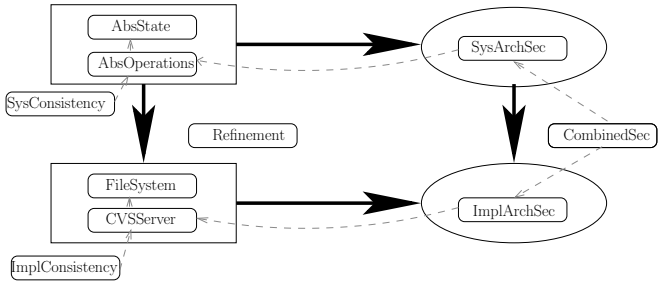

Fig. 3. The Specification Organization

The Z-sections `AbsState` and `AbsOperations` describe the system architecture, while the attached Z-section `SysConsistency` contains the consistency conditions (convervativity of axiomatic definitions, definedness of applications, deadlock-freeness of all operation schema). This is mirrored at the implementation architecture by the corresponding structures `FileSystem`, `CVSServer` and `ImplConsistency` respectively. The section `Refinement`, which contains the usual abstraction predicate relating abstract and concrete states, also contains

the proof obligations for the data refinements of the various operations. The security properties are defined and the proof obligations postulated in the sections `SysArchSec` and `ImplArchSec`; statements relating the security properties are kept in section `CombinedSec`.

5 Conclusion and Future Work

We have seen a security technology mapping based on an abstract role-based access control security model to concrete security mechanisms as offered by the traditional UNIX/POSIX security architecture. Our informal security analysis on the basis of our model establishes that the abstract security requirements were met by the properties of the concrete technologies.

Some methodological conclusion may be drawn from our case study:

- The good news is that refinement methodology can be used to greatly simplify the task of proving *some* security properties.
- The bad news is that a security technology mapping involves fairly concrete and complex models of the implementation technologies and the analysis of attacks against these. This represents for some vital security properties a barrier to our standard methodology to use as abstract models as possible.

This implies that *attacks against the implementation* must simply be taken more seriously, which means that models of implementation architectures deserve more attention as before, where more abstract models have been preferred. But in security, more abstract models are not necessarily better ones.

In the near future, we plan provide a more complete formal analysis in HOL-Z, i.e. more proofs of consistency, refinement and security properties on both levels of the abstraction. We believe, that the proof architecture and the model of the UNIX filesystem will be reusable for a wider range of similar applications.

References

1. `http://www.informatik.uni-freiburg.de/~softech/software/cvs/`.
2. A. D. Brucker, S. Friedrich, F. Rittinger, and B. Wolff. HOL-Z 2.0: A proof environment for Z-specifications, 2002. Submitted.
3. A. D. Brucker, F. Rittinger, and B. Wolff. A CVS-server security architecture — concepts and formal analysis. Technical report, Albert-Ludwigs-Universität Freiburg, Jan. 2002. An preliminary version is available at `http://wailoa.informatik.uni-freiburg.de/WebBIB/preliminary/cvs_sec.pdf`.
4. P. Cederqvist et al. *Version Management with CVS*, 2000.
5. D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39. World Scientific Publishing Company, Singapore, 1993.
6. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, Feb. 1996.
7. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Englewood Cliffs, NJ, 1996.
8. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.