Implementation of a Distributed Security Monitoring Tool: ESOPE

Rodolphe Ortalo[1]
ortalo@laas.fr

LAAS - CNRS
7, avenue du Colonel Roche
31077 Toulouse
France

Abstract

This article presents the details of the software architecture adopted for the implementation of the ESOPE security audit tool. ESOPE is a tool designed for operational monitoring of the security of networked Unix systems. It is implemented in the Perl language. ESOPE incorporates an original technique for audit results analysis: quantitative security evaluation. This method allows to isolate precise and noteworthy security events. This paper presents the overall operation of ESOPE, but focuses primarily on the implementation of the audit tool software. Various aspects of the architecture are presented, including: a meta-object protocol for providing encryption or transparent remote execution to software modules; a hierarchical distribution supervision tree; and a two layers object-oriented design for the implementation of vulnerability analysis procedures.

1 Introduction

Managing the security of a network of workstations remains a difficult task for system administrators. Several security audit tools are available in order to help them to monitor the computer system security, however such tools usually require significant human intervention to fit the specific needs of one organization. The raw output of common security audit tools can be extremely verbose and a cumbersome manual or semi-automatic analysis of the reports is necessary to extract significant facts from these reports. Such tools are rarely easy to configure and it is usually not possible to focus their activity on specific areas of the computer system, even if system administrators know the most sensitive parts of their system. Due to these problems, home-made scripts tailored to the specific needs of one computer system still play an important role in security management.

Given these problems, it seems that a security audit tool should provide detailed and exhaustive information concerning the security flaws and access rights configuration of all the hosts existing inside the organization, but it should not provide directly this information to system administrators (at least unless requested explicitly). The tool that analyses the audit results should try to cope itself with the difficult task of extracting interesting events from the exhaustive audit reports. The analysis tool should also allow the system administrators to specify the security objectives they are primarily interested in.

In previous work [1,2,3], we developed a quantitative security evaluation method in order to propose a solution to the analysis problem. Audit results analysis triggered by the evolution of the security measures proposed in [4] allows to identify security events that seems interesting enough to be presented to system administrators.

This original technique is embedded within an experimental tool, named ESOPE. The first version of this tool was primarily used for validation of the interest of this quantitative approach for security monitoring. The encouraging results obtained with this prototype as well as the experience obtained using conventional security audit tools (such as COPS [5], SATAN, etc. [6]) led to the development of a second version of ESOPE. In this second version, we adopted an originaL software architecture for performing the security

1. R. Ortalo is now at: ONERA - Centre de Toulouse, 2 avenue Edouard Belin, 31055 Toulouse, France.

audit of the computer system. This new audit tool allows a detailed analysis of a *distributed* Unix computer system, and should easily allow integration of new code for scanning additional vulnerabilities.

In this article, we focus on implementation issues. We explain the role of the security audit tool in the entire ESOPE execution process, and we present in more detail the software architecture adopted for performing the audit.

Unix is the primary target platform of ESOPE. We are especially interested in receiving feedback and opinions concerning the design choices made for ESOPE, both from potential users, i.e. Unix system administrators, and from experienced Unix security or audit tools developers.

The organization of this article is the following. Section 2 presents the overall operation of ESOPE and the relationship between the security audit tool and the quantitative evaluation and results analysis tools. Section 3 summarizes the design objectives of the current version of the software. Section 4 presents in detail the main elements of the software architecture which was adopted for the audit tool. Section 5 presents the overall implementation guidelines of the security vulnerabilities analysis software, executed within the audit tool context. Section 6 outlines some future plans for the development. Finally, section 7 draws a short conclusion.

2 Presentation of ESOPE

2.1 Overall operation

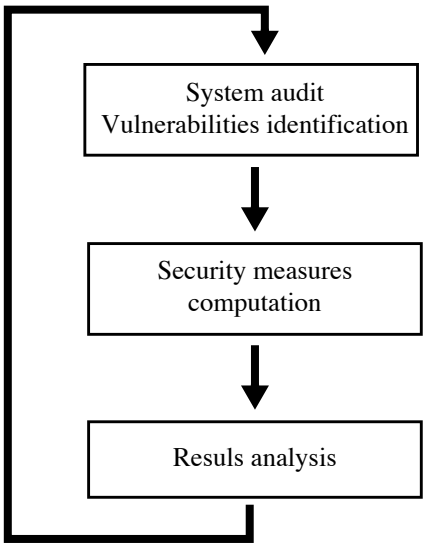

Figure 1 - ESOPE overview

One full execution of ESOPE involves three main steps:

- In the first step, the tool performs the audit of the target computer system. During this audit, several programs are run on all the computer hosts in order to identify the existing vulnerabilities. For example, vulnerabilities may be associated to erroneous configuration of access rights of sensitive files (either user-specific or host-related), inconsistent settings between different hosts, program binaries exhibiting known weaknesses, etc.
- In the second step, a quantitative evaluation technique presented in [3] is used to assess the overall impact on the security objectives of the various vulnerabilities identified in the system.
- In a third step, the security measures and especially their evolution between two successive executions are analysed in order to identify, if necessary, the precise security events that caused them. This analysis can reveal the occurrence of one or several new vulnerabilities which impact on the

security of the system is significant. In this case, the system administrator may decide to perform some corrective actions. Otherwise, executions of full runs proceed at regular intervals.

In this article we focus on the *first* step of a full execution: the audit. The audit tool should analyse all the resources of the computer system, both global and local to each computer host involved. Such accurate and detailed security audit necessitates a sophisticated architecture.

2.2 Overview of audit results analysis

In order to clarify further the overall operation of the entire tool, in this section we give a fast presentation of the analysis that can be performed over the data collected by the audit system. Such automatic analysis is absolutely necessary to provide to the system security administrator interesting, motivated and *concise* reports.

2.2.1 Raw results

Raw results of an audit can be useful to the system administrator. But in the absence of specific treatment, it is up to this administrator to guide the analysis. Of course, some tools are available in ESOPE to extract from the data repository some specific piece of information that may be requested. Furthermore, this data can be sorted and filtered in various ways thanks to Perl flexibility in this domain. But exploiting the raw audit information is usually feasible only when one already knows more or less precisely what he searches for. It is rare that reliable automatic security alarms can be raised directly on the basis of raw auditing results, except perhaps for the most severe security vulnerabilities classes. Hence, the exhaustive information logged in the data repository is mostly valuable for very critical alarms or *a posteriori* analysis.

2.2.2 The privilege graph model

The set of all the vulnerabilities identified by ESOPE include many possible ways of obtaining new privileges in a Unix system. This information covers all categories of users and data and includes *legitimate* privilege transfer mechanisms. Some of this information is relevant only due to the *transitivity* of privilege transfers. Isolated, one vulnerability may seem unimportant. But a combination of several vulnerabilities may be used by an attacker to progress in the system and gain more and more privileges. We use a model for representing all the security vulnerabilities of the system, called a privilege graph[2] [1], that takes into account this transitivity. Hence, the model allows to represent the global security state of the computer system, and the potential interactions between all vulnerabilities, including security flaws, permissive access rights, or legitimate privilege transfer mechanisms.

2.2.3 Security objectives and quantitative evaluation

The security objectives of the computer system are included by the identification in the privilege graph of two specific nodes corresponding to a target to protect and the potential privileges of an attacker[3]. Then, the quantitative evaluation technique presented in [3,4] can be used to evaluate the global effort needed by this attacker to obtain the privileges of the target using the vulnerabilities existing in the system. This effort value represents the ability of the computer system to resist on average to one successful attack. We shown in a previous experiment that the evolution of this measure allows to identify, among all the vulnerabilities recognized by the audit system, specific ones that appear to be of most interest for the security of the entire system. These security measures can be used to trigger alarms. The privilege graph is used to analyse the causes of a significant measure evolution and provide the security administrator the information

2. In this graph, nodes represent sets of privileges, e.g. one user's rights, and arcs represent vulnerabilities. An arc exists between two nodes if it is possible, using the privileges of the origin node to obtain, legitimately or not, the privileges of the target node.
3. Note that this technique allows to take into account attackers that are normal users of the computer system, i.e. *insiders*. Outsiders also appear in the privilege graph via a special (virtual) node.

necessary to understand the alarm, i.e. the relationship between the vulnerabilitie(s) brought to his attention and the security objectives he specified.

3 Design issues

In this section, we summarize the design objectives of the ESOPE auditing system:

- First, a typical target system for the audit tool is a *network* of Unix workstations. (Primarily, ESOPE focuses on the Unix operating system.) A secondary design objective was that it should be possible to monitor dedicated network devices (such as routers, etc.) and *potentially* extend the audit tool to other operating systems.
- The audit tool should analyse exhaustively all the resources of a Unix system, including shared global resources and resources local to each computer host. Therefore, a distributed execution infrastructure for remote execution of audit agents is necessary.
- It is our feeling that an efficient security administration tool should be fast to develop and extend (especially to take into account new vulnerability classes). Development speed and easy of maintenance is also a strong requirement.
- Given the huge amount of resources that such auditing tool should potentially be able to analyse, some care must be given to performance considerations. It was *not* an objective to provide the most efficient tool, but some optimization framework must be provided if such tool is expected to complete its work within a realistic period of time.
- Finally, this tool is our primary platform for experiments with the quantitative security evaluation method we research. Interoperability with the quantitative computation programs had minor impact on the design, but it is one of the requirements.

These requirements motivated the various design choices presented in the next sections.

4 Software architecture

4.1 Distributed architecture*: Supervisors* and *Executors*

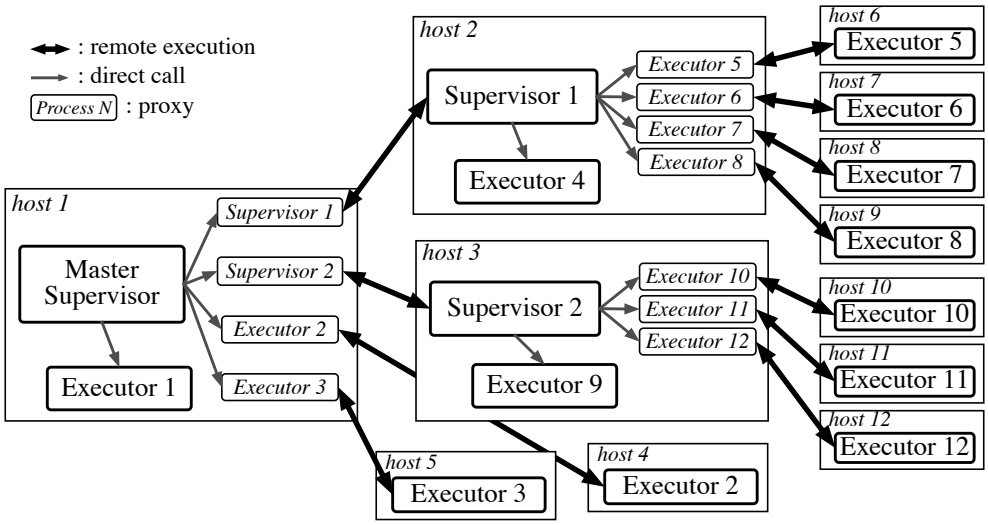

Figure 2 - Distributed execution

The need for an audit tool capable of remote execution led to the definition of a distributed execution strategy for ESOPE. The execution of the security audit on a set of interconnected computer hosts relies on two types of agents: *supervisors* and *executors*. *Executors* perform the execution of the desired security vulnerabilities detection modules over local resources of their home host. Some specific *executors* may be asked

to analyse also some shared resources such as NFS filesystems[4], SNMP-capable network devices, or resources managed by non-Unix hosts. The *supervisors* control the execution of *executor* agents: they allocate and spread *executors* to their target hosts (including their own local host), they configure their execution and monitor them. One *supervisor* agent can also decide to propagate recursively additional *supervisor* agents to limit the number of agents under its direct control. Therefore, a full execution of the ESOPE audit system involves the (concurrent) deployment of a tree of controlling agents (the *supervisors*), followed by the execution of actual workers (the *executors*) on every target host. Figure 2 shows an example of this architecture with the maximal number of remote agents controlled by each *supervisor* arbitrarily set to 4 (in real operation, this limit depends on the performance of each supervising host and is typically much higher).

4.2 Programming language and main modules

The programming language used in ESOPE is Perl5 [7]. This choice was motivated primarily by the wide availability of Perl5 interpreters on all variants of Unix platforms, and the fast development cycle that this language allows [8]. The use of an interpreted language also solves several of the platform independence issues that could have been problematic within ESOPE. Finally, Perl provides a convenient framework for structuring the entire tool in separate modules [7] and much of the modularity of ESOPE itself is based on Perl modules. The most interesting modules of the ESOPE audit system are listed in Table 1.

Module category	Module names
Library	My_DBFile MOP::MOP Observer Connection Mailbox and SocketMailbox
Meta-module	MOP::MetaModule Remote SecureConnection
Distribution control	Supervisor Executor
Security analysis	Fact::* Vulnerability::* Accessor and Thing

Table 1 - Main Perl5 modules

In the following sections, we provide additional information concerning the internals of most of these modules. But, from a global point of view, we can identify several modules categories:

- classical library modules provide sets of functions or procedures used at different steps of system audit, e.g. for network communication, or persistent data repository management;
- several *meta-modules* are used for remote execution control or network encryption; this type of module is based on an original programming technique used in ESOPE (via the MOP::MOP library module) and is covered in detail in section 4.4.1;
- the distributed execution strategy implemented in ESOPE (section 4.1) is implemented in two object-oriented modules named: Supervisor and Executor;
- finally, all the security audit procedures are implemented as object-oriented modules, organized within two hierarchies: Fact::*[5] and Vulnerability::*. The exact organization of these hierarchies, as

4. Especially when these filesystems are served by dedicated machines. However, when it is possible, it is generally more efficient to spread an executor directly on the NFS server, especially during light load hours.

well as the Accessor and Thing base modules used by all members of the hierarchy, is detailed in section 5.

4.3 Centralized data repository

ESOPE manages a centralized data repository containing all the data produced at the various steps of its execution. The security audit step of a full ESOPE execution regularly stores vulnerability reports in the data repository (typically every day). This information is complemented by quantitative measures computation and analysis of the measures and vulnerabilities evolution. Some specific Perl scripts can be used to generate text reports from the data stored in the repository. The data repository is currently implemented via a two-level *tiedhash* data structure supported by the MyDB_File module. Hence, it consists of a collection of regular (compressed) Unix files. MyDB_File is a simple interface layer (based on Perl *tiedhash*es) that allows to use directly the information stored in the repository from all Perl programs, like a regular in-memory hash table.

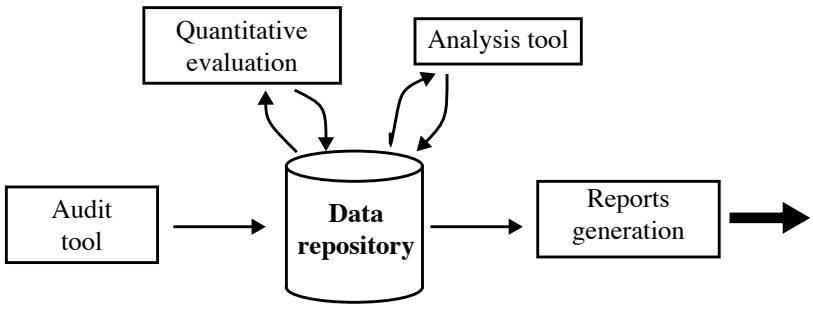

Figure 3 - Data repository

4.4 Programming paradigms

4.4.1 Meta-object protocol

ESOPE relies on an original programming paradigm for implementing remote execution capabilities and encryption: reflective programming. This programming technique has been well addressed in the literature. The approach taken within ESOPE is heavily inspired from [10]. In this approach, reflective programming is available at the object level and is called meta-object programming [13]. Meta-object programming relies on the existence in the target language of a meta-object protocol (or MOP).

Perl modules can be used as classes definitions, and special scalar values, *blessed references*, can be used as objects. The language then provides syntactical support analogous to method calls in regular object-based languages (especially C++) [7]. We have implemented a meta-object protocol for Perl objects in the MOP::MOP module.

The overall behaviour of a method call in the presence of the MOP and a meta-object is shown in Figure 4. Essentially, the MOP allows to intercept any method call my_method() made on a regular object (or class) and reroute this method call towards a special meta_method_call() method call on a different object the *meta-object*. Of course, this meta-object can subsequently call the actual method of the original target object, thanks to a specific method named handle_method_call() which is available at the meta-level. But the meta-object can also perform various operations on the method arguments and execute additional code before or after the real execution of the target method. It may also reroute the method call to a different object or a different method. The arguments passed by the application programmer to the initial method are also passed to the meta-object but it may choose to preprocess them before passing them to the base-level

5. In this notation, the * is used as a wildcard and designates any element of the module hierarchy. Actual module names are, for example, Fact::Files::UnixFilesystems or Vulnerability::Trusted::User.

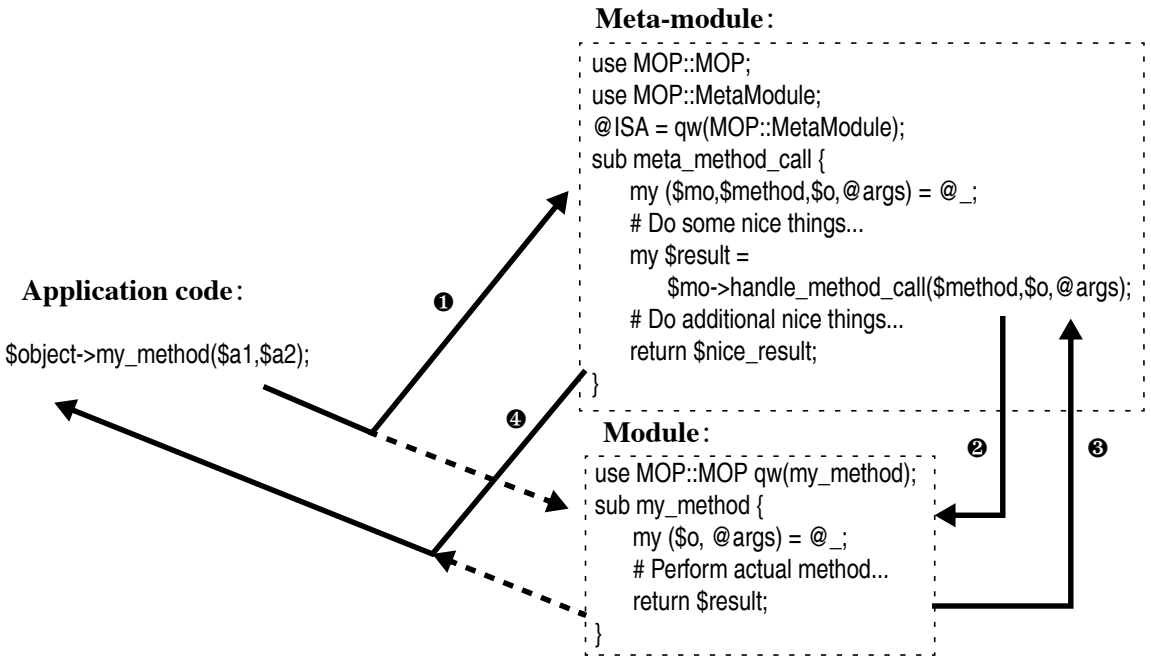

Figure 4 - Meta-object protocol

method (between ❶ and ❷), or postprocess the result of the method call before sending it back to the calling code (between ❸ and ❹).

In our context, in Perl5, we call *meta-modules* the modules defining the behaviour of meta-objects. All meta-modules must inherit from a base module named MOP::MetaModule which provides a default implementation of the meta_method_call() method[6] as well as the handle_method_call() method needed to execute the base-level method on the base-level object.

Both the meta-modules and the modules that wish to expose their interface to the control of a meta-object must also use the MOP::MOP module. This module provides a set of functions (mostly useful to meta-modules) that manages an application-wide mapping of (*module*,*meta-module*) and (*object*,*meta-object*) associations. MOP::MOP also activates the MOP control flow on all the exposed[7] methods of the base module.

Even though this is not used inside ESOPE, note that this process can be applied recursively. Hence, a meta-module itself can expose some of its methods (including meta_method_call()) to another meta-module. In such a scheme, we would end with three objects: a (base-level) object, a meta-object, and a meta-meta-object.

The interest of this programming paradigm for real-world issues lies first in the fact that it allows independent programming of the base-level methods which are mainly application-related, and of the meta-level methods which are usually associated to high level management tasks such as security management, remote execution control, or also checkpointing or replicas management for fault-tolerance, etc. However, this independence does not prevent the code executed in the meta-object from controlling effectively all the behaviour of the base-level object. For example, all attributes are visible and can be accessed and modified from the meta-level[8].

Second, it is very easy to activate the MOP on an object-oriented module. This allows a very flexible association of some specific meta-module (e.g. encryption-related) with different kind of modules. This ap-

6. This default implementation simply performs the base-level method call. Hence, the default meta-module essentially does nothing special with respect to the normal behaviour of a Perl method call.
7. It is possible to expose to the meta-object protocol a *subset* of all the methods available in a normal module. For example, a module containing three methods named method1(), method2() and method3() can restrict the MOP awareness to the first and third method. The syntax used is then: use MOP::MOP qw(method1 method3);
8. Especially in Perl. But this remark is also applicable to meta-object programming in less virtuous languages [14].

proach promotes separation of concerns and reuse for pieces of software traditionally difficult to design or reuse due to heavy integration with application-specific code (as is usually the case with security mechanisms, or replicas management). The interested reader may refer to [10,11,12] for additional information.

4.4.2 Message passing communications

ESOPE includes several communication modules, namely: Connection, Mailbox and SocketMailbox. This set of modules provide a message passing communication system for the Perl programs over conventional TCP/IP sockets.
The module Connection provides an object-oriented API for sockets, and allows to send or receive binary data on several connections. It is inspired from a similar module presented in [9]. Modules Mailbox and SocketMailbox provides a message passing mechanism based on the Connection module: they manage the creation and destruction of needed connections between hosts, allow to share a connection object between multiple mailboxes, and manage the conversion of Perl internal data structures to and from a format suitable for network communication (using the standard Data::Dumper module).

4.5 Remote execution

Remote execution features similar to RPC are implemented in ESOPE using meta-object programming. Unlike the situation presented in section 4.4.1, *two* meta-objects (from meta-module Remote) are involved: one local and one remote. The local meta-object takes advantage of its position to redirect all method calls made on a local object towards a second remote meta-object, via communication mailboxes. This second meta-object performs the execution of the method on the base-level object that it controls and returns the result to the first meta-object for delivery to the original caller.
In fact, the most specific part of this execution path takes place at creation of an object. The meta-module also relays object creation to the remote host, where the creation of the object is performed. On the local site, only a stateless entity is created and given back to the application to support the meta-object protocol. This entity[9] plays the role of a proxy. Two meta-objects are also created, both on the local and remote site, and associated to the proxy and remote object respectively. The application can use the proxy like a normal object but all method calls are performed remotely.

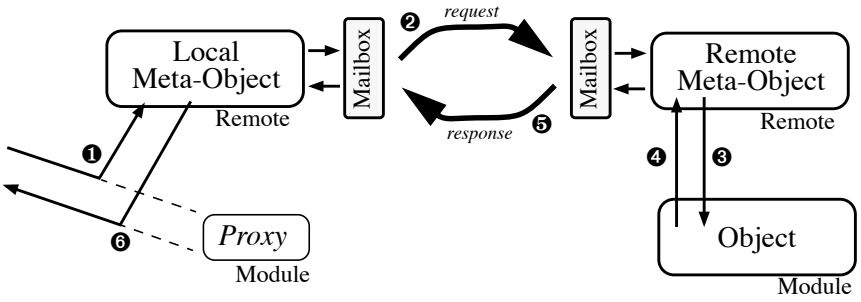

Figure 5 - Remote execution

However, due to the object-oriented mechanisms available in Perl5, such proxy object cannot be used exactly like a normal object. Method calls are trapped by the MOP, but direct access to the data elements of the blessed reference[10] are not intercepted. Hence, all attributes of an object should be accessed via a *method* and not directly via a hash look-up. But this is a known issue in Perl OO programming, and several mechanisms can be used to facilitate automatic management of object attributes, such as the AUTOLOAD mechanism [7].
Finally, the Remote module provides both blocking and non-blocking method calls[11]. In the blocking case, the local meta-object waits until the response of the remote meta-object is available before returning. In

9. Which is in fact a blessed reference to an *empty* hash.
10. Which is also a Perl reference in the first place.

the non-blocking case, the local meta-object sends a request message to the remote host, but returns the control to the calling application immediately after sending the message. Instead of returning a result, the meta-object returns a Perl anonymous subroutine to the application. Afterwards, the application can execute this subroutine to obtain (and potentially wait) the remote meta-object answer. This mechanism allows a single Perl program to perform some additional work while asynchronous procedures are executed remotely, or to start simultaneously *several* remote executions on different hosts and wait subsequently to gather all the results.

4.6 Security issues

Now, ESOPE self-security should be addressed. Some experimental mechanisms have been added to improve the confidentiality of the data exchanged on the network between the various agents during an audit. Especially, all the vulnerabilities identified by the *executor* agents should not be sent out in clear over the network.

Meta-object programming was used again in order to provide (relatively) secure communications between the various agents of ESOPE. As an object-oriented interface to socket-based communications between two computer hosts had already been included among the support modules of ESOPE (module Connection), a new meta-module (SecureConnection) was developed which generic behaviour is to encrypt method arguments at call time and decrypt method results. Activated on the methods available in the Connection module, this SecureConnection meta-module allows transparent encryption with a block cipher of the messages submitted to a connection object for network transmission, as well as decryption of the received messages before their delivery to the application program. This behaviour in the context of remote execution is presented in Figure 6.

In addition to mere encryption of the application data sent via TCP/IP connections, the SecureConnection meta-module also performs additional modifications of the usual behaviour of the Connection module. For example, on creation of one connection object with another host, the meta-module performs mutual authentication via a simple challenge-response protocol using the secret key provided by the application. It also generates a random internal key for actual encryption of the network link. Furthermore, this internal key is changed at regular intervals. It is noteworthy that integration of these enhancements was very easy thanks to the clean separation between the security mechanisms (in the meta-module) and socket programming (in the base-level module), achieved via meta-object programming.

4.7 Observation system

For demonstration and debugging purposes, ESOPE also incorporates an observation system. An Observer module provides several functions that an agent can use to send status reports, distribution state or error messages to an observing host. When observation is activated, the messages containing such information are sent over UDP connections towards some independent process (usually run on the initial master host). This *observer* process listens to UDP messages and display them. Different *observer* programs are currently available. A simple *observer* directly prints raw messages on a text terminal, possibly filtered according to their type. A more sophisticated *observer*, based on the Amulet graphical library [16], displays the status of the audit system graphically, as shown in Figure 7.

This graphical observation program is complemented by a graphical user interface that allows to start manually the various elements of ESOPE (security audit, quantitative measures computation, measures evolution analysis). However, ESOPE remains primarily a batch-oriented program: the GUI is mainly aimed at demonstrational purposes.

11. Currently, non-blocking behaviour is requested by adding a the 'NONBLOCKING' constant to the method call arguments. To identify this request, the meta-object inspects the last element of the arguments list of the trapped method and either recognizes it or leaves the list untouched.

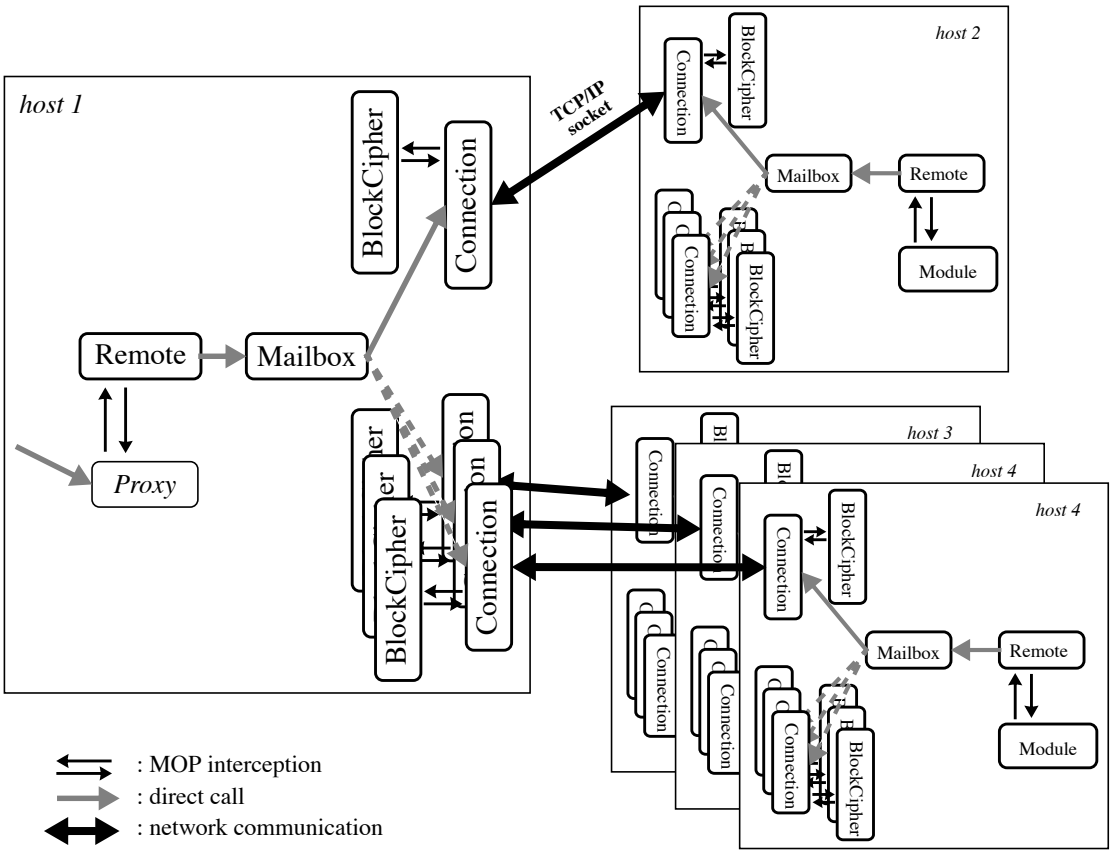

Figure 6 - Self-security

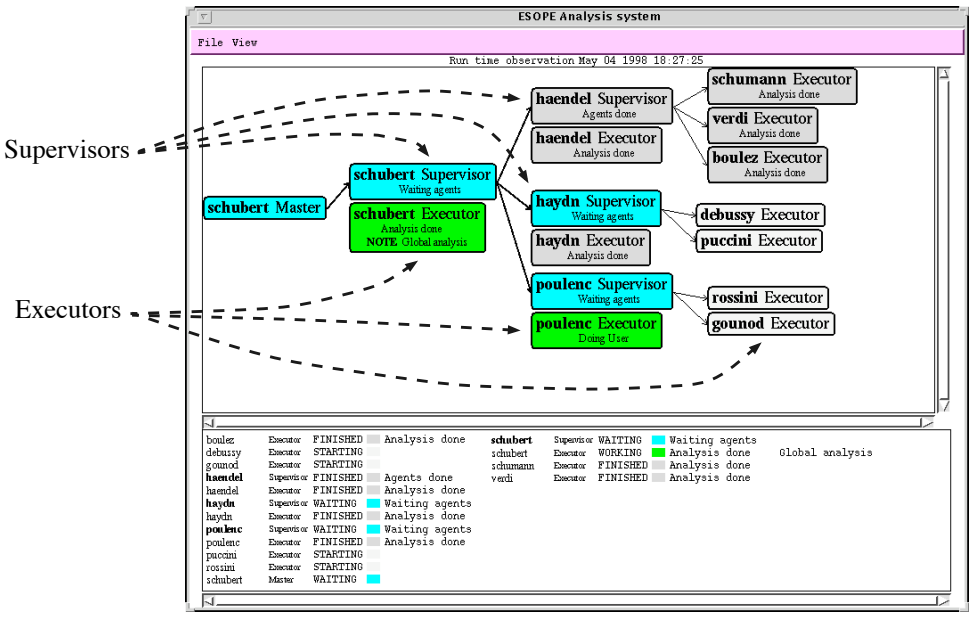

Figure 7 - Observation system

5 Vulnerabilities detection

5.1 Overview

The various vulnerability detection procedures of ESOPE (executed by *executors*) are implemented via Perl5 modules. All modules of the Vulnerability::* hierarchy correspond to types of security vulnerability that should be detected during the audit and recorded in the data repository. Such vulnerabilities may be associated to classical security flaws (such as defective programs), to user misuse (such as incorrect configuration of access rights on initialization files), or to usual privilege transfer mechanisms (e.g. `.rhosts`).

To facilitate the development of new vulnerability audit procedures without compromising efficiency, a two layers software design has been chosen. The Vulnerability::* modules do not directly consult the information available in the computer system. Vulnerabilities rely on information provided by the Fact::* modules. These Fact::* modules provide access to the information available from the operating system, such as the file names of files existing in a filesystem, the users lists, the content of some configuration files. Thanks to the layering, system-level information is available to Vulnerability::* modules via a uniform API. A single Fact::* module may be used by several vulnerability testing modules and OS accesses may be cached or optimized. The fact layer also hides from the higher vulnerability layer several system-dependent or platform-dependent problems, such as the full users list determination (e.g. `/etc/passwd`, NIS or NIS+ tables may be used) or the access rights semantics (e.g. with respect to group permission with BSD or SVR4, or with foreign shared filesystems such as NTFS). The vulnerability layer focuses on the decision of the presence or absence of a precise vulnerability given the information available. Of course, application code (such as the code of an *executor*) primarily accesses to the Vulnerability::* modules, and not directly to facts[12].

These two layers share a common API. This API is similar to the one of a *list* iterator, and is implemented in the Accessor module. To operate with such *accessors*, all Fact::* and Vulnerability::* modules share a common base class, implemented in the Thing module.

5.2 *Accessor* API

Access methods	General methods	Search methods
first() next() current() all()	copy() dump() restore()	contains() search() filter() find() constrain()

Table 2 - *Accessor* API

The various methods available via the Accessor module are presented in Table 2. They correspond to:

- usual access methods to iterate over the whole list of elements;
- the ability to copy, save and restore the whole list;
- and searching capabilities, with the ability to:
  - search one specific item within the list;
  - or to filter the entire list against some specific criteria (or filter function).

The search methods exist in two versions:

- search() and filter() are passed as a parameter an anonymous test function which should accept one object as argument and return a boolean value when this object should be selected;

12.This is not always true, as some Fact::* modules are used sometimes. For example, the Supervisor module also relies conveniently on the fact layer to obtain the list of the existing hosts on the local network.

- find() and constrain() are passed a list of pairs (*attribute*,*value*) . The searched object(s) are selected based on the values of the given attributes. These versions of the search methods exist as they may be optimized by the underlying fact module[13].

The Accessor module allows to iterate over the list content. It is associated to the Thing module which is a base class from which should inherit all the items that may be accessed via an *accessor*. The strict association is due to the fact that some specialization of the Accessor module are available for fully in-memory lists, progressive construction and caching of the list elements, or dynamically constructed list elements. Such specialization should be associated to the corresponding specialization of the Thing module. The entire diagram for these modules is presented in Figure 8.

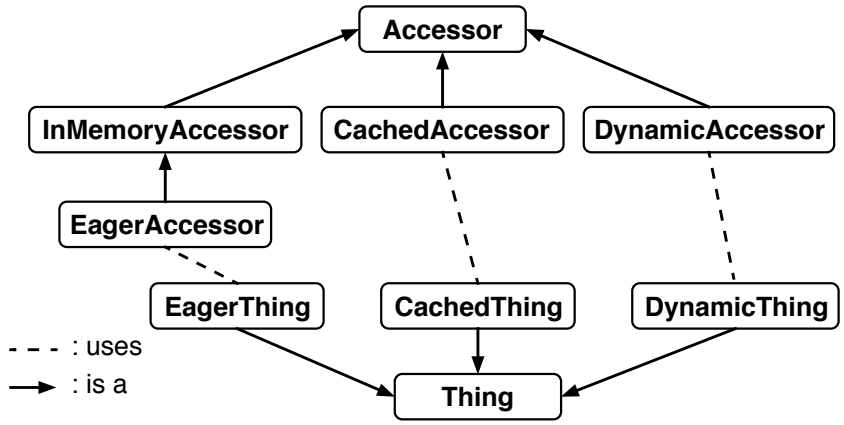

Figure 8 - Access modules class diagram

Each specific *thing* module provides suitable public methods for the corresponding *accessor* module. Each *thing* module also exhibits an internal private API that can be overloaded by Fact::* or Vulnerability::* modules (hence, we use a delegation pattern [15]). This internal API is presented in Table 3. It corresponds to the methods that one should implement in order to provide a new vulnerability audit module with the desired behaviour (optional methods are *italicized*).

EagerThing	CachedThing	DynamicThing
_all_in_system() *_how_to_index()*	_first_in_system() _next_in_system() *_how_to_index()* *_find_with_criteria()* *_can_do_constraint()* *_first_with_constraint()* *_next_with_constraint()*	_first_in_system() _next_in_system() *_find_with_criteria()* *_can_do_constraint()* *_first_with_constraint()* *_next_with_constraint()*

Table 3 - *Thing* private API

As can be seen from Table 3, in the simplest case (module EagerThing) only one method is required. Such an implementation may be inefficient as it corresponds to eager analysis of the whole target resource(s). Another specialization (CachedThing or DynamicThing) correspond to on-demand access and may be more efficient, but they require more implementation work, especially when search functions are provided.

6 Future plans

Running ESOPE over a set of several hundred workstations is feasible, but unreliable. The distribution control tree manages the execution with very good performance, but the deployment of the various agents

13.In the absence of such optimization, find() and constrain() simply fall back to search() and filter().

in the initial phase of the distributed execution is frequently compromised by failed hosts or failure of the remote execution system[14]. Improvements to ESOPE dependability should be addressed soon.
The two layers software architecture adopted for coding the audit modules (see section 5) offers the opportunity of lazy analysis of the system. We would like to study the efficiency of ESOPE for on-demand audit of specific aspects of the computer system, such as local configuration files, etc. This would allow to use ESOPE not only for periodic scheduled audit of the system security but also for dynamic reactive information gathering.
Even though ESOPE was primarily dedicated to security-related analysis of the computer system, it may be useful as a generally data gathering system. On the contrary of proprietary management consoles, ESOPE is largely non-intrusive and can be adapted to specific needs. However, this implies the integration in ESOPE of new sources of information. In a first step, we would like to adapt ESOPE to take into account data available via the SNMP protocol.

7 Conclusion

ESOPE is not readily available to the public in its entirety. The security-related nature of the tool prevented an immediate full distribution from being chosen initially by the supporting organization. However, the most interesting parts of its software architecture that are not security-related have been isolated and are distributed separately. Most notably, the meta-object protocol modules are available on the CPAN (`http://www.cpan.org/`) in the MOP package. *(Updates of the v.1.0 distribution should be ready very soon.)* Controlled distribution of the entire tool is also possible. Readers interested in obtaining the full system should contact the author.
In its current state, the ESOPE audit system easily performs the audit of 30 Unix workstations. The first results confirmed the interest of a distributed audit tool: the audit of the local resources of workstations often reveals surprising vulnerabilities. However, especially when non-critical security vulnerabilities are taken into account, the size of a single audit raw report is impressive. The quantitative evaluation technique developed for security assessment seems to be suitable for managing such important amount of raw audit results. The information which is finally provided to a system administrator is filtered, precise, and explained. The entire tool suite is based on classical Unix tools and could easily incorporate users feedback and third party development. Overall, we feel that the various choices made within ESOPE could be of practical interest to the Unix community, either for experimenting with ESOPE itself, or for designing similar tools.

Bibliography

[1] M. Dacier, Y. Deswarte, "Privilege Graph: an Extension to the Typed Access Matrix Model", in *Third European Symposium on Research in Computer Security (ESORICS 94)*, Brighton, United Kingdom, Lecture Notes in Computer Science 875, pp.317-334, ISBN 3-540-58618-0, Springer-Verlag, 1994.

[2] M. Dacier, Y. Deswarte, M. Kaâniche, "Models and Tools for Quantitative Assessment of Operational Security", in *12th IFIP Information Systems Security Conference (IFIP/SEC'96)*, Samos, Greece, May 21-24, pp.177-186, ISBN 0-412-78120-4, Chapman&Hall, 1996.

[3] R. Ortalo, Y. Deswarte, M. Kaâniche, "Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security", *IEEE Transactions on Software Engineering*, vol.25, no.5, September/October 1999. (extended version of [4], *to appear*)

[4] R. Ortalo, Y. Deswarte, M. Kaâniche, "Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security", in *Dependable Computing for Critical Applications 6* (DCCA-6), (M.D. Cin, C. Meadows, W. Sanders, eds.), 5-7 mars 1997, Grainau, Allemagne, Dependable Computing and Fault-Tolerant Systems, vol.11, IEEE Computer Press, ISBN 0-8186-8009-1, pp.307-328, 1998.

[5] D. Farmer, E. H. Spafford, "The COPS Security Checker Sytem", in *Summer Usenix Conference*, Anaheim, USA, 1990.

14.Especially when using the `rsh`-based remote execution starter. The deployment of a permanent service started by `initd` would be preferable, but `rsh` is the least intrusive technique for the network, especially during ESOPE own development.

[6] S. Garfinkel, E. Spafford, *Practical Unix & Internet Security*, 2nd edition, 971 p., ISBN 1-56592-148-8, O'Reilly & Associates, 1996.
[7] L. Wall, T. Christiansen, R. L. Schwartz, *Programming Perl*, 2nd edition, 645 p., ISBN 1-56592-149-6, O'Reilly & Associates, 1996.
[8] M.-J. Dominus, "Perl: Not Just for Web Programming", *IEEE Software*, pp.69-74, January-February, 1998.
[9] S. Srinivasan, *Advanced Perl Programming*, O'Reilly & Associates, 404 p., ISBN 1-56592-220-4, 1997.
[10] J.-C. Fabre, T. Perennou, "A Metaobject Architecture for Fault-Tolerant Distributed Systems: the FRIENDS Approach", in *IEEE Transactions on Computers*, *Special issue on Dependability of Computing Systems*, pp.78-95, 1998.
[11] M.-O. Killijian, J.-C. Fabre, J.-C. Ruiz-Garcia, S. Chiba, "A Metaobject Protocol for Fault-Tolerant CORBA Applications", in *IEEE Symposium on Reliable Distributed Systems*, West Lafayette, Indiana, USA, pp.127-134, 1998.
[12] J.-C. Ruiz-Garcia, M.-O. Killijian, J.-C. Fabre, S. Chiba, "Optimized Object State Chekpointing using Compile-Time Reflection", in *Workshop on Embedded Fault-Tolerant Systems (EFTS'98)*, Boston (USA), pp.46-48, 1998.
[13] G. Kiczales, J. des Rivières, D.G. Brobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.
[14] S. Chiba, "A Metaobject Protocol for C++", in *ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, Austin, Texas, USA, pp.285-299, 1995.
[15] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
[16] B.A. Myers, R.G. McDaniel, R.C. Miller, A.S. Ferrency, A. Faulring, B.D. Kyle, A. Mickish, A. Klimovitski, P. Doane, "The Amulet Environment: New Models for Effective User Interface Software Development", *IEEE Transactions on Software Engineering*, vol.23, no.6, pp.347-365, June, 1997.
[17] L. A. Stein, "Delegation is Inheritance", in *ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, Orlando, pp.138-146, 1987.