Secure Management by Delegation within the Internet Management Framework

J. Schönwälder
Computer Science Department
Technical University Braunschweig
Bültenweg 74/75
38104 Braunschweig
Germany
schoenw@ibr.cs.tu-bs.de

J. Quittek
C&C Research Laboratories
NEC Europe Ltd.
Hardenbergplatz 2
10623 Berlin
Germany
quittek@ccrle.nec.de

Abstract

This paper discusses the security aspects of the IETF Script MIB which integrates the Management by Delegation Model into the Internet Management Framework. The paper shows how SNMPv3 security mechanisms have been utilized to protect the Script MIB from unauthorized access. This paper also describes the security aspects of a prototype implementation which uses the Java virtual machine as a runtime system for delegated management functions.

Keywords

Internet Management, Management by Delegation, Scripting, Java, Security

1 Introduction

Large networks are difficult to manage with centralized systems. The Management by Delegation (MbD) model was proposed in 1991 to address this problem [1]. The key idea behind the MbD approach is to delegate management functions to remote devices in order to reduce communication costs, to abate the micro-management problem, to avoid a single point of failure and to increase the scalability of management applications.

The publication of the MbD model inspired a number of researchers to develop several prototype implementations. The network management standards bodies, namely the IETF and ISO, later started work on integrating the MbD model into their management frameworks. In 1998, ISO published the Command Sequencer Systems Management Function [2], which is a set of GDMO classes that can be used to transfer, launch and control management scripts via CMIP operations.

The IETF chartered the Distributed Management (DISMAN) working group in 1996 to work on (i) a standard expression and threshold monitoring MIB for decentralized event generation and (ii) a standard MIB for delegating management scripts within the Internet management framework [3]. An overview and a comparison of the two approaches taken by ISO and IETF to integrate MbD into their management frameworks can be found in [4].

This paper takes a closer look at the security aspects of the IETF Script MIB [3] and it describes a prototype implementation developed jointly between the Technical University of Braunschweig and C&C Research Laboratories of NEC Europe Ltd. at Berlin. The paper is organized as follows. The general structure of the IETF Script MIB is presented in Section 2. Section 3 reviews the security mechanisms provided by SNMP version 3 (SNMPv3). Section 4 explains how the SNMPv3 security mechanisms have been used to secure the Script MIB. An implementation of the Script MIB which uses the Java virtual machine [5] as a runtime environment is described in Section 5. We conclude with some remarks about future work items in Section 6.

2 Overview of the IETF Script MIB

The Script MIB [3] defines a standard MIB for the delegation and invocation of management functions based on the Internet management framework. In particular, it provides the following capabilities:

- Transfer of management scripts to a distributed manager.
- Initiating, suspending, resuming and terminating management scripts.
- Transfer of arguments for management scripts.
- Monitoring and control running management scripts.
- Transfer of the results produced by running management scripts.

The Script MIB makes the assumption that management functions are defined in the form of executable code written in a management scripting language. The MIB supports arbitrary languages and code formats. The Script MIB even allows the distribution of compiled native code, if an implementation is able to execute native code under the control of this MIB.

An entity relationship diagram of the logical structure of the Script MIB is shown in Figure 1. The MIB consists of 6 tables that are organized into 5 groups. The first group, called the smLanguageGroup, contains two read-only tables (`smLangTable` and `smExtnsTable`) which provide information about the languages and language extensions supported by a Script MIB implementation. These two tables were designed so that a higher-level manager can select the script version appropriate for a given Script MIB implementation.

The smScriptGroup consists of a single table. The `smScriptTable` lists all scripts known to a Script MIB implementation. It contains objects that allow a manager to download scripts from a URL (pull model), to read scripts from local non-volatile storage, to store scripts in local non-volatile storage, to delete scripts from

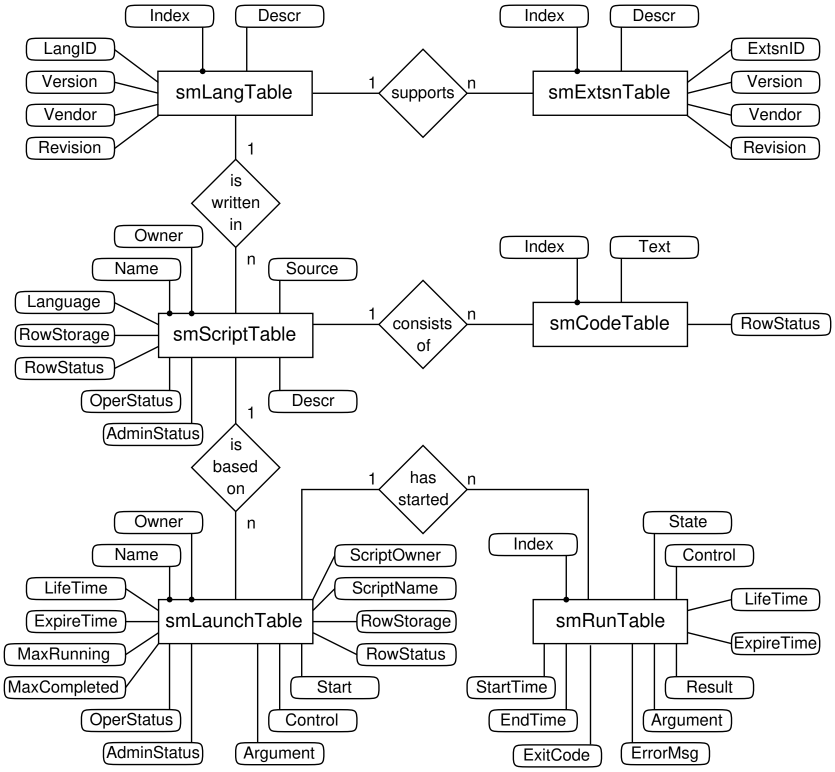

Figure 1: Structure of the IETF Script MIB

local non-volatile storage, to list permanent scripts (that can not be changed or removed), or to read and modify the script status (enabled, disabled, editing).

The smCodeGroup consists of a single table, called the `smCodeTable`, which provides the ability to transfer and modify scripts via SNMP set operations (push model). A script can be fragmented over multiple rows of the `smCodeTable` in order to deal with SNMP message size limitations. The implementation of the smCodeGroup is optional and not required for conformance with the Script MIB.

The smLaunchGroup contains a single table. The `smLaunchTable` describes scripts that are ready to be launched. An entry in the `smLaunchTable` associates a script with an owner used during script execution, provides arguments and parameters for script invocations and allows a manager to invoke scripts with a single SNMP set operation. The `smLaunchMaxRunning` object, for example, defines the maximum number of times that a script invoked from a particular row in the `smLaunchTable` may be running concurrently. The `smLaunchTable` can have multiple entries that point to the same script. This can be used to attach different arguments or permissions to a script.

Finally, the smRunGroup contains the `smRunTable`, which lists all scripts that are currently running or have terminated recently. The `smRunTable` contains objects that allow a manager to retrieve status information from running scripts, to control running scripts (suspend, resume, abort), to retrieve results from recently terminated scripts, to control the remaining maximum lifetime of a running script, or to control how long script results are stored in the MIB. Every row in the `smRunTable` contains the argument passed during script invocation, the result produced by the script and the script exit code. The `smRunTable` also provides information about the current run state, as well as start and end time-stamps. There are three writable objects in the `smRunTable`. The `smRunLifeTime` object defines the maximum time a running script may run before it is terminated by the Script MIB implementation. The `smRunExpireTime` object defines the time that a completed script may stay in the `smRunTable` before it is removed. The `smRunControl` object allows to suspend, resume, or abort running scripts.

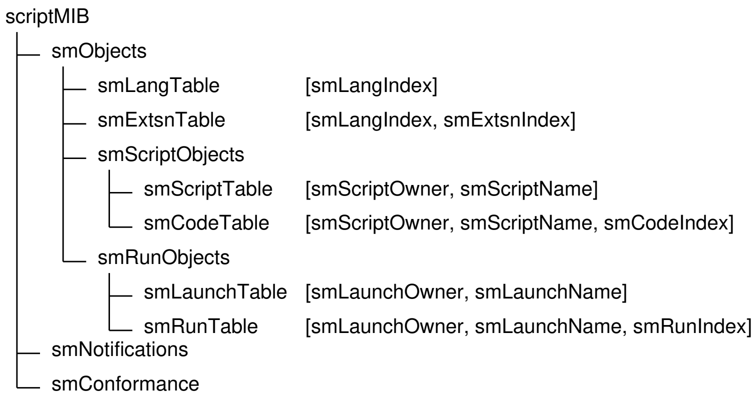

Figure 2: OID and INDEX Structure of the IETF Script MIB

The left part of Figure 2 shows how the Script MIB tables are registered in the object identifier (OID) tree. The right part of Figure 2 shows the table indexing structure. Index elements that match the pattern `*Index` are of type `Unsigned32`. Index elements that match `*Owner` or `*Name` are of type `SnmpAdminString`[1].

3 SNMP Version 3 Security

The SNMP version 3 working group published SNMPv3 as a Proposed Standard in January 1998 [6, 7, 8, 9, 10]. The SNMPv3 specifications define an IETF standard for SNMP security which is supported by several vendors. Work on SNMPv3 delayed the progress in the DISMAN working group as it was clear that any distributed

[1] `SnmpAdminString` is an UTF-8 string which contains administrative information [6].

management solution must be able to protect itself and the managed network from unauthorized access and abuse. It was therefore necessary to have SNMPv3 in place before DISMAN MIBs could go onto the standards track. There have in fact been many discussions between these working groups in order to make sure that the DISMAN needs are addressed by the SNMPv3 specifications. This section gives an overview of the SNMPv3 security mechanism in as much detail as required to understand how these mechanisms are used to protect the Script MIB.

The SNMP architecture [6] makes a distinction between message security services (integrity, authentication and encryption) and access control services. Both, message security and access control services can be provided by multiple security or access control models. The architecture allows the coexistence of multiple models in order to allow future updates, in case the security requirements change or cryptographic protocols need to be replaced. The security model independent identifier for a principal invoking an SNMP operation is the `securityName`. The `securityLevel` actually used in an SNMPv3 message exchange is either no authentication and no privacy (`noAuthNoPriv`), authentication without privacy (`authNoPriv`) or authentication with privacy (`authPriv`).

An access control model is used to decide whether a specific type of access (read, write, notify) to a particular object instance is allowed. This access type is mapped to a `viewType`. Access control models are independent of the security models and therefore use only security model independent parameters as input. The interface to an access control model is defined by the `isAccessAllowed()` abstract service interface (ASI). It takes the `securityModel`, the `securityName`, the `securityLevel` and the `viewType` as inputs in order to decide whether the access to a `variableName` in a given context[2] identified by `contextName` is allowed.

3.1 User-based Message Security

The User-based Security Model (USM) [9] provides integrity, authentication and privacy services and is the standard security model currently used with SNMPv3. The USM internal identifier for a principal is the `userName`, which maps to the model independent `securityName` by an identity transform.

Data integrity and authentication in USM is realized by computing message authentication codes with either `HMAC-MD5-96` or `HMAC-SHA-96`. The keys are localized for each SNMP engine [11]. Timeliness of SNMP messages is checked using loosely synchronized clocks (`snmpEngineBoots` and `snmpEngineTime`).

Privacy services are optional in USM. The data encryption standard (DES) [12] is used to encrypt the payload of an SNMP message if privacy is used. One interesting aspect of USM is that key changes have to be realized without encryption since encryption is optional. The solution used in USM has the property that a user can change his key without actually transmitting the key. However, if a key is broken

[2] A context is a naming scope. MIB variables are uniquely identified by an OID value within a given context. The same OID can however identify different instances of a variable in different contexts.

and if the attacker is able to monitor all key change requests, it will be easy to compute the key currently in use. This means that regular key changes do not limit the time available for an attacker to break a key if the attacker is able to monitor all key changes. USM therefore suggests to use encryption for key changes whenever possible.

3.2 View-based Access Control

The View-based Access Control Model (VACM) [10] allows a security administrator to restrict access to a subset of the management information within a context. This subset is called a MIB view and identified by its `viewName`. A MIB view is defined as the union of several view subtrees. A view subtree is the set of MIB object instances which have a common OID prefix. VACM allows to use wild-cards in the OID prefix in order to reduce the size of typical view subtree definitions.

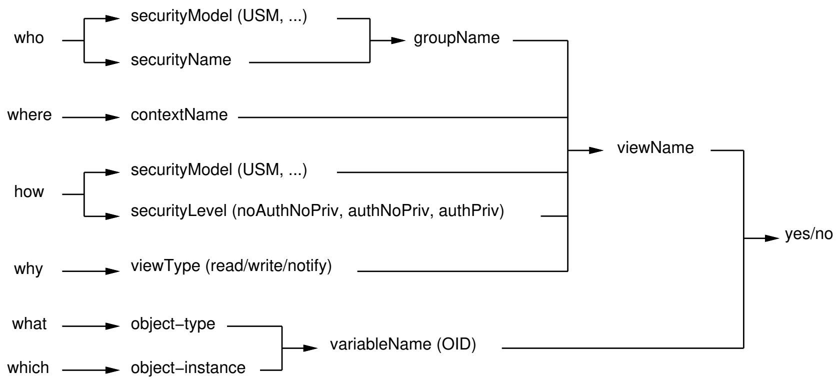

Figure 3: Description of the `isAccessAllowed()` ASI in VACM [10]

Figure 3 shows how VACM implements the `isAccessAllowed()` abstract service interface. The `securityName` and `securityModel` is first mapped into a `groupName`. The `contextName`, `securityModel` and `securityLevel` and the operation contained in an SNMP message are then mapped to a `viewName`. The variables addressed by the SNMP operation are then checked to see whether they are inside of the MIB view identified by `viewName`.

VACM groups are used to reduce the number of VACM configuration entries for multiple `securityName`, `securityModel` pairs. However, VACM does not allow a `securityName`, `securityModel` pair to be a member of multiple groups. There is also no other mechanism in VACM to reuse view subtree definitions. This means that one needs complete sets of view subtree definitions even if there is only a single difference in the views used by two different VACM groups.

4 Security Aspects of the Script MIB

The security requirements for the Script MIB are different from many other MIBs. The Script MIB needs to describe execute permissions and it must allow users and security administrators to control the permissions associated with a delegated script. The detailed requirements are:

- The Script MIB must control who is allowed to read, write and execute scripts. The View-based Access Control model distinguishes between read, write and notify MIB views. It is therefore necessary to express execute permissions with read/write/notify access control rules.
- The Script MIB must support different script calling conventions. The *normal* calling convention is used whenever a user starts a script which will execute with the permissions and credentials associated with that user. The *setuid* calling convention is used whenever a user starts a script which will execute with the permissions and credentials associated with the user who installed the launch button for that script.
- The Script MIB must protect script arguments and results.
- The Script MIB must provide a way to associate credentials with a running script that control the access to resources used during script execution.
- The Script MIB must allow the security administrator to provision access control so that users can independently create and modify scripts.

One of the most interesting questions was how the existing SNMPv3 security mechanisms and especially the VACM access control model can be used to realize these requirements. There are basically two different ways to tackle this problem:

1. The first approach gives each user of the Script MIB its own (logical) execution engine. This maps directly to the usage of different SNMP contexts to separate scripts and running scripts. VACM can be easily configured to allow or deny access to different contexts. (Using `where` in Figure 3.)
2. The second approach gives each user of the Script MIB its own playground with fences to separate the various users as appropriate. This approach requires choosing a table indexing structure which allows to use VACM efficiently to define different views in shared tables. (Using `which` in Figure 3.)

The DISMAN working group adopted the second approach. All the relevant tables are therefore indexed by either `smScriptOwner` or `smLaunchOwner` (see Figure 2). VACM can thus be configured to allow or deny read and write access to those parts of a table that are owned by a particular owner. Execute permissions are expressed by overloading write permissions to the `smLaunchStart` object. However, having write permission on a `smLaunchStart` instance is not sufficient to start a script. One also needs read access to the script itself to ensure that a user can not start arbitrary scripts by guessing script names and creating entries in the `smLaunchTable`. This requires to check the read access rights to the script using the `isAccessAllowed()` ASI while processing SNMP set-requests. Agent extensibility protocols such as AgentX [13] do not support such a call from a subagent to a master agent. This means that a Script MIB implementation must either be part of the master agent or it has to use nonstandard mechanisms.

```
 1:vacmAccessReadView."junior"."".usm.authNoPriv  = "juniorReadView"
 2:vacmAccessWriteView."junior"."".usm.authNoPriv = "juniorWriteView"
 3:
 4:vacmAccessReadView."senior"."".usm.authNoPriv  = "seniorReadView"
 5:vacmAccessWriteView."senior"."".usm.authNoPriv = "seniorWriteView"
 6:
 7:juniorReadView:                    juniorWriteView:
 8:  smLangTable
 9:  smExtsnTable
10:  smScriptObjects.*.*.*."junior"    smScriptObjects.*.*.*."junior"
11:  smRunObjects.*.*.*."junior"       smRunObjects.*.*.*."junior"
12:  smScriptObjects.*.*.*."utils"
13:  smScriptObjects.*.*.*."emergency" smLaunchStart."emergency"
14:  smRunObjects.*.*.*."emergency"    smLaunchArgument."emergency"
15:
16:seniorReadView:                    seniorWriteView:
17:  smLangTable
18:  smExtsnTable
19:  smScriptObjects.*.*.*."senior"    smScriptObjects.*.*.*."senior"
20:  smRunObjects.*.*.*."senior"       smRunObjects.*.*.*."senior"
21:  smScriptObjects.*.*.*."utils"     smScriptObjects.*.*.*."utils"
22:  smScriptObjects.*.*.*."emergency" smScriptObjects.*.*.*."emergency"
23:  smRunObjects.*.*.*."emergency"    smRunObjects.*.*.*."emergency"
```

Figure 4: Example VACM Configuration

Figure 4 shows an example VACM configuration. Lines 1-2 define that the members of the VACM group `junior` will have read (write) access to the objects in the view `juniorReadView` (`juniorWriteView`) if they access the agent using the USM `securityModel` with the `securityLevel` `authNoPriv`. Lines 4-5 define that the members of the VACM group `senior` will have read (write) access to the objects in the view `seniorReadView` (`seniorWriteView`) if they access the agent using the USM `securityModel` with the `securityLevel` `authNoPriv`.

Lines 8-9 grant access to the read-only `smLangTable` and `smExtsnTable` and the lines 10-11 grant read and write access to all the Script MIB table entries owned by `junior`. The members of the `junior` VACM group therefore have the permissions to install and launch their own scripts. The lines 17-20 define similar permissions for the members of the `senior` VACM group. Note that it is not possible for the members of the `junior` VACM group to access scripts installed and used by the members of the `senior` VACM group and the other way round.

It might be useful to give the members of the `senior` VACM group the right to install scripts that can be called by the members of the `junior` VACM group using the normal calling convention. This is realized by the lines 12 and 21. Line 12 grants read access to the scripts owned by `utils` so that the members of the `junior` VACM group can create launch buttons to start these scripts. Line 21 grants read and write permissions to the scripts owned by `utils` to the members of the `senior` VACM group.

Emergency situations may require that the members of the `junior` VACM group can execute recovery scripts that need more permissions than associated with the members of the `junior` VACM group. The lines 13-14 and 22-23 show how an `emergency` owner is used with a setuid calling convention to address this need. The members of the `junior` VACM group get read access to all the entries owned by `emergency` but only write permissions to the `smLaunchStart` object and the `smLaunchArgument` object. They can therefore start an emergency script and pass arguments, but they can not create new launch buttons owned by `emergency`. The lines 22-23 finally grant the right to the members of the `senior` VACM group to install and modify scripts and launch buttons for the `emergency` owner.

5 Jasmin: A Script MIB Implementation

An implementation of the Script MIB has been developed as a joint project between the Technical University of Braunschweig and C&C Research Laboratories of NEC Europe Ltd. at Berlin. It uses the Java virtual machine as the primary runtime system and is called Jasmin (JAva Script Mib ImplementatioN). The goal of this implementation was to prove that the specifications are implementable how security can be achieved.

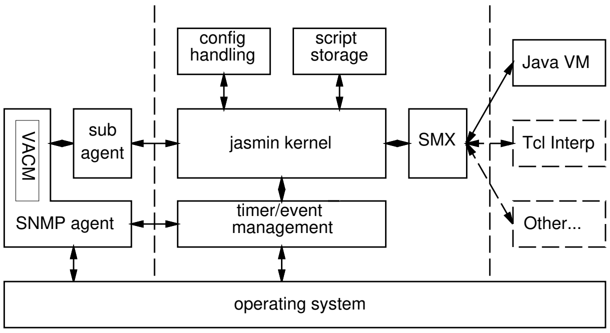

Figure 5: Structure of the Jasmin Script MIB Implementation

The general structure of the implementation is shown in Figure 5. The Jasmin implementation is designed to support multiple languages and runtime systems. We achieved this goal by separating the runtime systems from the SNMP agent through a lightweight protocol called SMX [14]. This separation is indicated by the right dashed line in Figure 5. We also decided to keep the core of the Script MIB implementation independent from the SNMP toolkit we were using. All data structures are kept in the toolkit-independent Jasmin kernel and the toolkit-dependent subagent calls functions exported from the Jasmin kernel where appropriate. This is indicated by the left dashed line in Figure 5. The kernel interacts with other modules that maintain the configuration file, manage the script storage, or resolve URLs.

5.1 SMX Protocol

The SMX protocol runs over a local TCP connection and uses simple ASCII encodings. This makes it easy to implement the protocol in various programming languages. Runtime systems are started by the Jasmin kernel. The kernel passes an authentication cookie to the runtime systems (stored in the process environment) which is verified after establishing the TCP connection.

The SMX protocol has primitives to start, suspend, resume or abort scripts and to query the hosting runtime system for status information. The runtime system can signal events such as script termination by sending asynchronous notification messages. One goal of SMX was simplicity. The integration of new runtime systems should be easy so that we can experiment with several different languages. SMX version 1.0 therefore does not support the following features:

- The Script MIB only defines a single script result, which is an opaque `OCTET STRING` value. It is often desirable to export results from running scripts as ordinary MIB objects. This allows one to use scripts as a mechanism to implement new MIBs. The SMX protocol does not provide protocol operations to export data from a runtime system via the associated SNMP agent. We argue that a runtime system which allows scripts to implement MIB objects should use an agent extensibility protocol such as AgentX [13].

- The SMX protocol does not allow to access data stored in the MIB of the associated SNMP agent. Instead, we expect that the runtime system uses normal SNMP operations to retrieve data from the "local" MIB. This decision has some consequences. First, access to "local" MIB objects requires normal SNMP interactions which might be more expensive than direct access. Further, using SNMP requires that the script has the appropriate credentials in order to access the "local" MIB. On the other hand, using SNMP interactions makes sure that scripts access the "local" MIB in exactly the same way as they access "remote" MIBs, which makes scripts more general and portable.

5.2 Security Profiles

The Jasmin prototype uses security profiles to define what a running script is allowed to do. Jasmin distinguishes between an operating system security profile and a runtime system security profile.

- The *operating system security profile* specifies the set of operating system services that may be used by the operating system level process which executes a script. Under UNIX, this maps to the effective UNIX user identity for the running process and the file system root used for the process executing the runtime system. Changing the file system root is an effective mechanism to prevent running scripts from accessing system files.
- Secure runtime systems usually allow further control over the set of services that can be used at a particular point in time during script execution. The *runtime security profile* is runtime system dependent and usually has much finer granularity than the operating system security profile.

The Jasmin Script MIB implementation maps `smLaunchOwner` values to an operating system security profile and a runtime security profile. The latter one is a string which is passed to the runtime system via the SMX protocol. It is the job of the runtime system to interpret the string and to map the string into concrete access and resource control decisions. The mapping of `smLaunchOwner` to the security profiles is currently defined in the Jasmin configuration file and read by the Jasmin kernel at startup time.

5.3 Java Runtime System

The Java runtime system of the Jasmin prototype is based on JDK 1.1. It interacts with the Script MIB SNMP agent via the SMX protocol, which is implemented in Java. Access to other network devices via SNMP is provided by using off the shelf Java SNMP classes.

5.3.1 Java Runtime Security

Runtime security for Java is based on the sandbox security model. For an applet or an application, a *sandbox* defines a subset of the system resources which are accessible [15]. The sandbox restrictions are enforced by the security manager and the class loader. Each access to crucial system resources, e.g. files or network connections, is mediated by the Java virtual machine, and is checked in advance by the security manager.

The security manager is an instance of a class that can be installed only once for an application. Before a resource is accessed, a corresponding method of the `SecurityManager` class is called. If this method returns, the access to the resource is granted. Otherwise, it throws a security exception.

The security manager controls file access (reading, writing, deleting), network access (opening and accepting a socket connection, usage of IP multicast), IO access (printing, opening a top level window, communicating with the system clipboard), thread and thread group access (stopping, suspending, resuming, changing priority), and access to some further resources as dynamic linked libraries, classes of different packages, and the system properties.

The class loader controls dynamic loading of code. If a class is called that has not been loaded so far, the class loader tries to load the code from local files, over the network, or from other sources. In order to allow loading of code which runs in a strongly restricted sandbox, the class loader is not restricted by the security manager when loading class code. However, the installation of a class loader is controlled by the security manager.

This description of Java runtime security is based on JDK 1.1 which has been used for the implementation of the Script MIB runtime engine. JDK 1.2 will offer several new security features. Concerning runtime security, a new and easier way of configuring the security manager is available, which is based on the concept of security domains. However, the functionality of the JDK 1.1 security manager is not changed or replaced, but used as the core of the new security model.

5.3.2 Java Implementation of Runtime Security Profiles

The Jasmin implementation of a runtime engine for the Script MIB is a Java application that communicates with the agent via the SMX protocol and executes scripts as Java threads. Scripts to be executed by this engine are Java archive files which contain all classes required by the script excluding classes provided by the JDK. When a script is started, a new thread group is created and the script is started as member of this thread group.

The security manager is created by the runtime engine before any script is started. The security manager developed for this runtime engine grants or denies access to resources according to an access table. The rows of this table are indexed by the thread group the requesting thread belongs to and the columns are indexed by the method called on the security manager. Entries of the access table are either boolean values or boolean methods to be called in order to decide about the request for this combination of resource and thread group.

For each script to be started, an new row of the access table is created specifying the access rights of the thread group used by the running script. The entries of the new row depend on the runtime security profile linked to this run of the script. Using thread groups instead of threads allows the script to create new groups inside its own thread group with the same access rights. However, if the script creates a new thread group, threads of this group are completely restricted because the security manager will not grant any request from this thread group.

In order to protect the security manager, direct access to the access table is only allowed for the thread that created the security manager, i.e. no script can access it. For each script, a class loader is created, that restricts loading of classes to the script's Java archive file and the JDK libraries.

The security manager's access table allows fine grained access control. However, we did not consider it desirable that all possible access table rows can be addressed by the runtime security profile. This would make the profile a long and hardly read-able string, and the selection of the right profile would be complicated. Furthermore, most of the possible rows do not represent useful combination of access rights. So, for this implementation of a Java runtime engine, we restricted the number of profiles to a set of combinations of access rights we considered desirable and sufficient for practical use of scripts. This set includes profiles for complete restriction, no restriction at all, file access to a temporary working directory only, access to the local SNMP agent via AgentX, and some other profiles restricting or allowing SNMP requests and event reports to the local agent or to remote agents, respectively.

6 Conclusions and Future Work

This paper discussed security aspects associated with the delegation of management scripts within the Internet management framework. The SNMPv3 security mechanisms can be used to protect a scripting engine from unauthorized access. However, the complexity of VACM access control rules increases with fine grained access control on scripts. The decision to disallow membership of a single `securityName`, `securityModel` pair in multiple groups makes the VACM access control configuration pretty complex and hard to maintain without specialized tools.

The requirement to call the `isAccessAllowed()` abstract service interface while processing SNMP set-requests to start a script causes some implementation problems in an AgentX [13] subagent environment. This makes it difficult to distribute Script MIB implementations as add-on software.

The delegation of credentials is currently not covered by the Script MIB. The Jasmin implementation described in this paper uses operating system security profiles and runtime system security profiles to address this problem. The mappings from `smLaunchOwner` values to the security profiles is currently obtained from a configuration file. It might be worthwhile to define a new MIB so that the security profiles can be administrated via SNMP.

The SMX protocol allows to add support for other languages with little costs. We are currently working on a second runtime system based on the Tool Command Language (Tcl) which will use the *padded cell* approach to protect Tcl scripts running in a safe Tcl interpreter [16]. Tcl is usually classified as a "pure" scripting language while Java is usually classified as a system programming language. Further experiments with the Jasmin prototype will be necessary to better understand the strength of these languages to solve real-world management problems.

Acknowledgments

The authors like to thank the members of the DISMAN working group for some insightful discussions. We also like to thank the Jasmin project members, namely Andreas Kind, Matthias Bolz, Cornelia Kappler, Sven Mertens and Jan Nicklisch for their contributions to the prototype implementation.

References

[1] Y. Yemini, G. Goldszmidt, and S. Yemini. Network Management by Delegation. In *Proc. International Symposium on Integrated Network Management*, pages 95–107, April 1991.

[2] ISO. Command Sequencer. International Standard ISO 10164-21, ISO, 1998.

[3] D. B. Levi and J. Schönwälder. Definitions of Managed Objects for the Delegation of Management Scripts. RFC (to be published), 1999.

[4] J. Schönwälder. Network Management by Delegation - From Research Prototypes Towards Standards. *Computer Networks and ISDN Systems*, 29(15):1843–1852, November 1997.

[5] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[6] D. Harrington, R. Presuhn, and B. Wijnen. An Architecture for Describing SNMP Management Frameworks. RFC 2271, January 1998.

[7] J. Case, D. Harrington, R. Presuhn, and B. Wijnen. Message Processing and Dispatching for the Simple Network Management Protocol. RFC 2272, January 1998.

[8] D. Levi, P. Meyer, and B. Stewart. SNMPv3 Applications. RFC 2273, January 1998.

[9] U. Blumenthal and B. Wijnen. User-based Security Model (USM) for version 3 of the Simple Network Management Protocol. RFC 2274, January 1998.

[10] B. Wijnen, R. Presuhn, and K. McCloghrie. View-based Access Control Model for the Simple Network Management Protocol. RFC 2275, January 1998.

[11] U. Blumenthal, N. C. Hien, and B. Wijnen. Key derivation for network management applications. *IEEE Network Magazine*, 11(4), 1997.

[12] National Bureau of Standards. Data Encryption Standard. Federal Information Processing Standard 46, National Bureau of Standards, January 1988.

[13] M. Daniele, B. Wijnen, and D. Francisco. Agent Extensibility (AgentX) Protocol Version 1. RFC 2257, January 1998.

[14] J. Schönwälder, M. Bolz, S. Mertens, J. Quittek, A. Kind, and J. Nicklisch. SMX - Script MIB Extensibility Protocol Version 1.0. Informatikbericht 98-07, TU Braunschweig, October 1998.

[15] J. S. Fritzinger and M. Mueller. Java Security. Technical Report, Sun Microsystems, 1996.

[16] J. Y. Levy, L. Demailly, J. K. Ousterhout, and B. Welch. The Safe-Tcl Security Model. In *Proc. USENIX Annual Technical Conference*, June 1998.