An Approach for Secure Software Installation[†]

V. N. Venkatakrishnan, R. Sekar, T. Kamat, S. Tsipa and Z. Liang
– Computer Science Dept., SUNY at Stony Brook

ABSTRACT

We present an approach that addresses the problem of securing software configurations from the security-relevant actions of poorly built/faulty installation packages. Our approach is based on a policy-based control of the package manager's actions and is customizable for site-specific policies. We discuss an implementation of this approach in the context of the Linux operating system for the Red Hat Package manager (RPM).

Introduction

Management of software installations has been one of the biggest problems facing system administrators.[1] Significant progress [6] has been made in some aspects of this problem, e.g., management of dependencies and conflicts among packages. In other areas that concern overall system security and interoperability with existing applications, package managers still fall short, as they make several unrealistic assumptions:

- *System administrators want to treat packages as "black boxes.* Although system administrators are not interested in the details of installation, they certainly care about package installation actions that have implications for overall system security and operation, e.g., addition of new users, modifications to boot-time scripts, addition of entries to crontab, modifications of system libraries, changes to global configuration files (e.g., /etc/inetd.conf) or application-specific configuration files or in the case of Windows, changes to system registry.
- *Package installation steps operate correctly.* Few provisions exist for dealing with poorly-written packages that crash in the middle of the installation process. Typically, such crashes would result from unanticipated conditions (involving the configuration of the system on which the package is being installed) encountered by install scripts that are included with many packages. System administrators are all too familiar with situations when packages can neither be fully installed, nor be fully uninstalled, leaving the system in an inconsistent state.
- *All software and system configuration updates are the result of package installation*: In practice, however, system administrators have to frequently configure systems or packages by manually editing config files. In addition, software is frequently installed outside of the package management system, e.g., by downloading and compiling a source-code archive. Package managers interact poorly with such situations, often ending up overwriting critical application files. Although package managers can save backup copies of certain config files, the sysadmin is usually not alerted that the config files have been updated.

We describe a new approach that augments existing package managers such as RPM to overcome the above problems. Our approach enables system administrators to reason about the security-critical actions of an installation/upgrade process, check whether these actions are compatible with specified security policies, and if so, allow the installation to proceed. During the installation, all actions are logged so that they can be rolled back in the event of an installation failure.[2] We have implemented our approach in the form of a tool called *RPMShield* that operates in conjunction with RedHat's package manager (RPM). As compared to existing package managers, our approach offers the following benefits:

- *Policy-based control of package installation actions.* While existing package managers such as RPM allow a system administrator to examine package contents and installation scripts in detail, this is a cumbersome process and hence seldom undertaken. In contrast, our approach presents a convenient interface through which a system administrator can exert control over installation actions that impact system security or the operation of existing applications.
- *Interoperability with changes made outside of package managers.* Our approach provides a

[†]This research is supported in part by a ONR University Research Initiative grant N000140110967 and NSF grant CCR-0098154.

[1]In this paper, we use the term "system administrator" to refer to professionals and end-users that perform software installation.

[2]Note that a complete rollback is impossible if the installation scripts communicate over the network, or when processes unrelated to the installation are allowed to make system changes after reading files modified by the installation process.

convenient mechanism to control updates to manually edited files, files shared among multiple packages, or more generally, files installed outside the scope of the package manager.
- *Normal-user installation of packages.* Individual users often want to install packages that are of interest to themselves. Since all RPM installation actions require super-user privilege, normal users are unable to install such packages for themselves. Our approach can support this capability through the use of security policies that limit installation actions so that the changes are restricted to a specific user directory.
- *Tolerance to failures*. Package managers offer poor support to revert to original system configuration when an installation upgrade/process fails. Our automatic recovery mechanism reverts the system to its original (consistent) state.

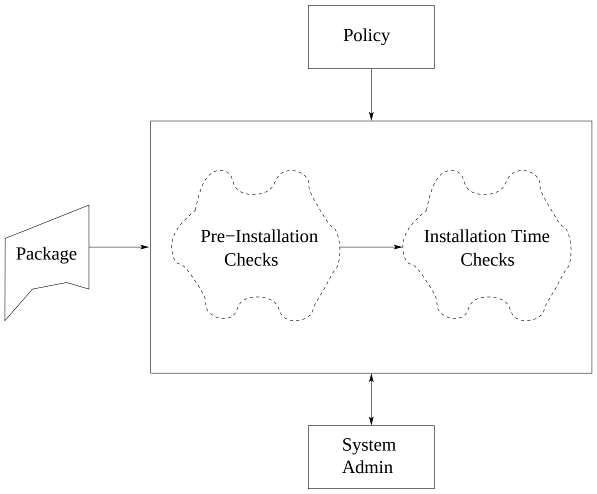


Figure 1: Approach overview.

We note that this paper is concerned with the package installation process, and does not address security implications of running the applications installed as a result. Ensuring safety of the system while supporting the execution of untrusted application is an orthogonal problem. This is the subject of many papers on digital signatures [7], sandboxing [9], proof-carrying code [11] and model-carrying code [12].

Overview of Approach

Our approach, presented in Figure 1 divides package installation into two phases. 1) In the *pre-installation phase*, a package is analyzed to determine its compatibility with a system administrator's policies. 2) In the *installation phase,* where the actual package installation takes place in a controlled environment. Each of these phases is described below.

Pre-installation Phase

The pre-installation phase (see Figure 2) consists of the following steps:
- *Generation of behavioral models.* In this phase, a package is analyzed to identify the security-relevant actions that will take place during its installation. The analysis involves two steps: 1) finding the list of files the package will install/upgrade, and 2) capturing the intended behavior of its (pre-install and post-install) scripts. The first step involves querying the package itself as well as the packages database. The second step involves learning the behavior of the scripts/make files. More details on the model generation process is presented in the section 'Model Generation'
- *Consistency resolution.* The model generated in the previous step is supplied to the *consistency resolver* which checks whether this behavior is in accordance with the security policy provided by the system administrator. A discussion of security policies that could be enforced though the system is presented in the 'Security Policies' section. Consistency resolution is described in its own section.

By performing consistency resolution before installation, our approach avoids the time-consuming step of actual installation when there is a conflict. In addition, all conflicts with the policy are identified and presented together, which enables a system administrator to make more informed decisions. This contrasts with conflict identification during actual installation, when each conflict must be presented individually to the system

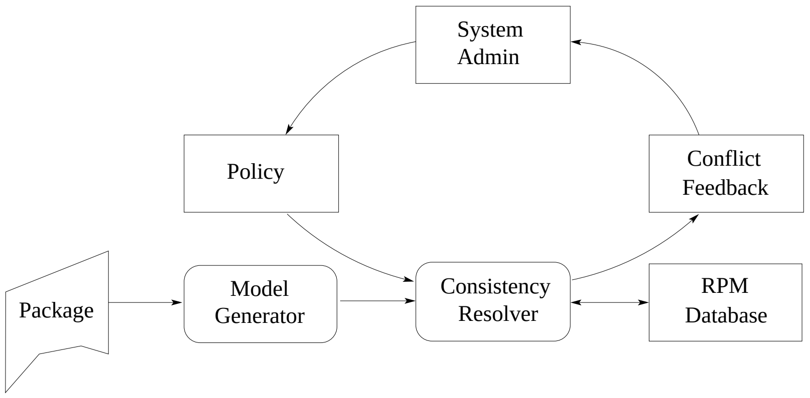


Figure 2: Pre-installation phase.

administrator for acceptance. This cumbersome process can lead to "click fatigue," sometimes causing the system administrator to make inappropriate decisions.

Installation Phase

While the benefits of pre-installation checks were identified above, it is not always possible to identify all conflicts statically. Scripts may perform complex computations (e.g., creating a file name from several command-line arguments or environment variables) whose results cannot always be statically determined. Such conflicts are dealt with during the second phase of package installation (refer to Figure 2), namely, the installation phase.

In this phase, as shown in Figure 3, the package manager is allowed to run in an environment where the system calls made by the package manager process and its children are monitored by RPMShield. These system calls are compared with the policy provided by the system administrator during the pre-installation phase. Typically there are no policy violations in this phase, as they would have been handled in the previous phase. However, if conflicts do arise, this information is presented to the system administrator. If the violation is accepted, then package installation proceeds. If not, installation is aborted, and the system state is restored as it was prior to the installation. The runtime-checking mechanism is described in the 'Runtime Interception' section.

Description

This section elaborates on the various components that were introduced in the preceding high-level discussion.

Security Policies

We use an expressive policy language that, in addition to capturing conventional access-control policies, can also express context sensitive policies such as "this application cannot modify files owned by other applications," or history sensitive policies such as "the installation process can only delete the files it has created." Access control policies and context sensitive policies are specified conveniently through a GUI, as shown in Figure 3. In this figure, the system administrator can specify whether the package installation process can possess the corresponding *capabilities*. The first row describes a capability where a package can create files in directories not owned by itself; In the second row, policy specification for writes to files that are owned by the package, but modified from the original installation (e.g., config files) are shown; and writes to files owned by other packages are shown in the succeeding row.

Similar capabilities that could be specified using the GUI include the ability to add users, perform network operations, update shared libraries, modify system services (e.g., files in the /etc/rc.d/* directories), execution of arbitrary system commands and so on. History sensitive policies are currently not expressed through the GUI, but could be specified using our underlying expressive policy language [13].

These security policies are internally represented as *extended finite state machines*. An finite state machine consists of *states* and *transitions*. The states of these machines correspond to various program points and the transitions are over an alphabet of system calls with their arguments. There are *condition guards* associated with transitions. Whenever, a condition guard is enabled, an optional *action* associated with the guard is triggered. A simple example of an extended finite state automaton is presented in Figure 5, which illustrates the use of these automata in keeping track of the number of bytes written to a file (denoted by MY_FILE).

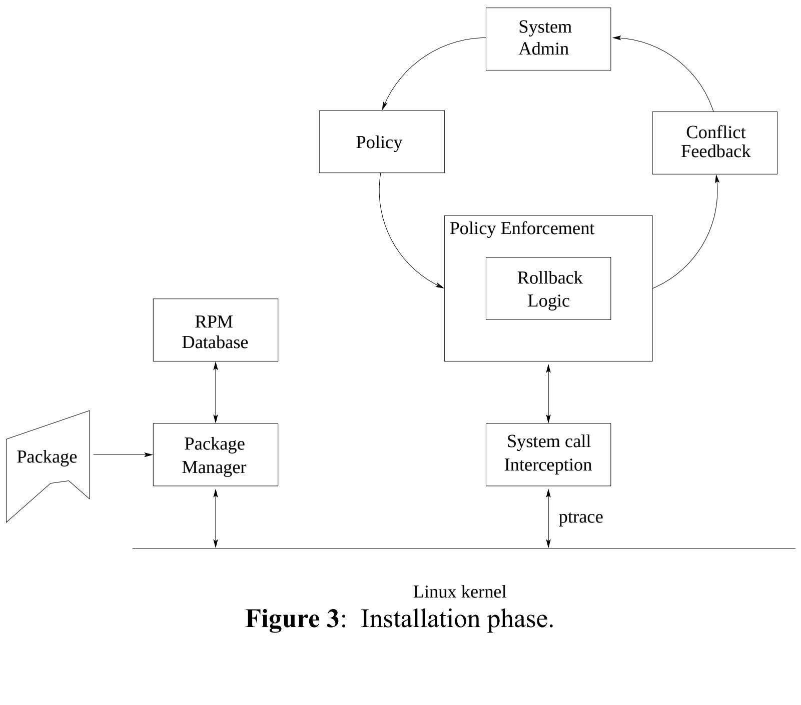


Figure 3: Installation phase.

In this figure, whenever an open system call is made, the condition guard associated with this transition checks whether the file opened is MY_FILE, and if so, the file descriptor is stored in the variable X. Later in the program, if a write operation is performed, the condition guard associated with this transition checks whether the file descriptor equals the descriptor stored in X, and if so, increments the variable count. (For simplicity, we do not show the states and transitions corresponding to the invocations of the close system call). The policy represented thus is a simple example of a history sensitive policy, and the variables that are associated with the transitions enable such policy specifications.

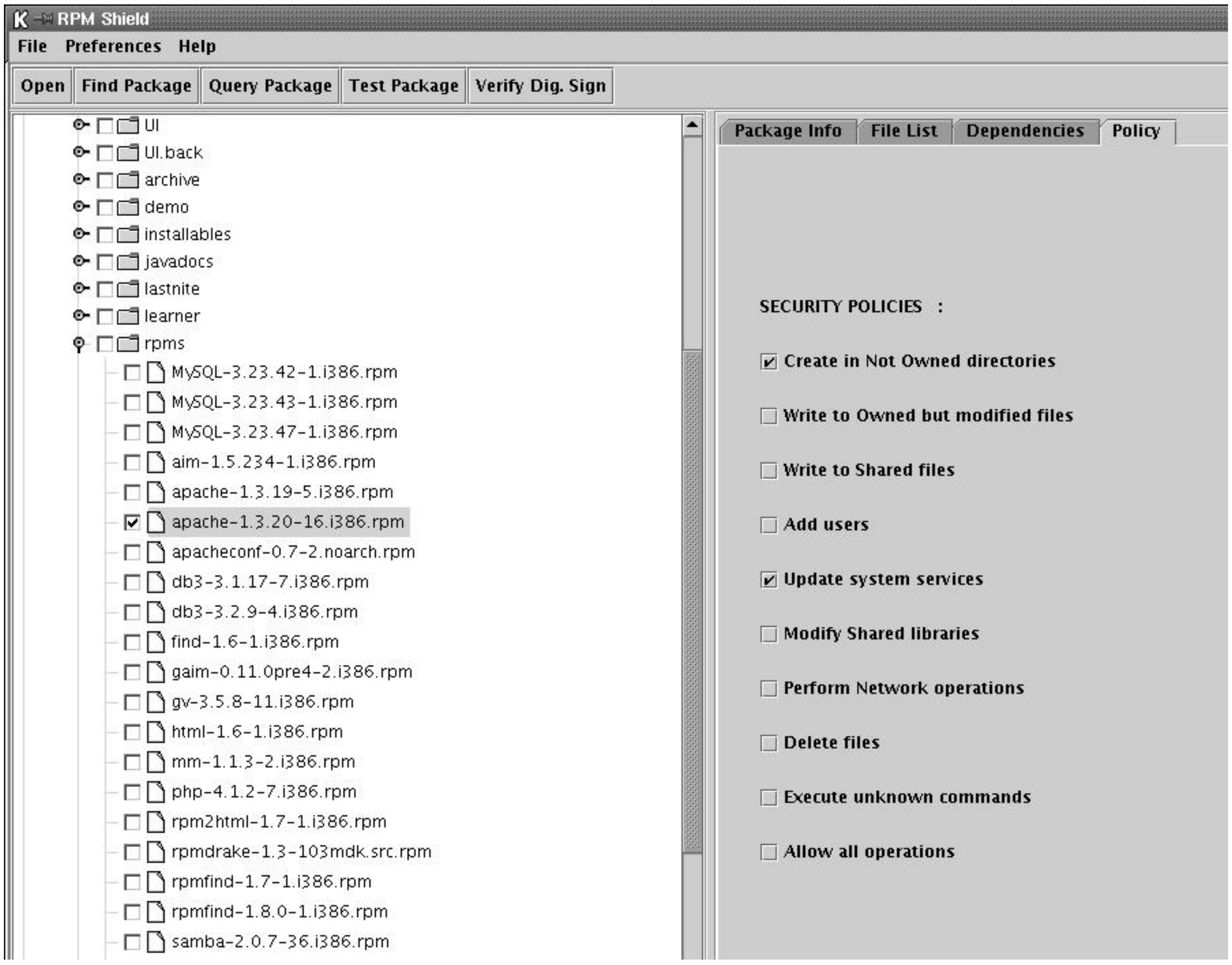
For more information on our work in compiling high level specifications into extended finite-state machines, we refer the reader to [13].

Model Generation

Model generation involves determining the security relevant actions of the scripts and obtaining the list of files the package plans to modify/upgrade. The latter is obtained directly by querying the package, so we describe the analysis of scripts in the following section.

Scripts

There are several approaches that address the problem of analyzing a shell script to determine its behavior. A *static analysis based* approach is one which would analyze the actions of a script without executing it. It usually involves parsing the input script, and interpret the script's actions to analyze its behavior. Such an approach could build on modifying the shell interpreter and performing operations in an


RPM Shield
File Preferences Help
Open Find Package Query Package Test Package Verify Dig. Sign
Package Info File List Dependencies Policy

UI
UI.back
archive
demo
installables
javadocs
lastnite
learner
rpms
- [ ] MySQL-3.23.42-1.i386.rpm
- [ ] MySQL-3.23.43-1.i386.rpm
- [ ] MySQL-3.23.47-1.i386.rpm
- [ ] aim-1.5.234-1.i386.rpm
- [ ] apache-1.3.19-5.i386.rpm
- [x] apache-1.3.20-16.i386.rpm
- [ ] apacheconf-0.7-2.noarch.rpm
- [ ] db3-3.1.17-7.i386.rpm
- [ ] db3-3.2.9-4.i386.rpm
- [ ] find-1.6-1.i386.rpm
- [ ] gaim-0.11.0pre4-2.i386.rpm
- [ ] gv-3.5.8-11.i386.rpm
- [ ] html-1.6-1.i386.rpm
- [ ] mm-1.1.3-2.i386.rpm
- [ ] php-4.1.2-7.i386.rpm
- [ ] rpm2html-1.7-1.i386.rpm
- [ ] rpmdrake-1.3-103mdk.src.rpm
- [ ] rpmfind-1.7-1.i386.rpm
- [ ] rpmfind-1.8.0-1.i386.rpm
- [ ] samba-2.0.7-36.i386.rpm

SECURITY POLICIES :

- [x] Create in Not Owned directories
- [ ] Write to Owned but modified files
- [ ] Write to Shared files
- [ ] Add users
- [x] Update system services
- [ ] Modify Shared libraries
- [ ] Perform Network operations
- [ ] Delete files
- [ ] Execute unknown commands
- [ ] Allow all operations

Figure 4: Screen shot of RPMShield.

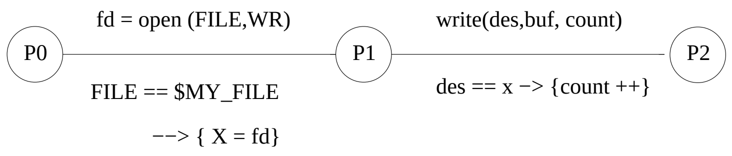


Figure 5: An extended finite state automaton.

abstract domain (a general technique called *abstract interpretation [8]*).

The main advantage of a static analysis based approach is that it has the ability to reason about the actions of the program at a level closer to the program source. However, there are a few disadvantages with such an approach. Shell scripts heavily depend on the environment and redirection for their successful completion. Any static analysis based approach has to *approximate* the environment and the effects of redirection. Thus, the behavior obtained through analysis is usually incomplete.

In addition, there is one more practical implementation problem: The number of shell interpreters that are available in a general purpose system abound. An analyzer that has to deal with an arbitrary script needs to have a front end that would support the idiosyncrasies of the syntax of a number of languages (bash, csh, perl to name a few!). Clearly, this is not a desirable situation.

We follow an alternate approach that follows the *program behavior learning* approach. In this approach, we intercept the system calls of the shell script. We inspect the system call and its arguments, and allow it based on whether it is trying to perform a security-critical operation. We allow access to all operations that are not security-critical: for example, reads to non-sensitive files, creation of temporary files, execution of commands that do not alter the system state and so on. When the program performs write-operations or security critical operations like adding a system service or a user, we simply *fake* the return value of the system call.

By *faking*, we mean returning success without executing the corresponding system call. Thus, the original operation is not performed, and the trace that is generated is the model of the program capturing the intended behavior of the program. All the environment related information is available to this approach (unlike the previous approach). See the 'Examples' section for an example of a model that is generated.

Consistency Resolution

The consistency resolution step involves checking of the model that is generated from the previous step against the security policies of interest. This step involves two operations. The first operation involves checking whether the scripts in the package conform to the security policy. The second operation involves checking whether the file creation / update operations of the package is in accordance to the policy.

Script Checking

Checking whether the execution of a script will violate the given security policy is an interesting problem. The model generated for the shell script contains the trace of the script execution. The security policy (discussed in its own section earlier), is represented in the form of an extended finite state automaton. The policy proscribes all the invalid traces. The model, obtained as an output of the previous step, is presented as an input string to this automaton. If the trace execution is a valid string that is accepted by this automaton, then we can conclude that the execution of this script will violate the specified policy.

As an example, consider an installation script that adds a new user to the /etc/passwd file, while the policy allows no such addition. This conflict information (including the specific action and/or file that caused the conflict) is presented to the system administrator, who may decide to abort the installation, or refine the policy to eliminate the conflict, e.g., permit addition of user to the password file.

Checking of File Updates

The consistency resolver detects any conflicts between the policy and the package behavior model, e.g., if the policy allows updates only to those files owned by a package, then a conflict will arise if the package updates a file that has been updated manually or by a different package. Similar violations are reported for creating files in directories that are not owned by the package, deletion of files and so on.

Runtime interception

The installation phase is realized by the following components.

- *System call interception environment.* System calls and arguments are forwarded to the *policy enforcement engine* using a ptrace-based system call interception facility that we had developed earlier for Linux [10]. This infrastructure provides the facilities for one program to inspect another (target) program whenever the target program performs system calls; check the arguments to the system call; and, if necessary, fake the return value without executing the system call.
- *Policy enforcement engine.* The policy enforcement engine is implemented as an extended finite state machine. which was discussed These automata enforce these policies with very low overheads (typically under 2%) [13].

  In addition, the policy enforcement engine incorporates *rollback logic*, which keeps track of the files modified by the package manager (since the original installation), as well as the original contents of these files. If the installation is to be aborted, then this information is used to reset the system state to what it was before the package installation began.

Other Features

Our tool has a convenient user interface and incorporates attractive usability features, which we describe below.

Query downloads. Most packages have dependencies, i.e., they can be installed only if certain other packages are present in the system. A novel feature in

RPMShield facilitates easy installation of such packages, by downloading them from RPM mirror servers (such as rpmfind.net). The user can configure this set of servers. When an installation encounters dependency requirements, our implementation searches for the existence of these dependency packages on any of the servers.

If the dependency package is found, it is reported to the user and downloaded at his/her discretion . Of course, the downloaded dependency is subjected to the same security checks. Finally, after a successful download, the entire installation process is resumed. RPMShield also takes care of transitive dependencies, e.g., if a package A is dependent on package B which in turn is dependent on package C and so on, then both, packages B and C (and further dependencies) are downloaded and installed first and finally package A is installed.

The implementation of this feature involves the construction of a directed-graph where the nodes represent packages and the edges represent the dependencies between such packages. The installation then starts by installing from the nodes farthest from the root and then proceeds back to the root.

Normal User Installation. Since RPM installations require root privileges, normal users (who do not have root privileges) do not have an opportunity to install packages that are of interest to themselves. Using a highly constrained security policy, we can provide a confined environment where the package can be installed by the normal user. However, not all packages can be installed by a normal user. The packages have to be re-locatable, and must not update any system owned files. Due to the nature of this constrained policies, it is not possible to change them through the graphical-user interface. (One could however change them by modifying the policies in the underlying language).

Examples

We illustrate our approach by running through the installation of the web-server program Apache through our tool. We illustrate the various stages of the installation process through this example.

Apache is a popular and freely-available Web server. The package consists of a set of installation files as well as pre-install and post-install scripts. This example pertains to the version of 1.3.20-16 of apache server.

The system first queries whether the package satisfies all dependency checks. If there are any dependencies on packages that are not yet installed on the system, then the user is queried for downloading of these packages from the download sites. These checks are run through the same security checks as the package. In the following discussion we assume that all dependencies are satisfied.

The pre-installation script is given in Listing 1. The script creates a user in the system. Obviously, this is a sensitive operation, and the system administrator is alerted of this operation. (We do not show the model for this script, but present the model for the post-install script).

```
# Add the "apache" user
/usr/sbin/useradd -c "Apache" -u 48 \
    -s /bin/false -r -d /var/www apache \
                    2> /dev/null || :
```

Listing 1: Pre-installation script.

Listing 2 is the post-installation script of apache server. The generated model is shown in the Figure 6. In this model the states refer to various program points, and the transitions refer to various system calls with their arguments. Due to constraints on the size of the figure, we omit some details such as system call arguments for a selected set of system calls.

```
/sbin/chkconfig --add httpd
# safely add .htm to mime types if it is not already there
[ -f /etc/mime.types ] || exit 0
TEMPTYPES=‘/bin/mktemp /tmp/mimetypes.XXXXXX‘
[ -z "$TEMPTYPES" ] && {
  echo "could not make temporary file, htm not added to /etc/mime.types" >&2
  exit 1
}
( grep -v "^text/html"  /etc/mime.types
  types=$(grep "^text/html" /etc/mime.types | cut -f2-)
  echo -en "text/html==>[ignored: t]<====>[ignored: t]<====>[ignored: t]<=="
  for val in $types ; do
      if [ "$val" = "htm" ] ; then
          continue
      fi
      echo -n "$val "
  done
  echo "htm"
) > $TEMPTYPES
cat $TEMPTYPES > /etc/mime.types && /bin/rm -f $TEMPTYPES
```

Listing 2: Post-installation script of apache server.

This script performs a update to the system services scripts. Also the script updates the file /etc/mime.types that is shared by other applications. The user is alerted of these operations. The other operations performed by the script such a creating and deleting of temporary and running other utilities such as cut, grep etc., are allowed by the security policy.

In addition, suppose if this installation of apache was actually an upgrade from a previous version, then the system keeps track of all the files that have been modified since the previous installation. This not only includes all configuration files, but other files such as local symbolic links, text files, etc. An attempt to delete/overwrite these files is presented to the system administrator. All through the installation, such files are backed up, such that in case the user decides to abort the installation, the system can be reverted to its original state.

Applicability to Other Systems

Although the approach presented in this paper is described only in the context of the Linux operating system and the RedHat package manager, the overall architecture can be ported to other Unix like environments with relative ease. We discuss about migrating to other package managers below.

We have used the Java environment for the implementation of the graphical user interface. Hence this portion of the implementation is portable. The model generation involves querying the package and hence this has to be customized for the particular package format. The consistency checking involves querying the package database and this requires customization as per the package manager's programming interface for querying and modifying its database. The model generation for scripts and the runtime checking steps involve system call tracing. Although, the ptrace system call is not completely portable across different Unix environments, similar facilities exist for other Unix variants, as evidenced by the implementation of gdb-like debuggers for these systems. In fact, our system call interceptor [10] provides a uniform programming interface implementation that abstracts the architecture dependencies. Our interceptor has been implemented for Linux and Solaris.

Other popular package managers like pkg [3] (used on Solaris systems), lpp [2] (used on AIX systems), dpkg [1] (used on Debian Linux distributions) have interfaces similar to that of RPM and hence our approach is applicable to these package managers with the corresponding implementation changes that were described above. There are a few other package managers like SEPP [4], SLP [5] (used on Stampede Linux) that simply do not offer convenient interfaces to query packages and the package databases, and hence are not particularly suitable for our approach.

Conclusion

In this paper, we have discussed the design of a secure software installation framework. Our framework allows a system administrator to control the installation process through a configurable set of policies and enforce these policies through static and runtime checks. Our future work would include exporting the prototype in the context of other operating systems and package managers.

Author Biographies

All the authors of this paper are members of the Secure Systems Laboratory of Stony Brook and their homepages are accessible on the web from the laboratory page at http://www.seclab.cs.sunysb.edu .

R. Sekar is currently an Associate professor of Computer Science and heads the Secure Systems laboratory at SUNY, Stony Brook. Prof. Sekar's research interests include computer system and network security, software and distributed systems, programming languages and software engineering. He can be reached by electronic mail at sekar@cs.sunysb.edu .

VN Venkatakrishnan is a Ph.D. student in the Computer science department at Stony Brook. His main research area is computer security and is currently working on combining static and runtime techniques for assuring software security. He is available by email at venkat@cs.sunysb.edu .

Tapan Kamat is a M.S. student in the Computer Science department at Stony Brook. Tapan does research in the area of computer security. He can be reached via email at tkamat@cs.sunysb.edu .

Sofia Tsipa is currently working as a System Administrator in the Network Operations Center at the University of Thessaloniki, Greece after completing

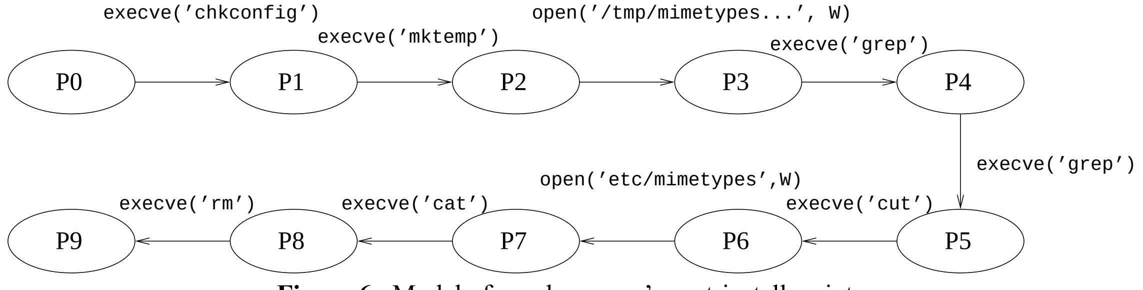


Figure 6: Model of apache server's post-install script.

her M.S. degree in the computer science department at SUNY, Stony Brook. She can be reached via email at sofia@cs.sunysb.edu.

Zhenkai Liang is a Ph.D. student in Computer Science at SUNY, Stony Brook. His research interests include computer security and algorithms. He can be reached by email at zliang@cs.sunysb.edu.

References

[1] *dpkg: A Medium Level Package Manager for Debian*, http://www.debian.org/doc/manuals/quick-reference/ch-package.

[2] *lpp: Aix package distribution*, http://usgibm.nersc.gov/doc_link/en_US/a_doc_lib/aixins/inslppkg/toc.htm.

[3] *pkg: Solaris package distribution*, Solaris man pages.

[4] *Sepp: Software Installation and Sharing System*, http://www.sepp.ee.ethz.ch/seppdoc.pdf.

[5] *Slp: Stampede Linux Packages,* http://www.marblehorse.org/projects/slp/SLPv5a-draft-specification.html.

[6] Anderson, E. and D. Patterson, "A Restrospective on Twelve Years of lisa Proceedings," *Proceedings of Usenix System Administration, LISA*, 1999.

[7] Mitchell, M. W. C. and P. Wild, *Contemporary Cryptology: The Science of Information Integrity*, Chapter Digital Signatures, IEEE Press, Piscataway, NJ, 1992.

[8] Cousot, P., "Static Determination of Dynamic Properties of Programs," *Proceedings of the Second International Symposium on Programming*, Paris, 1976.

[9] Gong, L., M. Mueller, H. Prafullchandra, and R. Schemers, "Going Beyond the Sandbox: An Overview of the New Security Architecture In the Java Development Kit 1.2," *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[10] Jain, K. and R. Sekar, "User-level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement," *ISOC Network and Distributed System Security*, 2000.

[11] Necula, G., "Proof Carrying Code," *ACM Principles of Programming Languages*, 1997.

[12] Sekar, R., C. Ramakrishnan, I. Ramakrishnan, and S. Smolka, "Model Carrying Code: A New Paradigm for Mobile Code Security," *Proceedings of the New Security Paradigms Workshop*, 2001.

[13] Sekar, R. and P. Uppuluri, "Synthesizing Fast Intrusion Prevention/Detection Systems From High-Level Specifications," *Proceedings of the USENIX Security Symposium*, 1999.