A Usability Study of Security Policy Management

Almut Herzog and Nahid Shahmehri

Department of Computer and Information Science
Linköpings universitet, Sweden
{almhe,nahsh}@ida.liu.se

Abstract. Security policy management is a difficult and security-critical task. We have evaluated Java's policytool with a usability study to see how well it can support users in setting up an appropriate security policy. The Java policytool is a graphical user interface tool integrated into Sun Microsystem Inc.'s Java 5.0 distribution for setting up security policies that can enable e.g. applets with more permissions than the default sandbox.
Results show that policytool is in line with other security tools, namely usability is poor. Policytool provides a certain degree of syntax help to novice users but it does not help with semantics, does not cater to expert users and actually does promote the accidental set-up of too lenient a policy. We show specific usability problems in policytool, comment on the differences in the policy files created by our study users, explore ways of solving the error-prone task of setting up a Java policy and relate this to the general subject of usability of security tools.

1 Introduction

Security policies come in many different shapes and sizes. They may be long, written documents on how employees must choose passwords, protect their office keys or handle documents. They may be short but intricate rules in a proprietary language to configure a firewall. What they all have in common is that the policy is highly specific to the environment (the company, the household, the user, the network etc.) in which it is used. There is no "one size fits all".

Our interest is in the area of security policies for software. In that area the property of "one size does not fit all" has especially dire implications. It means that the end user, be it a system administrator or regular home user, must be highly involved in the task of setting up a policy because only the end user knows "which size will fit". However, all too often end users do not perform such a setup because security is seldom a main user goal but rather a secondary exercise to the user's primary goal, which is to make the application work. Also, policy setup *is* difficult and time-consuming. "Finding the size that will fit" is a lengthy procedure.

Clearly, applications that target security need to be carefully designed to be useful and usable because "there is already plenty of evidence to suggest that end users do not need a great deal of excuse or encouragement to neglect their security responsibilities"[1]. Security software is usable if "the people who are expected to use it (1) are reliably made aware of the security task they need to perform; (2) are able to figure out

Please use the following format when citing this chapter:
Author(s) [insert Last name, First-name initial(s)], 2006, in IFIP International Federation for Information Processing, Volume 201, Security and Privacy in Dynamic Environments, eds. Fischer-Hubner, S., Rannenberg, K., Yngstrom, L., Lindskog, S., (Boston: Springer), pp. [insert page numbers].

```
1 keystore "file:///tmp/keys.data";

2 grant codeBase "file:///tmp/xx.jar" {
3  permission java.lang.RuntimePermission "getProtectionDomain";
4  permission java.net.SocketPermission "www.ida.liu.se:80", "connect, resolve";
5  permission java.util.PropertyPermission "user.dir", "read";
6  permission java.io.FilePermission "/tmp", "read";
7  permission java.io.FilePermission "/tmp/a", "write, read, delete";
8 };

9  grant signedBy "alice",  codeBase "file:///tmp/yy.jar" {
10   permission yy.YPermission "x", "set", signedBy "bob";
11 };
```

Fig. 1. A policy file with certificate database and two policy entries or grant statements.

how to successfully perform those tasks; (3) do not make dangerous errors; and (4) are sufficiently comfortable with the interface to continue using it."[2]. We will come back to this definition in the conclusion.

This paper examines how well the end user is supported in the task of setting up a security policy for Java applications. Sun Microsystems Inc. provides policytool [3], a tool with a graphical user interface (GUI) that aims at supporting the novice user in the task of setting up a security policy for a Java application or component.

We have evaluated the version of policytool that comes with Java 5.0. For the evaluation we used the think-aloud method [4] with ten graduate and undergraduate students from the area of computer science and with some degree of experience in Java programming. They were asked to individually set up a Java security policy using policytool while thinking aloud about what they were doing and about problems encountered. At the end of each evaluation the user was invited to comment on the difficulty of the tasks and the support that policytool had provided. Users found a total of 23 specific usability problems. Astonishing differences in the resulting policy could be seen. We have used the user observations to draw up suggestions and we comment on how to mitigate the difficulty of setting up Java security policies using examples from other security applications.

The paper is divided into the following sections. Section 2 describes the security architecture and policy files of Java. Section 3 describes the GUI of policytool that is used to set up a Java policy. Section 4 contains the study design. Results are shown in section 5. We discuss related studies and usability solutions in section 6 and conclude the paper with section 7.

2 Security Manager and Java policy files

All Java applications as well as other Java components such as servlets, Enterprise Java Beans (EJBs) or OSGi bundles (www.osgi.org) can be forced to execute under a so-called Security Manager. As of Java 2 the Security Manager can be set up with a set of permissions that modify the default policy of the Security Manager. Since the introduction of that new Security Manager it has been possible to allow fine-grained access control based on *where* the code was downloaded from (the so-called code base) and who has signed the code. Since the introduction of JAAS (Java Authorization and Authentication Service) it has also been possible to set up access control rules based

on *who* executes the code. In Sun's Java version, the security policy is by default put in policy files. These are text files that contain the permissions that are explicitly granted to code bases, the location from which a Java class is loaded.

An example of a policy file is shown in fig. 1. The file may start with the definition of the so-called *keystore*, a URL to a file that contains public key certificates for verifying signed Java bytecode (see line 1 in fig. 1). The policy file consists further of *policy entries* (also called *grant statements*) that define positive permissions given to the named code base. In fig. 1 there are two policy entries—one for code base `/tmp/xx.jar` (lines 2–8), one for code base `/tmp/yy.jar` with signer `alice` (lines 9–11). Each policy entry consists of one or more *permission statements*. A permission statement consists of the keyword `permission`, the permission class such as `java.io.FilePermission`, a target such as `/tmp` and an action such as `read` (line 6). *Basic permissions* such as RuntimePermissions do not have actions, only targets (in this case `getProtectionDomain` (line 3)).

Java's security architecture is thoroughly described in [5].

3 Java policytool

The Java policytool is a simple GUI application included in every Java distribution and written in Java. The policytool consists of three major windows. The first window named *Policy Tool* (see fig. 2(a)) shows information about where in the file system the policy file resides, where the keystore is located and which policy entries are present in the file. The *File* menu contains the items *Open, Save, Save As, View Warning Log, Exit.* The *Edit* menu contains the items *Add Policy Entry, Edit Policy Entry, Remove Policy Entry* (which are also available as buttons) and *Change KeyStore*. For details about each policy entry one must mark a policy entry and then press *Edit Policy Entry*. This opens the window called *Policy Entry* (see fig. 2(b)) where all details about the policy entry and its permissions are shown. Creating or editing permissions (*Add Permission* or *Edit Permission* buttons in fig. 2(b)) requires a third window titled *Permissions* (see fig. 2(c)) where one specific permission is set up.

4 Study design

The method of our evaluation was a think-aloud study [4] where end users are using the application with a given scenario and tell the evaluator what they are thinking while working with the application and how they reason about problems they encounter.

There were 10 users* in the study (graduate and undergraduate students in the area of computer science) who had all worked with Java before. Only one person had previously set up a Java security policy. The other nine users had heard of Java security policies but had never actually worked with them.

The scenario for the users in this think-aloud study was that they had just downloaded two Java components from the Internet. The advertisement for the components

* Literature [6, 4] suggests that 3-5 users are enough to identify the most important usability problems.

described certain functionality but users were told not to trust the components and thus to subject them to the Security Manager to see which permissions the components would actually need and to set up the components with minimal permissions (according to the principle of least privilege). Users were not given access to the source code.

The first component required permissions as shown in the first policy entry (fig. 1, lines 2–8).

The second component involved signed code, i.e. the code that arrived at the end user was signed by Alice and accompanied by Alice's certificate. This task required only one permission, but the difficulty was that it was not a standard permission but rather one that we had developed for this study (called yy.YPermission, fig. 1, line 10). This second task also required the set-up of the keystore (fig. 1, line 1).

The typical procedure was that users ran the component under a Security Manager until an access control exception occured (due to the lack of an appropriate permission). Then they examined the access control exception message, set up the policy file with the required permission and restarted the component.

While solving the two tasks users were encouraged to give feedback, to think aloud. When users could not proceed with their tasks because they did not know how, they were instructed to ask the evaluator, rather than searching for on-line documentation. As the tool did not provide any help features, this shortcut was chosen to keep end user frustration low and to focus the study on the tool and not on the end users' capabilities of browsing for suitable documentation on the Internet.

At the end of the evaluation, users were asked to summarise their experience with policytool. If they were not satisfied with it they were asked to give suggestions for additions or improvements or to propose different ideas for solving the task of setting up a security policy for Java applications.

5 Results

We present the results of the user trials in two groups: feedback that relates to policytool in terms of specific GUI problems and more general usability metrics; and feedback that is oriented towards finding more usable ways of setting up Java security policies.

5.1 Problems with the graphical user interface

Shneiderman and Plaisant [7] provide the following eight golden rules of interface design by which we categorised the usability problems encountered in the user interface.

1. *Strive for consistency* across the application wrt. terminology, colour, layout, fonts, etc.
2. *Cater to universal usability*, design for both novice and expert users.
3. *Offer informative feedback* for every user action by e.g. visual presentation.
4. *Design dialogs to yield closure* so that users know which steps must be accomplished and where in the process they are currently situated.
5. *Prevent errors* by disabling inappropriate fields, checking user input and supplying constructive error feedback.
6. *Permit easy reversal of actions* to relieve user anxiety and to encourage exploration.
7. *Support internal locus of control* by making the user initiate actions, not respond to system output.
8. *Reduce short-term memory load* by e.g. providing online access to syntax help, abbreviations, codes.

Table 1 contains the 23 specific problems that the users found. The problems are grouped by window in which they appear and categorised according to the above mentioned golden rules.

Table 1. Usability details found during the evaluation categorised by the golden rules of Shneiderman and Plaisant [7].

No.	Problems	Category
	The following issues were taken up for the main window (fig. 2(a)):	
1	At start-up without command line parameters policytool tries to read the user default policy file from $HOME/.java.policy. If this file does not exist (which will rarely be the case) a warning is generated. This puzzles the users. They have done nothing except starting policytool and already a warning is issued.	7
2	As a reaction many users wanted to enter the name of the policy file in the field labelled `Policy File`. But this field is not editable even though the user interface suggests it by using a text field with sunken relief. Its contents can only be changed through menu options (open an existing file, save a newly edited policy) or by starting policytool with a command line option that specifies the policy file.	1
3	After returning from the set-up of policy entries, one must save the policy file (and then start the Java application with the newly saved policy file). Every *Save*-operation results in a popup-window that informs the user that the *Save* completed successfully. Some users were annoyed by this. One user suggested that there should be a status bar at the bottom of the window that shows the result of the Save-operation and whether there are unsaved changes.	1
4	Sometimes users forgot to save their changed policy and restarted the application with an old policy file. When they went back to the policytool window, there was no hint to indicate whether or not the user had saved the file. A status bar or greyed-out menu item would have helped here.	3
5	There are no keyboard shortcuts for menu items, e.g. Ctrl-S for *Save* is not implemented. This annoys proficient programmers because they must make use of the mouse.	2
6	When users had to set up the second permission they were tempted to create a new policy entry (Button labelled *Add Policy Entry*), when the correct action would have been to add a permission to an existing policy entry (Button labelled *Edit Policy Entry*). In two cases, users created additional policy entries without any code bases (thus granting the permissions to all code), because they assumed that they *were* in fact editing the existing policy entry. Users suggested that showing the complete policy in the main window would be an advantage. Large policy files could be addressed by using collapsing trees (as found in file system browsers).	4
7	In the second task, it was necessary to also set up a keystore entry. Most users forgot that. The warning that a public key is not known when closing the policy entry window (fig. 2(b)) was not enough to remind people that the keystore was needed. Thus they would restart the application only to get the very same access control exception. This resulted in user bewilderment and frustration because they thought they had made a mistake in the permission definition but could not find any such error. However, while the permission statement was correct, the reference to the certificate database (the keystore) was missing. This would cause the code to fail with the same access control exception as if the policy entry was not there at all.	5
8	When entering the keystore location one user noticed that it did not matter if the location was entered as a URL or not. Policytool would automatically convert a non-URL location to URL notation. This behaviour is inconsistent with how the code base URL in the *Policy Entry* window is treated.	1
	The following issues were taken up for the policy entry window (fig. 2(b)):	
9	Even though users had understood that permissions are granted per code base, the text field `CodeBase` was sometimes misunderstood. One user wanted to fill in the host name needed for the socket permission.	5, 8
10	The code base must be filled in as a URL. If this is not done, an error message pops up when trying to close this window. An example of a code base would lead to less user frustration in this issue.	5, 8
11	Many users were confused by the *SignedBy* text field and did not know whether it was relevant to task 1 or not (it was not).	4, 8
12	The *SignedBy* text field was relevant in task 2 but many users hesitated what to put there: Should it be alice, Alice or yy.jar?	8
13	All users wondered about principals and whether they had to deal with them or not (principals are needed in JAAS which we did not use in this study). The UI should clarify how code bases, signers and principals interact.	4
	The following issues were taken up in the window for permission setting (see fig. 2(c)).	
14	A number of users were confused by the concepts of permission, target and action. One user was looking for the target *getProtectionDomain* in the permission list. All users were looking for targets such as */tmp/a* or *www.ida.liu.se:80* in the target list and did not expect to type the text themselves. Many users remarked that examples would have helped greatly.	4, 8
15	While permission names (FilePermission, SocketPermission etc.) are listed in alphabetical order, target names are not. Users were confused when *getProtectionDomain* was not readily visible in the list.	1
16	It is not possible to get through the lists for permissions, targets or actions more quickly by typing the first letter(s) of the permission, target or action that one wants to find in the list.	2
17	Most users did not understand that there are both basic permissions (such as a RuntimePermission at line 3 of fig. 1), which consist of a target (in this case *getProtectionDomain*) only, and regular permissions, which consist of target *and* action (see all other permissions in fig. 1). Using target for basic permissions was considered misleading by one user.	4
18	Many users did not understand that the action list would append its items to the text field. They would enter additional actions by typing the action text in the text field or by creating a new permission for the additional action.	1
19	Many permissions allow wildcards in their target and action strings. But this is represented in the graphical user interface. Only the *<<ALL FILES>>* keyword is shown for file permissions. However, hints for the wildcards * and – are missing.	8
20	Users did not understand what the text field *SignedBy* in the *Permission* window was about.	4
21	Entering a non-standard permission (the yy.YPermission) was difficult for all users. All were spending time in browsing the interface in search of a YPermission. When they entered the text manually, it could happen that the action field was greyed out and thus uneditable because users had chosen a basic permission previously. When they had entered the required text manually and closed the window for permission setting, a warning was issued that a YPermission was not known to the system. However, it was not possible to start policytool with a classpath set that would remedy this warning. This was annoying.	1, 4
22	Even users that had understood the concepts of permission, target and action were confused when these concepts were applied on a self-defined permission, even though the error message text is completely analogous (`...AccessControlException: access denied (yy.YPermission x set)`, cf. fig. 3).	4
23	When adding the non-standard YPermission, user were tempted to forget to put the full class name yy.YPermission and would only put YPermission.	5

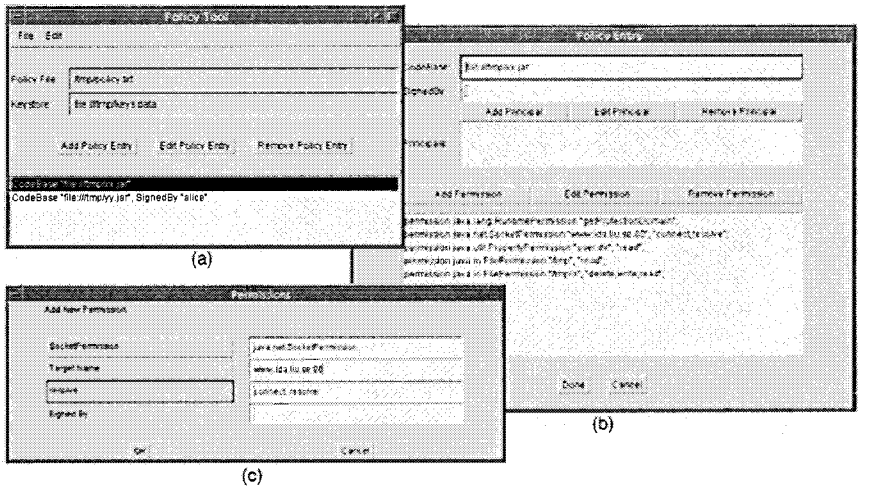

Fig. 2. (a) Main window of the Java policytool (b) Permission entries for the code base `/tmp/xx.jar` (c) Set-up of permissions.

```
Exception in thread "main" java.security.AccessControlException:
 access denied (\textit{java.net.SocketPermission www.ida.liu.se:80 connect,resolve})
        at java.security.AccessControlContext.checkPermission(AccessControlContext.
        java:264)
        at java.security.AccessController.checkPermission(AccessController.java:427)
        ...
        at java.net.URLConnection.getContentType(URLConnection.java:479)
        at xx.XX.main(XX.java:113)
%\end{alltt}
```

Fig. 3. Error message when executing the code for the first task without the proper socket permission. The text in italics shows the permission that is needed but not currently given. This is the text that corresponds almost verbatim to what must be put in the policy file, see line 4 in fig. 1.

The most frequent user problem was that users simply did not know how to proceed (categorised as no. 4—Design dialogs to yield closure): Users did not know which steps they had to accomplish now nor where in the process of policy set-up they were currently situated—and policytool offered no help. This is most serious and cannot be remedied easily. Categories 1 (inconsistencies) and 8 (reduce memory load) are runners-up. Subjects found a number of inconsistencies within the application (e.g. URL checking is handled differently in two different windows; some lists are ordered alphabetically, some are not) and with general GUI guidelines (e.g. successfully saving a policy causes a pop-up window to appear that must be acknowledged by the user).

It cannot be stressed enough that there was no help feature at all included in the GUI of policytool, not an example, not a pointer to a web page, no help on how to use policytool. No user could complete the task without help from the evaluator. Many said that they would have had to study the documentation extensively before being able to

solve the two tasks on their own. This clearly violates items 2 (universal usability) and 8 (reduce short-term memory load).

Working with signed files was considered difficult by all users, which is in line with other studies [2, 8] involving digital signatures. This task required a good understanding of the concepts of public key infrastructure and certificate databases. However, this understanding was made even more difficult due to the fact that keystore creation and import of certificates are not integrated into policytool but done in a separate command line tool called keytool. Users found this switch between applications most confusing. Also, keytool uses a lot of options that must be known to the user. As keytool was not part of the evaluation, users were supplied with the correct commands. Many users commented that it would probably have taken them a long time to figure out the command chain by themselves.

The lack of integration between policytool and the runtime environment also caused problems. Even reasonably experienced programmers did not realize—at first—that the access control exception message caused by the Java runtime contained the permission that needed to be put in the policy file (see bold face text of fig. 3). They had not seen access control exceptions before, though they had seec other Java exceptions, and started examining the stack output rather than the access control exception message.

After evaluating this tool, it is clear why applets as a substitute for plugins or for applications never had a breakthrough—managing their required permissions is an insurmountable task for non-Java security experts given a tool like policytool, especially when working with signed code.

5.2 Differences in the policy files

Of the ten created policy files, four show interesting deviations that have an impact on security and performance.

One user created three permissions where one would have been enough. Instead of the compact

```
permission java.io.FilePermission "/tmp/a", "read, write, delete";
```

the user created three permission entries: one for read, one for write, one for delete. This is not a security problem but an inefficient way of stating policies. Other users started out with this approach but realised what they were doing and changed their policy to the more compact way of stating the permission.

One user became impatient when setting up the required permissions for file access to `/tmp/a` and chose all actions. This gave the code the unneeded and dangerous permission to *execute* `/tmp/a`.

One user did not supply the port for the socket permissions, thus allowing the code to connect to any port at www.ida.liu.se (instead of port 80).

Two users mistakenly created policy entries that gave permissions to any code because they had not actively chosen the code base to which they wanted to add a permission:

```
grant {
  permission java.io.FilePermission "/tmp/a", "write";
};
```

Both users mistook the *Add Policy Entry* button for the *Edit Policy Entry → Add New Permission* sequence. This serious mistake is a direct result of the GUI design of policytool. In a hand-typed policy file this problem would not have arisen.

5.3 Effectiveness, efficiency, user satisfaction

The previous sections describe specific problems of policytool. This section summarises these findings and comments on the general usability of policytool following the usability-defining keywords of the ISO standard 9241-11: effectiveness, efficiency, and user satisfaction.

Effectiveness judges how well policytool supports the user in setting up a policy. If the user has all the knowledge about how Java security works, policytool may be quite a good syntax helper. However, novice users do not have that knowledge and are left on their own to find guidance in how to proceed to set up a policy. Effectiveness could be increased by guidance features such as wizards or safe staging (see section 6) that help identify the steps that must be taken to arrive at a functional policy. To arrive at a holistic solution, keytool must also be integrated.

Efficiency judges how resource-consuming (both computer and human) policytool is while accomplishing its task. Policytool fails on the human side. Users had to spend a lot of time and effort in figuring out what had to be done and received no help from policytool on this issue. Efficiency for novice users could be increased with wizards and help features; efficiency for expert users could be increased by keyboard shortcuts for buttons and menu items and a view that allows direct editing of the policy file.

User satisfaction judges how acceptable policytool is to the users and if there is a fun factor or enriching experience associated with using policytool. The study users were not that negative towards policytool as a tool for editing a policy file, but they were very negative in their satisfaction with setting up security policies. They would have prefered a tool that was integrated into the Java runtime, a tool that would automatically collect required permissions and interact with the user only to ask whether a permission should be allowed or not.

5.4 Suggestions for improvement

This section contains user suggestions for improving the usability (and security) of setting up Java security policies and comments about the feasibility of the suggestions . The suggestions are categorised into new language features, help for end users and help for developers.

Help for end users As one user suggested: "There should be a `-learn` option to the Java runtime so that I do not have to be the parser for the system. My task [of putting required permissions in the policy file] could have been automated."

Another user pointed out that there is a mismatch between the syntax of the policy file and the syntax of the access control exception text (see figures 1 and 3). It should be possible to create a smart copy-and-paste function that correctly pastes an access control exception text into a policy file.

One user suggested helplessly that there should be a check that ensures "that one does not allow too much". He had realised during the debriefing that his policy file allowed too much.

It is definitely possible to set up Java security policies at runtime. JSEF [9] has done this with direct user interaction through a GUI. The user is prompted to decide whether to accept the required permission or not. However, using a GUI is not always possible, especially not in container environments where no display is associated with a piece of code. We therefore suggest the implementation of a "policy learner" that will either grant all required permissions by default and write them to a policy file for post mortem analysis or that will interact with user preferences. An analyzer component, much like an intrusion detection module, could react to flaws in the policy and trigger an alert if a piece of code has read a file in the file system and then opens a network connection (implying a breach of confidentiality because data may leave the host system and be transmitted somewhere else without the consent of the host or material owner). A policy learner could have prevented all of the deviations described in 5.2.

Help for developers A tool could be created that semi-automatically detects required permissions in Java code (along the lines of C-tools like IBM Rational Purify, which detects runtime memory leaks). It must be ensured that all code is run (as with IBM Rational PureCoverage) so that even permissions for error situations (writing to an error log file) are detected. A problem is that e.g. automatically generated file names (e.g. date+time.txt) are then difficult to put in a policy.

New language features One user proposed that the Java language could enforce that permissions must be declared in the same way as exceptions are declared today. As in the previous item, dynamic targets would be difficult to handle. It would be possible to require declaration of permissions within the code or that a JIT(Just In Time)-Compiler could produce a footprint of required permissions at early-execution time. It would not be too difficult to find out which *classes* of permissions a piece of code needs, but the problem resides with the targets that could be dynamically created at runtime, e.g. a program may ask the user to supply the hostname of the computer to which it wants to connect. It is obvious at compile time that a socket permission is needed, but the target of the socket permission, the actual host address, is not known before runtime. Koved and others [10] have addressed static analysis of Java code in order to find required permissions. Their approach examines Java bytecode prior to execution and, using invocation graphs, finds required permissions. However, such a solution will not help for targets created at runtime (as described above).

One user suggested permissions that allow the deletion of the files that the program has created. This could be either a new permission class or an additional action for standard FilePermissions. A lot of work exists to make Java security policies more expressive, e.g. [11–13], allowing policies like e.g. "no write after a confidential read". However, none of the existing solutions have addressed usability issues. It is not likely that users would be able to handle more expressive security policies easily.

6 Related work

It may seem that it is rather lopsided to evaluate a small application, find a lot of usability problems and draw general conclusions about usability of security tools from that. Unfortunately, policytool is just one in a long line of security tools that display an astonishing lack of user focus even though they are targeted at novice users. E-mail software with encryption [2, 8], Internet banking [14, 15], Internet Explorer [1], MS Word [16] and firewalls [17, 18]—all these applications show severe shortcomings in the usability of their security functions.

Currently suggested ways for improving the usability of security functions comprise "good-enough security" [19] or "if we put usability first, how much security can we get" [20]. These approaches do not strive for perfect security, but for security that works in many cases. *Safe staging* [21] slowly and safely introduces the security novice to a security application through several stages. Also the concept of *social navigation*—showing how previous users have interacted or should interact with a user interface by leaving traces of earlier use on the artefact—has been used to enhance users' perception of security features [22]. Yee [23] proposes *security by designation*: User action designates what the user wants to do and thus also authorises an action.

In the end, there is a lot to be gained by following professional usability guidelines such as Jonston, Eloff and others [17, six criteria for successful human-computer interaction in security applications], Leveson [24, 60 guidelines for safe HMI design], Nielsen [4, 10 design slogans], Garfinkel [25, 6 general principles], ISO standard 9241 (parts 10–17), Shneiderman and Plaisant [7, 8 golden rules of interface design] and by subjecting the security product to usability testing, again and again.

7 Conclusion

In this paper we have evaluated the usability of the Java policytool for setting up security policies for Java applications or components. We have found a number of problems—some small and some larger—in the GUI of policytool. The small ones violate GUI design guidelines and thus easily annoy users. The larger problems lead to serious lapses in the policy, thus endowing a Java application with more rights than it should have.

The study users suggested a number of improvements that range from more expressive permissions to runtime policy setup with or without user interaction and to issues in static analysis such as changing the Java language in order to support the declaration of permissions.

According to the definition of usable security software (see introduction), the Java policytool is not usable. (1) Users are *not* reliably made aware of their security tasks because policytool is not integrated with the actual Java application. (2) Figuring out how to perform the policy setup takes a lot of help from web resources and is not explained explicitly or implicitly within policytool. (3) Dangerous errors can easily be made as was shown in our evaluation. (4) Expert users are quicker in directly editing the policy file than handling the somewhat cumbersome policytool.

Obviously there is a great potential for improvement and our future work will make use of findings from work in the field of usable security to improve the user task of setting up security policies.

References

1. Furnell, S.M.: Using security: easier said than done. Computer Fraud & Security **2004**(4) (2004) 6–10
2. Whitten, A., Tygar, J.D.: Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In: Proceedings of the 8th USENIX Security Symposium (Security'99), Usenix (1999)
3. Sun Microsystems Inc.: Policy tool—policy file creation and management tool. `http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/policytool.html` (2002)
4. Nielsen, J.: Usability Engineering. Morgan Kaufmann Publishers, Inc (1993)
5. Oaks, S.: Java Security. 2nd edn. O'Reilly & Associates, Inc. (2001)
6. Faulkner, X.: Usability Engineering. Macmillan Press Ltd (2000)
7. Shneiderman, B., Plaisant, C.: Designing the User Interface. 4th edn. Addison Wesley (2004)
8. Gerd tom Markotten, D.: Benutzbare Sicherheit in informationstechnischen Systemen. Rhombos Verlag, Berlin (2004) ISBN 3-937231-06-4.
9. Hauswirth, M., Kerer, C., Kurmanowytsch, R.: A secure execution framework for Java. In: Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS'00), ACM Press (2000) 43–52
10. Koved, L., Pistoia, M., Kershenbaum, A.: Access rights analysis for Java. In: Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02), ACM Press (2002) 359–372
11. Mehta, N.V., Sollins, K.R.: Expanding and extending the security features of Java. In: Proceedings of the 7th USENIX Security Symposium (Security'98), Usenix (1998)
12. Corradi, A., Montanari, R., Lupu, E., Sloman, M., Stefanelli, C.: A flexible access control service for Java mobile code. In: Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC'00), IEE (2000) 356–365
13. Venkatakrishnan, V., Peri, R., Sekar, R.: Empowering mobile code using expressive security policies. In: Proceedings of the New Security Paradigms Workshop (NSPW'02), ACM Press (2002) 61–68
14. Hertzum, M., Jørgensen, N., Nørgaard, M.: Usable security and e-banking: Ease of use vis-à-vis security. In: Proceedings of the Annual Conference of CHISIG (OZCHI'04). (2004)
15. Nilsson, M., Adams, A., Herd, S.: Building security and trust in online banking. In: Proceedings of the Conference on Human Factors in Computing Systems (CHI'05), ACM Press (2005) 1701–1704
16. Furnell, S.M.: Why users cannot use security. Computers & Security **24**(4) (2005) 274–279
17. Johnston, J., Eloff, J.H.P., Labuschagne, L.: Security and human computer interfaces. Computers & Security **22**(8) (2003) 675–684
18. Wool, A.: The use and usability of direction-based filtering in firewalls. Computers & Security **23**(6) (2004) 459–468
19. Sandhu, R.S.: Good-enough security. IEEE Internet Computing **7**(1) (2003) 66–68
20. Smetters, D., Grinter, R.E.: Moving from the design of usable security technologies to the design of useful secure applications. In: Proceedings of the New Security Paradigms Workshop (NSPW'02), ACM Press (2002) 82–89
21. Whitten, A., Tygar, J.: Safe staging for computer security. In: Proceedings of the CHI2003 Workshop on Human-Computer Interaction and Security Systems. (2003)
22. DiGioia, P., Dourish, P.: Social navigation as a model for usable security. In: Proceedings of the Symposium on usable privacy and security (SOUPS'05), ACM Press (2005) 101–108
23. Yee, K.P.: Guidelines and strategies for secure interaction design. [26]
24. Leveson, N.: Safeware: System Safety and Computers. Addison Wesley (1995)
25. Garfinkel, S.L.: Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable. PhD thesis, Massachusetts Institute of Technology (2005)
26. Cranor, L.F., Garfinkel, S.L.: Security and Usability. O'Reilly & Associates, Inc (2005)