Empowering End Users to Confine Their Own Applications: The Results of a Usability Study Comparing SELinux, AppArmor, and FBAC-LSM

Z. CLIFFE SCHREUDERS, TANYA MCGILL, and CHRISTIAN PAYNE, Murdoch University

Protecting end users from security threats is an extremely difficult, but increasingly critical, problem. Traditional security models that focused on separating users from each other have proven ineffective in an environment of widespread software vulnerabilities and rampant malware. However, alternative approaches that provide more finely grained security generally require greater expertise than typical end users can reasonably be expected to have, and consequently have had limited success.

The functionality-based application confinement (FBAC) model is designed to allow end users with limited expertise to assign applications hierarchical and parameterised policy abstractions based upon the functionalities each program is intended to perform. To validate the feasibility of this approach and assess the usability of existing mechanisms, a usability study was conducted comparing an implementation of the FBAC model with the widely used Linux-based SELinux and AppArmor security schemes. The results showed that the functionality-based mechanism enabled end users to effectively control the privileges of their applications with far greater success than widely used alternatives. In particular, policies created using FBAC were more likely to be enforced and exhibited significantly lower risk exposure, while not interfering with the ability of the application to perform its intended task. In addition to the success of the functionality-based approach, the usability study also highlighted a number of limitations and problems with existing mechanisms. These results indicate that a functionality-based approach has significant potential in terms of enabling end users with limited expertise to defend themselves against insecure and malicious software.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Security, Human Factors

Additional Key Words and Phrases: Application-oriented access controls, sandboxing, POLA, usability, HCISec, SELinux, AppArmor, FBAC-LSM, functionality-based application confinement

ACM Reference Format:
Schreuders, Z. C., McGill, T., and Payne, C. 2011. Empowering end users to confine their own applications: The results of a usability study comparing SELinux, AppArmor, and FBAC-LSM. ACM Trans. Inf. Syst. Secur. 14, 2, Article 19 (September 2011), 28 pages.
DOI = 10.1145/2019599.2019604 http://doi.acm.org/10.1145/2019599.2019604

1. INTRODUCTION

Popular operating systems, such as Windows and Unix, employ security mechanisms that control what each individual user may do. However, a process executed by a given user typically inherits all of that user's privileges. Such access control schemes do not protect the user against attacks performed by the programs they run, thereby leaving

Authors' address: Z. C. Schreuders, T. McGill, and C. Payne, School of Information Technology, Murdoch University, Murdoch 6150, Perth, WA, Australia; email: {c.schreuders, tmcgill, c.payne}@murdoch.edu.au.

DOI 10.1145/2019599.2019604 http://doi.acm.org/10.1145/2019599.2019604

users exposed to widespread, contemporary threats such as security vulnerabilities in their applications and malicious software.

Application-oriented access control schemes exist that restrict the actions of each application. By specifying what each application is authorised to do, potential damage from a misbehaving application is significantly limited. Examples of schemes that provide application restrictions include: chroot, FreeBSD Jails [Kamp and Watson 2000], Solaris Zones [Tucker and Comay 2004], domain and type enforcement (DTE) [Badger et al. 1995], Role Compatibility (RC) [Ott 2002], Bitfrost [Krsti and Garfinkel 2007], CapDesk [Miller et al. 2004], Polaris [Stiegler et al. 2006], TRON [Berman et al. 1995], Virtual Machines (VMs) [Madnick and Donovan 1973], PeaPod [Potter et al. 2007], Alcatraz [Liang et al. 2009], Janus [Wagner 1999], Systrace [Provos 2002], SELinux [Vance and Salamon 2001], and AppArmor [Cowan et al. 2000]. These schemes can be divided into broad categories such as isolation-based and rule-based restrictions.

Isolation-based application-oriented access controls simply confine the application to a limited name-space and isolate it from the rest of the system. Although providing a relatively straightforward mechanism, isolation does not suit typical user workflows where multiple applications need to share and exchange data. It is also often impractical to individually isolate all of a user's applications as many of these schemes demand significant redundancy in terms of resources.

Rule-based application-oriented access controls can enforce least privilege by permitting programs to only access the specified resources they require to carry out their legitimate functions. However, this finely grained level of control often leads to complex policies as sophisticated applications typically require access to a myriad of resources. Policies for these schemes also expose the complexity of the underlying platforms and applications, and therefore can be very hard to create and manage without significant levels of expertise on the part of the user.

Despite the fact that usability has long been acknowledged as an important aspect in the design of security systems [Saltzer and Schroeder 1975], the topic received limited attention in the literature until it was demonstrated that a poorly designed security user interface results in degraded protection [Hitchings 1995; Zurko and Simon 1996]. Although awareness of the importance of usability in security design has improved [Cranor and Garfinkel 2005], and the literature now contains many publications related to computer security usability, very little research has investigated or addressed the usability issues associated with application restrictions.

A study by Dewitt and Kuljis [2006] assessed the usability of the Polaris security mechanism [Stiegler et al. 2006], an application-oriented access control system for Windows designed with usability in mind. The Polaris study involved 10 participants utilising that system to carry out a number of tasks. As with the usability study described here, their success at the tasks was evaluated and perceived usability measured. After using Polaris to attempt a number of tasks, participants on average rated the system 44.2 out of 100 using the System Usability Scale (SUS) [Brooke 1996]. Consequently, the study concluded that further work was necessary to improve the usability of Polaris.

The study described here makes a significant contribution to the pool of research on the usability issues associated with application restrictions. A comparative study was conducted to evaluate the usability of a functionality-based approach to application restriction, where applications are restricted based on the expected high-level behaviour of the program. The usability study compared a Linux-based implementation, FBAC-LSM, with the widely deployed SELinux and AppArmor mechanisms. To date, this is the most comprehensive comparative usability study conducted on application confinement systems.

SELinux was originally developed by the US National Security Agency, and provides an implementation of mandatory controls for Linux. Access control decisions are made based on the security context resources are labelled with, combining role-based access control (RBAC), domain and type enforcement (DTE), and multilevel security (MLS). DTE forms the basis of application restrictions; rules determine the domain a program is associated with, and define how processes within particular domains may access resources labelled with specific types. Typically separate domains are specified for each program. A number of user-space tools to configure SELinux are available, and are available on Linux distributions such as Fedora. Although some management tasks can be achieved using GUI tools (such as the SELinux Policy Generator tool), most require the use of command line tools. SELinux rules can be very complex and are defined in terms of security contexts that are applied as labels on resources [Zanin and Mancini 2004]. The out-of-the-box configuration for SELinux typically aims to lock down system-wide services and remain out of sight from end users; most of their processes run unconfined. While primarily aimed at expert users and security administrators, SELinux is the most widely deployed Linux-based security mechanism capable of application confinement. For many Linux users it will be the only such scheme installed on their system, and recent work has been aimed at improving the usability of SELinux [Athey et al. 2007; Nakamura et al. 2009]. In any case, its maturity and wide deployment makes it the archetypal Linux enhanced security mechanism and worthy of study.

AppArmor, previously known as SubDomain [Cowan et al. 2000], also implements mandatory controls for Linux, although using a simpler model than SELinux. AppArmor defines a list of resources based on resource names (such as file paths) for each restricted program to specify what may be accessed. Simple abstractions such as dbus, kde, and nameservice are used to group privileges related to particular low-level program characteristics and can be used when constructing policies. User-space tools to configure AppArmor are available, including graphical tools that are available on Linux distributions such as openSUSE. These graphical tools can be used to create and manage policy, including a "learning mode" used to create application profiles based on the actions a program attempted previously. AppArmor policies can be long and detailed, reflecting the underlying complexity of the confined applications and the various platform layers these depend on. At one stage an online repository was available for users to share the application profiles they had created. AppArmor has been presented by Novell as an easier to use alternative to SELinux, with a focus on providing mandatory application-oriented controls.

FBAC-LSM takes a different approach: applications are confined based on the functionalities they are expected to perform [Schreuders and Payne 2008a]. FBAC-LSM is an implementation of the *functionality-based application confinement* (FBAC) model[1]. The FBAC model provides both mandatory and discretionary controls, allowing administrators and end users to define policies that are simultaneously enforced. This enables administrators and users to restrict applications in order to enforce their respective security goals. Policy abstractions, known as *functionalities*, are used to authorize programs to access resources. Functionalities model the privilege requirements of high-level application features such as "Web Browser," "Image Editor," or "Game" [Schreuders and Payne 2008b].

Functionalities are parameterized, which enables them to adapt to each program's specific needs. Application policies can specify arguments to the parameters of the functionalities assigned. These arguments can be passed in a fashion similar to subroutines in programming languages. Parameters can describe application-specific

[1]FBAC-LSM is free open source software available at: http://schreuders.org/FBAC-LSM.

details such as the location of files, directories, or network resources. For example, parameters may specify where an application stores its configuration files, the location the user intends to store files created using the application, or the hosts the application is authorised to communicate with.

Functionalities are also hierarchical; that is, they can contain other functionalities. Functionality hierarchies enable further policy modularity, and can provide layers of abstraction and encapsulation. For example, the high-level "Web Browser" functionality includes functionalities representing lower-level policy details, such as "HTTP client." As FBAC-LSM policies are defined in terms of hierarchical functionalities, low-level details are abstracted, and an overview of the policy can be presented to the user in terms of high-level security goals.

FBAC is designed to abstract policy details away from users and enable them to specify what applications are authorized to do based on high-level and conceptually simple goals. To restrict a program, functionalities that describe the behavior expected of the application are assigned and any parameters then specified.

The FBAC model can facilitate automation of certain stages of policy construction. The implementation provides a graphical policy manager tool that steps users through the process of defining application policies and performs automation where possible. The policy manager can suggest functionalities and parameter arguments based on analysis of the program and filesystem. Since manually specifying parameter argument values can require knowledge of the applications being confined (such as where configuration files are stored), this automation can further reduce the expertise required to manage the security scheme. Unlike most other rule-based application-oriented access controls, policies are created a priori: without needing applications to be first executed in order to generate confinement policy. FBAC-LSM includes a learning mode for situations where application policies do not provide all the privileges necessary.

Each of these three schemes allows users to restrict the actions of applications. The Linux security module (LSM) framework, as currently implemented, only allows a single security mechanism to be enabled at a time on a Linux system. Therefore users must choose between these security systems if they wish to confine their applications with one of these schemes.

2. EXPERIMENTAL OBJECTIVES

The usability study described here was designed to evaluate the usability of the FBAC approach relative to the two most mature and widely deployed Linux enhanced security modules, SELinux and AppArmor, and their configuration tools.[2] This was the first formal comparative usability study to examine any of these systems. Where possible the three systems were compared to each other, and the effects of the different approaches taken by each were investigated.

Although application-oriented access controls can be shipped with predefined policies specified by third parties that restrict specific known programs, this study focuses on the ability to specify policies that allow users or administrators to protect themselves against potentially unknown applications and enforce their own security goals.

In particular, the following aspects of usability and security were measured and analyzed:

(1) User perceptions of the usability of the three confinement schemes
(2) User success at creating and applying confinement policies

[2]This study considers the usability of the schemes in terms of configuration which is done via user-space tools.

(3) Ability of user-confined programs to continue to execute as expected
(4) Overall risk exposure after confinement
(5) Ability to successfully restrict well-behaved programs
(6) Ability to successfully restrict malicious programs
(7) Time-efficiency of the three confinement schemes

3. METHOD

The usability study employed a within-subjects design. Participants used all three security mechanisms to construct policies to confine two programs. Participants provided feedback regarding security system usability and preference, and the security properties of the resulting policies were analysed.

3.1. Participant Recruitment

Participants were primarily recruited from the information technology students at an Australian university, members of a Linux user group and an information security association. Participant recruitment targeted people who had previously used Linux systems, although this was not a requirement for participation. Participants were recruited using flyers on university notice boards, announcements in lectures and via email. A prize of an 8GB iPod Nano was used to encourage participation, and was awarded to a participant chosen at random.

A convenience sample of 46 people was used, made up of every potential participant available during the study period. Seven of those people left before completing the experiment and were excluded from analysis, leaving 39 participants considered during analysis. The size of this sample compares favourably with that of the earlier Polaris study [Dewitt and Kuljis 2006].

3.2. Environment and Logistics

The study was conducted over a number of sessions in a university computer laboratory, with between one and 10 subjects participating at a time. In order to ensure consistent dissemination of information, most information presented to participants was prerecorded and was presented via video files launched via batch scripts on the computer.

Participants were assigned individual copies of three Virtual Machines (VMs) setup for use with the three security systems. Since no single distribution supported all the user-space tools, the Linux distributions used for each of the systems were those with the most complete support for the security systems studied: Fedora 11 for SELinux and openSUSE 11.1 for AppArmor. openSUSE 10.3 was used for FBAC-LSM, since that was its development environment. Each of the environments were configured to look alike. Access to the VMs was via batch scripts that used VMWare player to run the appropriate VM, and logged the time VMs were started and when participants were finished. Each participant's VMs were then stored for later analysis.

3.3. Preparation

Participants were randomly assigned an order for using the three security systems to remove any biasing due to learning effects [Greenwald 1976]. Participants were supplied with headphones and the following handouts.

— an ID and the order in which they were to use the three systems;
— welcome page, system use, and task scenario information;

— a copy of the Filesystem Hierarchy Standard (FHS) reference. The complete FHS v2.3 was available as a PDF file on the lab computers;[3]
— a Unix/Linux command reference.[4]

A short presentation explained the various handouts and how to access the videos and VMs via the scripts. Participants were prompted to record their ID on the computers they were using to facilitate the collection and collation of data. The time constraints were also explained: participants were encouraged to spend a maximum of approximately one hour on each system, with a total maximum experiment time, including feedback, of four hours[5]. Participants were encouraged to ask for help if they were stuck on a task (as they might do in a workplace environment). Participants were also asked to notify the moderator if they encountered unrelated technical problems; for example, if a VM were to crash. Participants were then prompted to watch the introductory video. This video explained the goal of application confinement, and provided further details as to how the experiment would be run. As was explained in the video, the sequence of the experiment was as follows.

— viewing of the introductory video;
— preexperiment questionnaire;
— Linux filesystem video;
— for each security mechanism:
  — Mechanism videos;
  — Confining the programs;
  — Post-task questionnaire;
— postexperiment questionnaire;
— debriefing.

The pre-experiment questionnaire was used to identify demographic characteristics of participants. Information collected included self-assessed expertise and experience. Each of the participants rated their computer skill, knowledge of computer security, knowledge of Linux, and knowledge of how files are organised on Linux on semantic differential scales. The frequency with which they had used Linux was recorded using a multiple choice question.

The filesystem video elaborated on the FHS reference handout, illustrating the directory hierarchy on an example Linux system. This familiarized the participants with the Linux directory structure. This knowledge would assist participants in utilizing each of the three mechanisms being compared, and ensured a minimal level of awareness about the way files are organized on a Linux system.

Before participants used each system, they watched a video describing the way the security system worked and a demonstration of configuring the system. Each explanation video covered the same level of detail: describing policy components, how policy is represented on disk, the states that policies for applications can be in (either enforced or not), an overview of the steps involved in confining an application, and a list of helpful commands. Another video for each system gave a demonstration of creating a policy to confine the KWrite program as an example. When participants were ready to start learning about each system, they were given a hard copy of the demonstration script so they could more quickly access the information without re-watching the video.

[3]`http://www.pathname.com/fhs/`

[4]`http://fosswire.com/-post/2007/8/unixlinux-command-cheat-sheet/`

[5]This relatively generous maximum timeframe was intended as a rough guide for participants based on the time taken during the pilot study, and was intended to avoid disadvantaging SELinux, which during the pilot study took the longest to complete. Many participants finished sooner than these guidelines. Participants were provided with refreshments and in general seemed to remain receptive and responsive throughout.

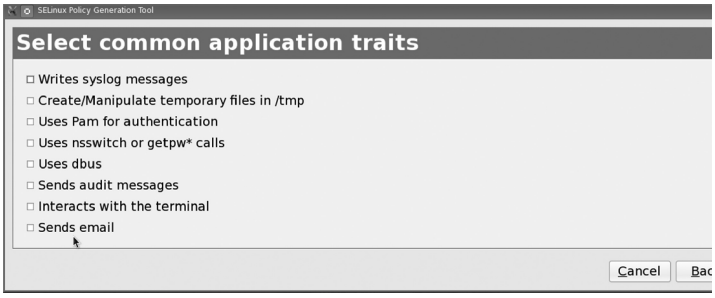

Fig. 1. SELinux Policy Generator specifying low-level application traits.

3.4. Tasks

Information about the programs to be confined was presented to the group on the task scenario handouts (Appendix A) and during the initial talk. Using each of the mechanisms in the random order allocated, participants consecutively created security policies for these programs with the goal of restricting the ability of each program to act maliciously, while allowing the programs to operate as described in the scenarios. At no point were participants given any indication that FBAC-LSM had been created by one of the authors.

The programs that the participants endeavored to confine were the Opera web browser and a simulation of a Trojan horse posing as a Tetris game, KSirtet, which was downloaded from an unauthenticated Web site. Participants were informed they should allow Opera to browse the Web, chat using IRC, and download files, while KSirtet should be permitted to operate as a game.

Both of these scenarios were designed to pose realistic risks and this was explained in the information presented to participants during the introduction video and on the scenario sheet. As Web browsers interact with external untrusted hosts, software vulnerabilities could lead to an attacker taking control of the program. A game originating from an unauthenticated source could be malicious code posing as a legitimate program. The following section describes how each of the security systems can be configured by end users.

3.4.1. Steps Involved. Using the tools typically available on an SELinux-enabled system (where possible using graphical tools), specifying a new application policy module usually involves using the Policy Generator GUI tool to create a barebones skeleton policy. Using this tool involves specifying the executable path, selecting ports and low-level application traits (as illustrated in Figure 1), and manually specifying any directories or files the application manages.

This process generates a policy module that is by default permissive (not enforced) and incomplete. Next, a number of command line tools need to be used, and the program being confined needs to be run, to generate the detailed rules that authorise the program to access the resources it requires to run. Figure 2 illustrates using command line tools to generate additional rules based on previous program activity. Subsequently, the .te file needs to be manually edited to put the domain into an enforced mode and the policy needs to be compiled and loaded into effect. Command line tools can be rerun to add any further rules.

Using the YAST AppArmor Add Profile Wizard GUI tool involves the following steps. First, the user specifies the name of the application to be confined. Next, the user is prompted to run the program. After the user has used the program, they are asked to review and vet (either allowing or denying) each of the low-level rules that would allow

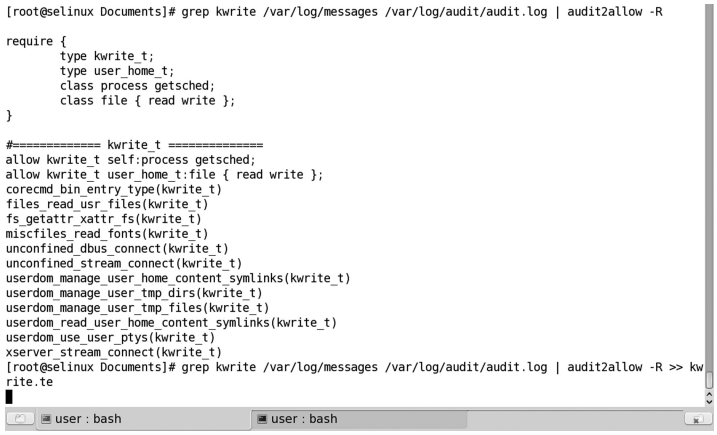

Fig. 2. SELinux command line configuration.

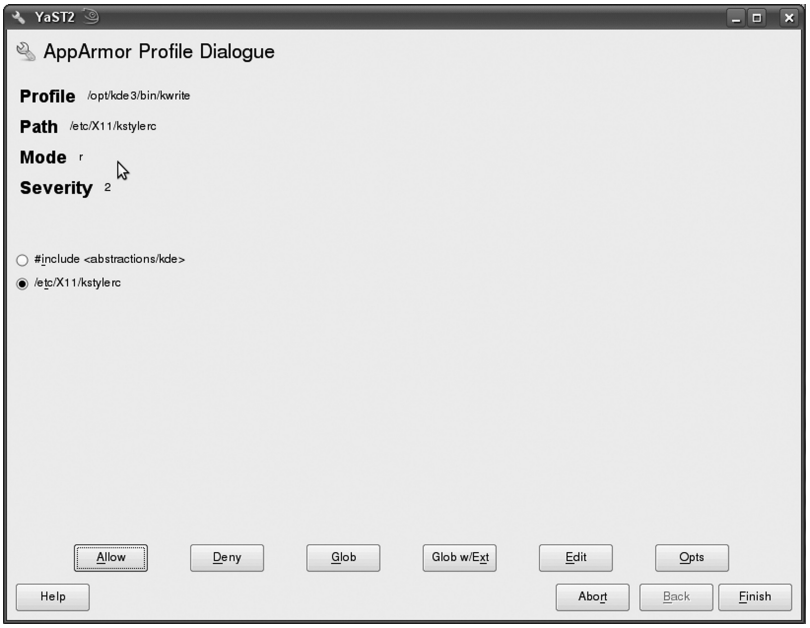

Fig. 3. AppArmor Add Profile Wizard vetting previous program activity.

the program to behave the same way in the future. Figure 3 illustrates the process of vetting the previous actions of an application using the Add Profile Wizard. The user is then presented with a text file view of the policy they have created. Depending on the way the tool is used the profile may be placed in effect, or remain in an unenforced state, in which case the user can use the AppArmor Control Panel GUI to put the profile into an enforced mode. The Update Profile Wizard can be used to add further rules.

Using the FBAC-LSM Policy Manager GUI tool to specify application policies involves the following steps. First, the user starts the Add Application Wizard, and specifies the name of the application being confined. Next, the executable paths are

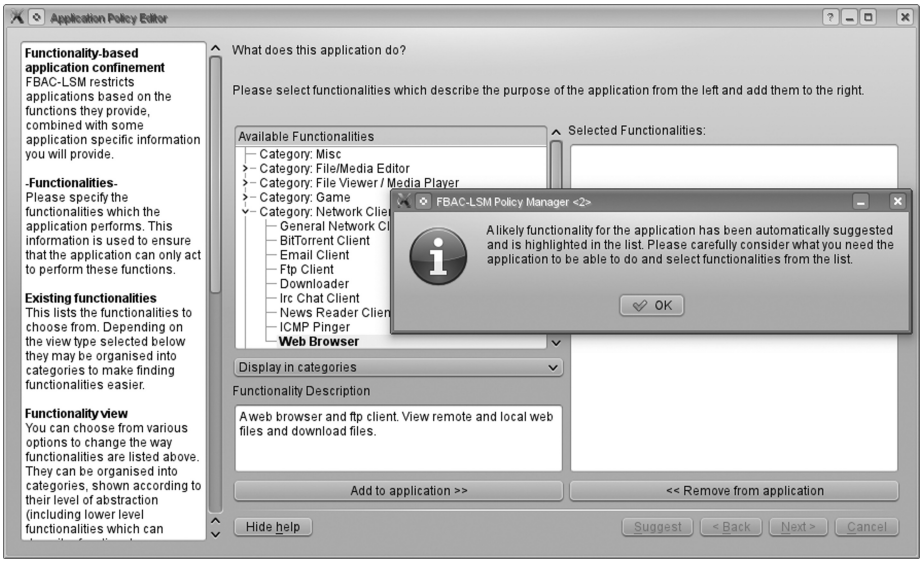

Fig. 4. FBAC-LSM Policy Manager functionality suggestions.

specified, which the wizard attempts to automatically detect. The type of program (technically the base-level functionally for the program), either command line or graphical is specified, this is typically suggested automatically and accurately. As illustrated in Figure 4, the tool then performs some analysis and suggests likely high-level functionalities that describe the features the application performs. The user selects which functionalities apply, taking the suggestions into consideration.

Then, for each of the functionalities, the argument values are specified for parameters. These specify all the application-specific information that allows the functionalities to adapt to the needs of the programs being confined. Depending on the purpose of each parameter, they can take the form of files, directories, ports, or IP addresses. In most user-independent cases (such as identifying the locations of application-specific configuration files), after some analysis the wizard can automatically suggest values, which the user can edit or accept. User-dependent cases (such as where the user chooses to store their own files) are specified manually by the user. Figure 5 illustrates the interface for specifying parameter arguments. After specifying values for all the parameters, the user chooses the name for the policy file in which the policy is stored. The user can then review the application policy that they have created in a number of ways. The user then saves the policy and loads it into effect using the Policy Manager main dialogue. The learning mode can be used to add further privileges to the application profile.

The tasks in the usability study were intentionally challenging for FBAC-LSM and, as mentioned, they posed realistic situations and threats. As with the majority of applications that have been studied, the automation provided by the FBAC-LSM policy manager assisted the user [Schreuders et al. 2011]. However, in order to meet all the requirements in the task scenario, the user had to deviate from the suggestions and also manually provide some details. For Opera, the FBAC-LSM policy manager suggested the functionality “Web Brower,” and the user should have also specified “IRC Client.” For KSirtet, the policy manager suggested both “Game” and “Network Game” functionalities, although KSirtet only required the “Game” functionality to function legitimately, since multiplayer features were provided locally with players sharing the

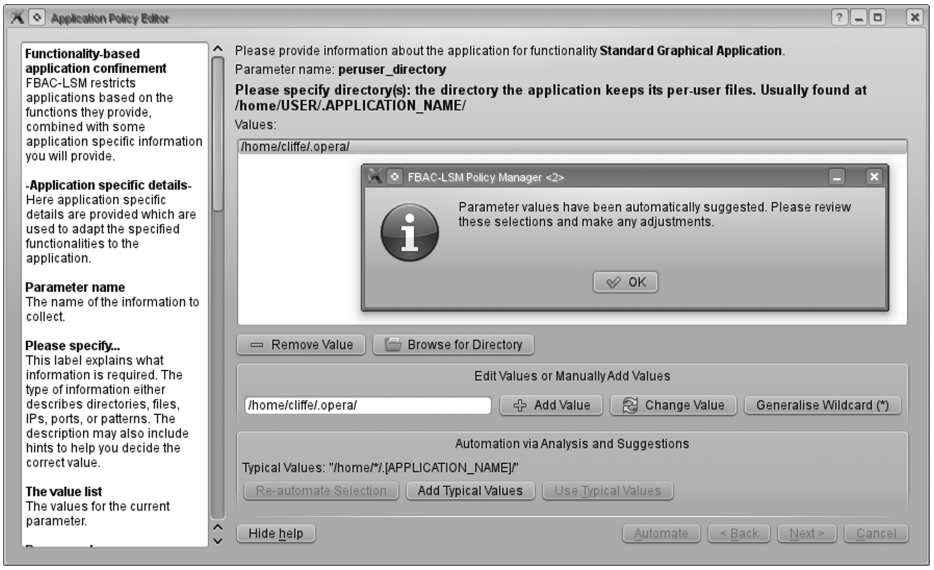

Fig. 5. FBAC-LSM Policy Manager parameter argument specification.

keyboard. Details such as the location for authorised Opera file downloads were specified by the user.

3.5. Trojan Horse Simulation

For the purposes of the experiment, the KSirtet Tetris game was modified to simulate a Trojan horse. The modified program attempts to access many resources that a game should not need to and, as a malicious program, would represent a serious security problem. Such a program should require very limited privileges to run. However, without the use of an application confinement security mechanism such as those studied, such a program would possess all of a user's privileges. In addition to attacks that could be performed on a correctly configured Linux system, the Trojan horse simulation also attempts to access resources normally protected by discretionary access controls (DAC). For the purposes of the experiment, the VMs were deliberately misconfigured to allow this access. This type of configuration could be caused by user error, malicious actions from other programs, or be standard on single user Linux systems such as embedded devices. This aspect of the experimental design was intended to illustrate the potential for application-oriented access controls to provide defence-in-depth as additional layers of security.

The list of malicious activity attempted by the Trojan simulation was developed to reflect a range of risks and malicious behaviour that could compromise a Linux system. These risks include privacy risks and system-wide or user-level compromise. Appendices B and C describe the activities the Trojan simulation attempted.

3.6. Measuring Perceived Usability

Perceived system usability was measured using [Brooke 1996] System Usability Scale (SUS), which is a widely employed and extensively verified tool [Bangor et al. 2008; Lewis and Sauro 2009]. The SUS is a 10 item Likert scale, with even-numbered items worded negatively and odd-numbered items worded positively. The scale yields a single

score ranging from 0 to 100, representing an assessment of the system's usability. After using each security system, participants completed the SUS questionnaire.

After they had used all three systems, participants completed a final questionnaire ranking the three systems in terms of how easy they were to use, how easy they were to understand and how likely they would be to use them again. Once participants had completed the exercises and questionnaires, they were taken into a separate room and the security systems were discussed in a debriefing session where additional opinions were collected.

3.7. Measuring Policy Quality and Task Success

The VMs from each participant were stored for subsequent data collection. Data collection involved testing the ability of the confined programs to run and the ability of the programs to access the security sensitive resources the Trojan horse attempted to use. Each program was tested manually to assess whether it could run and that all required features operated correctly. Whether or not policies were successfully created was also assessed manually. To determine the threats the programs still posed, the Opera and KSirtet executables on each VM were replaced with an assessment program that attempted to access the same resources as the Trojan simulation. While retaining the policies created by participants, the replacement scoring program output the result of each access attempt. These results showed whether the confined program was able to potentially act maliciously. Additionally, time-on-task was recorded. All data was stored in a database for statistical analysis.

3.8. Pilot Study

The study was carefully designed to eliminate potential biasing factors. For example, the order in which participants used the systems was randomised, the names of the systems studied were not advertised during participant recruitment, and participants were not allowed to search the Internet for information about the systems during the study. FBAC-LSM had not been released prior to conducting the experiment, and those already aware of it did not participate.

A pilot study was conducted with four participants, who had a range of expertise levels. The primary concern of the pilot study was to detect the potential for participant bias. The pilot group completed an additional pilot questionnaire regarding whether they noticed anything potentially biasing in the videos, presentations, and handouts supplied during the experiment. They were also interviewed during the debriefing. The pilot group reported no biasing factors. The pilot study did raise awareness of a number of technical problems, such as networking problems, missing codecs for video playback, and missing sound in one of the videos. All these issues were resolved prior to conducting the main study.

4. RESULTS

One way repeated measures analysis of variance (ANOVA), the nonparametric Friedman test, repeated measures logistic regression, and descriptive statistics were utilised to compare the within-subjects effects of the three security systems, SELinux, AppArmor, and FBAC-LSM.

4.1. Participant Demographics

Participants' ages ranged from 18 to 67 (mean: 31.1 std. deviation: 13.0). Five participants were female. Table I summarizes the self-reported expertise of the participants,

Table I. Participant Self-Assessment

Expertise	Mean	Std dev	Min	Max
Skill with computers	5.82	0.90	3	7
Knowledge of computer security	4.47	1.20	2	7
Frequency of Linux use	4.24	2.32	1	7
Knowledge of Linux	3.53	1.89	1	7
Knowledge of FHS	3.61	1.97	1	7

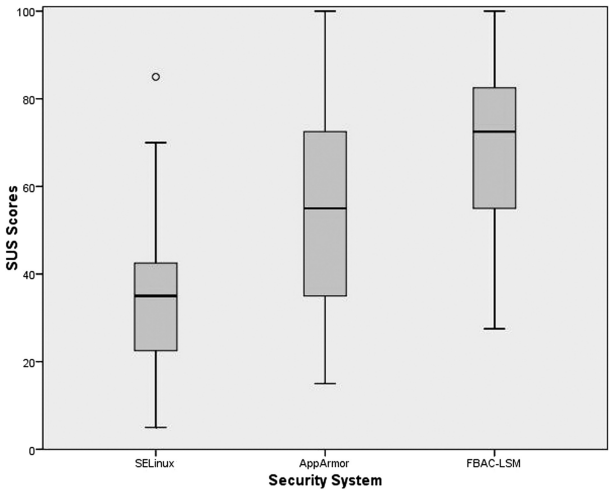

Fig. 6. Box plot comparing SELinux, AppArmor and FBAC-LSM System Usability Scale scores.

collected using the pre-experiment questionnaire. In each case responses could range from one to seven, with higher values representing higher levels of experience or expertise. As shown in the table, the majority of participants evaluated themselves as possessing above average computer skill, with a relatively wide range of responses for the computer security and Linux questions. As recommended for usability studies, the experiment included some *least competent users* (LCU); that is, users representing the minimum level of expertise that would be expected to utilise the systems [Rubin and Chisnell 2004]. The study also included some Linux and computer security experts, who work within industry managing Linux systems and providing IT security services.

4.2. Preference Evaluation—System Usability Scale

A one-way within subjects ANOVA was conducted to compare the effect of security system on SUS. The assumptions of the test were met. There was a significant effect of security system, Wilks' Lambda = 0.38, $F(2,35) = 28.99$, $p < .001$, $n = 37$. The effect size was .624. Post hoc analysis using the Tukey LSD test showed significant contrasts between each pairwise comparison. That is, all three systems were significantly different from each other in terms of perceived usability.

As illustrated in Figure 6, on average FBAC-LSM received the highest SUS scores ($M = 70.21$, $SD = 18.34$), followed by AppArmor ($M = 54.93$, $SD = 24.18$), and SELinux with the lowest scores ($M = 34.58$, $SD = 18.04$).

Table II. Mean Ranks

Security system	Mean rank for ease of use	Mean rank for ease of understanding	Mean rank for likeliness of reuse
SELinux	2.67	2.64	2.58
AppArmor	1.85	1.90	1.92
FBAC-LSM	1.49	1.46	1.45

4.3. Preference Evaluation—Ranking

Table II shows the mean rank (one, first, to three, last) for each system in terms of: how easy they were to use, how easy they were to understand, and how likely participants would be to use them again. In each case, FBAC-LSM was, on average, ranked highest, followed by AppArmor, then SELinux. FBAC-LSM was also ranked first most frequently, and SELinux was ranked last most frequently. A post hoc analysis confirmed that the order in which the systems were used did not influence these rankings.

The results of the SUS score differences and ranks showed that FBAC-LSM exhibited higher perceived usability than AppArmor and SELinux.

4.4. Performance Evaluation—Creation of Policies

In this section the extent to which participants were able to create policies to confine the programs is reported. The quality of the policies created is described in subsequent sections. The following results (including percentages) do not include participant records with "missing values" due to the following.

— Seven SELinux virtual machines that froze at start-up with a SELinux AVC message. This problem appeared to be the result of SELinux rules, created by participants, that inadvertently no longer allowed the VMs to start.
— Two VMs (one SELinux, one AppArmor) that did not start due to kernel panics. The exact cause of this was not clear.
— Existing SELinux rules for KSirtet which conflicted with the creation of a new policy to confine KSirtet. Due to the abstruseness of the command-line output that reports this conflict, the problem was not detected during initial environment setup, the pilot study, or by the majority of participants. After the problem was detected, it was remedied for the subsequent participants. Therefore, 22 participants could not create a policy to confine KSirtet due to this conflict. Ironically, the default policy did not provide any protection against the threats tested during the study.

As policies were either successfully created or not, repeated measures logistic regression was conducted to compare the effect of security system on the creation of enforced policies for Opera. There was a significant effect of security system, Wald Chi-Square (2, N = 105) = 31.30, $p < .001$. All three systems were significantly different from each other in terms of how many policies for Opera were successfully created and enforced. As shown in the Enforced Policy column of Table III, 90% of participants created enforced policies for Opera using FBAC-LSM, 66% using AppArmor, and only 23% using SELinux.

Repeated measures logistic regression was also conducted to compare the effect of the security system on the creation of enforced policies for KSirtet using each of the security systems. Again, there was a significant effect of security system, Wald Chi-Square (2, N = 86) = 10.03, p = .007. As with Opera policies, all three systems were significantly different from each other: 82% of participants created enforced policies for KSirtet using FBAC-LSM, 71% using AppArmor, and only 22% using SELinux.

Table III. Policy Creation Rates for Opera and KSirtet Using SELinux, AppArmor, and FBAC-LSM

Security System	Application	No Policy (unconfined)	Unenforced Policy (unconfined)	Enforced Policy
SELinux	Opera (n = 31)	21 (68%)	3 (10%)	7 (23%)
	KSirtet (n = 9)	6 (67%)	1 (11%)	2 (22%)
AppArmor	Opera (n = 38)	3 (8%)	10 (26%)	25 (66%)
	KSirtet (n = 38)	7 (18%)	4 (11%)	27 (71%)
FBAC-LSM	Opera (n = 39)	4 (10%)	0 (0%)	35 (90%)
	KSirtet (n = 39)	7 (18%)	0 (0%)	32 (82%)

Table IV. Extent to Which Opera Can Function while Confined by SELinux, AppArmor, and FBAC-LSM

Access to feature (Opera)	SELinux (n = 7)	AppArmor (n = 25)	FBAC-LSM (n = 35)
Program runs	3 (43%)	14 (56%)	34 (97%)
Can access web pages via HTTP	3 (43%)	14 (56%)	34 (97%)
Can access web pages via HTTPS	3 (43%)	13 (52%)	34 (97%)
Can access IRC	1 (14%)	8 (32%)	19 (54%)

Table V. Extent to Which KSirtet Can Function while Confined by SELinux, AppArmor, and FBAC-LSM

Access to feature (KSirtet)	SELinux (n = 2)	AppArmor (n = 27)	FBAC-LSM (n = 32)
Program runs	2 (100%)	19 (70%)	32 (100%)
Can play game	2 (100%)	19 (70%)	32 (100%)
Can store high scores	2 (100%)	19 (70%)	10 (31%)

In both tasks it was notable that participants were most likely to successfully create policies to confine applications using FBAC-LSM. In addition to participants who were not successful at creating a policy at all (refer to the No Policy column in Table III), SELinux and AppArmor resulted in a number of policies that were left in an unenforced state. The terminology for an unenforced policy differs for each system; SELinux: permissive domain, AppArmor: complaining mode, FBAC-LSM: complaining or disabled modes. The result of an unenforced policy is that the application is not confined, despite the fact that a policy exists. Although participants had seen videos describing the way policy enforcement works for each system, it is possible that many participants were unaware these policies were in an unenforced state. As illustrated in the Unenforced Policy column, a number of SELinux and AppArmor policies for both Opera and KSirtet were not in an enforced state. In contrast, 100% of the policies created using FBAC-LSM were in an enforced state.

4.5. Performance Evaluation—Confined Applications Can Run

As shown in Table IV and Table V, the extent to which the confined programs can actually operate is affected by the security system. Using FBAC-LSM 97% of the policies created for Opera allowed the program to run, compared to 56% and 43% for AppArmor and SELinux respectively. These results demonstrate that, compared to AppArmor and SELinux, FBAC-LSM is more successful at not interfering with programs performing their legitimate functions. This is an important practical measure of application confinement success, as a security mechanism or policy that disrupts the operation of a program is likely to be promptly disabled.

Table V shows that 100% of FBAC-LSM policies for KSirtet allowed the program to run, as opposed to 70% of AppArmor policies. However, only 31% of FBAC-LSM

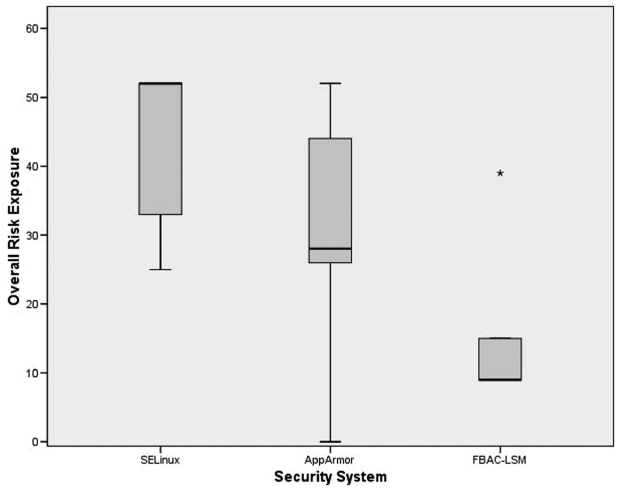

Fig. 7. Box plot comparing SELinux, AppArmor and FBAC-LSM overall risk exposure.

policies allowed the program to record high scores in the user's home directory. This did not affect game-play and was due to the fact that the score file did not necessarily exist when the policy was created. Of the nine participants not affected by the policy conflict described in Section 4.4, only two participants created policies for KSirtet using SELinux, both of which allowed the game to run.

4.6. Performance Evaluation—Risk Exposure

4.6.1. Overall Risk Exposure. Risk exposure was measured using a simple score, one *demerit* point for each security sensitive resource that was accessible for each of the two programs. This measure was designed to give a clear indication of the effect of the security system on the exposure to threats by simply recording the number of realistic threats the systems remained exposed to, rather than attempting to subjectively weight each threat. The nonparametric Friedman test was conducted to compare the effect of security system on the overall risk exposure. This test was used as an alternative to one-way within subjects ANOVA to ensure that violations of the assumptions of ANOVA did not impact on the interpretation of the results. Analysis included the data from participants who participated after the SELinux KSirtet default policy conflict was detected and resolved. There was a significant effect of security system, $\chi^2(2) = 36.32$, $p < 0.001$. Post hoc analysis was conducted using the Wilcoxon Signed Ranks test with a Bonferroni correction applied. This showed that FBAC-LSM (M = 14.3, SD = 9.7) had a significantly lower risk exposure than both AppArmor (M = 30.3, SD = 17.0) and SELinux (M = 43.0, SD = 12.0). AppArmor was also found to have a significantly lower risk exposure than SELinux. A post hoc analysis confirmed that the order in which the systems were used did not influence risk exposure. Appendix B gives further detail regarding the types of access permitted by each security system.

These results allow comparison between the level of protection each system provided in terms of user success at creating and enforcing correctly configured policies using each of the systems. As illustrated in Figure 7, SELinux was the least successful at reducing risk exposure. AppArmor had the highest degree of variation, resulting in a broad range of risk exposure values, from policies that did not allow anything to

policies that gave unrestricted access. Overall AppArmor averaged second most successful at reducing risk exposure. AppArmor's average score still indicates its policies exposed the user to high degrees of risk by allowing the programs undue access to resources. FBAC-LSM was both the most consistent, and provided the greatest protection. These results demonstrated that, compared to the other systems, FBAC-LSM resulted in the lowest overall risk exposure.

4.6.2. Opera Risk Exposure when Policies Exist. The nonparametric Friedman test was also used to compare the effect of security system on the Opera risk exposure. Risk exposure was measured using a simple score, one demerit point for each security sensitive resource that was accessible to Opera. Analyzing the data from participants who created an enforced policy for Opera using all three systems (n = 5) showed no significant effect of security system, $\chi^2(2) = 5.06$, p = 0.080. This was repeated analyzing the participants who created an enforced policy for Opera using AppArmor and FBAC-LSM (n = 23) to compare the effect of security system on the Opera risk exposure using these two systems. An effect was detected, $\chi^2(1) = 7$, p = 0.008. This result indicates that, when policies for Opera *were* successfully created and enforced, AppArmor policies were slightly more restrictive (M = 4.65, SD = 3.19) than those created using FBAC-LSM (M = 7.83, SD = 1.72). Both of these scores represent a significant reduction in exposure to risk, although FBAC-LSM authorized additional network access. However, only 57% of those AppArmor profiles actually allowed Opera to function, compared to 96% of the FBAC-LSM policies, which allowed Opera to run while reducing exposure to risks. Also, in practice there would be a difference in risk exposure between the three mechanisms due to FBAC-LSM's greater success at creating and enforcing policies. All three systems when successfully deployed to confine a nonmalicious application reduced the exposure to risk.

4.6.3. Trojan Horse Risk Exposure when Policies Exist. Due to the low number of participants who created an enforced policy for KSirtet using SELinux (n = 2), there was insufficient residual degrees of freedom to compare the effect of security system on the KSirtet risk exposure using all three systems. Instead, a Friedman test was conducted to compare the effect of security system on the KSirtet risk exposure using only AppArmor and FBAC-LSM. Risk exposure was measured in the same way as previously, one demerit point for each accessible security sensitive resource. Security system was found to have a significant effect, $\chi^2(1) = 5.26$, p = 0.022. Therefore in the case of malicious programs, the results showed that FBAC-LSM (M = 6.04, SD = 4.96) is likely to be superior in producing more secure confinement policies than AppArmor (M = 14.54, SD = 9.85).

Unlike in the case of Opera, the Trojan program KSirtet was attempting to behave maliciously. AppArmor policies are often built by the mechanism of observing program behavior and having users review the rules generated from this. Therefore, successful confinement relies on the user's ability to vet the actions of potentially misbehaving programs. FBAC-LSM on the other hand constructs policy based on the features the user wants the program to perform, and on the location of various application specific resources. These results indicate that users typically do not have the expertise necessary to vet the actions of programs as required by AppArmor. In contrast, the FBAC approach was found to be far more accessible by users, and thereby achieved greater levels of protection.

It is expected that, had more participants successfully created policies for KSirtet using SELinux, SELinux would have rated even worse than AppArmor. The userspace tools for SELinux automatically enumerate all the learned rules, and there is no GUI tool that assists users with the vetting process.

4.7. Performance Evaluation—Efficiency

To compare the effect of security system on the overall time-on-task, a Friedman test was conducted. Time-on-task was defined as the time spent using each security system, and was measured in minutes based on the start and end times as recorded by the batch scripts. There was a statistically significant effect of security system, $\chi^2(2) = 14.45$, $p = 0.001$. Post hoc analysis was conducted using the Wilcoxon Signed Ranks test with a Bonferroni correction applied. This showed there was a significant difference in the time-on-task for AppArmor (M = 29.3, SD = 14.3), which was significantly less than SELinux (M = 45.18, SD = 19.0), and also significantly less than FBAC-LSM (M = 40.1, SD = 15.8). No significant difference in time-on-task was found between SELinux and FBAC-LSM. This indicates that participants completed the tasks faster using AppArmor than the other two systems. During the study it was observed that, when faced with repetitive AppArmor dialogs with rules to vet, some participants simply clicked "Allow" as fast as possible for each rule, and this appears to explain the shorter completion time using AppArmor.

Two scales for each system were used to gauge the perceived time efficiency of the three systems. These results are in contrast to the actual time-on-tasks. On average participants rated FBAC-LSM as the most time efficient. The majority of participants indicated that they felt they had enough time to use AppArmor and FBAC-LSM. Participants indicated a more varied response (SD = 2.22) regarding whether they had enough time to use SELinux; on average response was relatively neutral (M = 3.90). Study participants occasionally encountered reliability issues with the new FBAC-LSM implementation. The moderator stated that any crashes were not the participant's fault, and asked them to restart the VM. Any saved policies were not lost. These problems did not appear to have a notable impact on user perception of the system.

5. DISCUSSION

The results of the usability study showed that, compared to AppArmor and SELinux, FBAC-LSM was rated and ranked as easiest to use, had significantly higher rates of policy creation and enforcement, had more policies that allowed the programs being confined to run and function correctly, and most reduced the risk to the user. While FBAC-LSM policies were no more protective than the other two mechanisms when confining the Opera Web browser to protect against vulnerabilities, the study demonstrated that users were more likely to successfully construct and enforce these policies, reducing the risk overall. Furthermore, FBAC-LSM policies provided far more protection against potential malware then either of the other two mechanisms. Users reported preferring FBAC-LSM from a time efficiency point of view. However, AppArmor confinement procedures were sometimes recorded as taking less time, apparently due to users rapidly clicking through dialogues without necessarily considering the message that was presented to them. These results will now be discussed in detail.

5.1. Perceived Usability

Based on their research, Bangor et al. [2008] have published advice for interpreting SUS results, which suggests that "products with scores less than 50 should be cause for significant concern and are judged to be unacceptable." SELinux scored 34.58, which suggests that SELinux suffers from major usability deficiencies and is in need of significant usability improvements. Based on observations made during the experiment, the primary factors limiting SELinux usability seem to be due to the complexity of the model used and the lack of an intuitive graphical interface for much of the task of policy specification. While SELinux is arguably intended to be configured by those with significant expertise, its widespread deployment on Linux systems, including personal

workstations, means these usability results cannot be ignored. If the scheme itself is deemed to be fundamentally unsuited to being managed by end users, an alternative, user-friendly approach may be better suited.

AppArmor scored 54.93. According to Bangor et al., "products with scores less than 70 should be considered candidates for increased scrutiny and continued improvement and should be judged to be marginal at best". This suggests that, while AppArmor is significantly preferred over SELinux in terms of usability, improvements are required in order for it to be considered "acceptable." Based on observations during the study, the primary factor limiting AppArmor usability seems to be due to the inability of typical users to make informed decisions about the files to which applications require access. This result supports the notion that end users cannot be expected to have the expertise to make low-level decisions about what access privileges their programs require.

Bangor et al. suggest that "products which are at least passable have scores above 70". Based on these interpretation guidelines, FBAC-LSM, with a score of 70.21, is the only system studied that can be classified as "acceptable." While FBAC-LSM was significantly preferred over AppArmor and SELinux, there is clearly still room for improvement. FBAC-LSM is a much newer, less mature implementation than the other two systems studied. Continuing development on FBAC-LSM will incorporate feedback from the experiment to further improve its usability. Nonetheless, these results indicate that the FBAC model in general is well suited to providing application-oriented controls capable of being managed by non-expert users.

5.2. Rate of Policy Enforcement

As described in Section 4.4, FBAC-LSM had the highest success rate in creating enforced policies, followed by AppArmor and then SELinux. The variety in rate of policy enforcement can be attributed to a number of factors. The successful creation of a policy is likely to be affected by the difficulty of using the system and the difficulty of understanding how to create policies. FBAC-LSM again ranked highest in both respects, followed by AppArmor and then SELinux.

One factor that affects policy enforcement is the behavior of the policy tools for each of the systems and the state in which they leave newly created policies. The SELinux Policy Generator tool left newly created skeleton policies in permissive mode so that users could further develop these policies before manually editing the appropriate file to set the domain to be enforced, and then run a shell script to recompile and load the policy. Some participants forgot to manually set the policy to be enforced, meaning that these policies were not enforced. Several participants reported that they were not comfortable with the requirement to run console commands and simply completed what they could using the graphical tools available. The result was that the policies were not created and assigned to the executables. A more complete graphical tool that steps users through the whole process would have improved success using SELinux.

AppArmor's policy tools left policies either enforced or in complaining mode (which is not enforced) depending on user choices. For example, exiting from the Add Profile Wizard resulted in the policy left in complaining mode. This resulted in some policies being left unenforced, most likely unbeknownst to the user. Making this behavior more obvious to users may have helped decrease the number of programs left unconfined.

FBAC-LSM created policies that were enforced by default unless the user specifically toggled the policy activation. This, combined with the wizard for creating policies that steps users through the process and the fact that the main window includes

information about policy enforcement, appear to be factors contributing to the high success rate in creating enforced policies.

5.3. Continued Program Operation

Of the cases where policies were in effect, FBAC-LSM allowed the highest percentage of programs to continue to run and function. SELinux often did not log all of the access attempts required to create a policy that allowed Opera to start. As described on the SELinux handout provided to participants, the solution was to set SELinux to log all events, which led to the display of a large number of unrelated messages which SELinux is normally configured to ignore. While using AppArmor, a number of participants created policies that denied access to resources that were required in order for the programs to continue to function correctly. This again demonstrates that users frequently do not have the expertise required to vet the actions of programs. Because FBAC-LSM builds policy using reusable, easily understandable policy abstractions, the required access rules were assigned when building the application policies. The study showed that this approach clearly led to higher program-feature-access success rates compared with requiring users to vet the actions of programs.

As noted in Section 4.5, some FBAC-LSM policies were missing rules for a file that did not exist when policy was created. To clarify, this does not imply that all files the application will require access to need to preexist, just that the parameter automation in this case failed to predict the existence of the configuration file. The planned addition of a notification feature to FBAC-LSM would address this issue, since this filename can already be identified as belonging to the application and a simple interactive prompt could add to the a priori policy.

5.4. Confinement of Trustworthy Programs

Considering policies that were successfully created and enforced, all three systems performed similarly where the program being confined was acting benevolently during policy construction. This result reflects the fact that, when the program is behaving nonmaliciously, adequate protection will be obtained by simply allowing all the rules suggested by the learning tools. In this scenario if, at a later point in time, the program is compromised or replaced by a malicious version, the user will be protected.

While in this scenario, where enforced policies had been created, FBAC-LSM was found to have no significant restriction advantages compared to AppArmor and SELinux, further research is required. SELinux makes decisions based on types assigned to files, which in the case of the user's home directory are typically very coarsely grained. It is therefore anticipated that, in most cases, user applications will be granted access to excessive rights; that is, access to almost anything in the user's home directory. This could be improved by creating more finely-grained SELinux types. Therefore a typical SELinux installation is perhaps better suited to improving system-wide user-oriented access controls than to application-oriented access controls for programs run by users.

The AppArmor policies created provided slightly tighter controls than those created using FBAC-LSM. FBAC-LSM was more permissive due to the high level of network access the "Web Browser" functionality authorized in order to allow Active FTP, and a policy mistake that granted access to Firefox's configuration files. The access that was granted is illustrated in Appendix B. However, this FBAC-LSM policy abstraction could be improved (in one central location) to provide tighter controls for all the policies that were created using it. As noted in Section 4.6.2, a large number of those AppArmor profiles stopped Opera from running at all, which significantly reduces their usefulness; if the security mechanism stops a user's applications from working, the

user is more likely to simply disable the mechanism, and therefore the actual protection provided in practice may be lower. Also, this analysis only considered policies that were successfully created and enforced, and FBAC-LSM had the highest rate of enforced policies.

Finally, it is important to note that these results are only applicable where the user can be completely certain that an application can be trusted at the point of policy construction. In practice this cannot be safely assumed in many cases, for example when downloading software. Therefore the ability of users to successfully confine potentially malicious programs is arguably more important.

5.5. Confinement of Potentially Malicious Programs

As established in Section 4.6.3, FBAC-LSM was significantly more successful at protecting resources from the Trojan horse simulation than AppArmor. Not enough participants successfully created policies using SELinux to confine KSirtet to make reliable comparisons between all three systems.

As described in Section 3.4.1, when using AppArmor a user will typically run the program to be confined, and is then stepped through the process of vetting the learnt rules to allow the program to perform the same actions in the future. Using this approach, protection from malicious programs is dependent upon users successfully vetting these rules. The results of the study demonstrated that typical users, and even security professionals and Linux system administrators, generally do not necessarily have the required expertise to successfully vet these actions.

A number of noteworthy examples of the tendency of users to allow malicious activity using AppArmor illustrate the extent of this problem.

— 74% of participants who managed to create an enforced policy using AppArmor allowed KSirtet to access the shadow file.[6] The shadow file is a very high profile target that contains the hashed user passwords for the system. With this file a malicious program could, for example, attempt dictionary attacks on passwords;
— 71% granted write access to the hosts file, which could allow the program to discretely redirect network traffic and perform man-in-the-middle attacks;
— 68% granted write access to the exports file used to configure network shares. This could be used to covertly share the contents of files to remote hosts;
— 58% granted access to private information in the users Firefox directory, potentially including saved web passwords such as those for Internet banking;
— 68% granted unrestricted network access, allowing the Trojan unlimited scope for sending information to, or receiving commands from, the network;
— 47% allowed the program to insert itself into KDE startup, which allows a malicious program to gain a persistent presence on the system.

Any single one of these potential breaches could have very serious security consequences. Several participants reported relying heavily on the severity level suggestions provided by AppArmor, as they did not have the expertise to vet rules based on resource names and access types alone. However, many participants reported that the severity level metric was ambiguous and often absent. Others seemed to not pay any attention to the specifics of the rules at all, simply clicking “Allow” to almost every rule. Participants who incorrectly did not notice any suspicious behavior included ICT PhD candidates, security professionals, and Linux system administrators. If experienced users of this caliber cannot use AppArmor to successfully confine a malicious

[6]As discussed in Section 3.5, user-oriented access controls were configured to allow this access in order to assess the protection provided by the application-oriented controls studied.

program, it is highly unlikely that typical end users could. Therefore, the results of this study strongly indicate that ordinary users cannot reliably employ learning mode application-oriented access controls that rely on users vetting generated rules. This result can be generalized to other systems such as Systrace, which in addition requires knowledge of system calls (a complex interface considered not suited to security mediation [Garfinkel 2003]) in order to vet the rules.

Rules for SELinux modules are typically generated in a similar fashion based on program behavior. Rules are described in terms of access to types rather than specific files. SELinux does not currently have a graphical tool to step users through the vetting process, which means that vetting is a manual process of editing complex rules that were generally not considered easy to understand by participants. As previously mentioned, some participants were uncomfortable with the command-line based nature of the SELinux tools used to create policy. It is believed that most participants did not edit or remove any lines of the rules generated by audit2allow, which created rules based on the previous actions of the program. Since this method of generating policy is also based on a learning mode and the rules are not easily vetted by users, it is likely that policies created by typical users using SELinux would also provide less protection than FBAC-LSM. Also, as demonstrated by the policy conflict encountered during this study, the ineffective default configuration of SELinux can inhibit users from specifying policies to enforce their own security goals.

Alternative tool sets exist for managing SELinux policy. However, the tools used were the ones that are standard with Fedora (the Linux distribution with the most complete SELinux support) and, unlike many SELinux tools, these provided graphical tools to step users through some of the process. It is believed that further development could yield more usable tools. However, due to the complex nature of the way rules are modeled, and the need for vetting the output of learning tools used to create rules for SELinux, it is believed that SELinux faces serious obstacles to improving usability. The results of this study support the argument that SELinux and its current set of configuration tools is not suited to end user configuration to protect themselves against misbehaving software.

FBAC-LSM substantially lowered the risk exposure compared to AppArmor. In each case, as previously described, FBAC-LSM lowers the risk. For example, 6% of participants granted access to the shadow, hosts, exports, and private user files (compared to 74%, 71%, 68%, and 58% respectively). Furthermore, only 50% allowed outgoing network connections, 12% incoming connections (compared to 68% full network access with AppArmor), and 19% allowed insertion into KDE startup (compared to 47%). Therefore, FBAC-LSM clearly provided the best protection of the systems studied. These results are attributable to the design of the FBAC model, which abstracts away the low-level privileges that are required in order for applications to provide various features. The techniques for automating the discovery of parameter arguments also helped the security decisions made by users, allowing them to focus on higher level (functionality-based) security goals.

FBAC-LSM was also able to step participants through the process of creating policies without requiring users to execute the program being confined. AppArmor and SELinux both generally rely on the execution of programs in order to generate rules. It is possible to add new rules while enforcing the policy being developed. However, using SELinux or AppArmor this can take an extremely large number of iterations to create a working policy, as incomplete policies will often cause the program to terminate. Therefore, the method demonstrated in the preparation videos was to generate the rules while the policy was not enforced. This is the approach that is typically used; however, it leaves the program unconfined while rules are generated. Therefore, if the program is malicious, it could compromise the system while a policy is being

developed. The risk of using this approach was stressed, and participants were encouraged to take the more restrictive approach, although few did. Furthermore, the few more experienced users who were observed attempting to take the safer approach were still ultimately unsuccessful at creating policies which restricted malicious activity. As discussed in the introduction, there are many reasons not to trust software, and therefore it is recommended to avoid these learning modes. FBAC-LSM provides an alternative approach to policy creation that avoids these risks, since policies are specified without executing the programs being confined. Note that the potential risk exposure during learning was *not* considered when assessing risk levels of the policies created during the study.

5.6. Overall Protection

Factors contributing to the overall risk exposure score include the success rate of creating policies and ensuring policies are enforced (as discussed in Section 4.4), the extent to which the legitimately behaving program (Opera) was restricted, and the extent to which the malicious program (the KSirtet Trojan) was restricted. The overall risk exposure score reflects the practical security-benefit of each system by measuring the extent that users are protected from misbehaving programs accessing sensitive resources.

As described in Section 4.6.1, FBAC-LSM had significantly lower overall risk exposures. This result can be attributed to the fact that the highest number of users were successful in creating policies, ensuring policies were enforced, and confining the Trojan simulation using FBAC-LSM.

5.7. Limitations of the Study

The primary limitation of this study was the SELinux KSirtet policy conflict, which excluded some results from a large number of participants from analysis. As discussed previously, existing SELinux rules for KSirtet prevented new rules from being enforced. Due to the abstruseness of the output from the SELinux command-line tools, participants were unaware that their new policies were not in effect. When this was detected, the VMWare image was modified to remove this conflict for the nine subsequent participants. Data from the previous participants regarding KSirtet was not included in analysis. As noted previously, the conflicting SELinux policy (included in the games policy module) did not provide any protection against the malicious activities attempted by the Trojan horse simulation.

The FBAC-LSM implementation is relatively new compared with the other two schemes and occasionally caused crashes, requiring the VM to be restarted. While no policies that had been created were lost, this bug may have impacted the time-on-task measurement, and also potentially negatively affected participants' perceptions of the system. However, the results suggest that any such effect was not a significant one.

5.8. Conclusions

Developing usable security software has long been acknowledged as a challenge that has not been given sufficient attention [Zurko and Simon 1996]. Application-oriented controls have the potential to improve security but pose new usability problems that, until recently, had not been considered [Dewitt and Kuljis 2006]. In particular, achieving sufficiently finely grained protection without exposing end users to complex low-level details of the application's operation has remained problematic.

This paper presents the results of a study into the usability and security outcomes of three different approaches to application confinement. SELinux provides a mature and

technically robust framework. However, due to its complex model and poor usability of its existing tools, the study showed it to have a low success rate. SELinux tools need significant work in order to be usable by end users. In the mean time SELinux seems to be better suited to enforcing system-wide policies constructed and managed by experts.

In contrast, AppArmor is relatively easy to learn. However, the security decisions made during policy construction still require expertise beyond that of most users. In particular, policies generated using the system's learning mode cannot be relied upon unless the user manually verifies these are correct. Not only is this process time-consuming, it often requires expertise that end users are unlikely to have. This was clearly shown in the study by the relatively low success of AppArmor in confining the KSirtet Trojan. This study therefore demonstrates that policy generating learning modes are not a viable method for end users to successfully confine their applications.

However, the results showed end user construction and management of protective application confinement policies was indeed feasible. The study demonstrated that the FBAC model can be effectively employed by end users with limited technical expertise to protect themselves against both vulnerabilities in otherwise trustworthy software and potentially malicious programs. These results also clearly indicated the large practical impact usability has on security. The hierarchical nature of the policy abstractions used by FBAC avoids the need for end users to deal with low-level platform complexities, while still achieving a very high level of confinement. In comparative terms, FBAC-LSM was markedly superior to both AppArmor and SELinux in creating policies, ensuring policies were enforced, allowing programs to function correctly while confined and the protection achieved from malicious code. The FBAC LSM implementation was also preferred by study participants for usability and time efficiency.

Therefore, although the FBAC model is a relatively new approach to application-oriented access control, the results of this study are highly encouraging in pointing the way forward to the use of functionality-based schemes that empower nonexpert end users to confine their applications and protect themselves from a variety of prevalent security threats.

APPENDIXES

APPENDIX A – TASK SCENARIOS

Scenario 1: Opera
You use the Opera Web browser for chatting online (using IRC), for downloading files, and to browse Web pages. You are concerned that since it often interacts with external servers you should confine it in case there are any exploitable vulnerabilities in opera.
Command to run: **opera**
To find the path used, open a console and type: **which opera**
Tips:

> You may want to create a new directory in your home directory for downloads
> There is a webpage at: www.murdoch.edu.au
> There is an IRC server at: 134.115.65.115 with a chat room named #chat

Scenario 2: KSirtet
You have just downloaded and installed a game from the Internet. It is similar to the classic game Tetris. Since this game was downloaded from an unauthenticated website you decide to confine the program.
Command to run: **ksirtet**
To find the path used, open a console and type: **which ksirtet**

Table VI. Mean Authorization Granted to Opera to Access Categories of Resources

Access to category of resource (Opera)	SELinux (n = 31)	AppArmor (n = 38)	FBAC-LSM (n = 39)
System misconfiguration information leak (MAX: 3)	2.45	1.03	0.31
System misconfiguration compromise (MAX: 5)	3.87	1.71	0.51
System information leak (MAX: 2)	1.81	1.13	0.23
Local privacy (MAX: 1)	0.77	0.37	0.97
Local compromise (MAX: 3)	2.61	1.03	0.92
Temporary file creation (MAX: 1)	0.81	0.61	0.97
Network ingress (MAX: 2)	1.61	1.58	2.00
Network egress (MAX: 3)	2.52	2.37	3.00
Execute commands using bash (MAX: 3)	2.32	1.03	0.51

Table VII. Mean Authorization Granted to KSirtet to Access Categories of Resources

Access to category of resource (KSirtet)	SELinux (n = 9)	AppArmor (n = 38)	FBAC-LSM (n = 39)
System misconfiguration information leak (MAX: 3)	3.00	2.21	0.69
System misconfiguration compromise (MAX: 5)	4.89	3.47	1.03
System information leak (MAX: 2)	2.00	1.53	0.41
Local privacy (MAX: 4)	3.89	2.32	0.82
Local compromise (MAX: 3)	2.67	1.37	0.64
Temporary file creation (MAX: 1)	0.89	0.68	0.97
Network ingress (MAX: 2)	1.78	1.37	0.56
Network egress (MAX: 6)	5.33	4.11	3.54
Execute commands using bash (MAX: 3)	2.67	0.97	0.59

APPENDIX B – RISK EXPOSURE

Table VI illustrates the mean number of security sensitive resources that were left in a state where Opera is authorised to access various security sensitive resources. This information is organised in terms of categories of resource (that is, the type of security risk they present). This includes VMs that did not have enforced policies. This information shows the practical impact of each security system. For each security system the table shows the portion of participants' VMs that were left in a state which allowed the program to access the specified resource. Table VII shows the same type of information for KSirtet; again including VMs without enforced policies.

The FBAC-LSM functionalities deployed for the experiment contained a mistake in the Web_Browser functionality which granted undue access to Firefox's files (the "local privacy" row for Opera in Table VI). Also, the liberal access to network access (as demonstrated in the "network ingress" and "network egress" rows for Opera in Table VI) is due to the FTP_Client functionality which grants this access to allow Active FTP which requires extensive network access. Also, FBAC-LSM does not restrict access to files in /tmp (the "temporary file creation" rows in the two tables above). As demonstrated in Table VI and Table VII, in all other 13 categories FBAC-LSM provided the tightest restrictions.

APPENDIX C – EXPOSURE SCORING

Table VIII lists all the resources the scoring program assessed and illustrates the number of VMs that were left in a state where Opera was authorized to access various security sensitive resources. This included VMs which did not have enforced policies.

Table VIII. Authorization Granted to Opera to Access Resources That Pose Security Risks

Access to resource (Opera)	SELinux (n = 31)	AppArmor (n = 38)	FBAC-LSM (n = 39)
/etc/sysctl.conf r	28 (90%)	13 (34%)	4 (10%)
/etc/group r	28 (90%)	30 (79%)	5 (13%)
/etc/login.defs rw	24 (77%)	13 (34%)	4 (10%)
/etc/inittab rw	24 (77%)	13 (34%)	4 (10%)
$HOME/.mozilla/firefox/X.default/ formhistory.dat r	24 (77%)	14 (37%)	38 (97%)
/tmp/JsXr.c w	25 (81%)	23 (61%)	38 (97%)
$HOME/.kde/Autostart/ksirtet w	27 (87%)	13 (34%)	12 (31%)
$HOME/Desktop/ksirtet.desktop w	27 (87%)	13 (34%)	12 (31%)
$HOME/.rhosts w	27 (87%)	13 (34%)	12 (31%)
/etc/shadow r	24 (77%)	13 (34%)	4 (10%)
/etc/ssh/sshd_config r	28 (90%)	13 (34%)	4 (10%)
/etc/gshadow r	24 (77%)	13 (34%)	4 (10%)
/etc/exports rw	24 (77%)	13 (34%)	4 (10%)
/etc/hosts rw	24 (77%)	13 (34%)	4 (10%)
/etc/logrotate.conf rw	24 (77%)	13 (34%)	4 (10%)
TCP 22 gateway.murdoch.edu.au	25 (81%)	30 (79%)	39 (100%)
TCP 995 www.mail.murdoch.edu.au	25 (81%)	30 (79%)	39 (100%)
UDP 1050 murdoch.edu.au	28 (90%)	30 (79%)	39 (100%)
TCP 5000	25 (81%)	30 (79%)	39 (100%)
UDP 5000	25 (81%)	30 (79%)	39 (100%)
netcat -h 2>&1	24 (77%)	13 (34%)	4 (10%)
2>&1 echo \“main(){printf(\\\“hello\\n\\\”);}\” > “HOMEDIR” JsXr2.c	24 (77%)	13 (34%)	12 (31%)
gcc “HOMEDIR” JsXr2.c -o “HOMEDIR”JsXr 2>&1	24 (77%)	13 (34%)	4 (10%)

For each security system the table shows the number of participants' VMs which were left in a state that allowed the program to access the specified resource. For example, the 10th row shows how many of the policies which were created successfully protect the contents of the shadow file from Opera. Of the FBAC-LSM VMs 10% allow this inappropriate access, compared to 34% of the AppArmor VMs, and 77% of the SELinux VMs.

Table IX shows the same type of information for KSirtet; again including VMs without enforced policies. The table lists all the resources the Trojan horse simulation attempted to access. KSirtet is allowed to access the shadow file with 23% of the FBAC-LSM VMs, compared to 74% of the AppArmor VMs, and 100% of the SELinux VMs. The differences in overall risk exposure are analyzed in the following section.

Table IX. Authorization Granted to KSirtet to Access Resources That Pose Security Risks

Access to resource (KSirtet)	SELinux (n = 9)	AppArmor (n = 38)	FBAC-LSM (n = 39)
/etc/sysctl.conf r	9 (100%)	28 (74%)	8 (21%)
/etc/group r	9 (100%)	30 (79%)	8 (21%)
/etc/login.defs rw	9 (100%)	27 (71%)	7 (18%)
/etc/inittab rw	9 (100%)	24 (63%)	7 (18%)
$HOME/.opera/typed_history.xml r	9 (100%)	22 (58%)	8 (21%)
$HOME/.opera/global.dat r	9 (100%)	22 (58%)	8 (21%)
$HOME/.opera/wand.dat r	9 (100%)	22 (58%)	8 (21%)
$HOME/.mozilla/firefox/we6ybhyi.default/ formhistory.dat r	8 (89%)	22 (58%)	8 (21%)
/tmp/JsXr.c w	8 (89%)	26 (68%)	38 (97%)
$HOME/.kde/Autostart/ksirtet w	8 (89%)	18 (47%)	12 (31%)
$HOME/Desktop/ksirtet.desktop w	8 (89%)	18 (47%)	6 (15%)
$HOME/.rhosts w	8 (89%)	16 (42%)	7 (18%)
/etc/shadow r	9 (100%)	28 (74%)	9 (23%)
/etc/ssh/sshd_config r	9 (100%)	27 (71%)	9 (23%)
/etc/gshadow r	9 (100%)	29 (76%)	9 (23%)
/etc/exports rw	9 (100%)	26 (68%)	9 (23%)
/etc/hosts rw	8 (89%)	27 (71%)	9 (23%)
/etc/logrotate.conf rw	9 (100%)	28 (74%)	8 (21%)
TCP 80 murdoch.edu.au	8 (89%)	26 (68%)	23 (59%)
TCP 22 gateway.murdoch.edu.au	8 (89%)	26 (68%)	23 (59%)
TCP 995 www.mail.murdoch.edu.au	8 (89%)	26 (68%)	23 (59%)
TCP 443 murdoch.edu.au	8 (89%)	26 (68%)	23 (59%)
UDP 53 murdoch.edu.au	8 (89%)	26 (68%)	23 (59%)
UDP 1050 murdoch.edu.au	8 (89%)	26 (68%)	23 (59%)
TCP 5000	8 (89%)	26 (68%)	11 (28%)
UDP 5000	8 (89%)	26 (68%)	11 (28%)
netcat -h 2>&1	8 (89%)	10 (26%)	8 (21%)
2>&1 echo \“main(){printf(\\\“hello\\n\\\”);}\” > “HOMEDIR”JsXr2.c	8 (89%)	16 (42%)	7 (18%)
gcc “HOMEDIR”JsXr2.c -o “HOMEDIR”JsXr 2>&1	8 (89%)	11 (29%)	8 (21%)

REFERENCES

ATHEY, J., ASHWORTH, C., MAYER, F., AND MINER, D. 2007. Towards intuitive tools for managing SELinux: Hiding the details but retaining the power. In *Proceedings of the Security Enhanced Linux Symposium*.

BADGER, L., STERNE, D. F., SHERMAN, D. L., WALKER, K. M., AND HAGHIGHAT, S. A. 1995. Practical domain and type enforcement for UNIX. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society.

BANGOR, A., KORTUM, P. T., AND MILLER, J. T. 2008. An empirical evaluation of the system usability scale. *Int. J. Hum.-Comput. Interact. 24*, 6, 574–594.

BERMAN, A., BOURASSA, V., AND SELBERG, E. 1995. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the Winter USENIX Conference*. USENIX Association.

BROOKE, J. 1996. SUS: A quick and dirty usability scale. In *Usability Evaluation in Industry*, P. W. Jordan, B. Thomas, B. A. Weerdmeester, and I. L. McClelland Eds. Taylor & Francis, London, 189–194.

COWAN, C., BEATTIE, S., KROAH-HARTMAN, G., PU, C., WAGLE, P., AND GLIGOR, V. 2000. SubDomain: Parsimonious server security. In *Proceedings of the USENIX 14th Systems Administration Conference (LISA)*. USENIX Association.

CRANOR, L. AND GARFINKEL, S. 2005. *Security and Usability: Designing Secure Systems That People Can Use*. O'Reilly Media, Inc.

DEWITT, A. J. AND KULJIS, J. 2006. Aligning usability and security: A usability study of Polaris. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*. ACM Press.

GARFINKEL, T. 2003. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the 10th Network and Distributed System Security Symposium*.

GREENWALD, A. G. 1976. Within-subjects designs: To use or not to use. *Psych. Bull. 83*, 2, 314–320.

HITCHINGS, J. 1995. Deficiencies of the traditional approach to information security and the requirements for a new methodology. *Comput. Sec. 14*, 5, 377–383.

KAMP, P.-H. AND WATSON, R. 2000. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE'00)*.

KRSTI, I. AND GARFINKEL, S. L. 2007. Bitfrost: The one laptop per child security model. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*. ACM Press.

LEWIS, J. R. AND SAURO, J. 2009. The factor structure of the system usability scale. In *Proceedings of the International Conference on Human Centered Design*. Springer-Verlag.

LIANG, Z., SUN, W., VENKATAKRISHNAN, V. N., AND SEKAR, R. 2009. Alcatraz: An isolated environment for experimenting with untrusted software. *ACM Trans. Info. Syst. Sec. 12*, 3, 1–37.

MADNICK, S. E. AND DONOVAN, J. J. 1973. Application and analysis of the virtual machine approach to information security. In *Proceedings of the ACM Workshop on Virtual Computer Systems*. Harvard University, Cambridge, MA.

MILLER, M. S., TULLOH, B., AND SHAPIRO, J. S. 2004. The structure of authority: Why security is not a separable concern. In *Proceedings of the Multiparadigm Programming in Mozart/Oz (MOZ)*. Springer-Verlag.

NAKAMURA, Y., SAMESHIMA, Y., AND TABATA, T. 2009. SEEdit: SELinux security policy configuration system with higher level language. In *Proceedings of the 23rd Large Installation System Administration Conference (LISA)*. USENIX Association.

NOVELL APPARMOR AND SELINUX COMPARISON. http://www.novell.com/linux/security/apparmor/selinux_comparison.html.

OTT, A. 2002. The Role Compatibility Security Model.

POTTER, S., NIEH, J., AND SELSKY, M. 2007. Secure isolation of untrusted legacy applications. In *Proceedings of the 21st Large Installation System Administration Conference (LISA'07)*. USENIX Association.

PROVOS, N. 2002. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*. USENIX Association.

RUBIN, J. AND CHISNELL, D. 2004. How to plan, design and conduct effective tests. In *Handbook of Usability Testing*. Wiley India Pvt. Ltd., 129.

SALTZER, J. H. AND SCHROEDER, M. D. 1975. The protection of information in computer systems. *Proc. IEEE 63*, 9 1278–1308.

SCHREUDERS, Z. C. AND PAYNE, C. 2008a. Functionality-based application confinement: parameterised hierarchical application restrictions. In *Proceedings of the International Conference on Security and Cryptography (SECRYPT'08)*. INSTICC Press.

SCHREUDERS, Z. C. AND PAYNE, C. 2008b. Reusability of functionality-based application confinement policy abstractions. In *Proceedings of the 10th International Conference on Information and Communications Security (ICICS'08)*. Springer.

SCHREUDERS, Z. C., PAYNE, C., AND MCGILL, T. 2011. Techniques for automating policy specification for application-oriented access controls. In *Proceedings of the 6th International Conference on Availability, Reliability and Security (ARES'11)*. IEEE Computer Society.

STIEGLER, M., KARP, A. H., YEE, K. P., CLOSE, T., AND MILLER, M. S. 2006. Polaris: Virus-safe computing for Windows XP. *Comm. ACM 49*, 9, 83–88.

TUCKER, A. AND COMAY, D. 2004. Solaris zones: Operating system support for server consolidation. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium Works-in-Progress*.

VANCE, C. AND SALAMON, W. 2001. Implementing SELinux as a Linux security module. NAI Labs rep. #01-043, NSA.

WAGNER, D. A. 1999. Janus: An approach for confinement of untrusted applications. Tech. rep. CSD-99-1056, University of California, Berkeley.

ZANIN, G. AND MANCINI, L. V. 2004. Towards a formal model for security policies specification and validation in the SELinux system. In *Proceedings of the 9th ACM Symposium on Access Control Models and Technologies*. ACM Press.

ZURKO, M. E. AND SIMON, R. T. 1996. User-centered security, ACM Press.

Received September 2010; revised March 2011; accepted June 2011