Schreuders, Z.C., McGill, T. and Payne, C. (2012) ***The state of the art of application restrictions and sandboxes: A survey of application-oriented access controls and their shortfalls.*** **Computers & Security, 32 . pp. 219-241.**

The State of the Art of Application Restrictions and Sandboxes: A Survey of Application-oriented Access Controls and their Shortfalls

Abstract: Under most widely-used security mechanisms the programs users run possess more authority than is strictly necessary, with each process typically capable of utilising all of the user's privileges. Consequently such security mechanisms often fail to protect against contemporary threats, such as previously unknown ('zero-day') malware and software vulnerabilities, as processes can misuse a user's privileges to behave maliciously. Application restrictions and sandboxes can mitigate threats that traditional approaches to access control fail to prevent by limiting the authority granted to each process. This developing field has become an active area of research, and a variety of solutions have been proposed. However, despite the seriousness of the problem and the security advantages these schemes provide, practical obstacles have restricted their adoption.

This paper describes the motivation for application restrictions and sandboxes, presenting an in-depth review of the literature covering existing systems. This is the most comprehensive review of the field to date. The paper outlines the broad categories of existing application-oriented access control schemes, such as isolation and rule-based schemes, and discusses their limitations. Adoption of these schemes has arguably been impeded by workflow, policy complexity, and usability issues. The paper concludes with a discussion on areas for future work, and points a way forward within this developing field of research with recommendations for usability and abstraction to be considered to a further extent when designing application-oriented access controls.

Keywords: literature review, application-oriented access controls, sandboxing, virtualization, confinement.

1. Introduction

Traditional approaches to access control have typically been user-oriented, that is, they make access decisions based primarily on the identity of users. These methods generally fail to protect users from previously unknown software vulnerabilities and malware, because when executed this malicious code may utilise all of the user's privileges. Application restrictions and sandboxing can mitigate this threat by limiting the authority granted to processes based on the privileges or resource name-space they require in order to carry out their legitimate functions. The field of application-oriented access control, schemes which primarily base access decisions on the programs involved rather than on the identity of users, has therefore become an active area of research, and a number of approaches have been proposed. However, despite the security advantages they provide, adoption of these schemes has been limited. This paper aims to review the various approaches taken and identify their limitations.

This paper is structured as follows. We begin with a discussion of the limitations of the traditional security approaches, introducing the concept of user-oriented access controls and the inability of this method to sufficiently mitigate the threat posed by malware and vulnerabilities. Limitations of trust-based selective execution (such as traditional anti-malware software) and integrity-level schemes are also discussed. This is followed by a comprehensive overview and comparison of application-oriented access control schemes. Isolation-based schemes, including traditional sandboxes, virtual machines and containers, are described and practical drawbacks to these approaches are explored. Rule-based schemes and their advantages are discussed, followed by a discussion of the problems with these systems. Rule-based application-oriented access control schemes include models such as domain and type enforcement (DTE), system call interposition schemes such as Janus and Systrace, and mechanisms such as AppArmor and SELinux. The paper concludes with recommendations, and highlights the opportunities for progress within this field of research.

2. The Motivation for Application-oriented Access Controls: Limitations of Traditional Approaches

A long held assumption has been that programs will always run with the full authority of the user who starts them. This is illustrated by a Microsoft technical essay describing so called "immutable laws of security", including the statement that "if a bad guy can persuade you to run his program on your computer, it's not your computer anymore" (Microsoft, 2012b). Access controls, which restrict the actions of processes, have typically been designed on the basis that programs are trusted to act on behalf of users and that the actions of software are equivalent to the actions of users. However, this assumption is increasingly proving to be false, and a number of prevalent types of security attacks leverage this weakness to misuse the authority of users. In contemporary computing environments, programs cannot safely be automatically trusted with these rights.

In the past access control has been almost exclusively considered in terms of user confinement. Access control models were developed to specify exactly what each user could do with shared resources (based on the user's clearance or roles) and to separate users from each other (Department of Defense, 1985, Ferraiolo and Kuhn, 1992). Essentially the goal has been to protect the confidentiality, integrity, and availability of the system's resources and files from malicious users. These models can be considered examples of user-oriented access controls; that is, they primarily base access decisions on the authority granted to users. With user-oriented access control it is typical for active entities within the system (known as 'subjects') to have access to all the user's privileges regardless of the privileges actually required by the program running. In the literature most access control models are designed with the assumption that subjects act on behalf of users, and therefore they base access decisions on the user identity associated with each subject. In practice this effectively involves treating processes as equivalent to the corresponding user.

Some examples of user-oriented access control include traditional discretionary access control (DAC) models (Department of Defense, 1985) as implemented in most commodity operating systems (Garfinkel et al., 2003, Govindavajhala and

Appel, 2006)[1], traditional mandatory access control (MAC) models (Bell and LaPadula, 1975, Biba, 1977, Brewer and Nash, 1989, Department of Defense, 1985, Lipner, 1982), and role-based access control (RBAC) (Ferraiolo and Kuhn, 1992).

Despite being widely deployed, user-oriented access control is often insufficient as the sole access control mechanism. Processes do not always act on behalf of the users they belong to. Two of the major threats to operating system access control, those posed by software vulnerabilities and by malware, are particularly effective due to attackers executing malicious code via processes run by local users.

The 2009 SANS Cyber Security Risks report (Dhamankar et al., 2009) highlights attacks against software vulnerabilities in applications as the highest priority security risk. The report states that attacks against vulnerable applications, such as Adobe PDF Reader, QuickTime, Adobe Flash and Microsoft Office, are "currently the primary initial infection vector used to compromise computers that have Internet access" (Dhamankar et al., 2009). Software vulnerabilities often exist that enable attackers to gain control of legitimate processes, and misuse the authority to perform malicious actions. The causes of software vulnerabilities are numerous and varied, and have been explored and categorised by various taxonomies in the literature (Bishop, 1995, Landwehr et al., 1994, Piessens, 2002, Weber et al., 2005). Causes include design and implementation flaws, such as buffer overflows, race conditions, and input validation errors (Cowan et al., 2001a, Cowan et al., 2001b, Cowan et al., 2000b).

Software that is malicious by design, known as malware, also poses a significant threat that user-oriented access controls do not sufficiently mitigate. Malware poses security risks to users' integrity (via malicious destructiveness), confidentiality (privacy concerns), and availability (denial of service). Classifications of malware include Trojan horses, adware, spyware, viruses and worms, and taxonomies proposed in the literature provide further categorisation (Dagon et al., 2007, Stafford and Urbaczewski, 2004, Weaver et al., 2003). There are many ways that malware can propagate to computers including:

- Man-in-the-middle attacks can intercept communications between hosts and insert malware via trusted websites and hosts. It is even often possible to intercept "secure" encrypted communications (Marlinspike, 2009).
- Viruses copy themselves to other programs.
- Worms propagate across networks, often by exploiting software vulnerabilities.
- Trojan horses pose as legitimate programs.
- Malware can be sent via email in targeted attacks.

[1] The security schemes of modern operating systems such as Unix and Windows are primarily based on DAC user-oriented access control. Unix and related systems have incorporated user-oriented access controls since the 1970s. Microsoft Windows Me and earlier did not provide user-oriented access controls. User confinement is limited in Windows XP, and most users run as an administrator as many programs do not execute correctly on user confinement restricted accounts. Windows Vista and later provides improved user-oriented controls, known as User Account Control (UAC). As discussed later, modern systems are starting to incorporate some application-oriented controls.

The impact of attacks involving software vulnerabilities and malware is often significant because the malicious code executes after assuming the identity of an authorised user and is able to fully utilise all of their privileges (Bishop, 1991). Attempting to mitigate this problem by applying patches is insufficient, as these do not protect against zero-day exploits, while anti-malware software fails to detect previously unknown zero-day malware (Kotadia, 2006, Moser et al., 2007, Vegge et al., 2009). User confinement, when utilised correctly, protects system resources and users' objects from other users, but does not protect users from the applications they execute.

2.1. Trust-based Selective Execution

One of the simplest access control techniques to mitigate the risk of running programs with all of the user's authority is to only allow particular programs to run. Using this approach, processes typically still have all the authority of the user; however, only those programs deemed trustworthy (or not "untrusted") are allowed to run. There are a variety of methods that have been developed to help decide whether to treat an application as trustworthy. Mansfield-Devine (2009) describes the use of simple white or black lists of programs to control which programs are authorised to run. White lists treat only specific programs as trustworthy, while black lists specify particular programs as untrusted and treat all other programs as if they are trustworthy. In addition to the filesystem paths and attributes attached by the author, digital signatures can be used by these systems to make decisions based on who authored the software (Schiavo, 2010). Examples of trust-based selective execution include Microsoft AppLocker and Microsoft Software Restriction Policies (SRP), which are typically configured by an administrator to specify the programs that are allowed to execute (Microsoft, 2008a, b). ActiveX controls are used to run native code embedded in a web page (Microsoft, 2012c). Before an ActiveX control runs, the user is typically prompted to decide whether to allow content from the author of the control, and subsequently code signed by the author runs with the full authority of the user. Due to the threat they can pose, Microsoft maintains a black list of known harmful ActiveX controls.

Another method of selective execution is to analyse source code or binary files to decide whether the program is trusted to run. This form of trust-based selective execution is used by many of the current generation of anti-malware suites, typically based on attributes that identify code as untrusted. Signature-based and heuristic lists identify programs based on characteristics of executable files and are typically used to specify black lists to prevent known types of malware from running. The techniques used to identify malware have become increasingly sophisticated, as have the various techniques employed by malware to avoid detection (Christodorescu et al., 2006, Christodorescu et al., 2005, Nachenberg, 1997, Patcha and Park, 2007, Szor, 2005). Reputation-based security, used by systems such as Symantec Quorum, Microsoft SpyNet, and McAfee Artemis, is a relatively new technique, outlined by Ford and Allen (2009), that uses information collected from a large number of users to make judgements about the likely trustworthiness of programs to decide if programs should be trusted to run.

These trust-based security systems provide protection from a limited number of specific threats. However, many legitimate "trusted" programs can be the source of malicious behaviour: for example, well-intended software authors may accidentally introduce design or implementation flaws, resulting in software vulnerabilities that enable attackers to execute malicious code. These approaches do not provide a mechanism for safely running programs without trusting them to run with all of a user's authority. However, it is not ideal to have to completely trust any software. Furthermore, in many cases it is overly restrictive or impractical to prohibit users from running code obtained from third parties via the Internet. For example, mobile phone "apps" are currently very popular, and the web is becoming increasingly dynamic, including client-side execution of mobile code. All of these mechanisms can fail to protect users from malware. At various times:

- digital signatures and certificates have failed to accurately reflect the actual origin of programs (Callas, 2005, Marlinspike, 2009, Microsoft, 2001);
- ActiveX has been a prevalent infection vector (Bellovin et al., 2000); and,
- anti-malware black list techniques have failed to identify malware (Christodorescu and Jha, 2004, Kotadia, 2006, Vegge et al., 2009).

Clearly a more comprehensive confinement approach that has the ability to treat applications as untrusted, and that facilitates the execution of untrusted software, can better mitigate the threats posed by the problem of malicious code.

2.2. Integrity-level Schemes

Some integrity-level security schemes are related to application-oriented access controls. However, rather than focusing on restricting particular programs, these schemes divide programs and resources based on how trustworthy they are considered and restrict lower level programs from interacting with higher level programs and resources. This involves labelling every subject and object with integrity levels, and restricting the interaction between different levels. For example, Microsoft Windows (since Vista) incorporates an integrity level scheme known as the Windows integrity mechanism described by Governa (2009). Windows runs most user programs at the same integrity level, while some high risk applications such as Internet Explorer are run at a lower level. Other similar security schemes include LOMAC, presented by Fraser (2000, Fraser, 2001), which takes a dynamic approach to integrity labelling, and Li et al.'s (2007) HIPP mechanism. Although these schemes reduce the risk of malicious code, they are not intended as mechanisms for restricting and sandboxing applications and do not limit programs to only the resources they require.

3. Application-oriented Access Control

The limitations of user-oriented access control have led to application-oriented access control being an increasingly active area of research. Application-oriented access controls restrict subjects based on the identity of the application or process, rather than just the identity of the user. This approach is designed to limit the ability of applications to access resources outside of those they require to perform legitimate actions. Application-oriented controls can restrict the damage caused by malware or exploited vulnerabilities by limiting the software to those actions authorised by whoever configures the security policy, whether end users, administrators, or software developers. The extent of the mediation to resources that is provided by each application-oriented access control scheme depends on the specifics of the implementation. For example, access to files on the system is typically mediated, while access to network resources and inter-process communication may or may not be mediated depending on the goals of the scheme.

The remainder of this paper provides a comprehensive overview of the various methods and techniques available to provide application-oriented restrictions. Policy can be specified by various parties, and policy can do one of the following: simply specify which programs are authorised to run, isolate programs and their effects from each other, or allow programs to co-exist in the same namespace and restrict what each is allowed to do.

3.1. Terminology

In the literature, the relationship between access control that bases its restriction on the confinement of users and that which provides restrictions based on the privileges required by specific applications or processes has not always been identified. In this paper the terms 'user-oriented access control' and 'application-oriented access control' are used to differentiate between access control based on the identity of the user and that based on the identity of the application or process. It is useful to introduce this distinction, not previously made explicit in the literature, to distinguish the relationship between these fields of research. With user-oriented access control, the security context is primarily the identity or authority of the user associated with the subject being confined. The identity of the program is of secondary concern and is generally not considered. However, with application-oriented access control the security context is primarily the identity or authority of the process or application and the identity of the user is of secondary concern, typically being considered separately. These terms are proposed to differentiate between these two related, but distinct, types of access control that, as areas of research, are often not considered in terms of each other.

Application-oriented access control has usually been considered separately from user restrictions and access control research. This may be due to an assumed implicit equivalence between access control and user restrictions. It may also be due to the tendency of application-oriented access control research to propose, design and implement systems, rather focus on a theoretical model of a solution (for example, (Garfinkel, 2003, Goldberg et al., 1996, Wagner, 1999)). A clear exception is Type Enforcement (TE) (Boebert and Kain, 1985), an application-oriented access control model that has been described and developed within the field of access control, although this distinction has typically not been made in the literature.

User-oriented access controls restrict the actions of users to specific system-wide resources, while application-oriented controls can further limit an application to the specific resources it requires, usually a subset of a user's authority. Essentially the theory developed in user-oriented (access control) research can apply to application-oriented research. This is demonstrated in the conceptualisation of the functionality-based application confinement (FBAC) model (Schreuders and Payne, 2008a) described at the end of this paper, which incorporates constructs adapted from RBAC. Although in the past application-oriented restrictions have usually been considered separately from user restrictions and access control, there is a close relationship between the two fields.

3.2. Policy Providers

Independent of the actual techniques used to enforce security decisions, application-oriented access control schemes can also be differentiated based on who provides the security policy. Policy defines the access rules that are enforced. In the case of rule-based controls, these are typically defined in the form of a set of rules in a policy language that the target mechanism can enforce. Policy rules can also be defined using other means, including via platform configuration, file permissions, or digital signatures. Other security mechanisms have an implied policy, for example, an isolated sandbox, in which case simply entering the sandbox environment may define the policy that applies. In each case, some entity, such as a user, administrator, or process has the volition to set the policy. Application restrictions can be discretionary (in the control of the user) or non-discretionary (imposed on the user, and thus may provide an additional element of user confinement). Policy for application-oriented access control can be defined by various stakeholders to improve security by restricting the behaviour of certain processes (Anderson et al., 2010). This section describes the main ways that application restriction policies can be provided. Each of the security mechanisms mentioned are subsequently described in further detail.

Policy may be defined by distributors (such as software repository administrators) or software vendors, in order to prevent their software from being subverted and behaving maliciously due to software vulnerabilities. MapBox is an example of an application-oriented access control mechanism where vendors specify what behaviour the application should exhibit and users are then able to alter the exact policy assigned to these behaviours (Acharya and Raje, 2000). Enck et al. (2009) outline a similar approach taken by Android applications, which are distributed with manifest files that specify the privilege requirements of each application (granting access to protected APIs). Users must accept these policies in order to install the application. Software vendors and authors are well-placed to create policies that effectively restrict the damage caused by exploitation of vulnerabilities, as they have access to internal documentation and source code that can facilitate least privilege policies. However, it is not always in the authors' interests to restrict their applications. Creating the policy increases their workload and it is the user, rather than the author, that primarily bears the risk of an unrestricted application. Distributors of software are unlikely to provide policies unless the security system is widespread enough to justify the additional work. Further, malware authors obviously have malicious motives and cannot be trusted to create secure policies under any circumstances.

Some schemes allow policy to be defined by software that can, at run-time, confine itself to prevent unexpected behaviour[2]. Programs can use these schemes to safely execute mobile code or reduce the risk of vulnerabilities. Some examples of such mechanisms available to applications are `chroot()` and Jails system calls (Kamp and Watson, 2000, Lessard, 2003). Also, the Mac OS X Seatbelt sandbox mechanism allows applications to isolate themselves (Edge et al., 2010). These policies are typically non-discretionary; that is, users usually cannot configure the policies applications request.

[2] The related field, sometimes referred to as ‘application security’, deals with applications that change their behaviour based on the user’s identity and authority, and is outside the scope of this article.

Policies may be defined by system administrators to limit the behaviour of applications installed on a system and the actions users can perform with applications. Server applications, or any Internet-enabled applications, can be confined to minimise the damage caused by exploited vulnerabilities. Also, the applications accessible to users can be restricted to certain authorized behaviour. Some examples of application-oriented access control mechanisms that are managed by system administrators are AppArmor (Cowan et al., 2000a), DTE (Hallyn and Kearns, 2000), and SELinux (Loscocco and Smalley, 2001a). These are non-discretionary controls, configured by administrators.

Policies may also be defined by end users, who can confine specific applications, thus restricting malware or exploits. Examples of systems that provide discretionary application controls include Systrace (Provos, 2002) and Alcatraz (Liang et al., 2009). End users who are the 'administrators' of their own systems may also use the previously described non-discretionary controls. Users may be well-positioned to restrict applications to behavioural expectations because they know what they require the application to do. However, for this to be effective, policy management must be simple and unobtrusive due to varying levels of end user expertise.

Each of these options addresses various security goals. Application-oriented access control models or mechanisms typically do not provide the flexibility to enforce policies defined at more than one of the levels described above, meaning that sometimes it is necessary to combine these mechanisms.

4. Isolation-based Application-oriented Access Controls: Sandboxes and Virtualisation

One way to restrict a program's ability to access resources is to run it in an environment where the application can only access objects within their so-called 'sandbox'. Although the terminology in use varies, in general a sandbox is separate from the access controls applied to all running programs. Typically sandboxes only apply to programs explicitly launched into or from within a sandbox. In most cases no security context changes take place when a new process is started, and all programs in a particular sandbox run with the same set of rights. Sandboxes can either be permanent where resource changes persist after the programs finish running, or ephemeral where changes are discarded after the sandbox is no longer in use (Potter and Nieh, 2010, Rutkowska and Wojtczuk, 2010).

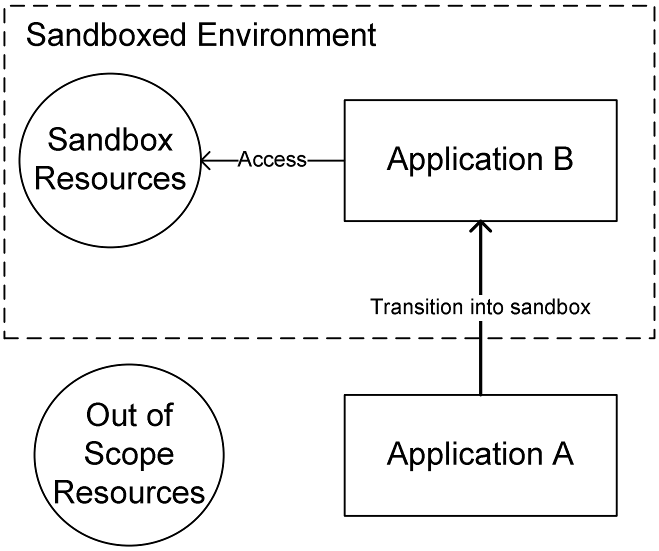

Figure 1: Typical Isolation-based Sandbox

Figure 1 illustrates a typical sandbox scheme, where an application (A) explicitly starts another application (B) within a sandboxed environment. Application B can access the resources available within the sandbox; however, the sandboxed application cannot access (or sometimes even name) resources outside of the sandbox. Children processes typically remain and co-exist within the same sandbox. Applications (such as Application A in the figure) that are not within the sandbox are typically outside the scope of the sandbox access controls, and are therefore (other security controls permitting) free to access any resources, including those within a sandbox.

Most sandboxes provide an isolation-based approach where the effect of programs run inside a sandbox is entirely isolated from resources outside the sandbox's authority. However, due to practical requirements, sandboxing schemes often provide ways of circumventing this isolation in order to copy data into and out of sandboxes.

System-level sandboxes provide complete operating environments to confined applications. One way of achieving this is through the use of hardware-level virtual machines (VMs). A virtual machine monitor (VMM) can be used to multiplex the physical hardware between multiple self-contained fully virtualised VM operating environments, each containing a complete operating system. As early as the 1970s Madnick and Donovan (1973) showed that VMs can be used to improve overall system security by providing an additional layer of controls. More recently researchers have proposed using VMs to confine applications to mitigate the risks associated with excess application authority. For example, Whitaker et al. (2002) presented a scheme designed to provide a lightweight VM and operating system for each application. It is possible to confine specific programs or sets of programs by containing them within separate VMs. This can be done manually using emulation based virtualisation software (such as VMware (Sugerman et al., 2001), VirtualBox (Oracle, 2012), Parallels (Parallels Inc, 2012b), Microsoft Virtual Server (Microsoft, 2012a) and QEMU (Bellard, 2005)) or using paravirtualisation software, which requires that guest operating systems are aware that they are running as VMs and participate in providing virtualisation.

Xen (Barham et al., 2003), User-mode Linux (UML) (Dike, 2000), and Denali (Whitaker et al., 2002) are examples of paravirtualisation.

Systems such as Qubes, released by Rutkowska and Wojtczuk (2010), provide an interface to simplify the management of VMs from a security perspective. Qubes provides lightweight ‘AppVMs’, which are used for different types of tasks and, unlike most hardware-level VMs, share the same read-only filesystem. For example, ‘personal’, ‘work’, and ‘bank’ VMs could be specified to keep the applications used and the risks associated with each task separate from each other. Although not based on system-level virtualisation, the earlier WindowBox scheme, proposed by Balfanz and Simon (2000), provides a similar mechanism where different desktop sandboxes are used for different types of tasks. Most of these types of systems provide ways for users to manually move files between VMs.

Container-based virtualisation, also known as operating-system level virtualisation, shares the one operating system kernel, but virtualises user space resources, allowing separate instances of the operating environment to be created. The Unix `chroot()` system call isolates a process and its children to a subset of the file system namespace. However, it was not designed as a security mechanism and consequently has significant limitations. The FreeBSD Jails mechanism, proposed by Kamp and Watson (2000), provides the `jail()` system call that aims to improve upon the security and functionality provided by `chroot()`. Jails confines each jail to a given IP address with particular network privileges, prevents well-documented means of escaping `chroot()` confinement, prevents certain super-user privileged actions from taking place, and each jail has a private copy of operating system objects such as shared memory and sockets (Kamp and Watson, 2004). Rule Set Based Access Control (RSBAC), presented by Ott et al. (2001), includes a jail module similar to FreeBSD Jails for Linux. Other similar systems include Solaris 10 Containers (Tucker and Comay, 2004), Linux Containers (lxc) (LXC, 2012), Linux VServer (des Ligneris, 2005), OpenVZ (Parallels Inc, 2012a), FreeVPS (Positive Software Corporation, 2012), Virtuozzo Containers (Parallels Inc, 2012c), and AIX Workload Partitions (WPARs) (Blanchard et al., 2007). Some of these systems are primarily designed for running complete operating environments and for isolating multiple virtual hosts. These solutions allow applications to be confined (along with the resources they require) by encapsulating them within a partitioned environment with no access to objects outside of the sandboxed environment. However, these container sandboxes are often easier to manage than hardware-level VMs, as different containers can be configured to use the same resources, such as portions of the operating system.

Potter and Nieh (2010) have proposed a more sophisticated approach to container-based sandboxes for application-oriented access control, with an implementation known as Apiary. Similar to how Qubes provides an interface to manage VMs for security purposes, Apiary presents a desktop environment that provides a user interface to launch applications into containers. Users can select whether to launch applications into ephemeral or persistent containers. Each container is isolated from each other, although there are ways for users to intervene to share directories with containers or copy data between containers.

Some sandbox schemes allow confined applications to read the host computer’s data with few restrictions, and provide copy-on-write features to write any modifications programs make to a virtual disk rather than to the actual hard drive. Systems such as Sandboxie (Tzur, 2012), Pastures (Bratus et al., 2007), Alcatraz (Liang et al., 2009, Liang et al., 2003, Sun et al., 2005), and Returnil (Returnil, 2012) take this approach. Typically, upon termination of the sandboxed programs, these systems present the user with a list of all the files that were modified, and the user has the option to commit changes to the regular file system.

Other sandbox solutions run single, self-contained applications using a VMM or an interpreter, which isolates the application from making changes to the host computer without user intervention. Java Applets (Gong et al., 1997), Silverlight (Farkas, 2007) and Flash (McCauley, 2008) use this approach to embed mobile code content into websites; very limited access is granted to storage, with additional access being granted via user interaction. Google native code (NaCl) executes native instructions inside an isolated sandbox, and interacts with JavaScript and other plugins that are responsible for granting restricted access to other resources (Yee et al., 2010). As previously mentioned, Whitaker and Shaw’s (2002) scheme Denali was originally designed to have VMs running single-application lightweight operating systems, to provide individual internet services that are confined. However, newer versions virtualise full commodity operating systems. Miller et al. (2004) developed CapDesk Caplets, which are self-contained programs that gain additional authority via a special type of file selection dialog, known as a ‘powerbox’, that grants access to files selected by the user.

Kato and Oyama (2002) have proposed a different approach. The scheme, known as SoftwarePot, restricts programs to an encapsulated filesystem that is mostly isolated but which can contain rules for mapping resources on the local computer to the name-space accessible to the restricted program.

4.1. Limitations of Isolation-based Approaches

All of the aforementioned security schemes attempt to mitigate the effects of malicious code by isolating programs or groups of programs from each other and the system in general. Although usually relatively straightforward to initially configure, isolated sandboxes present a number of problems in use. Many of these schemes suffer from redundancy of resources and management, and isolating programs entirely is often not practical.

Isolation-based security schemes generally require significant redundancy of resources, as any resources that an application requires must be contained within or accessible from the sandbox. In the case of VMs this typically involves duplicating a copy of the entire operating system for each group of programs that are confined. There is also often significant overhead in managing the configuration of the VMs. For container-based sandboxes and self-contained application sandboxes, often any shared libraries or resources must be duplicated within each sandbox that requires access to them (Kamp and Watson, 2000). Even copy-on-write sandboxes, which allow read access to the same operating environment, introduce management overhead when files are changed within a sandbox.

Isolation sandboxes inhibit applications operating in different sandboxes from easily and securely exchanging information or sharing resources. However, many applications frequently require access to shared resources and this is integral to the way some applications are used. For example, users often utilise separate applications for different tasks when working on a given set of files. A user may use one program to create a file, another to view it, and then use a third application to transfer the file over a network. Isolated sandboxes are not well suited to this scenario as the files they create would typically be completely isolated from each other. Alternatively, all three applications and the file could exist in a single sandbox. However, in this case each of the three applications are effectively entrusted with the same privileges, despite their potentially different requirements, with the result being that, for example, a viewing program could maliciously edit or send a file over the network.

Since the ability to share resources is such a common requirement, many of these systems provide ways of circumventing straightforward isolation. VMs can use the network to communicate and share files, containers can use hard-linked files to share the same resources, and others provide mechanisms that require user intervention. These methods of sharing pose additional challenges and new security problems, as risks are inevitably associated with requiring users to circumvent the security model.

Perhaps the greatest obstacle is that isolation sandboxes inhibit users from working with software in the way they are accustomed. In most cases users need to decide which applications they need to launch or install into sandboxes. As discussed by Potter and Nieh (2010), these applications are often not integrated into, or easily accessible from, the user's desktop, and such applications cannot leverage other unconfined applications. Also, requiring user interaction to manually copy files between sandboxes requires users to have a clear understanding of these concepts. Due to the large number of resources typically required to run an application, the decision making process for user intervention can be complicated (Jaeger et al., 2003, Marceau and Joyce, 2005).

Table 1 presents an approximate summary of the limitations that have been discussed and how they relate to each of the classes of isolation-based systems that have been described. In the table an "X" denotes that the limitation typically applies to this class of system, "X*" describes a limitation that applies to a lesser extent or with some exceptions. As an example, the following describes the "self-contained applications" row in the table. The self-contained applications category of system (as described in the previous section, examples include Java applets, Flash, and so on) avoids the issue of "OS configuration redundancy", where multiple instances of the operating environment can result in configuration overhead. This is in contrast to system-level sandboxes where entire copies of operating systems need to be maintained and administered. Continuing the example, self-contained application schemes typically isolate applications individually, rather than having "many processes run in the same context" (the second row in the table), which can diminish the restrictions on applications. Such is the case with the first three categories of schemes, where applications often share the same sandbox and subsequently a security compromise in one application can result in the entire sandbox and all the applications and data within the sandbox becoming compromised. The third row indicates that self-contained application schemes have limited ability to allow sandboxed applications to access specific files outside of the sandbox, like many isolation-based application-oriented access controls, these schemes are designed to enable very limited (if any) access to system resources, and sharing files between applications is often not possible, with the exception of Powerboxes which some of these systems, such as CapDesk Caplets, employ to grant access to individual files – this has the additional caveat that Powerboxes do not suit applications that require access to files without user interaction: for example, for searching through multiple files. The following row illustrates that this class of security scheme often requires libraries to be duplicated for each application; the exception being libraries provided by the framework itself. The table also shows that these schemes typically provide network restrictions. Also, there are not usually complex rules to review. As is the case with all of the isolation-based schemes, self-contained application schemes are not well suited for editing and viewing files with separate applications while applying appropriate security restrictions (for example, read-write or read for editing and viewing respectively); a rare exception would be a Powerbox system that prompts the user for this information whenever users open files. As shown in the table, these systems scale fairly well to restrict a number of applications without making management and policies more complex. They are typically not designed for separately confined applications to interact, which can be a practical limitation for building complex systems from multiple programs. Finally, perhaps their most significant limitation, self-contained application schemes require the applications to be developed specifically for the target platform and with the security restrictions in mind; the security scheme cannot typically be applied to "legacy applications", that is, other existing applications.

Although isolated sandboxes suit certain situations – such as when there are only a few untrusted programs that do not require interaction with other applications – in most practical scenarios sandboxes are not well-suited to restricting large numbers of programs, programs that require different privileges, or those that share resources. Consequently, isolation-based schemes often do not provide a sufficiently scaleable solution to the problems posed by malicious code. The evolution of sandboxed environments to provide increasing degrees of exceptions to straightforward isolation has led to further development of rule-based schemes, as discussed in the following sections.

Table 1: Limitations of isolation-based application-oriented access controls

Scheme category	*General properties*	*Example*	OS configuration redundancy	Many processes run in the same context	Sharing individual files (as required) between sandboxes is typically not supported, or is manual	Duplication of resources (OS, libraries)	Lack of networking and/or IPC mediation	Complex changes or rules to review	Does not support simple workflow for editing the same files between sandboxes	Does not scale well to a large number of separate sandboxes	Does not suit applications in different security contexts interacting	Cannot be applied to legacy applications
System-level sandbox	VMs containing complete OSs	VirtualBox	X	X	X	X	X		X	X	X	
Isolated desktop	Multiple desktops isolated from each other	Qubes	X*	X	X*	X*	X*		X*		X	
Container-based virtualisation	Shares the kernel, with separate user-space resources	Jails	X	X*	X	X	X		X	X	X	
Desktop sandbox launcher	Launches applications into various kinds of sandboxes	Apairy	X*		X*		X		X*		X*	
Copy-on-write sandbox	Applications can read system-wide resources, writes are isolated to sandbox	Sandboxie	X*				X	X*	X*		X	
Self-contained applications	Applications run via VMM or an interpreter, strictly isolated from OS	Flash			X*	X*			X*		X	X
Self-contained applications with mapped resources	As above, with rules defining access to other resources	SoftwarePot				X*		X*	X		X	

5. Rule-based Application-oriented Access Controls

5.1. Rule-based Sandboxes

Rather than focusing on completely isolating applications, rule-based schemes aim to control what each application is authorised to do. This approach allows applications to cooperate and share resources; although each application may still only perform the actions and access the resources specified in the security policy. These systems often include methods for policy transitions and the propagation of privileges when processes are started. Such techniques can restrict individual programs with different rules and mitigate the 'confused deputy problem' where a malicious program misuses another program's privileges by influencing the privileged program to act on its behalf (Benantar, 2006 p. 19, Hardy, 1988). This section discusses rule-based sandboxing schemes, and the following section covers system-wide rule-based application-oriented access controls.

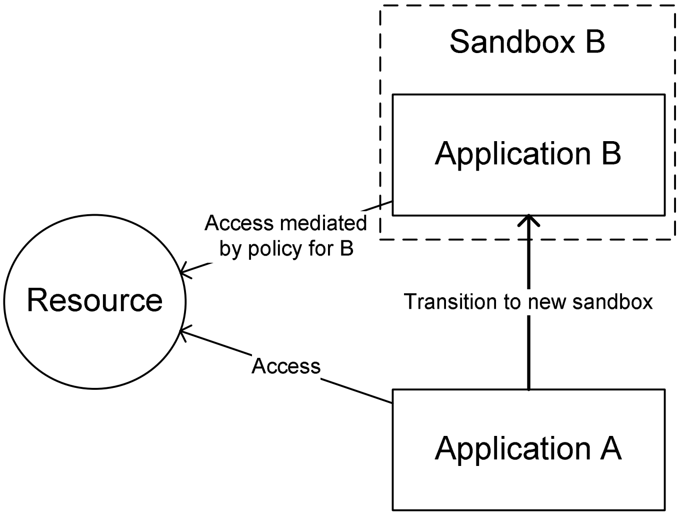

Figure 2: Typical Rule-based Sandbox

As illustrated in Figure 2, each rule-based sandbox enforces a specific policy regarding the resources that applications within the sandbox are allowed to access. Applications (such as A, in the figure), can implicitly launch applications (B) into a sandbox that enforces a policy. Depending on the specifics of the scheme and policy in place, children may have the same policy apply, or may transition to another sandbox policy. Applications that have not implicitly been launched into a sandbox are typically out of the scope of the mediation provided.

Berman et al. (1995) presented TRON, a discretionary application-oriented access control mechanism for Unix implemented on Ultrix. TRON confined applications to 'domains' that were assigned privileges based on text strings naming the files and directories a domain could access and the type of access allowed. Users then had the option of confining their applications by running them within a domain using a command line interface. TRON was implemented using system call interposition within the kernel.

System call interposition is a method for monitoring and regulating application behaviour by intercepting the program's system calls (Garfinkel, 2003). All external interactions, and hence potentially harmful operations, an application can perform occur via system calls. Therefore filtering calls based on a security policy can confine an application by restricting the application's access to the operating system's privileged kernel operations. System call interposition tools, such as Janus (Goldberg et al., 1996, Wagner, 1999), Systrace (Kurchuck and Keromytis, 2004, Provos, 2002), and ETrace (Jain and Sekar, 2000), can enforce extremely fine grained policies at the level of granularity of the operating system's system call infrastructure.

A number of system call interposition mediation mechanism designs have been proposed (Alexandrov et al., 1998, Jones, 1993) including library modifications (Krell and Krishnamurthy, 1992), kernel mediation (Fraser et al., 1999, Ghormley et al., 1998), and user-space mediation (Alexandrov et al., 1998, Garfinkel et al., 2004, Goldberg et al., 1996, Jain and Sekar, 2000). In most cases this involves a monitoring process enforcing the policy on a traced process. For each application being confined, its security policy is typically defined in terms of the specific system calls and arguments that are allowed. In many cases a single operation requires many system calls. For example, in order to make a network connection the socket is created, and then the protocol is specified. Consequently, system call interposition-based policies can be extremely lengthy and detailed, exposing the underlying complexity of the platform on which the software operates. To ease the burden of policy development, Systrace includes a policy learning mode that can generate policy rules based on recorded application actions. Some schemes such as Software Wrappers, proposed by Fraser et al. (1999), specify policy using a slightly higher level policy language that is independent of the actual system call interface. Seaborn (2008) has described the Plash scheme that uses system call interposition and jails to provide a form of capability-based security, thereby restricting programs to only the files specified by the user.

System call anomaly detection is a related field, pioneered by Forrest et al. (2008, 1994), which aims to monitor system calls in order to detect, and potentially react to, anomalies in system call sequences. These approaches attempt to apply concepts from immunology and epidemiology to the problem of detecting and reacting to abnormal behaviour from processes. A number of techniques for defining what constitutes valid sequences or parameters for system calls have been proposed: including learning modes that record program activity (Hofmeyr et al., 1998, Tandon and Chan, 2006), and static analysis of binary or source code to determine code path or data flow (Ben-Cohen and Wool, 2008, Giffin et al., 2002, Lam and Chiueh, 2005, Wagner and Dean, 2001).

Application virtualization is an alternative technique, primarily designed to allow portable application bytecode to run regardless of the platform. These interfaces can also provide controls on the resources available to applications. Java (Gong et al., 1997) and .NET (Thorsteinson and Ganesh, 2003) confinement models restrict applications based on both certain properties of the code and security policies defined by the user or administrator, explicitly permitting or denying certain operations from particular sources and authors. Each set of rules needs to be configured individually. These security models can only be applied to applications written in the corresponding programming languages. Due to the complexity of policy specification, in practice these restrictions are rarely employed. Language-based security schemes such as type checking and proof-carrying code (Lee and Necula, 1997, Necula, 1997, Necula and Lee, 2004), call/control-flow graph run-time verification (Ben-Cohen and Wool, 2008, Giffin et al., 2002, Lam and Chiueh, 2005, Wagner and Dean, 2001), object-capability schemes (Hardy, 1985, Mettler and Wagner, 2008, Miller, 2006), intra-component access controls (Levy et al., 1998), and code/binary transformation (Kiriansky et al., 2002, Pandey and

Hashii, 1999) can provide related additional layers of protection, but are outside of the scope of this paper.

Mac OS X includes a sandbox feature that can enforce two types of confinement (Edge et al., 2010). It allows an application to voluntarily confine itself and its subsequent children to one of five coarsely grained policies: for example, a ‘no Internet’ profile. Finely grained policies can also be applied to applications by launching them into the sandbox using the “sandbox-exec” command. The iOS platform applies a single finely-grained policy to all installed applications that largely inhibits their ability to access the data from other applications. However, this policy currently allows all iPhone applications to access various resources without notifying the user. For example, all iPhone applications have access to the address book, phone number, email account settings, and web search history (Seriot, 2010). With a wide variety of applications being available for the system, a ‘one size fits all’ approach to security policy is not always appropriate.

The MAPbox scheme, proposed by Raje and Acharya (Acharya and Raje, 2000, Raje, 1999), utilises a model where the vendor of the application is trusted to specify its security policy based on a so-called ‘behaviour class’. The application is then executed and confined based on user-assigned discretionary application-oriented access control policies for each allowed behaviour class. This allows the provider to communicate the general expected functionality of the program to the user, similar to the way in which MIME-types describe the expected format of data. Programs are described in terms of a behaviour class, and the resources they require are further clarified by providing parameters. For example, a program may be of class “editor”, requiring access to parameters that specify the program’s home directory and a list of files it can edit.

5.2. Rule-based System-wide Controls

Another, more comprehensive, approach is to restrict applications at the operating system security kernel level. System-wide controls involve confining all programs with an applicable policy, rather than just those explicitly launched into sandboxes. In these schemes, policy typically includes identification of programs and specification of rules for their authorised activity. Program identity may be established via file path naming, file labelling, or other techniques similar to those used to identify programs in trust-based selective execution schemes, as discussed in Section 2.1.

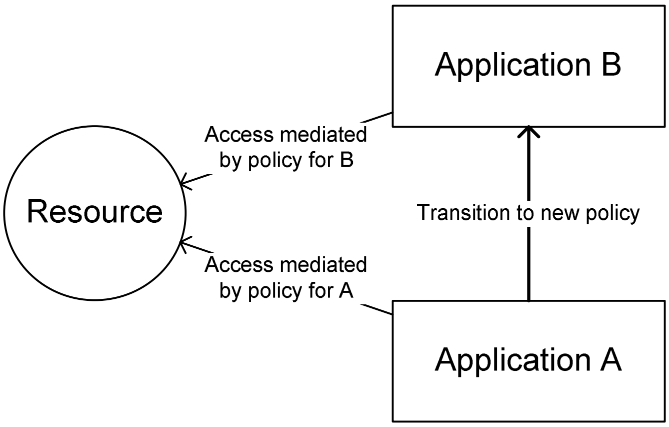
Application B
Access mediated by policy for B
Resource
Transition to new policy
Access mediated by policy for A
Application A

Figure 3: Typical Rule-based System-wide Control

As illustrated in Figure 3, all applications are confined based on the policies that apply. In the figure, Application A's access to any resource is subject to the policy for that application, while another policy may apply to Application B. Policy transitions are typically automatic, and applications actions are mediated without the need to launch them into sandboxed environments.

The simplest form of rule-based system-wide control is achieved by overloading the user-oriented access controls already in place to restrict applications. For example, on Linux systems the standard DAC mechanism can be used to control the actions of a particular program by creating an artificial user account specifically for that program. If the program has sufficient authority and is trusted to do so, it can restrict itself by changing its own user identity (for example, when running the Apache web server as root it typically automatically changes its user identity to the "nobody" or "apache" account), setuid file permissions can be used to launch the program as the owner of the executable, or the program can be launched as a specific user manually. Some application-oriented access control schemes are based on this technique. Rainbow provides an "exec-wrapper" that can automate the creation of 'synthetic' user accounts and launching of programs into those user accounts (OLPC, 2012). Stiegler et al. (2006) present the Polaris system, which leverages this approach to confine programs: applications are launched into restricted user accounts that are granted access to the user's files only when the user selects a file via a Windows file selection dialog. Access is granted via a mechanism that copies files into the application's home directory, and subsequently copies changes back to the user's home directory. Snowberger's (2007) sub-identities scheme aims to allow users to arbitrarily create user accounts that are limited to a subset of the user's rights.

Another approach is to divide the actions an application can be authorised to perform into simple, coarsely-grained privileges. Linux capabilities (also known as 'POSIX capabilities', since they are based on a withdrawn POSIX standard draft) partition the powerful root privilege into a fixed set of distinct privileges; for example, the ability to send signals to processes owned by any user, or to listen on TCP/UDP sockets below 1024 (Hallyn and Morgan, 2008, IEEE, 1997). Linux capabilities are not associated with any object, are represented as flag variables, and simply permit certain types of actions. Most other rule-based application-oriented access controls take a finer-grained approach that allows further control over applications.

Some schemes combine overloaded user confinement with additional coarse privileges. Krsti and Garfinkel (2007) combine the previously-described Rainbow scheme with coarsely grained privileges to create BitFrost. This scheme is used as the security model for the ‘One Laptop Per Child’ project to authorise additional restricted actions such as using the laptop microphone and camera. Authors of software specify which privileges are required by their programs at installation. Certain BitFrost privileges cannot normally be combined, which is intended to limit possible malicious activity. The Android platform creates separate user accounts for each application installed as part of its security scheme (Enck et al., 2009). Application authors specify all the coarsely grained kernel-protected privileges required in an `AndroidManifest.xml` file and, once an application is installed, it is prevented from exceeding these privileges. Applications run in a modified version of Java known as Dalvik, and additional controls are enforced by an application virtualisation layer. Native code is not subject to these additional controls (Barrera, 2011).

Program access control lists (PACLs), first proposed by Wichers et al. (1990), label resources with a list of programs, specifying the types of access allowed. Schemes such as Tivoli Access Control Facility (TACF) (Manoel et al., 2000) and eTrust Access Control (Computer Associates, 2001) provide PACLs that specify the names of the programs or shell scripts users must execute in order to access PACL protected resources. Rather than configuring policy to describe every resource accessible to every application, these controls typically limit their protection to specified resources. Proposed by Enck et al. (2008, 2007), PinUP provides a related form of restrictions; cryptographic hashes identify unmodified applications that are allowed to access files. When files are created, detailed rules are used to automatically assign a list of applications that are allowed access to them, and users can manually authorise additional application-file interactions that occur. Other related approaches exist. Schmid et al. (2002) describe the FileMonster scheme, which tags particular files as requiring user confirmation before applications can access them. The SubOS scheme, proposed by Ioannidis et al. (2002), associates policies with files based on their origin on the network, and processes are restricted by the policies of all the files they have accessed. However, this combination of numerous remote sources and multiple files can lead to significant complexity in practice.

An early approach presented by Boebert and Kain (1985), Type enforcement (TE) is a non-discretionary table-based access control model that confines applications based on grouping subjects into domains, and objects into types by labelling them. Access decisions are then made based on the contents of a domain interaction table (DIT) and a domain definition table (DDT). Walker et al. (1996) extended type enforcement to create Domain and type enforcement (DTE), which includes a policy specification language, DTE language (DTEL), to replace the DIT and DDT tables by describing what domain transitions can occur when executing programs and what interactions are allowed between domains and types. Several DTE mechanisms exist, many of which deviate from the original model and policy language: DTE mechanisms have been developed for Linux (Hallyn and Kearns, 2000) and FreeBSD (Badger, 1996, Badger et al., 1995) among others.

Ott’s (2002) RSBAC framework for Linux includes the role compatibility access control model, which is very similar to type enforcement. Although termed ‘role compatibility’, it has much more in common with DTE than RBAC: the initial role is specified, one role is active at a time, role transitions are specified by compatibility and it may not be possible to return to the original role, objects are classified as types, and role-type interactions are mediated.

SELinux, originally developed by the US National Security Agency (NSA), provides non-discretionary controls for Linux (Loscocco and Smalley, 2001a, Smalley et al., 2001). SELinux is arguably the most complete (and complex) non-discretionary access control scheme available in any operating system. Access control decisions are made based on the security context resources are labelled with, implementing a combination of access control models such as RBAC, DTE, and multilevel security (MLS) (Loscocco and Smalley, 2001b). DTE forms the primary basis of application restrictions: rules specify when domain transitions occur, which determines the domain a program is associated with, and rules define how processes within particular domains can access resources labelled with specific types. SELinux combines DTE with a non-standard RBAC model to also provide user confinement in terms of domains. Additionally, the new SELinux-Sandbox script provides a form of isolation-based application-oriented access control; it allows users to restrict programs by launching them via a command that generates and applies an SELinux policy to isolate the program to the files specified as arguments (Walsh, 2009). Like DTE and RSBAC, SELinux policies involve significant complexity.

A common approach to rule-based application-oriented access control is to simply specify a list of all the resources each application is authorised to access. As proposed by Cowan *et al.* (2000a), AppArmor (originally known as SubDomain) takes this approach, where a lengthy security policy in the form of an application profile defines all the files the confined program can access. AppArmor is similar to the previously described TRON (Berman et al., 1995); the main differences being that AppArmor allows processes to “change hats” depending on the tasks they are carrying out, and AppArmor is non-discretionary; policy is defined by an administrator and enforced system wide (Canonical, 2012). AppArmor is supplied with a number of Linux distributions including the popular Ubuntu and SuSE systems. Official Novell AppArmor documentation presents it as a user-friendly alternative to SELinux (Novell, 2012). AppArmor focuses on confining applications that are at a high risk of vulnerability (applications that are network-enabled) and confines them to only access the files they need to perform their tasks. Policies can also contain simple abstractions (common groups of privileges) and can be generated using a learning mode. Applications without policies are typically not confined. Harada et al. (2004) have presented TOMOYO, a similar system that applies policies based on the process invocation ancestry. TOMOYO also includes a learning mode to develop policies.

The PeaPod application-oriented access control takes a hybrid approach by combining an isolation-based ‘pod’ (PrOcess Domain), which provides container-based virtualisation, with ‘peas’ (Protection and Encapsulation Abstraction), which specify a list of all the resources each process is authorised to access within a pod (Potter et al., 2007, Potter et al., 2004). Pea policies are similar to the AppArmor policy language.

The authors have proposed a model known as functionality-based application confinement (FBAC) (Schreuders and Payne, 2008a), and an LSM-based (Linux security module) implementation has been developed (Schreuders, 2012). FBAC manages the authority assigned to applications based on the features they provide. Reusable modular policy abstractions known as *functionalities* can be employed by end users or administrators to confine programs based on high-level security goals (Schreuders and Payne, 2008b). For example, an application can be assigned a “web browser” functionality, which authorises the application to access appropriate resources. FBAC functionalities are similar in purpose to MAPbox behavioural classes, while overcoming the previously discussed limitations. Policy is represented using the FBAC policy language (Schreuders et al., 2011b). Functionalities are hierarchical and parameterised, which enables them to provide layers of abstraction and encapsulation of policy details and to adapt to the needs of individual applications. For example, the “web browser” functionality can be adapted to the needs of various browsers by specifying parameters, such as where downloads and configuration is stored for each application.

5.3. Limitations of Rule-based Application Restriction Schemes

Rule-based application restrictions can greatly mitigate the threat posed by malicious code, and avoid many of the previously discussed problems with isolation-based approaches. However, despite the large number of rule-based application restriction schemes that have been developed, these have largely not been embraced by users. Although they provide significant security advantages, rule-based schemes have a number of drawbacks that hinder their use. Coarsely grained controls are less able to protect against a diverse range of threats, while finely grained controls typically result in policy management complexity and significant usability problems. This section discusses issues with the previously described rule-based schemes and explains the need for further work in this area.

Simple coarsely grained policies can complement user-oriented access controls to reduce the risks associated with running programs with all of a user’s privileges; for example, Linux capabilities separate many of the root user’s special privileges. However, these restrictions typically only mediate access to specific resources, which often means that the policy allows more access than is necessary for the application to function and malicious programs have free rein with regards to those resources not specified. For example, flag-based privileges such as Linux capabilities cannot mediate access to specific files or network ports, so programs are still free to misuse these resources. Coarsely grained policies such as those currently used by the Mac OS X sandbox provide an all-or-nothing type of protection. Mac OS X programs can, for example, state that they don’t require network connectivity or that they don’t need write access to the file system. However, programs cannot state what specific access they do require, so a program that needs to write to any file is granted write access to all of the user’s files. This type of control is best suited to very simple programs that require very few privileges to run.

Sandboxes that enforce the same rule-based policy for every program effectively provide a limited form of isolation. Rather than providing controls that authorise access to the resources that particular programs need to function, all programs are granted the same privileges. For example, since some iOS applications require sensitive permissions, all applications have been granted these privileges (Seriot, 2010). This highlights the need for each application to be confined based on its specific requirements.

Other approaches, such as overloading user-oriented access controls and PACLs, also have drawbacks. User-oriented access controls are not designed to provide application-oriented access control and, while simple to implement, introduce complications. For example, as Cowan et al. (2000a) discuss, the Unix DAC security model allows all user accounts access to all files that permit access to "others", which results in a complex management task to ensure that applications confined by synthetic user accounts cannot misuse these permissions. In addition, typically all user accounts have permission to use resources such as the network. Overloading user-oriented control schemes makes it difficult to simultaneously provide user restrictions and application-oriented restrictions, therefore this technique is better suited for systems used by only one person. Also, having separate home directories for applications results in a form of isolation-based confinement with potentially complex or permissive file permissions used to specify exceptions and allow applications to share resources. The sub-identities scheme allows for multiple users, but introduces further complexity since additional ACL policies need to be specified to allow sub-identities to access any resources that are not granted to every user.

PACLs and related schemes can protect selected resources from malicious programs. However, labelling every resource with all the applications that require access is very complex and arguably impractical (Berman et al., 1995). Adding a new program involves changing permissions for all the resources required, and creating new files involves a complex decision making process to specify every application that should be authorised to access the files. Due to this complexity, these schemes typically only protect particular files. Therefore this approach cannot comprehensively mitigate the problem of programs running with too much authority. Related approaches pose similar concerns. SubOS is particularly complex since individual policies need to be specified for each remote host files come from, and processes are simultaneously subject to a myriad of policies from all of the files each process has accessed. As Provos (2002) points out, not all exploits are the result of accessing files, and it is sufficient to restrict processes based on the application's needs rather than enforcing policies for all the files the application has accessed.

Finely-grained per-program policies can provide very powerful controls that can specify with precision the privileges granted to particular applications. This approach has the greatest control over individual programs, and can allow confined processes to interact with resources and other processes in authorised ways. However, finely-grained application restrictions typically result in highly detailed, low-level policies that expose the internal complexity of applications and underlying platforms. Translating high level security goals into finely-grained policies has proven to be problematic, typically requiring expertise and knowledge of low level operations and interactions of programs. Also, once constructed, policies can be hard to verify for completeness and correctness (Garfinkel, 2003, Jaeger et al., 2003, Zanin and Mancini, 2004).

System call interposition mechanisms typically specify finely-grained rules in terms of allowed system calls and system call arguments. However, system call interposition systems have been criticised for inherent complexity and design flaws (Garfinkel, 2003, Watson, 2007). The main problem with these types of security policies is that they are so finely grained that the task of managing them is especially complex. This is also partially due to the complexity of the system call interface. In many cases a single operation requires multiple system calls. This complicates policy management and security system implementation as any nontrivial process utilises a vast number of system calls. This means that creating system call interposition policies using some schemes requires additional expert knowledge of low-level system call semantics. Arguably a consequence of this complexity is that system-call interposition schemes are not practical solutions since most users do not have the expertise required to translate high level requirements into meaningful low level policies. Deferring enforcement to user-space also introduces design challenges, such as the risk of race conditions (Spencer et al., 1999, Watson, 2007).

System call anomaly detection monitoring faces challenges similar to that of system call interposition. As a result of the complexity of system calls, anomaly classification is typically based on a simplified view of activity. For example, policy is often based on the system calls themselves (such as read, open, mmap) regardless of what the system calls are actually doing. Due to a simplified view of system activity, anomaly detection can be subject to mimicry attacks, where malicious activity can be disguised by interspersing normal-looking system call sequences (Wagner and Dean, 2001). Furthermore, policy generation faces challenges. Static analysis of binary or source code can produce policy that does not detect malicious activity that follows existing code paths, regardless of the security implications. For example, programs that include logic flaws or have been misconfigured can be exploited without corrupting the code of the executable, and therefore static analysis policies consider this to be normal behaviour. Additionally, static analysis requires access to all source or binary code, which is not always practical in the case of dynamically linked libraries and plug-ins. Learning modes can produce policy that is incomplete, since uncommon code paths may not be profiled, and limited protections are in place while initially profiling. Due to the high rate of false positives, system call anomaly detection systems are often passive and do not enforce policy (Forrest et al., 2008).

Type enforcement (TE) and similar models such as role compatibility (RC) define rules that specify the domains used to confine programs, the types accessible to domains, and the types associated with files. Although these concepts provide forms of abstraction, the policies remain complex and these concepts are arguably not intuitive. Although domains serve as policy abstractions that associate rules with programs, typically each application is assigned a unique domain consisting of complex rules specifying allowed file and domain transitions and interactions with types (similarly labelled objects). SELinux also includes m4 macros, which are generally very low level abstractions granting a number of access rules to specified resources. SELinux policy is complex, hard to comprehend and, despite the maturity of SELinux, few graphical tools have been developed to ease policy management. SELinux is frequently criticised for its complexity (Bratus et al., 2007, Harada et al., 2004, Jaeger et al., 2003, Li et al., 2007, Novell, 2012, Zanin and Mancini, 2004).

Many finely-grained rule-based application-oriented access controls simply specify a list of all the resources programs require. As previously discussed, these include sandboxes such as TRON, higher-level system call interposition schemes such as Software Wrappers, and system wide controls such as AppArmor and PeaPod. All of these systems require complex and typically lengthy policies to restrict applications. Because these policies specify all the low level resources programs need, they expose the complexity of the operating environment and the resource requirements of applications. As previously discussed, the list of resources applications typically need to access can be extremely complicated (Marceau and Joyce, 2005). Creating, managing and reviewing these policies therefore generally requires a relatively high level of expertise beyond that of most end users. Some of these systems include simple policy abstractions that group lists of rules and can be included in policies. However, due to the low level nature of the rules and the complexity of the needs of application, these abstractions represent low level attributes of programs. For example, AppArmor includes abstractions such as ‘fonts’, ‘bash’, ‘nameservice’, ‘dbus’, ‘orbit2’, and ‘aspell’, which still require expertise to understand and utilise. Application virtualization schemes use similar methods to define what actions programs are authorised to perform, with the additional complexity of intra-component rules and source based rules.

MAPbox makes the important contribution of behavioural classes to create reusable associations between fine grained rules and programs (Acharya and Raje, 2000, Raje, 1999). However, under MAPbox programs are limited to a single behaviour class (14 classes identified), and programs that exhibit multiple behaviours (such as most web browsers) are unable to be confined. Also, although it has been acknowledged that overlaps exist between classes, each policy is defined individually. Although author-influenced policies may be useful for simplifying policy management for users and may reduce the threat from vulnerabilities, the threat of untrustworthy authors defining insecure policies remains. An untrusted author will likely choose the most liberal class and parameters that users will accept in order to improve the chances that the program will have all the required authority to act as intended, be that benevolently or maliciously. Also, allowing the user to arbitrarily specify the privileges assigned to behaviours means that even authors who specify the behavioural class of applications do not know what privileges they will be authorised. As discussed, the FBAC model builds on this basic idea and overcomes a number of shortcomings of the MAPbox model. FBAC was designed to overcome the common limitations of rule-based application-oriented access controls, which is discussed further below.

Table 2 presents an approximate summary of the limitations that have been discussed and how they relate to each of the classes of rule-based systems that have been described. As with the previous table, an “X” denotes that the limitation typically applies to this class of system, while “X*” describes a limitation that applies to a lesser extent or with some exceptions. As with the approach used to explain Table 1, as an example one row is described in detail, in this case the “non-discretionary framework combining access control models” row, which describes security frameworks such as SELinux, and RSBAC: schemes that combine a number of label-based non-discretionary access control models. The first row indicates the complexity of the way policy is represented: the policy language is complicated by the multiple models being represented (such as RBAC, TE, and MLS), and is defined in terms of labels, which can be unintuitive for people unfamiliar with the details of the security mechanism and requires the text-based policies for restricting applications to be considered in terms of how the files on the system are labelled. The policies are also typically compiled into a non-human readable binary format before they are enforced. The second row indicates that for this class of security system the text-based policies are typically lengthy and complex, due to the large number of varied resources that applications often require. The third row indicates that the policies expose the complexity of underlying systems; that is, understanding policies requires knowledge of low-level system details such as the filesystem hierarchy standard, security labels, security models, and network ports. As shown in the next two rows, these systems typically mediate access to network as well as file resources. The following row represents the fact that each application's policy is largely independent and the scheme lacks easily reusable policy abstractions, which is a common limitation amongst many rule-based schemes. As discussed, in the case of SELinux m4 macros can provide some limited low-level abstraction, and multiple executables can be assigned to the same domains. However, in practice each application policy is typically defined individually due to the inability for these abstractions to adapt to individual needs of applications. The next row indicates that these schemes can confine legacy applications; that is, existing software. The next two rows indicate that these systems are non-discretionary and are therefore maintained by administrators, and cannot be used by normal users to confine their applications unless they are the administrator of their computer. The final row shows that, like many rule-based application-oriented access controls, non-discretionary frameworks combining access control models typically do facilitate applications sharing the same resources with different permissions.

Table 2: Limitations of rule-based application-oriented access controls

Scheme category	***General properties***	***Example***	Complex policy representation	Complex and/or lengthy application policies	Application policy exposes the complexity of underlying systems	Lack networking and/or IPC mediation	Lack of file mediation	Does not scale well, lacks reusable abstractions for simplifying policy	Cannot be applied to legacy applications	Does not suit use by normal users to protect themselves with discretionary controls	Does not suit use by admin to enforce non-discretionary controls	Does not support simple workflow of editing the same files between sandboxes, while limiting potential damage
System call interposition	Fine-grained rules, typically in terms of system calls	Systrace	X	X	X			X			X*	

Scheme category	***General properties***	***Example***	Complex policy representation	Complex and/or lengthy application policies	Application policy exposes the complexity of underlying systems	Lack networking and/or IPC mediation	Lack of file mediation	Does not scale well, lacks reusable abstractions for simplifying policy	Cannot be applied to legacy applications	Does not suit use by normal users to protect themselves with discretionary controls	Does not suit use by admin to enforce non-discretionary controls	Does not support simple workflow of editing the same files between sandboxes, while limiting potential damage
System call anomaly detection	Detects anomalies in system call sequences	Callgraph	X	X	X	X	X	X				
Application virtualization	Portable application bytecode run by a VM with a security policy	Java	X	X	X			X	X		X	
Fine-grained rules sandbox	Fine-grained rules, in terms of resources	Mac OS X Sandbox		X	X			X*			X	
One-size-fits-all fine-grained policy	The same detailed policy is applied to all apps	iOS				X*	X*			X	X	X
Artificial user accounts	Applications are confined by assigning them "user" accounts	Rainbow				X		X		X*		X
Coarsely-grained privileges	Flag-based privileges, assigned to applications	Linux capabilities				X*	X*	X		X		
Artificial user accounts + coarsely-grained privileges	Applications are assigned user accounts and flag-based privileges by authors	Android	X*			X*		X		X* (single user)		X*
ACL extensions	Files are labelled with a list of the programs that allowed access	PACLs		X	X	X		X			X	
Type enforcement	Label-based permissions between types (objects) and domains (processes)	DTE	X	X	X			X*		X		

Scheme category	***General properties***	***Example***	Complex policy representation	Complex and/or lengthy application policies	Application policy exposes the complexity of underlying systems	Lack networking and/or IPC mediation	Lack of file mediation	Does not scale well, lacks reusable abstractions for simplifying policy	Cannot be applied to legacy applications	Does not suit use by normal users to protect themselves with discretionary controls	Does not suit use by admin to enforce non-discretionary controls	Does not support simple workflow of editing the same files between sandboxes, while limiting potential damage
Non-discretionary framework combining access control models	Complex label-based framework combining multiple access control models	SELinux	X	X	X			X*		X		
Fine-grained system-wide rules	Fine-grained rules, in terms of resources, applied automatically when the application starts	AppArmor		X	X			X*		X		
Hybrid container-based virtualisation + fine-grained rules sandbox	Isolation-based virtualisation with additional rules for contained applications	PeaPod		X*	X			X		X*		
Functionality-based	Applications are confined based on the features they are expected to perform	FBAC										

As illustrated in the table, many application-confinement schemes suffer from policy complexity. Due to this complexity, using many of these systems it is impractical to specify policy *a priori;* that is, without analysing the activity of each application. For this reason many finely-grained rule-based mechanisms rely on learning modes to automatically generate policy based on the observation of programs operating. The program to be confined is executed and all security sensitive actions are logged. These logged activities are then typically assumed to be standard behaviour and are used to create a policy for confining the application. Since only primitive policy abstractions are used (if any), resulting policies are often large, complex and difficult to review, especially with system call interposition schemes. As previously mentioned, examples of security mechanisms that include learning modes include: AppArmor (Cowan et al., 2000a), Systrace (Provos, 2002), LIDS (LIDS, 2012), SELinux (audit2allow tool) (Habib, 2007), TOMOYO (Harada et al., 2004) and grsecurity (Spengler, 2012).

Learning modes develop policies either while the program is confined or when it is unconfined. Using systems such as AppArmor, incrementally building policy while the program is confined by the rules being developed can involve a large number of iterations due to program failures where the program is unable to continue without access to required privileges. Alternatively, building policy while the program is not confined poses security risks if the software acts maliciously during this process. Dynamic policy authorisation, such as used by PULSE (Murray and Grove, 2008), can somewhat alleviate this risk by waiting for user confirmation before authorising the program access to each resource.

Although learning modes can make policy generation easier, the policy requires review to ensure that the logged behaviour does in fact represent legitimate behaviour. However, reviewing policy requires expertise and meticulous inspection of the generated policy, and even simple applications can involve a very large number of rules. If the application is carrying out malicious behaviour at the time of logging, and this is not detected at review, then the application-oriented access control will not prevent the application from continuing these malicious activities. Also, if the application does not access all the resources it requires during policy generation, in the future it may be restricted from carrying out its legitimate tasks, or require excessive user interaction and policy refinement. This makes it difficult to learn behaviour without exposing the application to the production environment, and possibly attack, during policy generation.

Using previous finely-grained application restrictions, individual application-oriented access control policies typically apply primarily to only one specific application or program. The work involved in constructing policies for all applications utilised is considerable; the management task generally increases more or less in proportion to the number of applications being confined.

Furthermore, most schemes either provide user-specified policies (DAC), administrator-specified policies (MAC), author-specified policies, or software-specified policies. Each system can address one of these security goals. However, these are all valid restrictions that should be capable of being combined to provide defence-in-depth and to enforce the security goals of each party. Using the schemes described, enforcing these goals simultaneously requires the management of multiple security systems.

The FBAC approach demonstrably overcomes common limitations in rule-based controls. Application policies are defined in terms of high-level abstractions, which results in relatively simple (yet fine-grained) policies mediating access to files and the network while abstracting the details of the underlying platform (Schreuders et al., 2011b). FBAC enforces discretionary and non-discretionary policy defined by a number of parties. The model is designed to provide controls for users and administrators to enforce security goals. Policies have been created to demonstrate that this approach can be applied to legacy applications. FBAC also facilitates the development of policies *a priori* using high-level security concepts, eliminating the need for learning modes (Schreuders et al., 2011c).

6. Application Restrictions and Usability

The result of all these limitations is primarily a usability problem. Despite the fact that it has long been acknowledged as an important aspect in the design of security systems (Saltzer and Schroeder, 1975), usability received little attention in the literature until the importance of applying human-computer interaction (HCI) techniques to the field of computer security was emphasised by studies which demonstrated that poorly designed security user interfaces resulted 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 numerous publications related to computer security usability, very little research has investigated or addressed the usability issues associated with application-oriented access controls.

Research has explored usability within the wider field of access control and policy specification. A study by Motiee et al. (2010) identified general problems with existing user-oriented schemes. The conflict between usability and access control policy complexity has been acknowledged a number of times, and methods for improving the usability of policy specification have been explored (Brodie et al., 2006, Cao and Iverson, 2006, Johnson et al., 2010a, Johnson et al., 2010b, Karat et al., 2005, Reeder et al., 2008, Reeder et al., 2007, Zurko et al., 1999, Zurko and Simon, 1996). A number of policy authoring techniques have been developed by usable security researchers to overcome usability and policy complexity problems. The Adage (Zurko et al., 1999) and MAP (Zurko and Simon, 1996) projects were designed to provide usable RBAC for distributed organisations. SPARCLE is a natural language policy management tool that guides users through the task of specifying policy (Brodie et al., 2006, Karat et al., 2005). Expandable Grids is a graphical method of managing and viewing policy using a hierarchical matrix (Reeder et al., 2008), and Intentional Access Management produces user-oriented DAC ACL policy rules based on low-level access goals of end users (Cao and Iverson, 2006). Johnson et al. (2010a, 2010b) have recently proposed techniques for improving the usability of guided natural language policy specification using policy templates. Findings of studies such as these have supported the idea that abstraction improves the usability of access controls and eases policy specification.

However, little research has explored the usability of application-oriented access control schemes. The isolation-based Apiary scheme and the data-centric FileMonster scheme have been the subjects of limited usability studies (Potter and Nieh, 2010, Schmid et al., 2002). Potter and Nieh (2010) conducted a simple user study with 24 participants to evaluate the ability of users to use applications in the Apiary environment. The use of the Apiary desktop was compared with Xfce, a lightweight Linux desktop environment. Usability evaluation was simply measured by time-on-task and participants were "asked to rate their perceived ease of use of each environment on a scale of 1 to 5". Participants were also asked some other questions "including, would the Apiary environment be an environment they could imagine using full time and would it be an environment they would prefer to use full time if it would keep their desktop secure". Results were reported as affirmative, although no inferential statistics were employed. Schmid et al. (2002) also conducted a simple evaluation of the FileMonster scheme that measured the number of times the tool required user interaction.

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 study involved 10 participants utilising Polaris to carry out a number of tasks. Their success at the tasks was evaluated and perceived usability was 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), and the study concluded that further work was necessary to improve the usability of Polaris.

A within-subjects usability study was performed to compare the usability of SELinux, AppArmor, and FBAC-LSM and to identify factors that have an effect on the usability of application-oriented access controls (Schreuders et al., 2011a, Schreuders et al., 2012). Each of the 39 participants used all three systems to confine a web browser and a simulation of a Trojan horse posing as a game. Perceived usability was measured using the SUS, and policy success was measured using a score based on security-sensitive resources available after attempting the tasks. Inferential statistics indicated that significant differences in usability existed between the systems. FBAC received a SUS score of 70.21, AppArmor scored 54.93, and SELinux scored 34.58 (Schreuders et al., 2011a). Risk exposure scores followed a similar trend with FBAC-LSM outperforming AppArmor, which in turn successfully provided more protection than SELinux. Qualitative analysis suggested various factors had an effect on usability, including the kind of policy abstractions provided (Schreuders et al., 2012).

7. Recommendations and Future Work

This review shows that there are many opportunities for future research in the field of application restrictions. This paper has identified a number of common limitations in application-oriented access controls. These identified limitations can serve as means of assessing existing schemes for suitability for various purposes, can provide considerations for the design of new systems, and can form the subject of future research.

In general, it was found that those schemes that were designed with usability in mind were able to overcome many common limitations. Consequently, we recommend usability be considered further during the design of application-oriented access control schemes.

Work by the authors has taken the approach that the usability problems with rule-based application oriented access controls can be overcome by reusable policy abstractions representing high-level goals. Previous approaches either lacked policy abstractions entirely, or utilised abstractions that do not provide the flexibility and reusability necessary to represent high-level goals and abstract low-level details from application restriction policies. Most application-oriented access control models simply associate a list of privileges directly with an application. Others provide policy abstractions that group privileges and can be reused only to a limited extent. With most isolation sandboxes, the only policy abstraction available is an isolated container that groups subjects with the objects they can interact with; any exceptions are typically specified individually. Models that restrict access to shared resources are generally either devoid of policy abstraction (privileges are listed for each application), or are defined in terms of large monolithic self-contained policy abstractions, such as is the case with DTE domains and RC roles. These policy abstractions typically have limited reusability as they only apply to the specific needs of an individual application. These models lack the flexibility required to apply the policy abstractions to different applications with shared high-level goals. Unless the applications have exactly the same privilege requirements, the abstractions cannot be reused. Results of our research have supported the idea that policy abstraction can facilitate overcoming various limitations, and therefore we recommend serious consideration of policy abstraction when designing application-oriented access controls.

While many different schemes for application-oriented controls currently exist (as illustrated throughout this paper), a conflict exists between developing new innovative and more usable methods of control, and improving and standardising on existing systems which have received limited adoption. The recent popularity of mobile device operating systems, such as Android and iOS, has demonstrated that it is possible to provide more widespread use of application restrictions by standardising on controls. This is a contentious issue for the Linux security community, where a number of alternatives exist. The authors encourage the design and adoption of usable controls that suit legacy applications on commodity operating systems to provide security controls to limit the privilege assigned to applications.

8. Conclusions

This paper has presented an overview of the motivation for application-oriented access controls, and described in depth the application-oriented access controls that have been proposed in the literature. Unlike user-oriented access controls, application-oriented access controls specify policy primarily in terms of programs or groups of programs, rather than users or groups of users. By identifying the privileges required by each application and restricting their actions to those privileges, the ability of applications to act beyond their legitimate purpose is limited, and attempts to act maliciously are restricted. Application-oriented schemes include models that isolate programs to only access the set of resources available from within the isolated environment, and rule-based schemes that provide controls over shared resources while allowing applications to access the same resources in a restricted manner.

Both of these main approaches pose a number of challenges. Isolation does not suit typical user workflows, can result in redundancy of resources, and requires frequent circumvention of the security model to be practical. Rule-based schemes typically result in complex policies that are hard to specify, review, and manage.

There are many opportunities for future research, and there is a need for usable application-oriented security solutions. Although application-oriented access controls can provide substantial security benefits by restricting the activities of individual applications, to date adoption remains relatively rare and targeted, arguably due to the inadequate usability of these schemes.

References

Acharya A, Raje M. MAPbox: Using Parameterized Behavior Classes to Confine Applications. In

9th USENIX Security Symposium. Denver, CO, USA: USENIX Association, 2000.

Alexandrov A, Kmiec P, Schauser K, University of California, Technical Report: "Consh: Confined Execution Environment for Internet Computations," Santa Barbara, CA, USA, 1998.

Anderson J, Bonneau J, Stajano F. Inglorious Installers: Security in the Application Marketplace. In 9th Workshop on the Economics of Information Security (WEIS 2010). Cambridge, MA, USA: Harvard University, 2010.

Badger L. A Domain and Type Enforcement UNIX Prototype. Computing Systems. 1996; 9, 1: pp. 47-83.

Badger L, Sterne DF, Sherman DL, Walker KM, Haghighat SA. Practical Domain and Type Enforcement for UNIX. In 16th IEEE Symposium on Security and Privacy. Oakland, CA, USA: IEEE Computer Society, 1995, pp. 66-77.

Balfanz D, Simon DR. WindowBox: A Simple Security Model for the Connected Desktop. In 4th USENIX Windows Systems Symposium. Seattle, WA, USA: USENIX Association, 2000, pp. 37–48.

Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, et al. Xen and the Art of Virtualization. In 19th ACM Symposium on Operating Systems Principles. Lake George, NY, USA: ACM Press, 2003, pp. 164-77.

Barrera D. Secure Software Installation on Smartphones. IEEE Security & Privacy. 2011; 9, 3: pp. 42-8.

Bell D, LaPadula L, MITRE Corporation, Technical Report: ESD-TR-75-306 "Secure Computer System: Unified Exposition and Multics Interpretation," Bedford, MA, USA, 1975.

Bellard F. QEMU, a Fast and Portable Dynamic Translator. In FREENIX Track, USENIX Annual Technical Conference. Anaheim, CA, USA: USENIX Association, 2005, pp. 41–6.

Bellovin SM, Cohen C, Havrilla J, Hernan S, King B, Lanza J, et al. Results of the Security in ActiveX Workshop. In CERT Coordination Center Security in ActiveX Workshop. Pittsburgh, PA, USA: CERT, 2000.

Ben-Cohen O, Wool A. Korset: Automated, Zero False-Alarm Intrusion Detection for Linux. In The Linux Symposium. Ottawa, ON, Canada, 2008.

Benantar M. Access Control Systems: Security, Identity Management and Trust Models. New York: Springer; 2006.

Berman A, Bourassa V, Selberg E. TRON: Process-Specific File Protection for the UNIX Operating System. In Winter USENIX Conference. New Orleans, LA, USA: USENIX Association, 1995, pp. 165-75.

Biba KJ, Mitre Corp, National Technical Information Service, Technical Report: MTR-3153 "Integrity Considerations for Secure Computer Systems," Bedford, MA, USA, 1977.

Bishop M. An Overview of Computer Viruses in a Research Environment. In 4th Annual Computer Virus and Security Conference. New York, NY, USA, 1991, pp. 111-44.

Bishop M, Department of Computer Science, University of California at Davis, Technical Report: CSE-95-8 "A Taxonomy of UNIX System and Network Vulnerabilities," Davis, CA, USA, 1995.

Blanchard B, Coelho P, Hazuka M, Petru J, Thitayanun T, Almond C, IBM Corp., Technical Report: 073848654X "Introduction to Workload Partition Management in IBM AIX Version 6.1," 2007.

Boebert WE, Kain RY. A Practical Alternative to Hierarchical Integrity Policies. In 8th National Computer Security Conference. Gaithersburg, MD, USA: NIST, 1985, pp. 18-27.

Bratus S, Ferguson A, McIlroy D, Smith S. Pastures: Towards Usable Security Policy Engineering. In 2nd International Conference on Availability, Reliability and Security. Vienna, Austria, 2007, pp. 1052-9.

Brewer DFC, Nash MJ. The Chinese Wall Security Policy. In 10th IEEE Symposium on Security and Privacy. Oakland, CA, USA: IEEE Computer Society, 1989, pp. 206–14.

Brodie CA, Karat C-M, Karat J. An Empirical Study of Natural Language Parsing of Privacy Policy Rules Using the SPARCLE Policy Workbench. In 2nd Symposium on Usable Privacy and Security (SOUPS). Pittsburgh, PA, USA: ACM Press, 2006.

Brooke J. SUS: A Quick and Dirty Usability Scale. In: Jordan PW, Thomas B, Weerdmeester BA, McClelland IL, editors. Usability Evaluation in Industry. London: Taylor & Francis; 1996. p. 189-94.

Callas J. Hacking PGP (Presentation). In Black Hat Europe. Amsterdam, Netherlands, 2005.

Canonical. AppArmor Linux Application Security Framework. https://launchpad.net/apparmor, Accessed 2012.

Cao X, Iverson L. Intentional Access Management: Making Access Control Usable for End-users. In 2nd Symposium on Usable Privacy and Security (SOUPS). Pittsburgh, PA, USA: ACM Press, 2006.

Christodorescu M, Jha S. Testing Malware Detectors. In ACM SIGSOFT International

Symposium on Software Testing and Analysis. Boston, MA, USA: ACM Press, 2004.

Christodorescu M, Jha S, Maughan D, Song D, Wang C. Malware Detection (Advances in Information Security). New York: Springer-Verlag, Inc.; 2006.

Christodorescu M, Jha S, Seshia SA, Song D, Bryant RE. Semantics-Aware Malware Detection. In 26th IEEE Symposium on Security and Privacy. Berkeley, CA, USA: IEEE Computer Society, 2005.

Computer Associates, Technical Report: "eTrust Access Control for UNIX," 2001.

Cowan C, Barringer M, Beattie S, Kroah-Hartman G. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In 10th USENIX Security Symposium. Washington, DC, USA: USENIX Association, 2001a.

Cowan C, Beattie S, Kroah-Hartman G, Pu C, Wagle P, Gligor V. SubDomain: Parsimonious Server Security. In USENIX 14th Systems Administration Conference. New Orleans, LA, USA: USENIX Association, 2000a.

Cowan C, Beattie S, Wright C, Kroah-Hartman G. RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities. In 10th USENIX Security Symposium. Washington, DC, USA: USENIX Association, 2001b.

Cowan C, Wagle P, Pu C, Beattie S, Walpole J. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In DARPA Information Survivability Conference and Exposition (DISCEX). Hilton Head, SC, USA: DARPA, 2000b, pp. 227-37.

Cranor L, Garfinkel S. Security and Usability: Designing Secure Systems That People Can Use: O'Reilly Media, Inc.; 2005.

Dagon D, Gu G, Lee CP, Lee W. A Taxonomy of Botnet Structures. In 23rd Annual Computer Security Applications Conference (ACSAC). Miami Beach, FL, USA: IEEE Computer Society, 2007, pp. 325-39.

Department of Defense, United States Government DOD 5200.28-STD "Trusted Computer Security Evaluation Criteria," USA, 1985.

des Ligneris B. Virtualization of Linux Based Computers: The Linux-VServer Project. In 19th International Symposium on High Performance Computing Systems and Applications (HPCS'05). Guelph, Ontario, Canada: Springer, 2005, pp. 340-6.

DeWitt AJ, Kuljis J. Aligning Usability and Security: A Usability Study of Polaris. In 2nd Symposium on Usable Privacy and Security (SOUPS). Pittsburgh, PA, USA: ACM Press, 2006, pp. 1-7.

Dhamankar R, Dausin M, Eisenbarth M, King J, SANS, Technical Report: "The Top Cyber Security Risks," 2009.

Dike J. A User-mode Port of the Linux Kernel. In 4th Annual Linux Showcase and Conference. Oakland, CA, USA: USENIX Association, 2000, pp. 63–72.

Edge C, Barker W, Hunter B, Sullivan G. Enterprise Mac Security: Mac OS X Snow Leopard, Second Edition: Apress; 2010.

Enck W, McDaniel P, Jaeger T. PinUP: Pinning User Files to Known Applications. In 24th Annual Computer Security Applications Conference (ACSAC). Anaheim, CA, USA, 2008, pp. 55-64.

Enck W, Ongtang M, McDaniel P. Understanding Android Security. IEEE Security & Privacy. 2009; 7, 1: pp. 50-7.

Enck W, Rueda S, Schiffman J, Sreenivasan Y, Clair LS, Jaeger T, et al. Protecting Users From Themselves. In ACM Workshop on Computer Security Architecture. Fairfax, Virginia, USA: ACM Press, 2007.

Farkas S. The Silverlight Security Model. http://blogs.msdn.com/b/shawnfa/archive/2007/05/09/the-silverlight-security-model.aspx, 2007.

Ferraiolo D, Kuhn R. Role-Based Access Control. In 15th National Computer Security Conference. Baltimore, MD, USA: NIST, 1992, pp. 554-63.

Ford R, Allen WH. Malware Shall Greatly Increase. IEEE Security & Privacy. 2009; 7, 6: pp. 69-71.

Forrest S, Hofmeyr S, Somayaji A. The Evolution of System-call Monitoring. In 24th Annual Computer Security Applications Conference (ACSAC). Anaheim, California, USA: IEEE Computer Society, 2008, pp. 418-30.

Forrest S, Perelson AS, Allen L, Cherukuri R. Self-nonself Discrimination in a Computer. In 15th IEEE Symposium on Security and Privacy. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 202-12.

Fraser T. LOMAC: Low Water-mark Integrity Protection for COTS Environments. In 21st IEEE Symposium on Security and Privacy. Berkeley, CA, USA: IEEE Computer Society, 2000, pp. 230-45.

Fraser T. LOMAC: MAC You Can Live With. In FREENIX Track, USENIX Annual Technical

Conference. Boston, MA, USA: USENIX Association, 2001.

Fraser T, Badger L, Feldman M. Hardening COTS Software with Generic Software Wrappers. In 20th IEEE Symposium on Security and Privacy. Oakland, CA, USA: IEEE Computer Society, 1999, pp. 2-16.

Garfinkel S, Spafford G, Schwartz A. Practical Unix and Internet Security. 3 ed. Sebastopol, CA, USA: O'Reilly; 2003.

Garfinkel T. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In 10th Network and Distributed System Security Symposium. San Diego, CA, USA: Stanford University, 2003, pp. 163-76.

Garfinkel T, Pfaff B, Rosenblum M. Ostia: A Delegating Architecture for Secure System Call Interposition. In Network and Distributed Systems Security Symposium. San Diego, CA, USA, 2004.

Ghormley DP, Petrou D, Rodrigues SH, Anderson TE. SLIC: An Extensibility System for Commodity Operating Systems. In USENIX Annual Technical Conference. New Orleans, LA, USA: USENIX Association, 1998.

Giffin JT, Jha S, Miller BP. Detecting Manipulated Remote Call Streams. In 11th USENIX Security Symposium. San Francisco, CA, USA: USENIX Association, 2002.

Goldberg I, Wagner D, Thomas R, Brewer EA. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In 6th USENIX Security Symposium. San Jose, CA, USA: USENIX Association, 1996.

Gong L, Mueller M, Prafullchandra H, Schemers R. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In USENIX Symposium on Internet Technologies and Systems. Monterey, CA, USA: Prentice Hall PTR, 1997.

Governa F. Fighting Spyware with Mandatory Access Control in MS Windows Vista: A Concept for Fighting Spyware in the Microsoft Windows Vista OS: VDM Verlag; 2009.

Govindavajhala S, Appel AW. Windows Access Control Demystified. http://packetstorm.security-guide.de/papers/attack/winval.pdf, 2006.

Habib I. Creating SELinux Policies Simplified. Linux Journal. 2007; 154.

Hallyn SE, Kearns P. Domain and Type Enforcement for Linux. In 4th Annual Linux Showcase and Conference. Atlanta, GA, USA, 2000, pp. 247–60.

Hallyn SE, Morgan AG. Linux Capabilities: Making Them Work. In The Linux Symposium. Ottawa, ON, Canada, 2008.

Harada T, Horie T, Tanaka K. Task Oriented Management Obviates Your Onus on Linux. In Linux Conference 2004. Tokyo, Japan, 2004.

Hardy N. The KeyKOS Architecture. Operating Systems Review. 1985; 19, 4: pp. 8-25.

Hardy N. The Confused Deputy: Or Why Capabilities Might Have Been Invented. ACM SIGOPS Operating Systems Review. 1988; 22, 4: pp. 36-8.

Hitchings J. Deficiencies of the Traditional Approach to Information Security and the Requirements for a New Methodology. Computers & Security. 1995; 14, 5: pp. 377-83.

Hofmeyr SA, Forrest S, Somayaji A. Intrusion Detection Using Sequences of System Calls. Journal of Computer Security. 1998; 6, 3: pp. 151-80.

IEEE, Institute of Electrical and Electronics Engineers, Inc., Withdrawn Standards Draft: PSSG/D17 "POSIX 1003.1e," 1997.

Ioannidis S, Bellovin SM, Smith JM. Sub-operating Systems: A New Approach to Application Security. In 10th Workshop on ACM SIGOPS European Workshop. Saint-Emilion, France: ACM Press, 2002.

Jaeger T, Sailer R, Zhang X. Analyzing Integrity Protection in the SELinux Example Policy. In 12th USENIX Security Symposium, 2003, pp. 59–74.

Jain K, Sekar R. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In ISOC Network and Distributed Systems Symposium (NDSS). San Diego, CA, USA, 2000.

Johnson M, Karat J, Karat C-M, Grueneberg K. Optimizing a Policy Authoring Framework for Security and Privacy Policies. In 6th Symposium on Usable Privacy and Security (SOUPS). Redmond, Washington, DC, USA: ACM Press, 2010a.

Johnson M, Karat J, Karat CM, Grueneberg K. Usable Policy Template Authoring for Iterative Policy Refinement. In IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2011). Fairfax, VA, USA: IEEE Computer Society, 2010b.

Jones MB. Interposition Agents: Transparently Interposing User Code at the System Interface. ACM SIGOPS Operating Systems Review. 1993; 27, 5: pp. 80-93.

Kamp P-H, Watson R. Jails: Confining the Omnipotent Root. In 2nd International System Administration and Networking Conference (SANE 2000). Maastricht, The Netherlands, 2000.

Kamp P-H, Watson R. Building Systems to be Shared Securely. ACM Queue. 2004; 2, 5: pp. 42-

51.

Karat J, Karat C-M, Brodie C, Feng J. Privacy in Information Technology: Designing to Enable Privacy Policy Management in Organizations. International Journal of Human-Computer Studies. 2005; 63, 1-2: pp. 153-74.

Kato K, Oyama Y. SoftwarePot: An Encapsulated Transferable File System for Secure Software Circulation. In Mext-NSF-JSPS International Conference on Software Security: Theories and Systems. Tokyo, Japan: Springer-Verlag, 2002.

Kiriansky V, Bruening D, Amarasinghe S. Secure Execution Via Program Shepherding. In 11th USENIX Security Symposium. Berkeley, CA, USA: USENIX Association, 2002.

Kotadia M. Eighty Percent of New Malware Defeats Antivirus. http://www.zdnet.com.au/news/security/soa/Eighty_percent_of_new_malware_defeats_antivirus/0,2000061744,39263949,00.htm, 2006.

Krell E, Krishnamurthy B. COLA: Customized Overlaying. In USENIX Winter Conference. San Francisco, CA, USA: USENIX Association, 1992.

Krsti I, Garfinkel SL. Bitfrost: The One Laptop Per Child Security Model. In 3rd Symposium on Usable Privacy and Security (SOUPS). Pittsburgh, PA, USA: ACM Press, 2007, pp. 132-42.

Kurchuck A, Keromytis AD. Recursive Sandboxes: Extending Systrace to Empower Applications. In 19th IFIP International Information Security Conference (SEC). Toulouse, France, 2004.

Lam LC, Chiueh TC. Automatic Extraction of Accurate Application-specific Sandboxing Policy. In IEEE Military Communications Conference (MILCOM). Atlantic City, NJ, USA: IEEE Computer Society, 2005, pp. 713-9.

Landwehr CE, Bull AR, McDermott JP, Choi WS. A Taxonomy of Computer Program Security Flaws. ACM Computing Surveys (CSUR). 1994; 26, 3: pp. 211-54.

Lee P, Necula G. Research on Proof-Carrying Code for Mobile-Code Security. In DARPA Workshop on Foundations for Secure Mobile Code. Pittsburgh, PA, USA: United States Department of Defense, Defense Advanced Research Projects Agency (DARPA), 1997.

Lessard P, SANS Institute, InfoSec Reading Room Report: "Linux Process Containment – A Practical Look at chroot and User Mode Linux," 2003.

Levy JY, Demailly L, Ousterhout JK, Welch BB. The Safe-Tcl Security Model. In USENIX Annual Technical Conference. New Orleans, LA, USA: USENIX Association, 1998.

Li N, Mao Z, Chen H, Purdue University, Center for Education and Research in Information Assurance and Security, Technical Report: CERIAS TR 2006-38 "Host Integrity Protection Through Usable Non-discretionary Access Control," West Lafayette, IN, USA, 2007.

Liang Z, Sun W, Venkatakrishnan VN, Sekar R. Alcatraz: An Isolated Environment for Experimenting with Untrusted Software. ACM Transactions on Information and System Security (TISSEC). 2009; 12, 3: pp. 1-37.

Liang Z, Venkatakrishnan VN, Sekar R. Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs. In 19th Annual Computer Security Applications Conference (ACSAC). Las Vegas, NV, USA, 2003, pp. 182-91.

LIDS. Linux Intrusion Detection System (LIDS) - Secure Linux System. http://www.lids.org, Accessed 2012.

Lipner SB. Non-Discretionary Controls for Commercial Applications. In 3rd IEEE Symposium on Security and Privacy. Oakland, CA, USA: IEEE Computer Society, 1982, pp. 2-10.

Loscocco P, Smalley S. Integrating Flexible Support for Security Policies into the Linux Operating System. In FREENIX Track: 2001 USENIX Annual Technical Conference. Boston, MA, USA: USENIX Association, 2001a, pp. 29-42.

Loscocco P, Smalley S. Meeting Critical Security Objectives with Security-Enhanced Linux. In Ottawa Linux Symposium. Berkeley, CA, USA, 2001b.

LXC. lxc Linux Containers. http://lxc.sourceforge.net/, Accessed 2012.

Madnick SE, Donovan JJ. Application and Analysis of the Virtual Machine Approach to Information Security. In ACM Workshop on Virtual Computer Systems. Cambridge, MA, USA: Harvard University, 1973, pp. 210-24.

Manoel E, Budai V, Buecker A, Edwards D, Samson A, IBM Corporation, Technical Report: SG24-5520-00 "Enterprise Security Management with Tivoli," 2000.

Mansfield-Devine S. The Promise of Whitelisting. Network Security. 2009; 7: pp. 4-6.

Marceau C, Joyce R. Empirical Privilege Profiling. In Workshop on New Security Paradigms (NSPW). Lake Arrowhead, CA, USA: ACM Press, 2005, pp. 111-8.

Marlinspike M. More Tricks For Defeating SSL In Practice (Presentation). In Black Hat USA. Las Vegas, NV, USA, 2009.

McCauley T. Understanding the Security Changes in Flash Player 10. http://www.addlifetotheweb.com/devnet/flashplayer/articles/fplayer10_security_changes.html, 2008.

Mettler A, Wagner D, University of California at Berkeley, Technical Report: UCB/EECS-2008-91 "The Joe-E Language Specification," Berkeley, CA, USA, 2008.

Microsoft. Microsoft Security Bulletin MS01-017: Erroneous VeriSign-Issued Digital Certificates Pose Spoofing Hazard. http://www.microsoft.com/technet/security/bulletin/MS01-017.mspx, 2001.

Microsoft. Technet Report: Introducing AppLocker. http://technet.microsoft.com/en-us/library/dd560656(WS.10).aspx, 2008a.

Microsoft. Technet Report: Software Restriction Policies (SRP). http://technet.microsoft.com/en-us/library/dd348653(WS.10).aspx, 2008b.

Microsoft. Microsoft Virtual Server 2005 R2. http://www.microsoft.com/windowsserversystem/virtualserver/default.aspx, Accessed 2012a.

Microsoft. TechNet Essay: 10 Immutable Laws of Security. http://technet.microsoft.com/en-au/library/cc722487.aspx, Accessed 2012b.

Microsoft. Technet Report: Introduction to ActiveX Controls. http://msdn.microsoft.com/en-us/library/aa751972(VS.85).aspx, Accessed 2012c.

Miller MS. Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. Baltimore, MD, USA: Johns Hopkins University; 2006.

Miller MS, Tulloh B, Shapiro JS. The Structure of Authority: Why Security Is Not a Separable Concern. In Multiparadigm Programming in Mozart/Oz (MOZ). Charleroi, Belgium: Springer-Verlag, 2004, pp. 2-20.

Moser A, Kruegel C, Kirda E. Limits of Static Analysis for Malware Detection. In 23rd Annual Computer Security Applications Conference (ACSAC). Miami Beach, FL, USA, 2007.

Motiee S, Hawkey K, Beznosov K. Do Windows Users Follow the Principle of Least Privilege?: Investigating User Account Control Practices. In 6th Symposium on Usable Privacy and Security (SOUPS). Redmond, Washington, DC, USA: ACM Press, 2010.

Murray AP, Grove DA. PULSE: A Pluggable User-space Linux Security Environment. In 6th Australasian Conference on Information Security. Wollongong, NSW, Australia: Australian Computer Society, Inc., 2008.

Nachenberg C. Computer Virus-antivirus Coevolution. Communications of the ACM. 1997; 40, 1: pp. 46-51.

Necula GC. Proof-carrying Code. In 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Paris, France: ACM Press, 1997.

Necula GC, Lee P. The Design and Implementation of a Certifying Compiler. ACM SIGPLAN Notices. 2004; 39, 4: pp. 612-25.

Novell. AppArmor and SELinux Comparison. http://www.novell.com/linux/security/apparmor/selinux_comparison.html, Accessed 2012.

OLPC. Rainbow. http://wiki.laptop.org/go/Rainbow, Accessed 2012.

Oracle. VirtualBox. http://www.virtualbox.org/, Accessed 2012.

Ott A. The Role Compatibility Security Model. In 7th Nordic Workshop on Secure IT Systems (NordSec). Karlstad, Värmland, Sweden, 2002.

Ott A, Ievlev S, Heinrich WK. The Rule Set Based Access Control (RSBAC) Linux Kernel Security Extension. In 8th International Linux Kongress. Enschede, Netherlands, 2001.

Pandey R, Hashii B. Providing Fine-Grained Access Control for Java Programs. In 13th European Conference on Object-Oriented Programming (ECOOP). Lisbon, Portugal: Springer Berlin / Heidelberg, 1999, p. 668.

Parallels Inc. OpenVZ - Server Virtualization Open Source Project. http://openvz.org/, Accessed 2012a.

Parallels Inc. Virtual PC, Virtual Machine and Multiple Operating System Solutions by Parallels, Inc. http://www.parallels.com/, Accessed 2012b.

Parallels Inc. Virtuozzo Containers. http://www.parallels.com/au/products/pvc46/, Accessed 2012c.

Patcha A, Park J-M. An Overview of Anomaly Detection Techniques: Existing Solutions and Latest Technological Trends. Computer Networks: The International Journal of Computer and Telecommunications Networking. 2007; 51, 12: pp. 3448-70.

Piessens F. A Taxonomy of Causes of Software Vulnerabilities in Internet Software. In The 13th International Symposium on Software Reliability Engineering. Annapolis, MD, USA: IEEE Computer Society, 2002, pp. 47-52.

Positive Software Corporation. Free Virtual Private Server Solution. http://www.freecode.com/projects/freevps, Accessed 2012.

Potter S, Nieh J. Apiary: Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems. In USENIX Annual Technical Conference. Boston, MA, USA: USENIX Association, 2010.

Potter S, Nieh J, Selsky M. Secure Isolation of Untrusted Legacy Applications. In 21st Large Installation System Administration Conference (LISA '07). Dallas, TX, USA: USENIX Association, 2007, pp. 117-30.

Potter S, Nieh J, Subhraveti D, Columbia University, Technical Report: CUCS-005-04 "Secure Isolation and Migration of Untrusted Legacy Applications," New York, NY, USA, 2004.

Provos N. Improving Host Security with System Call Policies. In 12th USENIX Security Symposium. Washington, DC, USA: USENIX Association, 2002.

Raje M. TRCS 99-12: Behavior-based Confinement of Untrusted Applications. Santa Barbara: University of Calfornia; 1999.

Reeder RW, Bauer L, Cranor LF, Reiter MK, Bacon K, How K, et al. Expandable Grids for Visualizing and Authoring Computer Security Policies. In 26th Annual SIGCHI Conference on Human Factors in Computing Systems. Florence, Italy: ACM Press, 2008.

Reeder RW, Karat C-M, Karat J, Brodie C. Usability Challenges in Security and Privacy Policy-authoring Interfaces. In 11th IFIP TC 13 International Conference on Human-computer Interaction. Rio de Janeiro, Brazil: Springer-Verlag, 2007.

Returnil. Returnil Virtual System. http://www.returnilvirtualsystem.com/returnil-system-safe, Accessed 2012.

Rutkowska J, Wojtczuk R, Invisible Things Lab, Technical Report: Version 0.3 "Qubes OS Architecture," 2010.

Saltzer JH, Schroeder MD. The Protection of Information in Computer Systems. Proceedings of the IEEE. 1975; 63, 9: pp. 1278-308.

Schiavo J. Code Signing for End-user Peace of Mind. Network Security. 2010; 7: pp. 11-3.

Schmid M, Hill F, Ghosh AK. Protecting Data from Malicious Software. In 18th Annual Computer Security Applications Conference (ACSAC). Washington, DC, USA: IEEE Computer Society, 2002.

Schreuders ZC. FBAC-LSM: Protect Yourself From Your Apps. http://schreuders.org/FBAC-LSM, Accessed 2012.

Schreuders ZC, McGill T, Payne C. Empowering End Users to Confine Their Own Applications: The Results of a Usability Study Comparing SELinux, AppArmor and FBAC-LSM. ACM Transactions on Information and System Security (TISSEC). 2011a; 14, 2: pp. 1-28.

Schreuders ZC, McGill T, Payne C. Towards Usable Application-oriented Access Controls: Qualitative Results from a Usability Study of SELinux, AppArmor and FBAC-LSM. International Journal of Information Security and Privacy. 2012; 6, 1: pp. 57-76.

Schreuders ZC, Payne C. Functionality-Based Application Confinement: Parameterised Hierarchical Application Restrictions. In International Conference on Security and Cryptography (SECRYPT 2008). Porto, Portugal: INSTICC Press, 2008a, pp. 72-7.

Schreuders ZC, Payne C. Reusability of Functionality-Based Application Confinement Policy Abstractions. In 10th International Conference on Information and Communications Security (ICICS 2008). Birmingham, UK: Springer, 2008b, pp. 206-21.

Schreuders ZC, Payne C, McGill T. A Policy Language for Abstraction and Automation in Application-oriented Access Controls: The Functionality-based Application Confinement Policy Language. In IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2011). Italy, Pisa: IEEE Computer Society, 2011b.

Schreuders ZC, Payne C, McGill T. Techniques for Automating Policy Specification for Application-oriented Access Controls. In 6th International Conference on Availability, Reliability and Security (ARES 2011) Vienna, Austria: IEEE Computer Society, 2011c.

Seaborn M. Plash. http://plash.beasts.org/, 2008.

Seriot N. iPhone Privacy. In Black Hat DC. Arlington, VA, USA, 2010.

Smalley S, Vance C, Salamon W, National Security Agency (NSA), Technical Report: NAI Labs Report #01-043 "Implementing SELinux as a Linux Security Module," USA, 2001.

Snowberger P. Sub-identities: A Hierarchical Identity Model For Practical Containment. Notre Dame, IN, USA: University of Notre Dame; 2007.

Spencer R, Smalley S, Loscocco P, Hibler M, Andersen D, Lepreau J. The Flask Security Architecture: System Support for Diverse Security Policies. In 8th USENIX Security Symposium. Washington, DC, USA: USENIX Association, 1999, pp. 123-39.

Spengler B. grsecurity. http://www.grsecurity.net/, Accessed 2012.

Stafford TF, Urbaczewski A. Spyware: The Ghost in the Machine. Communications of the Association for Information Systems. 2004; 14: pp. 291-306.

Stiegler M, Karp AH, Yee KP, Close T, Miller MS. Polaris: Virus-safe Computing for Windows XP. Communications of the ACM. 2006; 49, 9: pp. 83-8.

Sugerman J, Venkitachalam G, Lim BH. Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor. In USENIX Annual Technical Conference. Boston, MA, USA:

USENIX Association, 2001, pp. 1–14.
Sun W, Liang Z, Sekar R, Venkatakrishnan VN. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In 12th ISOC Symposium on Network and Distributed Systems Security (SNDSS). San Diego, CA, USA, 2005.
Szor P. The Art of Computer Virus Research and Defense: Addison-Wesley Professional; 2005.
Tandon G, Chan PK. On the Learning of System Call Attributes for Host-based Anomaly Detection. International Journal of Artificial Intelligence Tools. 2006; 15, 6: pp. 875-92.
Thorsteinson P, Ganesh GGA. .Net Security and Cryptography: Prentice Hall PTR; 2003.
Tucker A, Comay D. Solaris Zones: Operating System Support for Server Consolidation. In 3rd Virtual Machine Research and Technology Symposium Works-in-Progress. San Jose, CA, USA, 2004.
Tzur R. Sandboxie. http://www.sandboxie.com/, Accessed 2012.
Vegge H, Halvorsen FM, Nergard RW, Jaatun MG, Jensen J. Where Only Fools Dare to Tread: An Empirical Study on the Prevalence of Zero-Day Malware. In 4th International Conference on Internet Monitoring and Protection (ICIMP 2009). Venice/Mestre, Italy: IEEE Computer Society, 2009.
Wagner D, Dean R. Intrusion Detection Via Static Analysis. In 22nd IEEE Symposium on Security and Privacy. Oakland, CA, USA: IEEE Computer Society, 2001, pp. 156-68.
Wagner DA, University of California, Technical Report: CSD-99-1056 "Janus: An Approach for Confinement of Untrusted Applications," Berkeley, CA, USA, 1999.
Walker K, Sterne D, Badger M, Petkac M, Sherman D, Oostendorp K. Confining Root Programs with Domain and Type Enforcement. In 6th USENIX Security Symposium. San Jose, CA, USA: USENIX Association, 1996.
Walsh D. Dan Walsh's Blog: Introducing the SELinux Sandbox. http://danwalsh.livejournal.com/28545.html, 2009.
Watson RNM. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In 1st USENIX Workshop on Offensive Technologies. Boston, MA, USA: USENIX Association, 2007.
Weaver N, Paxson V, Staniford S, Cunningham R. A Taxonomy of Computer Worms. In ACM Workshop on Rapid Malcode. Washington, DC, USA: ACM Press, 2003, pp. 11-8.
Weber S, Karger PA, Paradkar A. A Software Flaw Taxonomy: Aiming Tools at Security. ACM SIGSOFT Software Engineering Notes: Software Engineering for Secure Systems (SESS) - Building Trustworthy Applications 2005; 30, 4: pp. 1-7.
Whitaker A, Shaw M, Gribble SD. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In 5th USENIX Symposium on Operating Systems Design and Implementation. Boston, MA, USA: USENIX Association, 2002, pp. 195–209.
Wichers DR, Cook DM, Olsson RA, Crossley J, Levitt PKKN, Lo R. PACL’s: An Access Control List Approach to Anti-viral Security. In 13th National Computer Security Conference. Washington, DC, USA: NIST, 1990.
Yee B, Sehr D, Dardyk G, Chen JB, Muth R, Ormandy T, et al. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. Communications of the ACM. 2010; 53, 1: pp. 91-9.
Zanin G, Mancini LV. Towards a Formal Model for Security Policies Specification and Validation in the SELinux System. In 9th ACM Symposium on Access Control Models and Technologies. Yorktown Heights, NY, USA: ACM Press, 2004, pp. 136-45.
Zurko ME, Simon R, Sanfilippo T. A User-centered, Modular Authorization Service Built on an RBAC Foundation. In IEEE Symposium on Security and Privacy. Oakland, CA, USA: IEEE Computer Society, 1999, pp. 57-71.
Zurko ME, Simon RT. User-Centered Security. In New Security Paradigms Workshop (NSPW). Lake Arrowhead, CA, USA: ACM Press, 1996, pp. 27-33.