WindowBox: A Simple Security Model for the Connected Desktop

Dirk Balfanz *
balfanz@cs.princeton.edu
Princeton University

Daniel R. Simon †
dansimon@microsoft.com
Microsoft Research

Abstract

Breaches in computer security do not just exploit bugs in applications; they are often also the result of mismanaged protection mechanisms. The tools available to protect sensitive resources and networks are tedious to use, non-intuitive, and often require expert knowledge. As a result, many PC and workstation users end up administering their system security poorly, creating serious security vulnerabilities. This paper presents a new security model, WindowBox, which presents the user with a model in which the workstation is divided into multiple desktops. Each desktop is sealed off from the others, giving users a means to confine the possibly dangerous results of their actions. We have implemented our security model on Windows 2000, leveraging the existing desktop metaphor, the ability to switch between multiple desktops, and specific kernel security mechanisms.

1 Introduction

Today's typical computing environment no longer consists of a centrally managed mainframe accessed through terminals; more often it consists of networked, Internet-aware PCs and workstations. The security threat model has thus changed in several ways:

- Because individual users have greater control over the PCs or workstations they typically use, they also have greater responsibility for the administration of these machines and their security. Where knowledgeable administrators once made professional decisions about security risks, those decisions (such as permission and denial of access to system and data resources) are increasingly in the hands of relatively uninformed users.
- The more open, distributed architecture of PCs and workstations makes it possible for users to introduce new applications and even operating system modifications to their machines – opening up another avenue of attack, through viruses, Trojan horses and other malicious applications.
- Internet connectivity has greatly expanded the range of possible attackers with access to a user's machine.

Various tools are already available to combat each of these threats. For example, existing discretionary access control mechanisms are usually available to limit access by (potentially hostile) applications to files. Firewalls and proxies can thwart certain kinds of network attacks. Finally, sandboxing has become a popular method for restricting a process' privileges to a subset of its owner's privileges, usually in the case where that process is executing some untrusted code. However, in practice we often see these techniques fail, not necessarily because they were poorly implemented, but because they are being poorly applied. The reason for this is that the currently used techniques, taken on their own, tend to be too complex for ordinary users to administer effectively. Taken together, they can be completely overwhelming even to fairly experienced users. As a result, confused users, faced with the responsibility of using and managing these techniques, often make ill-informed decisions with misunderstood implications – including, perhaps, serious security vulnerabilities.

It is thus imperative that user-administered security tools present the user with an easily comprehensible, intuitive model that allows users to understand the implications of their security policy decisions. For example, one idea they can understand is that of absolute physical separation. Given a set of unconnected machines, between which all information must be explicitly carried (say, on a floppy disk), a user can understand that sensitive information or privileges on one machine are safe from potential threats on other machines.

* Department of Computer Science, Princeton, NJ 08544
† Microsoft Research, One Microsoft Way, Redmond, WA 98052

In this paper, we describe a model, WindowBox, which is based on this idea of complete separation, and thus allows users to make reasonable security decisions with a clear understanding of their implications. We have implemented the model on a Windows 2000 workstation complete with a simple user interface that permits the user to manage the workstation's security in a natural, intuitive way.

In the next section, we will analyze existing security mechanisms and explain why they fail to deliver, especially in the hands of "average" users. In Section 3 we introduce the WindowBox model in more detail, before we explain our implementation on Windows 2000 in Section 4.

2 Existing Security Mechanisms

2.1 Access Control Lists

In the traditional, mainframe-based model, users wish to restrict access to their resources (which are almost always data files) to some limited set of users. Conceptually, an access matrix [3] describes what each user can do to each file. We often find that in a given implementation, the access matrix is replaced by a slightly less general, but more efficient mechanism. In Windows 2000, and other operating systems, Access Control Lists (ACLs) are used to restrict access of users to files. In practice, the vast majority of such restrictions limit access to a resource to a single owner (or not at all). In this manner files, for instance, can be fairly easily made "private" (accessible only to the file's owner) or "public" (accessible to all the system's users).

In the PC/workstation setting, however, a user may own many other objects – systems resources, communications resources, and so on. These are often more difficult to control by ACL, for various reasons: the object may perhaps be created and used by applications without the owner's knowledge in the first place; its ACL may not be accessible to the owner, for lack of an appropriate user interface; or it may be (as in the case of remotely accessible resources) difficult for the owner to tell who should or should not have access to the object – or even who is capable of attempting to access it.

Moreover, deciding on the appropriate ACL for every individual object (even whether it is "public" or "private") is a complicated, tedious and distracting task. Hence systems typically apply defaults to construct an ACL without consulting the user (such as when a new file is created). Since these default ACLs are assigned in the absence of information about the context in which they are created, they often conflict with the user's intentions.

2.2 Application-Level Security

When the resource in question is an application, it is typically the application itself that is made responsible for its own security. The operating system provides applications with various security services (such as authentication for remote connections); using these services, an application can limit access to itself as specified by the user running it, thus (hopefully) preventing its own exploitation by an external attacker.

This application-oriented approach has two major disadvantages:

- The user making the access decisions must deal with a different configuration procedure for each application – hopefully involving the same system-provided infrastructure, but sometimes even depending on application-specific authentication and access control mechanisms.
- Each application's security is implemented separately – and therefore has its own independently buggy security. A user would need to know all the security holes in each application, and how to defend against attacks on each one, in order to secure his or her system against attack.

A preferable alternative would be for the user to be able to set access control rights for any application from "above", requiring any inter-process communication reaching the application to be authenticated as coming from a permitted user or user/application pair. A natural analogy is the file system: just as access (of any sort) to a file can be limited to a particular set of users, so too should access to an application. (Indeed, as the distinction between applications and data blurs, so does the distinction between the two types of access.)

2.3 Firewalls and Proxies

The problem of controlling access to an application becomes even more difficult when the application is intended to be "network-aware". A well-connected PC or

workstation, for example, runs a number of applications of various types; these may include both traditionally server-based applications (HTTP servers, FTP servers and the like) and traditionally user-oriented (but increasingly network-aware) applications. All of these may at various times send data onto the network or monitor and read arriving network data, using network services provided by the operating system. We might assume that the careful user avoids installing deliberately malicious applications; however, even established commercial applications can have security holes which may be exploited by hackers sending them unexpected data.

To protect against the possible existence of such holes, many administrators of large networks of PCs/workstations install firewalls or proxies, which filter out network traffic that does not conform to the formats that the applications are expected to handle correctly. But applications may still have bugs which allow an attacker to subvert them by sending them data that is allowed through by the firewall or proxy, but has unanticipated effects due to bugs or design oversights in the application.

Therefore, the problem would be much more effectively addressed by enforcement of the following two restrictions:

- Limiting applications' access to the network to authenticated connections, with access control applied to these connections "from the outside," as discussed above.
- User-imposed blocking of applications involved in network communications (especially unauthenticated connections) from accessing sensitive resources on the same machine.

For example, a personal HTTP server might be allowed to accept unauthenticated connections with the outside world, but be forbidden to write to files or communicate with other applications beyond an "internal firewall." The other side of this firewall might contain "private" applications permitted to communicate with each other, but not with the HTTP server, and not with the network at all except over authenticated connections to members of a particular group of trusted users (say, an Intranet).

2.4 Sandboxing

Meanwhile, the recent proliferation of mobile code via the Internet has spurred demand for mechanisms by which such mobile applications can be "sandboxed" (granted only restricted access to system resources, possibly including files, communication channels, miscellaneous system services and other applications). Originally, mobile code consisted in practice only of small "applets" with very limited functionality (and thus easy to sandbox tightly). Increasingly, however, the crisp lines between simple, casually distributed applets and "store-bought" applications, and between entire applications and individual components, have begun to blur. Sophisticated software is now being distributed over the Internet in the form of applets or even entire applications, and components distributed in this fashion are being used together in complicated ways. Even operating system updates, for instance, are being distributed over the Internet. The distinction between "trusted" and "untrusted" software is thus increasingly a continuum, with no software enjoying the confidence once (possibly too naively) accorded store-bought, "shrink-wrapped" software. It follows that some degree of sandboxing will become desirable for a wide range of applications and components.

A prominent example of the necessity of ubiquitous sandboxing is the use of cryptographic keys for sensitive functions such as electronic commerce or Intranet authentication. The susceptibility of such keys to compromise by viruses or security holes, and the dire consequences of such compromises, make it imperative that keys be accessed only by applications that can be trusted to use them properly (requesting direct user approval, for instance). A broadly enforced sandboxing policy could make such protection possible.

The major existing sandboxing framework is the Java virtual machine (VM), which can be used to allow "applets" (typically embedded in Web pages) restricted access to resources. This VM is only available for applications distributed in the form of a particular Java byte code, and then translated (with some performance overhead) into native machine code. The Java security model allows applications to be accompanied by digitally signed requests for various system access privileges; the user or administrator can decide whether to grant applets their requested privileges based on the identities of the digital signers.

This model ultimately requires the user to make difficult qualitative trust judgments, such as which permissions are too sensitive to trust to which applications. The model can thus lead to a proliferation of too many identities and permissions for the user to keep track of. At its worst, it simply adds an extra layer of complication to the ACL model: resources must be access-controlled

based not only on user identity, but also application origin. While it may be a useful tool for implementing a higher-level model, it is still far too complex for direct manipulation by ordinary users when applied to a great many applications.

3 A Model For Sandboxing-Based Security

The above issues suggest the following goals for a security model:

- It should unify the per-user access control functions of the ACL model with the security properties of firewall- and sandboxing-based models, so that users only have to deal with a single model;
- It should allow basic application-level security (restricting both access to the application and its range of permitted behavior) to be applied independent of the application;
- It should be simple enough to be managed via a natural, intuitive user interface.

The last goal is perhaps the most crucial, as well as the most difficult. Creating an understandable user interface for a security model is generally a daunting task, given the complexity and criticality of security administration. The only hope is to design a simple and natural model that users can grasp and manipulate intuitively, then present a UI which reflects this model. Otherwise, confused users will inevitably make dangerously mistaken security decisions – such as disabling all security features entirely, to avoid the inconvenience of dealing with them.

3.1 The Multi-Desktop Premise

We propose here a model based on an extremely simple idea: while users cannot really intuit the complex rules associated with zones or ACLs, one idea they can understand is that of absolute physical separation. Given a set of group or zone permission rules, a user will have a difficult time determining if it expresses his or her idea of security. But given a set of unconnected machines, between which all information must be explicitly carried (say, on a floppy disk), a user can understand that sensitive information or privileges on one machine are safe from potential threats on other machines. It may be that many users (particularly small businesses) are using multiple machines in this manner today to secure their sensitive data and applications from the Internet. And of course, large enterprises use proxies all the time to protect their internal machines from the world outside.

Moreover, users' business (both information and – to a lesser extent – applications) tends to divide up fairly cleanly into categories, such as "personal finance," "business/office/intranet," "Internet gaming," and so on. The amount of information flowing between these categories is typically relatively small, and therefore should be manageable through direct user intervention (carrying floppy disks between machines, or its drag-and-drop analog in the virtual context) without excessive strain on the user.

3.2 The WindowBox Security Model

In the WindowBox security model, the user can construct multiple desktops, which are kept completely isolated, except by explicit user action (such as a direct point-and-click command or response to a dialog or warning box). To a first approximation, the desktops start off completely identical, and are provided just so that the user can use different desktops for different tasks. As the user uses the different desktops, each one accumulates its own files, and applications in one desktop cannot access files in other desktops except by the aforementioned user action. To aid the user in consistently using the desktops for their specific purpose, each desktop has its own network access restrictions and code-verifying criteria (although the user would manipulate these only indirectly, by defining and configuring desktops). Many special-purpose applications would be confined to a single desktop; other, more general applications would be "installed" in multiple desktops, but would have different access rights, and possibly even different behaviors, in each desktop (for example, a word processor might have different defaults depending on whether it is being run in an enterprise/Intranet desktop or a personal one).

In some ways, the multiple desktops can be considered as representing different users logged on simultaneously. A key difference, however, is that simple user-mediated actions would always be able to transfer data between one desktop and another, and (if necessary) create new desktops or change the properties of existing ones. From the network's perspective, on the other hand, these desk-

tops would have very different security properties. For example, access to the private key necessary for authenticating as the user in a secure connection to a particular server may be restricted to applications in a particular desktop. Hence a server that only accepts secure connections would implicitly require that the user access it from only that desktop.

3.3 Examples

Most of the work of defining and configuring desktops should be a matter of choosing among standard desktop types with preset, mildly customizable attributes. We suggest a few natural ones here.

3.3.1 The Personal Desktop

A simple example of a useful separate desktop is a "personal" desktop to isolate sensitive personal (e.g., financial) applications and data from the rest of the user's machine. Applications in such a desktop would be limited to those handling such personal matters, plus a few trusted standard ones such as basic word processing. These applications would also be isolated from all interprocess communication with applications on other desktops, and files created by them would be inaccessible from any other desktop. Network access in this desktop would be limited to secure, authenticated connections with a small number of trusted parties, such as the user's bank(s) and broker(s); no general browsing or Internet connections would be permitted. Similarly, the authentication credentials required to establish authenticated connections to these trusted parties would be isolated in this desktop. A user with such a separate desktop should feel comfortable using the same machine for other purposes without fear of exposing sensitive personal data or functionality to attackers.

3.3.2 The Enterprise Desktop

Like personal data, a user's work-related data and applications are best kept isolated from the rest of the user's machine. In an "enterprise" desktop, only enterprise-approved work-related applications would be allowed to run, and network access would be limited to secure authenticated connections to the organizational network or intranet (and hence to the rest of the Internet only through the enterprise proxy/firewall). Again, all applications on this desktop would be "sandboxed" together, and denied inter- process communication with applications outside the desktop. The capability to authenticate to the enterprise network would also be isolated in this desktop. Such isolation would allow a user to access an enterprise Intranet safely from the same machine used for other, less safe activities.

Note that enterprise-based client-server applications actually benefit enormously from such isolation, because they typically allow the user's client machine to act in the user's name for server access. Thus if an insufficiently isolated client application opens a security hole in the client machine, it may implicitly open a hole in the server's security, by allowing unauthorized attackers access to the server as if they were at the same authorization level as the attacked client. On the other hand, if the client application and all associated access rights are isolated in an "enterprise desktop," then malicious or vulnerable applications introduced onto the client machine for purposes unrelated to the enterprise are no threat to the enterprise server's security.

3.3.3 The "Play" Desktop

For games, testing of untrusted applications, and other risky activities, a separate desktop should be available with full Internet access but absolutely no contact with the rest of the machine. There may be multiple play desktops; for example, a Java-like sandbox for untrusted network-based applets would look a lot like an instance of a play desktop.

3.3.4 The Personal Communication Desktop

Since users are accustomed to dealing with email, Web browsing, telephony/conferencing and other forms of personal communication in an integrated way (as opposed to, for instance, receiving email in different desktops), these functions are best protected by collecting them in a single separate desktop. This desktop should run only trusted communications applications; those communications (email, Web pages, and so on) which contain executable code (or data associated with non-communications applications) would have to be explicitly moved into some other desktop to be run or used. For example, a financial data file contained in an email message from the user's bank would have to be moved into the personal desktop before being opened by the appropriate financial application. Note that some communications functions could also be performed in other

desktops; for example, a Web browser in the enterprise desktop might be used to browse the enterprise Intranet (to which applications – including browsers – in other desktops would have no access).

3.4 How WindowBox Protects the User

Before we look at our implementation of the WindowBox model, let us recap how WindowBox can prevent common security disasters. Consider, for example, users who like to download games from questionable Web sites. Every now and then, one of the downloaded games may contain a virus, which can destroy valuable data on people's machines. If the game is a Trojan Horse, it might also inconspicuously try to access confidential files on the PC and send them out to the Internet. If the users were employing WindowBox, they would download the games into a special desktop, from which potential viruses could not spread to other desktops. Likewise, a downloaded Trojan Horse would not be able to access data in another desktop.

As a second example, let us now consider the recent spread of worms contained in email attachments. For example, the Melissa worm (often called the "Melissa virus"), resends itself to email addresses found in the user's address book. On a WindowBox-equipped system, users would open email attachments in a desktop that is different from the desktop in which the email application is installed. This might be because the attachments logically belong in a different desktop, or simply because the user judges them not trustworthy enough for the email desktop. From that other desktop, the worm cannot access the email application, or the network, to spread itself to other hosts.

4 Implementation

We implemented the WindowBox security architecture on a beta version of Windows 2000 (formerly known as Windows NT 5.0 Workstation). In our implementation, the various desktops are presented to the user very much in the manner of standard "virtual desktop" tools: at any given time, the user interacts with exactly one desktop (although applications running on other desktops keep running in the background). There are GUI elements that allow the user to switch between desktops. When the user decides to switch to a different desktop, all application windows belonging to the current desktop are removed from the screen, and the windows of applications running in the new desktop are displayed. Windows 2000 already has built-in support for multiple desktops. For example, if a user currently works in desktop A, and an application in desktop B pops up a dialog box, that dialog box will not be shown to the user until he or she switches to desktop B. Windows 2000 provides an API to launch processes in different desktops and to switch between them. We simply had to provide GUI elements to make that functionality available to the user.

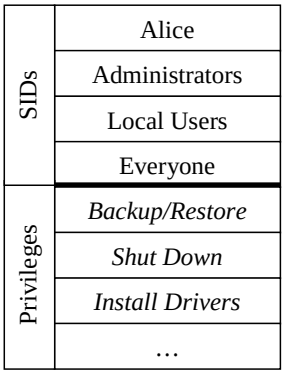

Figure 1: A sample access token

However, the desktops provided by Windows 2000 do not, in any way, provide security mechanisms in the sense of the WindowBox security architecture. Our implementation therefore had to extend beyond what is offered in Windows 2000. In this section we describe these extensions.

Before explaining how we represent desktops as user groups, and what changes we made to the NT kernel to implement WindowBox security, we will briefly recap the Windows NT security architecture.

4.1 The Windows NT Security Architecture

In Windows NT, every process has a so-called access token. An access token contains security information for a process. It includes identifiers of the person who owns the process, and of all user groups that person is a member of. It further includes a list of all privileges that the process has. Let's assume that Alice is logged on to her Windows NT workstation and has just launched a process. Figure 1 shows what the access token of such a process might look like: It includes an identifier (also called Security Identifier, or SID) of Alice as well as of all the groups she is a member of. These include groups

1. *Allow* Write Administrators
2. *Deny* Read Alice
3. *Allow* Read Everyone

Figure 2: A sample DACL

that she has explicitly made herself a member of, such as "Administrators," as well as groups that she implicitly is a member of (like "Everyone"). Since she is an administrator on her workstation, her processes get a set of powerful privileges (these privileges are associated with the user group "Administrators"). The figure shows a few examples: The "Backup/Restore" privilege allows this process to read any file in the file system, regardless of the file's access permissions. The "shut down" privilege allows this process to power down the computer, etc.

Access tokens are tagged onto processes by the NT kernel and cannot be modified by user-level processes[1]. When a user logs on to the system, an access token describing that user's security information is created and tagged onto a shell process (usually Windows Explorer). From then on, access tokens are inherited from parent to child process.

The second fundamental data structure in the Windows NT security architecture is the so-called security descriptor. A security descriptor is tagged onto every securable object, i.e. to every object that would like to restrict access to itself. Examples of securable objects include files or communication endpoints. A security descriptor contains, among other things, the SID of the object's owner and an access control list. The access control list (also called Discretionary Access Control List, or DACL) specifies which individual (or group) has what access rights to the object at hand. It is matched against the access token of every process that tries to access the object. For example, consider a file with the access control list shown in Figure 2. What happens when Alice tries to access this file? Since her access token includes the Administrators group, she will have write access granted. However, the DACL of this file explicitly denies read access for Alice (let's forget for a moment that Alice's access token also contains the Restore/Backup privilege, which enables her processes to override this decision). Because the order of the DACL entries matters, it is not enough that the group "Everyone" (of which Alice is a member, as her access token specifies) has read access. The entry that denies Alice read access comes first, effectively allowing everyone but Alice read access. We can see that the combination of access tokens and security descriptors provides for an expressive and powerful mechanism to specify a variety of access policies.

[1]This is an oversimplification. In reality, there are some limited operations that a user-level process can do to an access token: It can switch privileges on and off and even (temporarily) change the access token of a process (for example, when a server would like to impersonate the client calling it). However, a process can never, of its own volition, gain more access rights than were originally assigned to it by the kernel.

SIDs	Alice
	Local Users
	Everyone
	Administrators
	Alice.Personal
	Alice.Enterprise
	Alice.Play
Privileges	*Shut Down*
	Install Drivers
	Backup/Restore
	…

Figure 3: A sample access token with desktop SIDs

Only the kernel can modify the security descriptor of an object, and it will only do so if the process that is requesting modifications belongs to the owner of that object. Also, the access check described above happens inside the kernel. No user process can, for example, go ahead and read a file if the file's DACL forbids this.

4.2 Desktops as User Groups

Apart from the graphical representation to the user, we internally represent each desktop as a user group. For example, if Alice wanted three desktops for her home, work, and leisure activities, she could create three user groups called Alice.Personal, Alice.Enterprise, and Alice.Play. She would make herself a member of all three groups[2]. Now, whenever she logs on to her computer, her access token would look like the one shown in Figure 3. Note that the SIDs that represent her desktops are added to the access token. This happens automatically since Alice is a member of all these groups. We call these SID's desktop SIDs. They are marked as desk-

[2]In our prototype, this process is automated and happens when a new desktop is created.

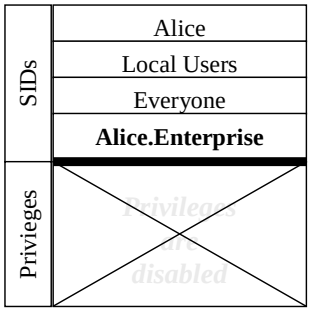

Figure 4: A sample access token of a desktop process

top SIDs in the access token (denoted by bold typeface in the picture), but are otherwise just normal group SIDs.

In our implementation, we create a desktop for every desktop SID in the access token when the user logs on. Continuing our example, when Alice logs on, we create three desktops. In every desktop, we start a shell (Windows Explorer)[3]. However, we limit what each shell can do by giving it a restricted token (the restricted token API has been introduced in Windows 2000 and allows processes to limit their own, or their children's, privileges): First, we remove all privileges from the access token. Then we remove all desktop SIDs except the one representing the desktop for which we are preparing the access token[4]. Lastly, we remove the "Administrators" SID from the token in a move to restrict access to system files, which usually allow write access to the Administrators group. Each shell is launched with this restricted token. Figure 4 shows the access token of the shell running in Alice's enterprise desktop.

When Alice now starts applications or processes in one of her desktops, they will inherit the restricted token of the desktop's shell. Note that this also holds for ActiveX components and other executable content downloaded from the network, which runs inside descendants of the desktop's shell.

With the system described so far, we could already implement some kind of WindowBox security. If Alice judiciously restricted access to some files to the user group Alice.Enterprise (and only to that group), these files could only be accessed by applications running in the enterprise desktop. However, one of the major goals of the WindowBox security architecture is to relieve the user from the burden of making difficult access control decisions, and to "automate" this process. Furthermore, while the user can modify the DACLs of files, this is not true for all objects. For example, processes in one desktop can still communicate with processes in another desktop, and there is nothing that the user can do about this. For this reason, we introduced the concept of "confined" objects.

4.3 Confined Objects

A confined object is an object that belongs to a certain desktop. The idea is that an object confined in desktop A should not be accessible by any process from desktop B. We confine objects by tagging their security descriptor with the SID of the desktop they should be confined to. For example, to confine a file to Alice's enterprise desktop, we would add the SID Alice.Enterprise to the file's security descriptor. This extension of the security descriptor is our first modification to the NT kernel.

Our second modification makes sure that objects automatically become confined: Whenever the kernel creates a securable object (i.e., an object with a security descriptor) such as a file or a communications endpoint on behalf of a process, we confine that object to the desktop that the creating process runs in. Note that not all objects have to be confined. There are processes on the system (for example system services) that do not belong to any desktop because they are not descendants of any of the desktops' shells. The kernel can also create objects on its own behalf.

Our third modification concerns the access check performed in the kernel. The original access check implements the semantics of the access control list of an object with respect to the accessing process' access token as explained above. We changed the access check as follows:

1. Is the object confined? If so, go to 2; otherwise go to 3.
2. Check whether the process' access token contains a SID that is equal to the desktop SID the object is confined to. If so, go to 3; otherwise deny access.
3. Perform a normal access check.

[3]For the record we should mention that there will also be a fourth desktop that serves as a "root" desktop in which all applications have full privileges and access to the system. Alice should stay away from that desktop for her day-to-day work, but can use it for administrative tasks.

[4]The diligent reader will object that removing a group SID from an access token doesn't necessarily restrict the process' rights. Windows 2000 does the right thing: The SID is not completely removed, it is "disabled for positive DACL entries": If you disable the SID "Group A" in an access token, and then try to open a file that allows only access to Group A, this access will not be allowed. However, if you try to access a file that explicitly denies access to Group A, access *will* be denied.

Our modifications implement the WindowBox security architecture: The desktops are completely sealed off from each other. For example, a file saved by a word processor in one desktop will be confined to that desktop and cannot be accessed by any process from another desktop, including potentially malicious applications downloaded from the outside world. This goes beyond what default ACLs in Windows 2000 or the umask feature in UNIX offer: First, the confinement cannot be undone by an ordinarily privileged process (see below). Second, there is no customizable setting that the user has to decide on (i.e., what should the default ACL or umask be?). Object confinement is mandatory, and cannot be customized, or mismanaged, by the user. Also note that confined files are usually visible to other desktops, they are merely inaccessible. An access to a file confined to another desktop would fail in the same way that access to file belonging to another user would. However, by the same token, this also implies that files within a directory in another desktop will *not* be visible.

Notice that processes also become confined as they are created (they, too, are securable objects).

Once an object is confined, it takes a special privilege to "un-confine" it (by removing the confining SID from its security descriptor) or to confine it to another desktop. Since we strip all privileges from the shells (and their children) in each desktop, no application in any desktop can move objects from one desktop to another. However, in each desktop we provide one process that is not restricted in the same way as the shell and its children are. This process provides the GUI to switch to other desktops (the restricted processes in the desktops would not even have enough privileges to make that switch). That privileged process also serves as a COM server exporting the service to move objects between desktops. Every process can connect to that server and ask it to move an object from one desktop to another. The server can then decide whether or not to do so. In our implementation, it asks the user for confirmation before any object (such as a file) is moved between desktops.

4.4 Restricting Network Access

The system, as described so far, can already safeguard against a number of attacks if used consistently. Our user Alice should never do her finances in her play desktop, for example. Nor should she visit untrusted Web sites while she is in her enterprise desktop. To encourage her to abide by the latter rule, we have restricted network access for processes running in desktops.

We modified the kernel to deny any network access to a process running in a desktop (i.e., a process with a desktop SID in its access token). However, we also added a layer in the network stack that relays network calls of a desktop process (which would fail in the kernel) to the privileged COM server mentioned above. This privileged process can connect to the network, but will only do so if the requested network address satisfies the desktop's policy (e.g., it is explicitly included in a list of permitted addresses). If so, it connects to the requested network address and returns the handle representing the network connection back into the unprivileged process.

In our implementation, users can specify a different network access policy for each desktop. The mechanism used is powerful enough to express policies like: "only allow connections to the corporate intranet," "only allow connections to www.mybank.com," "deny any network access," etc.

4.5 Security Analysis

How secure is WindowBox? This question does not have a generic answer. Rather, we need to ask how secure a specific implementation of the WindowBox security model is. A "secure" WindowBox implementation does not allow malicious code to affect another desktop. For example, a virus should not be able to infect files in a different desktop, a data-gathering Trojan Horse should not be able to read files in other desktops, etc. In other words, there should be no *channels* between desktops, which malicious code could exploit. The work on covert channels has shown that it is not possible to close every covert channel. For the purposes of this paper, a covert channel can be defined as a means by which information could leak from one desktop to another. Note that this is considerably less powerful a channel than one which a virus could actually exploit to propagate itself to a different desktop. For that purpose, a virus would have to be able to write a file in one desktop, and then cause that file to be executed in a different desktop. While our situation is not as hopeless as with covert channels, we still believe that it is impossible to get formal assurance that no dangerous channels exist. Windows 2000 is too complex a system for us to hope to model it in a way that would yield relevant statements about a given WindowBox implementation.

What, then, can we say about the security of our WindowBox implementation? We tried to make our implementation as secure as possible by implementing it at as "low" a level as possible inside the kernel. The rationale

is that every malicious program has to go through certain parts of the kernel - most notably, the access control reference monitor - in order to do anything useful, including any attempt to access another desktop. Therefore, we placed the WindowBox enforcement code inside that reference monitor.

There are, however, covert channels left between desktops in our current implementation. For example, files that are not confined to a desktop could potentially be writable by one desktop and then readable by another desktop. To prevent this covert channel from turning into a channel that a virus could exploit, we made sure that none of the executable system files are writable from any desktop. Ultimately, we believe that a lot of scrutiny will be necessary to find and deal with other potential channels, and that for this reason a production version of WindowBox would need extensive testing and prolonged exposure to the security community.

The last part of the answer is that a WindowBox implementation would be most secure on top of an operating system that was designed in anticipation of this kind of security model. For example, we mentioned above that certain applications should be installed in only one desktop, or that applications should be installed separately in multiple desktops. Windows 2000 does not really allow us to do that. For example, applications written for Windows 2000 like to keep configuration data in the (system-wide) registry or use well-known (system-global) files to store information. Ideally, the operating system should be designed from the ground up with the WindowBox model in mind, thus eliminating potential cross-desktop channels that malicious programs could exploit. However, a few key modifications to Windows 2000 would go a long way towards supporting WindowBox more securely; for example, some parts of the registry could be replicated, with separate copies for each desktop, to allow applications to install transparently on some desktops but not others.

Finally, we would like to remind the reader that no system is more secure than the decisions of its user or administrator, and that defining a security model in which the protection of sensitive data is relatively convenient (as is the case in WindowBox) creates the possibility of implementations converging towards security that is not only free of serious intrinsic holes, but also usable enough to avoid many of the types of holes introduced by the unsafe practices of users battling cumbersome systems.

5 Related Work

We are not the first to recognize the specific security requirements of a ubiquitously networked world, especially in the light of mobile code. A standard goal is to prevent mobile code, or compromised network applications, from penetrating the system. The generic term for ways of achieving this goal is “sandboxing.” The term was first used in [5] to describe a system that used software fault isolation to protect system (trusted) software components from potentially faulty (untrusted) software components. Perhaps the best-known example of sandboxing is the Java Virtual Machine. It interprets programs written in a special language (Java bytecode). It can limit what each program can do based on who has digitally signed the program and a policy specified by the local user. The biggest drawback of the Java approach is that it can only sandbox programs written in Java.

Our system sandboxes processes regardless of which language they have been written in. That, too, is not new. In [2], Goldberg et al. show how any Web browser helper application can be sandboxed. Their work, however, only targets processes that are directly exposed to downloaded content, and requires expertise in writing and/or configuring security modules.

Our system has certain resemblance to role-based access control in that one could think of our desktop SIDs as a user’s different roles. In fact, in [4], Sandhu et al. imagine a system in which “a user might have multiple sessions open simultaneously, each in a different window on a workstation screen.” In their terminology, each “session” comprises a certain subset of the user’s roles. Hence, in each window the user would have a different set of permissions.

Another concept with similarities to our desktops is that of “compartments” in Compartmented Mode Workstations (CMWs). CMWs are implementations of the Bell-LaPadula model [1] found in high-security military or government systems. Like CMW, we introduce “mandatory” security to an otherwise “discretionary” security model. In [8], Zhong explains how vulnerable network applications – such as a Web server – can be made less of a threat to the rest of the system if they are run in special compartments, shielding the rest of the system. The WindowBox security model is in some sense weaker than the Bell-LaPadula model. We are merely trying to assist the user in separating his or her different roles. For example, covert channels from one desktop to another are much less of a concern to us than they are for

a Compartmented Mode Workstation: In CMWs, an application voluntarily surrendering its data is a problem, in WindowBox it is not (it is simply not part of the threat model).

Domain and Type Enforcement (DTE) can also be used to sandbox processes in a way similar to ours. In [6], Walker et al. explain how they limit what compromised network applications can do by putting them in "domains" that don't have write access to certain "types" of objects, for example system files.

What distinguishes our work from the long list of other sandboxing approaches is that all of the above use sandboxing as a flexible, configurable technique to enforce a given – usually complicated – security policy. We argue that figuring out a complex, customized security policy is too difficult a task for most users to handle. In contrast, in our system we do not use a general sandboxing mechanism to implement specific security policies. In fact, we don't have any security policy in the traditional sense, except for the requirement of strict separation of desktops. It is this simplicity of the model that we think shows great promise for personal computer security.

Another area of related research is that of user interfaces and security. Recently, more and more people have realized that poor user interface design can seriously jeopardize security (see, for example, [7]). We very much concur with the thrust of that research. Our approach, though, is more radical. Instead of suggesting better user interfaces for existing security tools, we propose a completely redefined security model. One of the features of WindowBox is that is naturally lends itself to a more intuitive user interface for security management. Users have to understand, and learn tools that visualize, only *one* concept - the fact that separated desktops can confine the potentially harmful actions of code.

6 Conclusions

In this paper, we argue that existing security mechanisms – while possibly adequate in theory – fail in practice because they are too difficult to administer, especially for a networked personal computer or workstation under the control of a non-expert user. We present an alternative to this dilemma, WindowBox, a security model based on the concept of complete separation. While it has similarities to some existing security mechanisms, it is unique in that we do not try to provide a general mechanism to enforce all sorts of security policies. In contrast, we have only one policy – that of complete separation of desktops. We believe that this "policy" is easy to understand and has promise for the connected desktop.

We found Windows 2000 to be a good platform to implement WindowBox on: We could leverage the existing desktop API and were able to restrict changes to the NT kernel to a minimum.

The WindowBox model and prototype raise several interesting questions: What kinds of hidden security vulnerabilities might they contain, and how might they be eliminated? For instance, instead of using our own user interface, should we have used Window 2000's "Secure Attention Sequence"[5] to make sure that an application cannot trick the users into believing they are in a different desktop than they really are? How usable is a multiple-desktop environment for average users? (We conjecture that users' typical activities divide themselves up naturally in ways that correspond well with distinct desktops, but we have done no large-scale usability testing.) Should the separation be branched out into other parts of the system, e.g., would it be useful to have a separate clipboard for every desktop? Finally, are there corresponding simple, intuitive models that would apply to other environments, such as the professionally administered server? Further research may help us to answer these questions.

References

[1] D. Elliot Bell and Leonard J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical report, MITRE Corporation, March 1976.

[2] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the Sixth USENIX Security Symposium*, 1996.

[3] Butler Lampson. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, March 1971.

[4] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[5] This is the Ctrl-Alt-Del feature, which provides a trusted path to an unforgeable screen.

[5] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating System Principles*, 1993.

[6] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Sherman, and Karen A. Oostendorp. Confining root programs with domain and type enforcement. In *Proceedings of the Sixth USENIX Security Symposium*, 1996.

[7] Alma Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proceedings of the 8th USENIX Security Symposium*, Washington, DC, August 1999.

[8] Qun Zhong. Providing secure environments for untrusted network applications - with case studies using virtual vault and trusted sendmail proxy. In *Proceedings of Second IEEE International Workshop on Enterprise Security*, pages 277–283, Los Alamitos, CA, 1997. IEEE CS Press.