Apiary: Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems

Shaya Potter Jason Nieh
Computer Science Department
Columbia University
{spotter, nieh}@cs.columbia.edu

Columbia University Technical Report CUCS-034-09, August 2009

Abstract

Desktop computers are often compromised by the interaction of untrusted data and buggy software. To address this problem, we present Apiary, a system that provides transparent application fault containment while retaining the ease of use of a traditional integrated desktop environment. Apiary accomplishes this with three key mechanisms. It isolates applications in containers that integrate in a controlled manner at the display and file system. It introduces ephemeral containers that are quickly instantiated for single application execution and then removed, to prevent any exploit that occurs from persisting and to protect user privacy. It introduces the virtual layered file system to make instantiating containers fast and space efficient, and to make managing many containers no more complex than having a single traditional desktop. We have implemented Apiary on Linux without any application or operating system kernel changes. Our results from running real applications, known exploits, and a 24-person user study show that Apiary has modest performance overhead, is effective in limiting the damage from real vulnerabilities to enable quick recovery, and is as easy to use as a traditional desktop while improving desktop computer security and privacy.

1 Introduction

In today's world of highly connected computers, desktop security and privacy are major issues. Desktop users interact constantly with untrusted data they receive from the Internet. Users visit new web sites, download files, and email with strangers, all in the course of a regular day. All these activities involve making use of information that the user has no way to verify for safety. Untrusted data can be constructed in a malicious way to take advantage of bugs and vulnerabilities in application software to enable an attacker to subvert and take control of users' desktops. For example, a major flaw was recently discovered in Adobe's Acrobat products that enables an attacker to take control of a desktop when a maliciously constructed PDF file is viewed [2]. Adobe's estimate to release a fix is nearly a month after the exploit was released into the wild. Even in the absence of bugs, untrusted data can be constructed to invade users' privacy. For example, cookies are often stored when visiting web sites that allow advertisers to track user behavior across multiple web sites while web surfing.

The prevalence of untrusted data and buggy software has made application fault containment increasingly important. To address this important problem, many approaches have been proposed to isolate applications from one another using mechanisms such as process containers [26, 22] or virtual machines [32]. Faults are confined so that if an application is compromised, only that application and the data it can access become available to an attacker, not the entire system. By having one application per container, each individual container becomes a simpler system, making it easier to determine if unwanted processes are running within it.

However, existing approaches to isolating applications suffer from an unresolved tension between ease of use and degree of fault containment. On the one hand, there are approaches [19, 13] that provide a more integrated desktop feel but only provide partial isolation of applications. These approaches are relatively easy to use, but do not prevent vulnerable applications from compromising the system itself. On the other hand, there are approaches [24, 29] that have a less integrated desktop feel but full isolation of applications typically by using separate virtual machines. These approaches effectively limit the impact of compromised applications, but are harder to use since users are forced to deal with and manage multiple separate desktops. Virtual machine (VM) approaches also require the complexity of managing multiple machine instances, incur high overhead from supporting multiple operating system instances, and are too expensive to support more than a couple fault containment units for a user's desktop.

To address these problems, we introduce Apiary, a system that provides strong isolation for robust application fault containment while retaining the integrated look, feel, and ease of use of a traditional desktop environment. Apiary accomplishes this by combining three key mechanisms. First, it decomposes a desktop's applications into isolated containers. Each application container is an independent software appliance that provides all the system services the application needs to execute. To retain traditional desktop semantics, Apiary integrates these containers in a controlled manner at the display and file system. Apiary's containerized desktop prevents an application exploit from compromising the user's entire desktop. For example, by having separate web browser and personal finance containers, any compromise from web browsing would not be able to access or corrupt personal financial information. At the same time, Apiary makes the web browser and personal finance containers look and feel like part of the same integrated desktop, with all normal windowing functions and cut-and-paste operations operating seamlessly across containers.

Second, it introduces the concept of ephemeral containers. An ephemeral container is an execution environment that has no access to user data and is instantiated from a clean state for only a single application execution. Once the application is finished execution, the container is disposed of and never used again. Apiary uses ephemeral containers as a fundamental building block that enables the integrated desktop look and feel while preventing cross-contamination across containers. For example, users often expect to view PDF documents from the web, but should have separate web browser and PDF viewer containers for fault containment. If a user would always view the PDF documents in the same PDF viewer container, a single malicious document could exploit the container and have access to future documents the user desires to remain secret, such as bank and billing statements. Instead, Apiary enables the web browser to automatically instantiate an ephemeral PDF viewer containers to view individual PDF documents. Even if the PDF file is malicious, it will have no effect on the viewing of other PDF files as the container instance it exploited will never be used again.

As illustrated by this PDF example, ephemeral containers provide three benefits. First, ephemeral containers prevent compromises even when used with untrusted data that trigger application exploits because exploits cannot persist. Second, ephemeral containers protect users from compromised applications. Even when an application has been compromised due to usage with untrusted data, a new ephemeral container running that application in parallel will remain uncompromised because it is guaranteed to start from a clean state. Third, ephemeral containers protect user privacy when using the Internet. For example, while cookies must be accepted to use many web sites, web browsers in separate ephemeral containers can be used for different web sites to prevent cookies from tracking user behavior across web sites.

Apiary third mechanism is the Virtual Layered File System (VLFS). Apiary introduces the VLFS to efficiently store and instantiate containers. Each software package or application is stored as a read-only software file system layer. A VLFS dynamically composes together a private ready-write layer with a set of software layers into single file system view. Each container has its own VLFS. Since VLFS composition requires no data copying, instantiating containers is fast and space efficient. Since read-only software layers are shared across containers, multiple containers are centrally managed and upgraded as a single traditional desktop. By making containers so fast to instantiate, space efficient, and easy to manage, Apiary enables containers to be used in new ways that make possible an easy-to-use desktop with strong isolation of applications.

We have implemented an Apiary Linux prototype without any application or operating system kernel changes. To evaluate its effectiveness, we have conducted various experiments with real applications, real vulnerabilities, and real users in a user study. Our results show that Apiary can instantiate application containers in under a second, can upgrade a set of containers in just a few seconds, has scalable storage requirements, and modest file system performance overhead. Our results show that Apiary is effective at containing real exploits and preventing them from compromising a user's entire desktop. It quickly returns the desktop to a clean uncompromised state in cases where the exploit forces a complete reinstall when it occurs on a traditional desktop system. Finally, our results from a blind user study show that users find Apiary as easy to use as a traditional desktop, and given its improved security and privacy features, would prefer using it over a traditional desktop for everyday use.

2 Apiary Usage Model

Figure 1 shows the Apiary desktop. It looks like and is used in the same manner as a regular desktop. Users launch applications from a menu or from within each other, switch among all launched applications using a taskbar, interact with their running applications using their keyboard and mouse, and have a single display with integrated window system and clipboard functionality that contains all their running applications.

Although Apiary provides a similar look and feel to a regular desktop, it provides fault containment by isolating applications into separate containers. Containers enforce namespace isolation so that applications running

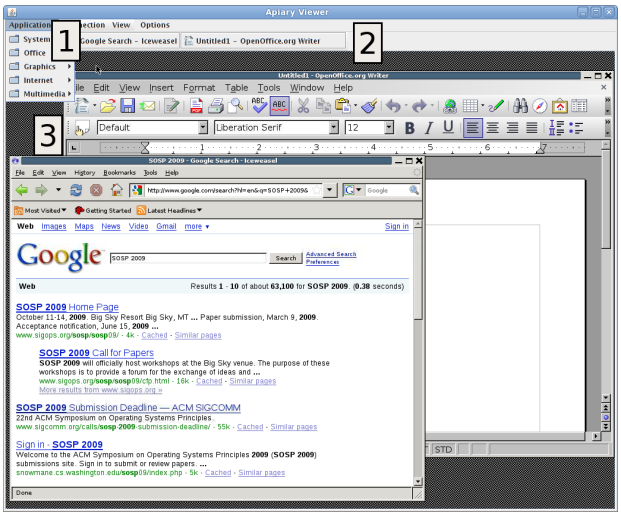

Figure 1: Apiary screenshot showing a user's desktop session. At the the topmost left, (1) An application menu that provides access to all the applications available to the desktop. Just below it, the window list (2) allows users to easily switch among running applications and the composited display view (3) of all the visible running applications.

inside cannot get out, and applications running outside cannot get in. Apiary isolates individual applications, not individual programs. An Application in Apiary can be viewed as a software appliance that is made up of multiple programs that are used together in a single environment to accomplish a specific task. For instance, a user's web browser and word processor would be considered separate applications and isolated from one another. This software appliance model means that users can install separate applications that contain many or all of the same programs but are used for different purposes and isolated from one another. For example, a user could have a banking application that contains a web browser for accessing a bank's website, and a web surfing application that also contains a web browser but is for general web browsing. While both appliances would make use of the same web browser program, they would be listed as different applications in the user's application menu.

Apiary provides two types of application containers, ephemeral and persistent. An ephemeral container is created fresh for each application execution. A persistent container maintains its state across application executions, like a traditional desktop. Apiary provides users with an option when launching an application from the menu to select whether it should be launched within an ephemeral or persistent container. Windows belonging to ephemeral applications are, by default, given distinct border colors so that users can quickly identify based on appearance in which mode an application is executing.

Ephemeral containers provide a powerful mechanism for providing desktop security and protecting user privacy. Users will typically run multiple ephemeral containers at the same time, and in some cases, multiple ephemeral containers for the same application at the same time. They provide important benefits for a wide range of usage scenarios.

Ephemeral containers prevent compromises when used with untrusted data that trigger application exploits because exploits cannot persist. For example, a malicious PDF document that exploits an ephemeral PDF viewer will have no persistent effect on the system as the exploit is isolated in the container, and the exploit will disappear when the container finishes execution.

Ephemeral containers provide an easy-to-use mechanism for protecting user privacy when using the Internet. For example, many web sites require that web browsers store cookies for them to function correctly. Many cookies are delivered by advertisers or other large companies such as Google that provide content across many web sites, allowing such organizations to track user behavior and compromise user privacy. Although web browsers provide mechanisms such as selective cookie rejection, this is typically too burdensome to use as the user must repeatedly respond to multiple dialog boxes concerning cookies for each web page visited. Similarly, web browsers that provide a privacy mode, must have application specific modifications made to them to provide this functionality. Apiary makes it easy to simply launch multiple ephemeral web browser containers simultaneously, each using separate cookies, making it much harder to track users across web sites.

Ephemeral containers provide a mechanism for protecting users from compromises that may have already occurred on a user's desktop. For example, if a web browser has been compromised due to its interaction with untrusted data during usage, parallel and future uses of the web browser could allow an attacker to steal sensitive information when the user accesses important web sites such as those for online banking. Ephemeral containers are guaranteed to launch from clean slate and not be affected by previous application usage. By using a separate ephemeral web browser container for accessing an online banking site, Apiary ensures that even an already exploited web browser installation does not compromise user privacy since the ephemeral web browser container will not have been exploited and will be isolated from other exploits.

Ephemeral containers provide a mechanism for allowing applications to launch other applications safely. For example, users often receive email attachments such as PDF documents that they would like to view. To avoid compromising an email container, Apiary creates a separate ephemeral PDF viewer container to view the PDF when the user selects it for viewing. Even if the PDF file is malicious, it will have no effect on the user's email as the PDF is isolated in a separate container. Simi-

larly, ephemeral Word or Excel containers will be created for viewing Word or Excel email attachments to prevent malicious Word or Excel files from compromising the system. In general, Apiary allows applications to cause other applications to be safely launched in ephemeral containers by default to support usage scenarios that involve multiple applications.

Persistent containers are a necessary mechanism for supporting applications that maintain persistent state across executions while preventing any compromises in such applications from affecting the entire system. In contrast to having many ephemeral containers on a desktop, we expect that users will typically run one persistent container for each application to avoid the need to track which persistent application container contains which pieces of persistent information. Some applications will be run only in persistent containers while other applications may be run in both persistent and ephemeral containers. For example, an email application will typically be used in a persistent container to maintain email state across application executions. On the other hand, a word processing application will be used in a persistent container to access a user's local trusted documents, but may also be used in an ephemeral container to view documents downloaded from the Web. Similarly, a web browser application may be used in a persistent container to remember browsing history, plugins, and bookmarks, but may also be used in an ephemeral container when accessing untrusted web sites. Note that files stored in each container are by default private and not accessible outside their respective container.

Persistent and ephemeral containers work together to provide a security system that is fundamentally different that common security schemes that attempt to lock down applications within a restricted privilege environment. First, in Apiary each application container is a fully independent entity that is fully isolated from every other application container within the Apiary desktop. As it is the only application that can access these files, one does not have to apply any security analysis or complex isolation rules to determine which files a specific application should be able to access. Second, in most other schemes, an application that gets exploited will remain exploited, even if the exploited application is restricted from accessing the data of other applications. By leveraging ephemeral containers, this is no longer a danger due to an exploits inability to persist between execution instances.

Apiary enables users to manage the files in all of their containers, by providing every desktop with a special persistent container that provides a file explorer with access to all of a user's containers. This container is special in that it can access all of the file systems in a read-write manner, enabling a user to move files between containers. This is useful if a user decides they want to preserve a file from an ephemeral container, or move a file from one persistent container to another so that it can be used by it. For instance, to email a set of files. This container cannot be used in an ephemeral manner, and its functionality cannot be invoked by any other application on the system. This prevents an exploited container from programatically propagating files it has corrupted into other containers.

3 Architecture

To support its containerized application model, Apiary must enable four abilities. First, Apiary must be able to run applications within secure containers to isolate applications from each other. Second, to provide a normal desktop display, Apiary must provide a single integrated display view that contains all of one's running applications. Third, Apiary must provide the ability for individual containers to be instantiated quickly and efficiently, as well as to manage them to enable the efficient creation of ephemeral containers. Finally, for the containers to provide a cohesive desktop experience, Apiary must provide the ability for applications within different containers to interact in a controlled manner

Apiary provides these abilities by using a virtualization architecture that consists of three main components: an operating system container that provides a virtual execution environment, a virtual display system that provides a virtual display server and viewer, and the Virtual Layered File System. Additionally, Apiary provides a desktop daemon that runs on the host, outside of any container. This daemon instantiates containers, manages their life times and ensures that they are correctly integrated together.

3.1 Process Container

Apiary's containers are essential to supporting Apiary's ability to isolate applications from one another. By providing isolated containers, individual applications can run in parallel within separate containers, and have no conception that there are other applications running. This enforces fault containment in the presence of an exploited application, as the exploited process will only have access to whatever files are available within the container itself.

Apiary's containers leverage modern operating system abilities, such as provided by Solaris's zones [22], FreeBSD's jails [14] and Linux's containers [17], to create isolated and independent execution environment containers. Apiary provides each container with its own private kernel namespace, FS and display server to provide total isolation at the process, file system and display lev-

els. Programs within separate containers can only interact with each other using normal network communication mechanisms. In addition, each container is provided with an application control daemon that enables the virtual display viewer to query the container for its contents and interact with it.

3.2 Display

Apiary's virtual display system is vital to ensuring the complete isolation of processes, as well as enabling the creation of a cohesive desktop experience for two reasons. First, if all the containers directly shared a single display, malicious and exploited applications can leverage built in mechanisms in commodity display architectures [9] to insert events and messages into other applications sharing the display. This enables the malicious application to remotely control the other applications, effectively exploiting them as well. Existing commodity security systems do not attempt to isolate applications at the display level, providing an easy mechanism for attackers to further exploit the desktop once a single application is compromised. Second, while providing each application with its own display will isolate the applications from each other, it does not provide the single coherent display users expect from their desktop. Providing this cohesive display has two elements. First, the actual display views have to be integrated into a single view. Second, Apiary has to provide the normal desktop usage metaphors that users expect. This includes a single menu structure for launching applications and an integrated task switcher that lets one switch focus between every running application.

Apiary's virtual display system solves both of these issues. First, Apiary's virtual display provides each container with its own virtual display similar to existing systems [8, 30, 3, 25]. This virtual display operates by decoupling the display state from the underlying hardware and enabling the display output to be redirected anywhere. Apiary's virtual display system operates as a client-server architecture and transparently provides a virtual display by leveraging the standard video driver interface, a well-defined, low-level, device-dependent layer that exposes the video hardware to the display system. Instead of providing a real driver for a particular display hardware, Apiary introduces a virtual display driver that intercepts drawing commands and redirects them to the Apiary client for display. All persistent display state is maintained by the display server within each independent container; the client is simple and stateless. This provides complete isolation at the display level, preventing an application within one container from leveraging the display to exploit applications in separate containers.

Second, Apiary enables these independent displays to be integrated into a single display view. While a regular remote framework provides all the information needed to display each desktop, they expect that they will be the only display in use, and therefore expect to be able to draw the entire display area. In Apiary, where one wants to use multiple containers, this expectation does not hold. Therefore, to enable multiple displays to be integrated into a single view, Apiary requires each display to provide an alpha channel color for its desktop background, so that the Apiary viewer can do Porter-Duff compositing [21] of the displays using the **over** compositing operation. This operation views object composition as a series of layers in a stack where objects higher in the stack will obscure elements of objects lower in the stack. Apiary uses this operation to stack the container displays and display all the windows associated with each display view. Apiary stacks the display views based on the currently used application and reorders the display view stack as one switches between applications. As each display provides the same resolution, and the displays are completely overlayed, this enables windows to appear anywhere in the composited desktop display.

As each container provides its own set of applications, Apiary must provide an integrated menu system that lists all the applications users are able to launch. Apiary achieves this by leveraging the application control daemon running within each container. This daemon uses the built in menu specification of the system to enumerate all the available applications within the container, much like a regular menu application in a traditional desktop. However, instead of providing the menu in the screen, it transmits the collected data back to the viewer, which integrates this information into its own menu, associating the menu entry with the container it came from. The viewer is able to communicate with every container and provide a single complete application menu. When a user selects a program from the viewer's menu, the viewer instructs the appropriate daemon to execute the program within its container. This causes it to be displayed on the correct container's display, while the Apiary viewer reorders the display stack, bringing that container's display to the top.

Similarly, to enable the effective management of running applications, Apiary provides a single taskbar that enables one to switch between all applications running within one's integrated desktop. Apiary leverages the system's ability to enumerate windows and switch application to have the daemon enumerate all the windows provided by its container and transmit this information to the viewer. The viewer is then able to integrate this information into a single taskbar that provides buttons that correspond to application windows. When the user uses the taskbar to switch which window is focused, the viewer communicates with the daemon and instructs it to

bring the correct window the foreground.

It should be noted that by stacking the independent displays, the windowing semantic is slightly changed from a traditional desktop. In a traditional desktop one can have multiple windows and when one raises a window to the foreground, only that single window will be raised. Similarly, in Apiary, each display supports the ability to display multiple windows that can be raised to the foreground. However, in Apiary, raising a window also involves raising its entire display layer to the foreground. Consequently, all the other windows provided by its display will be raised above the windows provided by other displays as well.

3.3 Virtual Layered File System

Apiary requires containers to be efficient in storage space and in instantiating time. Containers have to be efficient in storage space to enable regular desktops to support the large number of application containers that will be used within the Apiary desktop. Containers have to be efficient in being instantiated to provide fast interactive response time, especially for launching ephemeral containers. Both of these would be difficult to meet using traditional independent file systems (FSs) for each container. Each container's FS would be using its own storage space, which would be inefficient with a large number of containers, as there will be many duplicated files. More importantly, this would make a desktop much harder to maintain, as each independent FS will have to be updated individually. Similarly, instantiating the container involves making a copy of the FS, which can include many megabytes or gigabytes of storage space. This time to copy prevents the container from being instantiated in a timely manner. While any FS that supports a branching semantic [27, 4] can be used to quickly provision a new container FS from a template image, each template image will still be independent and therefore still be inefficient in space and in regards to upgrades and maintenance.

Apiary introduces the concept of a virtual layered filesystem to meet these requirements. The VLFS enables FSs to be created by composing layers together into a single FS namespace view. VLFSs are built by combining a set of shared software layers together in a read-only manner with a per container private read-write layer. Multiple VLFSs providing multiple applications are as efficient as a single regular FS as all files that are common between them will be stored once in the set of shared layers. Therefore, Apiary is able to store the FSs needed by its containers in an efficient manner. This also enables Apiary to manage its containers easily, as all one has to do is replace the single layer that contains the files that have to be updated to update each VLFS that uses it. The VLFS also enables Apiary to efficiently instantiate each container's FS. As no data has to be copied into place, as each of the software layers is shared in a read-only manner, instantiating a FS is a nearly instantaneous, and occurs transparently to the end-user.

Layers are the primary building block of a VLFS. Layers are composed of three elements: the metadata files that describe the layers, configuration scripts that enable the layer to be added and removed from the VLFS correctly, and the primary component, its FS namespace. The layer's FS namespace is a self-contained set of files providing a specific set of functionality. The files are the individual items in the layer that are composed into a larger VLFS. There are no restrictions on the type of files. They can be regular files, symbolic links, hard links or device nodes. The layer's FS namespace can be viewed as a directory stored on the shared FS that contains the same file and directory structure that would be created if the individual items were installed into a traditional FS. On a traditional UNIX system, the directory structure would typically contain directories such as `/usr`, `/bin` and `/etc`. Symbolic links work as expected between layers since they work on path names, but a limitation is that hard links cannot exist between layers.

Layers are stored on disk, as a directory tree that is named by the layer's name and its version. For instance, version 3.0.6 of the Firefox web browser, with a layer version of 1 would be stored under the directory `firefox_3.0.6-1`. Within this directory, VLFSs defines a `filesystem` directory that stores this layer's FS namespace, as well as a metadata file and scripts directory that stores those components. Each machine that hosts Apiary, stores its layers within a repository directory on its local FS. This repository's contents are just the individual layers that make up each container's VLFS.

To support the VLFS, Apiary must solve a number of FS related problems. First, to enable quick instantiation, the VLFS must support the ability to quickly compose numerous distinct FS layers into a single static view. Second, as users expect to be able to interact with the VLFS as a normal FS, such as by creating and modifying files, Apiary has to enable an instantiated VLFS to be fully modifiable, while enforcing the read-only semantics for the software layers. Finally, Apiary has to support the ability to dynamically add and remove layers without taking the FS off-line. This is equivalent to installing, removing or upgrading a software package that can be done while a monolithic FS is online.

To solve these problems, Apiary leverages and expands upon unioning file systems [33]. Unioning file systems enable Apiary to solve the first problem as they allow the system to join multiple distinct FS namespaces into a single namespace view. These directories are unioned by layering directories on top of one another,

joining all the files provide by all the layers into a single FS namespace view. As unioning requires no copying, it occurs quickly, enabling Apiary to be efficient in terms of provisioning.

To solve the second problem, the union semantic is extended [33] to enable the assignment of properties to the layers, defining some layers to be read only, while others are read-write. This results in a model that borrows from copy-on-write (COW) FSs, where modifying a file on a lower read-only layer will cause it to be copied to the topmost writable layer in a COW fashion. The VLFS leverages this property to enable multiple VLFSs to share a set of software layers in a read-only manner, while providing each instantiated VLFS with its own read-write private layer to store FS modifications. This enables Apiary to be efficient in terms of storage.

This layering model also provides a semantic that directory entries located at higher layers in the stack obscure the equivalent directory entries at lower levels. To provide a consistent semantic, if a file is deleted, a white-out mark is also created to ensure that files existing on a lower layer are not revealed. The white-out mechanism enables obscuring files on the read only lower layers, by just creating the white-out file on the topmost read-write private layer.

However, this creates a problem where a file deleted from a read-only share will never be able to reappear. Unlike a traditional FS, where a deleted system file can be recovered by simply reinstalling the package that provided that file, in a VLFS, white-outs that exist in the private layer will persist and continue to obscure the file even if the layer is replaced. The VLFS solves this problem by providing a private writable layer associated with each shared read-only layer in the VLFS for the storage of white-outs. Instead of writing a white-out file to the top-most layer, the white-out will be stored in the associated white-out layer. When a layer from which a file was deleted is replaced, its associated white-out layer will be replaced with an empty white-out layer as well, enabling it to be revealed.

Similarly, the VLFS has to handle the case where a file belonging to a shared read-only layer was modified and therefore copied up to the VLFS's private read-write layer. Apiary provides a *revert* command that enables the owner of a file that has been modified to revert the file's state to its original pristine state. While a regular VLFS *unlink* operation would remove the modified file from the private layer and create a white-out mark to obscure the original file, *revert* only removes the copy in the private layer thereby revealing the original copy below it.

Finally, VLFSs also have to support being managed while they are in use. In a traditional FS, an administrator can remove a package containing files in use, as deleting a file does not remove its contents from the FS until the file is no longer in use. However, if a layer is removed from a union, the data is effectively removed as well as unions only operate on FS namespaces and not on the date the underlying files contain. If an administrator wanted to modify the VLFS by removing a layer due to deletion or upgrade maintenance, one would be forced to perform the maintenance off-line due to not being able to remove layers that are in use.

The VLFS solves this problem by emulating what the `unlink` operation does on a single files and applies it to layer removal. `Unlink` operates in two steps. It first deletes the file name from the FS's namespace, while only freeing up the space taken up by the file's contents when its no longer in use. Traditional package management systems rely on this semantic to enable them to upgrade packages, even if files are in use, by unlinking and then recreating them instead of directly overwriting the files. Apiary applies this same semantic to layers. When a layer is removed from a VLFS, Apiary marks the layer as `unlinked`, removing it from the FS namespace. While this layer is no longer part of the FS namespace and therefore cannot be used by any operations that work on the FS namespace, such as `open`, it remains part of the VLFS enabling data operations, such as `read` and `write`, to continue to work correctly for files that were previously opened.

3.4 Inter-Application Integration

Apiary provides independent containers for fault containment, but must also ensure that they do not limit the ability of users to effectively use their desktops. For instance, if Firefox is totally isolated from the PDF viewer, how would users view a PDF file? While the PDF viewer can be included within the Firefox container, this breaks the isolation that should exist between Firefox and a PDF viewer that is viewing untrusted content. Similarly, users can copy the file from the Firefox container to the PDF viewer container to view it. However, this breaks the integrated feel users expect from their desktop as the application can no longer automatically launch.

Apiary solves this problem by enabling applications in one container to cause the instantiation of ephemeral containers and to cause the execution of a program within that new ephemeral container. Every container that is used within Apiary, is preconfigured with a list of programs that it enables other applications to use in an ephemeral manner. Apiary refers to these as **global** programs. For instance, a Firefox container can specify `/usr/bin/firefox` and a Xpdf container can specify `/usr/bin/xpdf` as global programs. Program paths that are marked global exist in all containers. Apiary accomplishes this by populating a single global layer, shared by all the container's VLFSs, with a wrapper pro-

gram for each global program. This wrapper program is used to instantiate a new ephemeral container and execute the requested process within it. Apiary only allows for the execution in a new ephemeral container and not in a preexisting persistent or ephemeral container, as that would break Apiary isolation constraints, and cannot be done in a safe manner to the preexisting container.

When executed, the wrapper program determines how it was executed and what options were passed to it. It connects over the network to the Apiary desktop daemon on the same host and passes this information to it. The daemon maintains a mapping of global programs to containers and determines which container is being requested to be instantiated in an ephemeral manner. This ensures that only the specified global programs' containers will be instantiated. preventing an attacker from instantiating and executing arbitrary programs. Apiary is then able to instantiate the correct fresh ephemeral container, along with all the required desktop services, including a display server. The display server is then automatically connected to the viewer. Finally, the daemon executes the program as it was initially called in the new ephemeral container.

To ensure that ephemeral containers are disposed of when they are no longer needed, Apiary desktop daemon monitors the process executed within the container, when it terminates, Apiary terminates the container. Similarly, as the Apiary viewer knows which containers are providing windows to it, if it determines that no more windows are being provided by the container, it will instruct the desktop daemon to terminate the container. This is to ensure that an exploited process does not stick around running in the background.

However, just running a new program in a fresh container is not enough to integrate applications correctly. When Firefox downloads a PDF and executes a PDF viewer, it has to enable the PDF viewer to view the file. As the Firefox and ephemeral PDF viewer containers do not share the same FS, this will fail. To enable this functionality, Apiary enables small private read-only file shares between a parent container and the child ephemeral container it instantiated. As well behaved applications, such as Firefox, Thunderbird and OpenOffice only use the system's temp directory to pass files between them, Apiary restricts this automatic file sharing ability to files located under `/tmp`. To ensure that there is no namespace conflicts between containers, Apiary provides containers with their own private directory under `/tmp` to use to store temporary files, and they are preconfigured to use that directory as their temp directory.

However, providing a fully shared temp directory will allow an exploited container to access private files that are placed there when passed to an ephemeral container. For instance, if a user downloads a malicious PDF and a bank statement in close succession, they will both exist in the temp directory at the same time. To prevent this, Apiary provides a special FS that enhances the read-only shares with an access control list (ACL) that determines which containers can access which files. By default, these directories will appear empty of files to the rest of the containers, as they do not have access to any of the files. This prevents an exploited container from accessing data that was not explicitly given to it. A file will only be visible within the directories if the Apiary desktop daemon instructs the FS to reveal that file by adding the container to the file's ACL. This occurs when a global program's wrapper is executed and the daemon determines that a file was passed to it as an option. The daemon then adds the ephemeral container to the file's ACL. As the directory structure is consistent between containers, simply executing the requested program in the new ephemeral container with the same options works without any modifications to the program.

Apiary enables its file explorer container discussed in Section 2 in a similar way. The file explorer container is set up like all other containers in Apiary. It is fully isolated from the rest of the containers and one interacts with it via the regular display viewer. It differs from the rest of the containers in that they are not fully isolated from it. Every container has two primary areas where the users files are written, the container's temporary directory under `/tmp`, and the container's version of the user's home directory. Apiary's file explorer provides two unique views of the these container directories. Each of these areas, from every container, is made available as a file share within the file explorer's FS namespace. Apiary provides this container with read-write access to each container, but prevents the explorer from executing any program provided within these FS. Users are able to use normal copy/paste file semantics to copy and move files between containers. While this is more difficult than a normal desktop that maintains only a single namespace, in general users do not have to move files between containers.

The primary situation where users might desire to move files between containers is when they are interacting with an ephemeral container, as a user might desire to preserve a file from that ephemeral container. For instance, a user can run their web browser in an ephemeral container to maintain privacy, but also downloaded a file they desire to keep. While the ephemeral container is active, a user can just use the file explorer to view all active containers. To avoid situations where one only remembers after one terminated the ephemeral application that there were files they desired to keep, Apiary also maintains access to the ephemeral container's FS for a user defined period after the ephemeral container is terminated, which by default it sets to be 30 minutes. This

gives the user enough time, even after they quit the application running within the ephemeral container, to access its FS and move the files they desire to preserve. After the time has elapsed and the FS share is no longer in use, Apiary removes the share that is associated with it from the file explorer's container FS namespace, and deletes it contents.

Apiary also turns the desktop viewer into an interprocess communication (IPC) proxy that can enable IPC state to be shared among containers in a controlled manner. For example, one of the most basic ways desktop applications share state is via the shared desktop clipboard. To handle the clipboard, each container's desktop daemon monitors the clipboard for changes. Whenever a change is made to one container's clipboard, this update is sent to the Apiary viewer, and then propagated to all the other containers. The Apiary viewer also keeps a copy of the clipboard so that any future container can be initialized with the current clipboard state. This enables users to continue to use the clipboard with applications in different containers in a manner that is consistent with a traditional desktop. This model can be extended to other IPC state and operations that one wants to share between containers in a controlled manner.

4 Experimental Results

We have implemented a remote desktop Apiary prototype system for Linux desktop environments. The prototype consists of a virtual display driver for the X window system that provides a virtual display for individual containers based on MetaVNC [25], a set of user space utilities that enable container integration and a loadable kernel module for the Linux 2.6 kernel that provides the ability to create and mount VLFSs. Apiary uses a Linux containers like mechanism to provide the isolated containers. [20]. The VLFS is implemented as an in-kernel stackable file system that implements the architecture described in Section 3.3.

Using this prototype, we use real exploits to evaluate Apiary's ability to contain and recover from attacks. We conduct a user study to evaluate Apiary's ease-of-use compared to a traditional desktop. We also measure Apiary's performance with real applications in terms of runtime overhead, startup time and storage efficiency. For our experiments, we compare a plain Linux desktop with common applications installed against an Apiary desktop that has the applications available to be used in persistent and ephemeral containers. The applications we used are the Pidgin instant messenger, the Firefox web browser, the Thunderbird email client, the OpenOffice.org office suite, the MPlayer media player and the Xpdf PDF viewing program. All experiments were conducted on an IBM HS20 eServer blade with dual 3.06 GHz Intel Xeon CPUs and 2.5 GB RAM. Participants in the usage study connected to the BladeCenter via a Thinkpad T42p laptop, with a 1.8 Ghz Intel Pentium-M CPU and 2GB of RAM.

4.1 Handling Exploits

We tested two scenarios that illustrate Apiary's ability to contain and recover from a desktop application exploit, as well as explore how different decisions can affect the security of Apiary's containers.

4.1.1 Malicious Files

Many desktop applications have been shown to be vulnerable to maliciously created files, that enable an attacker to subvert one's machine, as well as destroy a user's data. These attacks are prevalent on the Internet, as many users will download and view whatever files are sent to them. To demonstrate this problem, we use 2 malicious files [10, 11] that exploit old versions of Xpdf and mpg123 respectively. The mpg123 program was stored within the MPlayer container. The mpg123 exploit works by creating an invalid mp3 file that triggers a buffer overflow in old versions of mpg123 enabling the exploit to execute any program it desires. The Xpdf exploit works by exploiting a behavior of how Xpdf launched helper programs, namely by passing a string to `sh -c`. By including a back-tick (` `) string within a URL embedded in the PDF file, an attacker could get Xpdf to launch unknown programs. Both of these exploits are able to leverage sudo to perform privileged tasks, in this case, deleting the entire file system. Sudo is leveraged, as popular distributions require users to use it to gain root privileges, and have it configured to run any applications. Additionally, sudo, by default, caches the user's credentials to avoid having to authenticate the user each time they are required to perform a privileged action. However, this enables local exploits to leverage the cached credentials to also gain root privileges.

In the plain Linux system, recovering from these exploits required us to spend a significant amount of time reinstalling the system from scratch, as we had to install many individual programs, not just the one that was exploited. Additionally, we have to recover a user's 23GB home directory from backup. Reinstalling a basic Debian installation took 19 minutes. However, reinstalling the complete desktop environment took a total of 50 minutes. Recovering the user's home directory, which included many multimedia files, research papers, email and many other assorted files, took an additional 88 minutes when transfered over a 1Gbps LAN.

Apiary protected the desktop as well as enabled easier recovery. It protected the desktop by letting the malicious files be viewed within an ephemeral container. Even

though the exploit proceeded as expected and deleted the container's entire file system, the damage it caused is invisible to the user, as this ephemeral container will never be used again. Even when we let the exploit execute within a persistent container, Apiary enabled significantly easier recovery from the exploit. As shown in Table 2, Apiary can provision a file system in just a few milliseconds. This is nearly 6 orders of magnitude faster than the traditional method for recovering a system by reinstallation. Furthermore, Apiary's persistent containers divide up home directory content between them, eliminating the need to recover the entire home directory if one application gets exploited.

This also shows how persistent containers can be constructed in a more secure manner, to prevent exploits from harming the user. As a large amount of this user's data is only accessed in a read-only manner, such as multimedia files, the data can be stored on FS shares. This enables the user to allow the different containers to have different levels of access to the share. The file explorer container can access it in a read-write manner, enabling a user to manage the contents of the FS share, while the actual applications that view these files can be restricted to accessing them in a read-only manner, protecting the files from any exploit to the application.

4.1.2 Malicious Plugins

Applications are also exploited via malware that users are tricked into downloading and installing. This can be an independent program, or a plugin that integrates with an application a user already has installed. For example, malicious attackers try to convince users to download a "codec" they need to view a video. Recently, a malicious Firefox extension was discovered [6] that leveraged Firefox's extension and plugin mechanism to extract a user's banking username and password from the browser when the user visited their bank's website, and send the information to the attacker. These attacks are common as users are badly conditioned to allow a browser to install what it needs when it asks to install something. When installed into a traditional environment, this malicious extension persists until the user, or the user's anti-virus software discovers and removes it. As it does not impact the regular use of the browser, there is very little to tip off users that they have been attacked. As this exploit is not readily available to the public, we simulated its presence with the non-malicious Greasemonkey Firefox extension. Much like the malicious file example, Apiary prevented the extension from persisting when installed into an ephemeral container. Even when a user allowed the installation of the extension, it did not persist to future executions of Firefox.

However, this exploit poses a significant risk if it enters the user's persistent web browser container. While one might think that Firefox extensions should be uninstallable through Firefox's extension manager, this is only for extensions that are installed through it. If an extension is installed directly into the FS, it cannot be uninstalled this way, though it can be disabled, and one has to remove it from the file-system. This applies equally to Apiary and traditional machines. While users can quickly recreate the entire persistent Firefox container, that involves knowing that the installation was exploited. Apiary enables us to handle this situation in a more elegant manner by enabling a user to use the Firefox program in multiple web browsing application containers. In this case, we created a general purpose web browsing container for regular usage, as well as a financial web browsing container to only use to access our bank's website. By refusing to install any addons into the financial web browsing container, it remained isolated and secure, even when we exploited our general purpose web browsing container.

This scenario indicates how one does not need to be stuck in the mode of having only a single application for all types of actions that fall within a specific type of task. Apiary enables the creation of multiple independent application containers, that contain all of the same application, but are meant to perform different tasks, such as visiting one's bank website in this example. As the large majority of the VLFS's layers are shared, there is very little cost to the user for enabling these multiple independent containers. This can be extended to other related but independent tasks, for instance using a media player to listen to one's personal collection of music, as opposed to listening to Internet radio from an untrusted source.

4.2 Usage Study

We performed a usage study that evaluated the ability of users to use Apiary's containerized application model with our prototype environment. Participants were mostly recruited from within our local university, including faculty, staff and students. All of the users were experienced computer users. 24 participants took part in the study.

For our study, we created three distinct environments: A plain Linux environment running the Xfce4 desktop, A full Apiary desktop, and a neutered Apiary desktop that did not launch child applications in ephemeral containers but within a preexisting containers. These three environments enable us to compare the participants experience along two axis. First, we can compare the plain Linux environment where each application is only installed once and always run from the same environment against the neutered Apiary desktop where each application is also only installed once and run from the same environment. This enables us to measure the cost of using the Apiary viewer with its built in taskbar and appli-

cation menu against plain Linux where the taskbar and application menu are regular applications within the environment. Second, the full and neutered Apiary desktops enable us to isolate the actual and perceived cost to the participants of instantiating ephemeral containers for application execution. We presented the environments to the participants in random order and iterated through all 6 permutations equally.

We timed the participants as they performed a number of specific multi-step tasks in each environment that in summary are: (1) Download and view a PDF file with Firefox and Xpdf and follow a link embedded in the PDF back to the web. (2) Read an email in Thunderbird that contains an attachment that is to be edited in OpenOffice and returned to the sender. (3) Create a document in OpenOffice that contains text copied and pasted from the web and sent by e-mail as a PDF file. (4) Create and preview a web page in OpenOffice and Firefox. (5) Launch a link received in the Pidgin IM client in Firefox.

As Figure 2 shows, the average time to complete each task when averaged over all the users doing the tasks in random order only differed by a few seconds in any different direction for all tasks in all environments. Figure 2 shows that in all cases users performed their tasks quicker in the neutered Apiary environment than in the plain Linux environment. This indicates that Apiary's simpler environment is actually faster to use than the plain Linux environment that contains bells and whistles that make it friendlier to the user, such as application launchers and running applets within their taskbar panels. Furthermore, even though users were a little slower in the full Apiary environment compared to the neutered version, they were still generally faster than plain Linux environment. This indicates that while the fully environment has a small amount of overhead, in practice users are just as effective in it as in a plain Linux environment.

We also asked to rate their perceived ease of use of each environment. Most users perceived the prototype environments to be as easy to use as the plain Linux environment. While some users preferred the polish of the plain Linux environment, more preferred the simplicity of the environment provided by Apiary. Most users could not determine a difference between the full and neutered Apiary's desktops.

We also asked the participants a number of 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. All of the participants expressed a willingness to use this environment as their full time environment, and a large majority indicated that they would prefer to use Apiary over the plain Linux environment if it would keep their applications more secure. The majority of those who would not prefer to use Apiary, indicated it was because of bugs they perceived in their interaction with the prototype.

Test	Description
Untar	Extract Linux 2.6.19 kernel source archive
Gzip	Compress a 250MB Linux kernel source archive
Octave	Octave 3.0.1 running a numerical benchmark
Kernel	Build the 2.6.19 kernel

Table 1: Application Benchmarks

4.3 Performance Measurements

4.3.1 Application Performance

To measure the performance overhead of Apiary on real applications, we compare the runtime performance of a number of applications within the Apiary environment against the performance in a traditional environment.

Table 1 lists our application tests. We focus mostly on FS benchmarks as others have shown [20, 3] that display and operating system virtualization have little overhead. The untar tests file creation and throughput, while the gzip tests file system throughput and computation. The Octave benchmark is a pure computation benchmarks. The kernel build benchmark tests both computation, as well as stresses the FS due to the large amount of lookups that occur due to the large size of the kernel source tree and the repeated execution of the preprocessor, compiler and linker. To stress the system with many containers and to provide a conservative measure of performance each test was run in parallel with 25 instances. To avoid out of memory conditions, as the Octave benchmark requires 100-200MB of memory at various points during its execution, we ran the benchmarks staggered 5s apart to ensure they kept their high memory usage areas isolated to avoid the benchmark being killed by Linux's out-of-memory handler. As is shown in Figure 3, Apiary imposes almost no overhead in most cases, with about 10% overhead in the kernel build case due to the constant need to perform lookups on the FS which the VLFS imposes an extra cost. This demonstrates that Apiary is able to scale to a large number of concurrent containers with minimal overhead.

4.3.2 Container Creation

For ephemeral containers to be useful, container instantiation must be quick. We measured this cost in two ways. First, how long does it takes to instantiate its VLFS. Second, how long does it take the application to start up within the container. We quantify how long it takes to instantiate a container and compare Apiary against other common approaches. We compare how long it takes to setup a VLFS against how long it takes to setup a con-

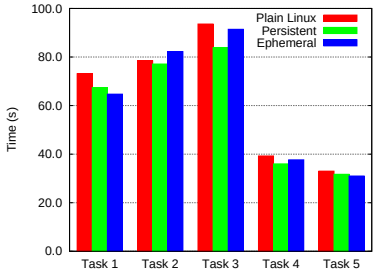

Figure 2: Usage Study Task Times

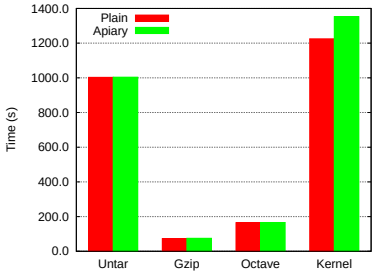

Figure 3: Overhead at Scale

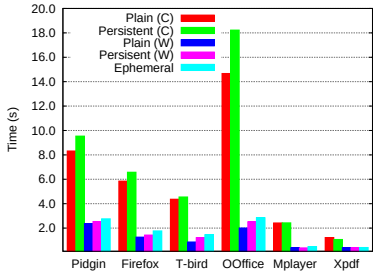

Figure 4: Application Startup Time

	P	**F**	**T**	**O**	**X**	**M**
Create	317s	276s	294s	365s	291s	294s
Extract	82s	86s	87s	150s	81s	81s
FS-Snap	.016s	.015s	.016s	.020s	.009s	.010s
Apiary	.005s	.005s	.005s	.005s	.005s	.005s

Table 2: FS Instantiating Times for (P)idgin, (F)irefox, (T)hunderbird, (O)penOffice, (X)pdf and (M)Player

tainer FS using Debian's traditional bootstrapping tools (**Create**), how long it would take to extract the same file system from a tar archive (**Extract**), and how long it takes a FS with a snapshot operation to create a new snapshot and branch of a preexisting file system namespace (**FS-Snap** as shown in Table 2. To minimize network affects with the bootstrapping tools, we used a local Debian mirror on the local 100Mbps campus network, and we were able to saturate the connection while fetching the packages that were to be installed.

Table 2 shows that Apiary instantiates containers with a VLFS composed of nearly 200 layers nearly instantaneously. This compares very positively with traditional ways of setting up a system. Table 2 show that it takes a significant amount of time to create a FS for the application container using Debian's bootstrapping tool and even extracting a tar archive takes a significant amount of time. This would prevent one from creating ephemeral application containers as users will not want to wait minutes for their applications to start. Tar archives also suffer from the fact that they have to be actively maintained and rebuilt whenever fixes need to be applied to them. Therefore, the amount of administrative work increases linearly with the number of applications in use. As Apiary creates the FS nearly instantaneously, it is able to support the creation of ephemeral application containers with no noticeable overhead to the users. While Table 2 shows that file systems, in this case Btrfs, with a snapshot and branch operation can also perform it quickly, the user would have to manage each of the application's independent file-systems separately.

To quantify startup time, we measured how long it takes for the application to open and then be automatically closed. In the case of Firefox, Xpdf and OpenOffice.org, this includes the time it takes to display the initial page of a document, while Pidgin, MPlayer and Thunderbird are all just loading the program. For ephemeral containers, we measure the complete time it takes to setup the container and execute the application within it. We compare these results against cold and warm cache application startup times for both plain Linux and Apiary's persistent containers. We include cold cache results for benchmarking purposes and warm cache results to demonstrate the results users would normally see.

As Figure 4, shows, while running within a container induced some overhead on startup, it's generally under 25% in both cold and warm cache scenarios. This overhead is mostly due to the added overhead of opening the many files needed by today's complex applications. The most complex application, OpenOffice, requires the most, while the least complex application, Xpdf, is almost equivalent to the plain Linux case. In addition, while the maximum absolute extra time spent in the cold cache case was nearly 5s for OpenOffice, in the warm cache case it dropped to under .5s. In addition, Ephemeral containers provide an interesting result. Even though they have a fresh new FS and would be thought to be equivalent to a cold cache startup, they are nearly equivalent to the warm cache case. This is due to the fact that their underlying layers are already cached by the system. The ephemeral case has a slightly higher overhead due to the need to create the container and execute a display server inside of it in addition to the regular application startup time, but this takes under 10ms and adds only a minimal amount to the ephemeral application startup time.

4.4 File-System Efficiency

To support a large number of containers, Apiary must store and manage its FS efficiently. This means, that storage space should not significantly increase with an increasing number of instantiated containers, as well as be easily manageable in terms of application updates. For each application's VLFS, Table 3 shows its size, its number of layers, and the amount of state shared with the other application VLFSs and the amount of state unique to it. For instance, the 129 layers that make up Firefox's VLFS require 353MB of which 330MB are shared with other applications in this scenario and 23MB are unique

	P	F	T	O	X	M
Size (MB)	394	353	367	645	339	355
# Layers	147	129	125	186	130	162
Shared (MB)	322	330	335	329	330	326
Unique (MB)	72	23	32	316	9	29

Table 3: VLFS Layer Storage Breakdown for (P)idgin, (F)irefox, (T)hunderbird, (O)penOffice, (X)pdf and (M)Player

	Single FS	Multiple FSs	VLFSs
Size	743MB	2.1GB	743MB

Table 4: Single Desktop vs. Multiple Container FSs

to the Firefox VLFS. In general, Table 3 shows, there is a lot of duplication among the containers, as the layer repository of 214 distinct layers, needed to build the different VLFSs for the different applications, is the same magnitude as the largest application.

Table 4 shows that using individual VLFSs for each application container consumes approximately the same amount of FS space as a regular desktop FS containing all the applications, as each layer only has to be stored once. This is comparison to the traditional method of provisioning multiple independent FSs for each application container which consumes a significantly larger amount of disk space. Similarly, if one would provide multiple desktops on a server, the VLFS usage would remain constant to the size of the repository, while the other cases would grow linearly with the number of desktops.

To demonstrate how Apiary improves the ability of users to maintain their many containers, we instantiated one container for each of the five applications previously mentioned. When a security update was necessary, we iterated through each container applying the security update. Table 5 shows the average times for the five application container FSs. This demonstrates that while individual updates by themselves are not too long, when one has multiple container FSs for each individual user, the amount of time one spends applying common updates will rise lineraly, and as the traditional method is two orders of magnitude greater than Apiary, will impact to a much greater extent.

5 Related Work

Isolation mechanism, such as VMs [28, 32] and OS containers [22, 26], have long been used to increase the security of applications. As it is confined to a virtualized environment, it is isolated from the rest of a user's applications and data. However, this means the applications are not integrated into the user's desktop experience. For instance, each application is totally independent and cannot leverage each other. VMs also suffer high overhead due to running independent operating systems. This impacts performance, as well as making them unsuitable for ephemeral usage due to their long startup times. Products, like VMware's Unity [29], attempt to solve part of this issue by combining the applications from multiple VMs into a single display, with a single menu and taskbar, as well as providing file-system sharing between the host and the VMs. However, the applications are still fully isolated from one another, preventing them from leveraging other applications installed into separate VMs.

	Traditional	Apiary
Avg. Time	18 s	0.12s

Table 5: Update Times

Tahoma [24] is similar to Apiary in that it creates fully isolated application environments that remain part of a single desktop environment. Tahoma lets one create a browser applications that are limited to what resources, such as URLs they are allowed to access and that are fully isolated from each other. Tahoma is similar to Apiary in that it enables the creation of isolated application environment. However, it only provides these isolated application environments for web browsers. It does not provide any way for these isolated environments to be integrated together and does not provide for ephemeral application environments.

Google's Chrome web browser [12] builds upon some of these ideas to provide isolation between web browser pages within a single browser. However the browser as a whole does not offer any isolation from the system, allowing an exploit that can escape the browser to become unrestricted. Chrome also provides an incognito mode, to provide additional privacy by preventing browser state from being committed to disk. While it serves a similar to ephemeral containers, incognito mode had to be written into the program itself and only provide basic means of privacy. For instance, it cannot prevent a plugin from writing state to disk. Apiary's ephemeral containers make the entire execution private and support any application with state a user desires to remain private without any application modifications.

Apiary's ability to run multiple applications in parallel resembles Lampson's Red/Green isolation [15]. Red/Green isolation involves users running two separate environments, a red environment for regular usage, and a green environment for environments they require a trusted environment. However, unlike Apiary's ephemeral containers, if an exploit can enter the green container, it will persist. Furthermore, by requiring two separate virtual machines, one increases the amount of work a user has to do to manage their machines. Apiary, by leveraging the VLFS minimizes the overhead required required to manage multiple machines.

FSs with a branching semantic [27, 4] can be used to quickly create a fresh FS namespace for a new container. However, these FSs do not help manage the large number of containers that will exist within Apiary. As each container has a unique FS, with differing sets of appli-

cations, administrators will have to create individual FSs tailored to each application. They cannot create a single template FS with all applications as applications can have conflicting dependency requirements or desire to use the same FS path locations. Furthermore, by putting all the applications into a single FS, they will no longer be isolated from each other. This results in a set of FSs that are inefficient in space, as each FS will have an independent copy of many files common to other. This inefficiency also makes management harder. When security holes are discovered and fixed, one will have to update each individual FS independently.

Many systems have been created that attempt to provide security through isolation mechanisms [18, 31, 23, 1, 5, 7, 16]. All these systems differ from Apiary in that they try to isolate the many different components that make up a standard fully integrated single system using sets of rules to determine which of the machine's resources the application should be able to access. This many times results in two outcomes. First, a policy is created that is too strict and does not let the application run correctly. Second, a policy is created that is too lenient and lets an exploited application interact with data and applications it should not have access to. Apiary, on the other hand, forces each components to be fully isolated within its own container before determining on what levels it should be integrated. As each container provides all the resources that the application needs to execute in an isolated environment, no complicated rule sets have to be created to determine what it needs access to.

Solitude [13] provides isolation via its Isolation FS (IFS) which a user can throw away. This is similar to Apiary's ephemeral containers. However, the IFSs are not fully isolated. First, Solitude does not create a new IFS for each application execution. Second, the IFS is built on top of a base file system which it can share data with, breaking the isolation. To handle this, Solitude implements taint tracking on files shared with the underlying base file system. This helps determine, post-facto, what other applications may have been corrupted by a maliciously constructed file. Similarly, Solitude only provides isolation at the file system level. As each application shares a single display, malicious and exploited applications can leverage built in mechanisms in commodity display architectures [9] to insert events and messages into other applications sharing the display.

6 Conclusions

Apiary introduces a new compartmentalized application desktop paradigm. Instead of running one's applications in a single environment with complex rules to isolate the applications from each other, Apiary enables them to be easily and completely isolated while retaining the integrated feel users expect from their desktop computer. The key innovations that make this possible are the introduction of virtual layered file-systems and the ephemeral containers they enable. The VLFS enables the multiple containers to be stored as efficiently as a single regular desktop, while also allowing containers to be instantiated almost instantly. This functionality enables the creation of the ephemeral containers that provide an always fresh and clean environment for applications to run in. Ephemeral containers prevent malicious data from having any persistent effect on the system and isolate the fault to only that single application instance.

We have implemented Apiary on Linux without requiring any operating system kernel or application changes. Our results demonstrate that Apiary's containerized desktop severely reduces the threat posed by malicious files and plugins by isolating them in ephemeral containers and enabling users to quickly recover if they penetrate a persistent container. Our 24 person usage-study demonstrates that Apiary is as easy to use as a regular Linux desktop by both measuring the time it took users to perform their tasks and their subjective opinions. Furthermore, we demonstrate that Apiary add minimal overhead to application performance, is as efficient as a regular desktop in its use of storage space and instantiates ephemeral containers in less than .5s.

References

[1] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine applications. In *Proceedings of the 2000 USENIX Security Symposium*, August 2000.

[2] Adobe Systems Incorporated. Buffer overflow issue in versions 9.0 and earlier of Adobe Reader and Acrobat. `http://www.adobe.com/support/security/advisories/apsa09-01.html`, Feb 2009.

[3] R. Baratto, L. Kim, and J. Nieh. THINC: A Virtual Display Architecture for Thin-Client Computing. In *Proceedings of the* 20^{th} *ACM Symposium on Operating Systems Principles*, Oct. 2005.

[4] M. R. Ben Pfaff, Tal Garfinkel. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *3rd Symposium of Networked Systems Design and Implementation*, May 2006.

[5] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proc. of 1995 USENIX Winter Technical Conference*, pages 165–175, 1995.

[6] bitdefender. Trojan.pws.chromeinject.b. http://www.bitdefender.com/VIRUS-1000451-en--Trojan.PWS.ChromeInject.B.html, Nov 2008.

[7] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Server Security. In *14th USENIX Systems Administration Conference*, New Orleans, LA, Dec. 2000.

[8] B. Cumberland, G. Carius, and A. Muir. *Microsoft Windows NT Server 4.0, Terminal Server Edition: Technical Reference*. Microsoft Press, Redmond, WA, Aug. 1999.

[9] J. Gettys and R. W. Scheifler. *Xlib - C Language X Interface*. X Consortium, Inc., 1996. p. 224.

[10] M. Gilmore. 10day cert advisory on pdf files. http://seclists.org/fulldisclosure/2003/Jun/0463.html, Jun 2003.

[11] GOBBLES Security. Local/remote mpg123 exploit. http://www.opennet.ru/base/exploits/1042565884_668.txt.html.

[12] Google. Google Chrome - Features. http://www.google.com/chrome/intl/en/features.html.

[13] S. Jain, F. Shafique, V. Djeric, and A. Goel. Application-level Isolation and Recovery with Solitude. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 95–107, 2008.

[14] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *2nd International SANE Conference*, MECC, Maastricht, The Netherlands, May 2000.

[15] B. Lampson. Accountability and Freedom. http://research.microsoft.com/en-us/um/people/blampson/slides/accountabilityandfreedom.ppt, Sept. 2005.

[16] Z. Liang, V. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *19th Annual Computer Security Applications Conference*, December 2003.

[17] Linux Containers. http://lxc.sourceforge.net/.

[18] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June 2001.

[19] Microsoft. Microsoft Application Virtualization. http://www.microsoft.com/systemcenter/appv/default.mspx.

[20] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[21] T. Porter and T. Duff. Compositing digital images. *Computer Graphics*, 18(3):253–259, July 1984.

[22] D. Price and A. Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *18th Large Installation System Administration Conference*, November 2004.

[23] N. Provos. Improving Host Security with System Call Policies. In *12th USENIX Security Symposium*, Washington, DC, Aug. 2003.

[24] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *4th ACM European Conference on Computer Systems (EuroSys 2009)*, Nuremberg, Germany, Mar 2009.

[25] U. Satoshi. Metavnc - a window aware vnc. http://metavnc.sourceforge.net/.

[26] S. Soltesz, H. Pötzl, M. e. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, 2007.

[27] Sun Microsystems, Inc. *Solariz ZFS Administration Guide*. 2009.

[28] VMware, Inc. http://www.vmware.com.

[29] VMware Inc. VMware Worksation 6.5 Release Notes. http://www.vmware.com/support/ws65/doc/releasenotes_ws65.html, Oct 2008.

[30] Virtual Network Computing. http://www.realvnc.com/.

[31] D. Wagner. Janus: an approach for confinement of untrusted applications. Master's thesis, University of California, Berkeley, 1999.

[32] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[33] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage*, 2(1):1–32, February 2006.