A Survey on Application Sandboxing Techniques

[placeholder for review]

Abstract:

The principle of least privilege *states that components in a system should only be allowed to perform actions that are required for them to function. The wish to limit what programs can access has given rise to a set of application-level sandboxing solutions. In this paper, we survey recent research on application-level sandboxing. We discuss the properties of the major implementations and highlight the key differences between them. In addition, we highlight how recent features in mainline Linux kernel have altered the sandboxing landscape.*

***Key words**: Security, Sandboxing, Program Isolation*

INTRODUCTION

Protecting operating system and user data from untrusted applications has been a major issue in computer security. Even a non-technical user would have a hard time avoiding hearing guidelines like *"be careful of unknown email attachments"*. The problem with unknown email attachments is two-fold: in case of an executable attachment, the attached application can typically do everything the users themselves can. However, even attachments that appear to be safe, that is, those that do not usually contain executable code, can cause problems if they manage to exploit a flaw in the program used to inspect the file. Opening a malicious photo in a trusted but vulnerable image viewing application could result in untrusted code being executed.

The reason why such guidelines are necessary is insufficient application isolation and overly broad default capabilities of programs. Users are often able to download and run programs of unknown origin, which, in turn, have wide-ranging capabilities to inspect and share user's personal data. Because of the lack of restrictions, even programs that are nominally safe can be turned malicious through code injection attacks. Why should the aforementioned image viewer ever have access to, for example, user's email? What is clearly needed is a more fine-grained way to limit capabilities granted to applications.

This need for better management of what applications can and cannot do, has created a whole field of different isolation solutions and technologies. The solutions range from traditional full-system virtualization to mechanisms relying on custom kernel-level components to sandboxes that work entirely in user space.

In this paper, we survey the existing research on per-application sandboxing solutions and discuss the mechanisms that are used for application containment. Our focus is in implementations that provide what Li et al. [10] call *one-way protection*, that is, they protect the operating system and user's data from the contained application but do not try to protect the application from the operating system.

BACKGROUND

Traditionally, Unix-like operating systems have granted applications with a wide set of capabilities. For example, a typical Linux application is able to use any of the hundreds of system calls supported by the operating system as well as access all the files available to the user.

The underlying adversary model is concerned about protecting the system from its users: a malicious user cannot access files belonging to other users and is prevented from performing operations that could have system-wide consequences. What is not protected, however, are users from the applications they execute.

System calls play a large role in application sandboxing. System calls are the mechanisms that enables applications to request services from the underlying operating system and reach outside the tight confines of their own address space. As Goldberg et al. [7] stated: *"An application can do little harm if its access to the underlying operating system is appropriately restricted."*

Considering their central role, looking at how applications can access system calls is a natural starting point for those wishing to limit what applications can do.

System Call Interposition (SCI) based application containment solutions work by listening the contained application for any system call invocations. When a system call invocation happens, the sandbox checks if the application is allowed to perform the call. If the call is not allowed, the contained application can be terminated.

PREVIOUS RESEARCH

There has been a number of publications covering the application sandboxing landscape. Garfinkel [5] discussed the common problems that system call interposition based software security solutions face. They based their analysis partly on the experiences gained from implementing Janus [4, 7], SCI-based sandboxing system targeting Solaris.

Al Ameiri and Salah [1] surveyed available sandboxing implementations and provided benchmarks of their effect on program execution times and memory, disk, and network performance. However, their focus was mainly in implementations targeting Windows environment.

Shu et al. [16] presented a comprehensive survey of security isolation techniques. The authors provided a hierarchical classification of different isolation techniques and considered the different solutions in terms of the mechanisms used and the way policies are handled.

METHODOLOGY

For this survey, we reviewed literature that dealt with containing individual applications and not, for example, whole systems. As an additional criteria for inclusion, we wanted the sandboxes to be general in that they were not restricted to programs written in particular language or against particular framework. However, even a generic sandbox might impose some restrictions on the kinds of programs it can successfully run, in these cases, we erred on the side of inclusion. More importantly, we only consider implementations targeting Unix-like operating systems (Linux, Solaris, various BSD variants). Additionally, we left out solutions specifically targeting mobile use.

Focusing our survey this way allows us to better highlight the similarities between different implementations and how the different solutions tackle the common challenges. By leaving out software offering radically different solutions to program isolation, we are able to highlight the differences between the included solutions that might otherwise look subtle if viewed as a part of wider context of isolation mechanisms.

IMPLEMENTATIONS

Janus

Janus [4, 7] was one of the earliest system call interposition based sandboxing solutions. While the early versions of the project were exclusive to Solaris due to its better support for system call tracing, later versions of the project added support for Linux with a help of a custom kernel module that extends the standard `ptrace` API.

Policies deciding which system calls to allow are defined through configuration files. Each configuration directive references a particular policy module implementing logic for deciding whether or not any particular system call should be allowed. By default, all system calls are denied. As an example, the `path` module could be used to deny or allow certain file IO related system calls based on file paths.

Based on the experiences gained from implementing the system, the authors have discussed [5] the challenges associated with implementing a system call interposition based process sandboxes.

Ostia

Where most of the surveyed sandboxes are based on filtering system calls from the sandboxed application, Ostia [6] introduces an architectural alternative based on system call delegation. In delegating architecture, the sandboxed process itself does not perform the system calls but instead, the system calls are performed by an outside agent and the results are transferred back to the sandboxed application. According to the authors, this change of design makes the system simpler and protects it from argument races since the arguments are transferred to the agent.

Ostia is implemented as a small kernel module and a user space component. The kernel module is responsible for preventing the sandboxed program from directly executing any unsafe system calls. Instead, the kernel module invokes a callback placed within the address space of the sandboxed process. The callback then transforms the system call request into an IPC call to the agent. The agent, in turn, receives the system call requests and, after deciding if the system call should be allowed, executes the request and passes the results back over the IPC link. However, it is noteworthy that not all system calls have to go through the delegation process: Some system calls can always be permitted while others are always denied.

Consh

Consh [2] is a sandboxing project that re-uses components from Janus, but extends its functionality with a virtualized file system. Consh targets Solaris and is implemented entirely in user space and requires no superuser privileges.

The core of Consh is Catcher, its system call interposition pipeline. System calls performed by the sandboxed process are handled by Catcher, which is able to sequentially route them through a series of processing steps. Each step of the process is able to perform arbitrary operations based on the call and its arguments and before passing the possibly transformed call to the subsequent steps. One such step is the decision engine from Janus. As a last step of the presented pipeline, Janus is able to deny any system call formulations produced by the earlier steps.

What separates Consh from Janus is its ability to provide sandboxed processes with a virtualized view of the file system. This is achieved through a pipeline step that transforms file system related calls. On a conceptual level, this approach to providing virtualized file system access places Consh closer to MBOX, a more recent sandbox implementation.

This ability to provide virtualized views of the file system is used to present applications with initially empty file system and that can be selectively expanded by the user. In addition, Consh provides access to network resources through the same file system abstraction. A special directory can be used to access HTTP and FTP end points using regular file system operations.

Systrace

Systrace [13] provides application sandboxing and limited privilege elevation through system call interposition. The system contains a custom kernel component that either consults an in-kernel database for system calls that are known to be safe or, alternatively, asks a user space for policy decision. The decision to make some of the choices in kernel is argued to lessen the performance overhead associated with sandboxing.

The system solves the argument race problem by copying system call arguments to the kernel memory before the user space is consulted for policy decision. This prevents a competing thread from changing the arguments before the system call is executed. Additionally, the system normalizes system call arguments to reduce the problem of indirect paths to resources.

An interesting feature of the Systrace system is the ability to partially elevate the privileges of a process by allowing individual system calls that would normally require super user privileges to complete.

All sandboxes need policies to decide which system calls to allow. Systrace provides users with the ability to interactively create policies as the application is running. When a policy decision needs to be done the user can be prompted to allow or deny the call in question. In addition to this interactive process, Systrace can learn applications' regular system call behaviors by recording their system call use and creating a policy based on the results.

MiniBox

MiniBox [10] is two-way sandbox protecting both the operating system from the application and the application from the operating system. In order to protect the underlying operating system, MiniBox re-uses Software Fault Isolation (SFI) techniques from NaCl [20] to prove that the confined program does not perform unsafe operations. To protect the application from a malicious OS, MiniBox contains a modified version of TrustVisor [11] hypervisor. This enables MiniBox to prevent memory access from OS to the sandboxed application.

In MiniBox, sandboxed applications are confined to a Mutually Isolated Execution Environment (MIEE). The hypervisor represents the only channel of communication between this isolated environment and the regular OS. When the confined application makes a system call, it is first handled and checked by the hypervisor and then transmitted to the regular operating system.

Additionally, MiniBox makes use of Trusted Platform Modules (TPM) to provide remote attestation capabilities. The system is capable of proving the integrity of sandboxed applications by recording their hashes into the TPM.

MBOX

MBOX [9] is sandboxing solution making use of recent features of the Linux kernel, namely seccomp/BPF-based system call filtering (BPF = Berkeley Packet Filter). It requires no custom kernel components but is built entirely on existing APIs. It is also notable that the program requires no superuser privileges.

While the traditional `ptrace`-based process tracing API was problematic for application sandboxing [5] because of the lack of fine-grained control over the traced system calls, Linux's seccomp facility supports defining BPF-based programs for reacting to different system calls. MBOX uses this feature to selectively invoke the monitoring process based on the system call.

Race conditions related to system call arguments are a common pitfall that sandboxing implementations have to solve. MBOX tackles this problem by allocating a read-only page in the contained process and copying the arguments there. Since the application cannot change the access permissions on the write protected page without a system call, the arguments stay safe from competing threads.

MBOX restricts changes the programs can enact on the file system by exposing them a transparent layered file system with copy-on-write semantics. If a sandboxed program tries to write to a file, the changes will be stored in a separate location and the user is able to later decide if they want to commit the changes to the original files. It is interesting to note that the implementation of this system does not depend on any “real” file system implementations, but the copy-on-write semantics are provided entirely through the use of system call interposition techniques.

Firejail

Like MBOX, Firejail [12] takes advantage of recent kernel features in order to sandbox programs. However, where MBOX intercepted system calls and rewrote their arguments, Firejail restricts applications access to file system through the use of mount namespaces, a kernel feature allowing processes to have their own view of the file system. In the same vein, namespaces are used to provide sandboxed application with their own view of running processes and network devices. However, it is noteworthy that the use of these features requires Firejail to contain a `setuid` helper binary that is able to create the required changes.

Sandboxing environment can be defined through a set of command line arguments, or through profile definitions. By default, Firejail tries to find an appropriate profile based on application name. Profiles define which resources should be made available to the application and which system calls they can perform. In addition, profiles can inherit from other profiles making creation of derived profiles simpler.

nsjail

nsjail [8] is another sandboxing solution primary utilizing namespaces and seccomp for isolation. A distinctive feature of the system is its use of control groups for resource limits. For example, nsjail allows the user to limit the amount of memory or CPU time that the sandboxed program can consume.

nsroot

nsroot [14] is a minimalistic sandboxing tool making use of Linux namespaces to provide sandboxed applications with their own view of the file system and restricted access to networks. In many respects, nsroot is similar to Firejail and nsjail in its functionality and implementation if less limited in its apparent configurability. The authors note the similarities and suggest that nsroot has the advantage of supporting similar usage patterns as the traditional `chroot` command.

Flatpak

Previously known under the name xdg-app, Flatpak [3] aims to provide means for portable application packaging and sandboxing solution for Linux. The sandboxing functionality of Flatpak was recently split into a separate tool, Bubblewrap. The sandbox provides unprivileged users with the ability to confine applications using namespaces and Seccomp.

EVALUATION

Implementation Mechanisms

The more recent work on application sandboxing shows increased homogeneity in implementation strategies as relevant functionality has found its way into the mainline Linux kernel. Where earlier sandboxing solutions made use of custom kernel modules for remedying limitations in the operating system APIs, newer solutions like MBOX [9], Firejail [12], nsjail [8], and nsroot [14] increasingly make use of the features found in standard kernels

	Janus	Ostia	Consh	Systrace	MiniBox	MBOX	Firejail	nsjail	nsroot	Flatpak
Release*	1996	2004	1998	2003	2011	2013	2015	2014	2016	2014
Modified OS	✗	✗		✗	✗					
Arguments†	✗	✗	✗	✗	✗	✗				
Namespaces							✗	✗	✗	✗
Seccomp						✗	✗	✗		✗
Resource Limits								✗		

Table 1: Comparison of sandbox implementations.
*: If the project has no associated publication, the time of the first commit is used. If the project has multiple associated publications, the earliest one is used.
†: Does the sandbox modify system call arguments.

and do not require custom kernel-level changes.

Related to this, there appears to be an influx of recent process sandboxing projects as Table 1 highlights. A possible explanation for this is the maturing of namespace technology in the Linux kernel, which has enabled the recent implementations and arguably lowered the bar for deployment. However, this reliance on mainline kernel features has arguably made the different implementations very similar in terms of the provided feature sets.

Race Condition Prevention

Different types of race conditions are a challenge that sandbox implementers have to solve. The problem was thoroughly described by Garfinkel [5]: Race conditions related to handling of system call arguments can lead to malicious system calls being allowed. Because deciding if a particular system call should be allowed and actually executing the call is often not an atomic operation, an attacker could manage to swap the arguments after the decision has been made but before the call is actually executed. This is commonly referred as Time of Check/Time of Use (TOCTOU) problem.

The surveyed implementations approached the problem with slightly different solutions. Ostia follows a different architectural pattern. Potentially harmful system calls are not performed in the sandboxed process, instead a separate agent process is used to decide whether or not the call should be allowed and for subsequently performing the call. Because the arguments are copied to the agent they are out of reach for attackers. In comparison, Systrace copies system call arguments to the kernel memory before consulting the user space about a policy decision. MBOX also employs copying but uses a separate page with read-only access rights for storing the arguments. Firejail and nsjail both employ Seccomp/BPF for preventing system calls that should always be denied. However, Seccomp only allows limited inspection into arguments of the calls.

Restricting File System Access

A file system is a shared resource and as such sandbox implementations have to control access to it. Consh approached the problem of restricting file system access by having a private virtualized file system available for sandboxed applications. This was achieved by inspecting resource acquisition related system calls and rewriting all paths to referer inside a particular directory inside the host system. Similar approach was later taken by MBOX, which exposed host's file system but transformed modification attempts to include copy-on-write semantics.

Recent sandbox implementations making use of Linux namespaces are able to control how the process sees the file system without having to resort analyzing system call arguments. Firejail, nsjail, and nsroot can all confine applications using mount namespaces.

It is also worth considering which files are exposed to the sandboxed application and how the decisions are made. For example, a user trying to open a document from a sandboxed application expects that they are able to browse through their files and select the file they wish to edit. Obviously for this to be possible, the process hosting the open file dialog has to be able to access user's files. As these dialogs have traditionally been part of the program itself, implementing this becomes a challenge. Flatpak solves this problem by introducing the concept of "portals". A sandboxed application can invoke a file chooser portal over an IPC mechanism. A portal provides the user with a familiar file selection dialog that exists outside the confines of the sandbox. When the user selects a file it is selectively exposed to the sandboxed application. Aside from file access, the portal facility is also used to control access to various other shared resources including printing and screenshots. In comparison, Firejail supports exposing files to a sandboxed program without requiring the program to be restarted.

Graphical Interface Isolation

Traditionally, X-server [19] has been a common choice for graphical user interfaces in Linux-based environments. Clients wishing to display graphics communicate with the server using a Unix domain socket or a regular socket. By default, X places few restrictions on clients' ability to interact with other clients. For example, clients are able to inspect contents of other windows or listen for all key press events. This is obviously problematic for those wishing to sandbox graphical programs: one would like sandboxed programs to be limited to interacting with windows belonging to them.

Wagner [17] noted this problem: access control in X is all-or-nothing. The authors highlight the possibility of using a sanitizing X-proxy that would filter malicious X-requests. However, for deploying Janus, they used a nested X-server, Xnest [18]. In this setup, the sandboxed application has its own X-server which itself is just another client for the parent X-server. This way, the sandboxed application is limited to only interacting with its own windows. Firejail supports a similar approach.

However, it is worth noting that X has gained some abilities to restrict inter-client communication through a Security extension [15]. The security extension makes it possible for clients to be labeled as trusted or untrusted and communication between these is limited.

CONCLUSIONS

In this survey, we have looked at ten application sandboxing solutions and characterized their notable features and provided some points of comparison. We have highlighted the division between older implementations and the recent trend of namespace oriented sandboxes.

REFERENCES

[1] Faisal Al Ameiri and Khaled Salah. "Evaluation of popular application sandboxing". In: *Internet Technology and Secured Transactions (ICITST), 2011 International Conference for*. IEEE. 2011, pp. 358–362.

[2] Albert Alexandrov, Paul Kmiec, and Klaus Schauser. *Consh: A confined execution environment for internet computations*. 1998.

[3] *Flatpak - the future of application distribution*. URL: http://flatpak.org.

[4] T Garfinkel and D Wagner. *Janus: A practical tool for application sandboxing*.

[5] Tal Garfinkel et al. "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools." In: *NDSS*. Vol. 3. 2003, pp. 163–176.

[6] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum, et al. "Ostia: A Delegating Architecture for Secure System Call Interposition." In: *NDSS*. 2004.

[7] Ian Goldberg et al. "A secure environment for untrusted helper applications: Confining the wily hacker". In: *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*. Vol. 6. 1996, pp. 1–1.

[8] *google/nsjail: A light-weight process isolation tool, making use of Linux namespaces and seccomp-bpf syscall filters*. URL: https://github.com/google/nsjail.

[9] Taesoo Kim and Nickolai Zeldovich. "Practical and Effective Sandboxing for Non-root Users." In: *USENIX Annual Technical Conference*. 2013, pp. 139–144.

[10] Yanlin Li et al. "MiniBox: A Two-Way Sandbox for x86 Native Code." In: *USENIX Annual Technical Conference*. 2014, pp. 409–420.

[11] Jonathan M McCune et al. "TrustVisor: Efficient TCB reduction and attestation". In: *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE. 2010, pp. 143–158.

[12] *netblue30/firejail: Linux namespaces and seccomp-bpf sandbox*. URL: https://github.com/netblue30/firejail.

[13] Niels Provos. "Improving Host Security with System Call Policies." In: *Usenix Security*. Vol. 3. 2003, p. 19.

[14] Inge Alexander Raknes, Bjørn Fjukstad, and Lars Ailo Bongo. "nsroot: Minimalist Process Isolation Tool Implemented With Linux Namespaces". In: (2016).

[15] *Security Extension Specification*. URL: https://www.x.org/releases/X11R7.6/doc/xextproto/security.html.

[16] Rui Shu et al. "A Study of Security Isolation Techniques". In: *ACM Computing Surveys (CSUR)* 49.3 (2016), p. 50.

[17] David A Wagner. "Janus: an approach for confinement of untrusted applications". PhD thesis. Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1999.

[18] *Xnest(1) manual page*. URL: https://www.x.org/archive/X11R7.5/doc/man/man1/Xnest.1.html.

[19] *X.Org*. URL: https://www.x.org.

[20] Bennet Yee et al. "Native client: A sandbox for portable, untrusted x86 native code". In: *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE. 2009, pp. 79–93.

ABOUT THE AUTHORS

[placeholder for review]

ACKNOWLEDGEMENTS

[Placeholder for acknowledgements]