Sandboxer: Light-Weight Application Isolation in Mobile Internet Devices

Rajesh Banginwar
Intel Corporation
rajesh.banginwar@intel.com

Michael Leibowitz
Intel Corporation
michael.leibowitz@intel.com

Thomas Tanaka
Department of Computer Engineering
San Jose State University
thomas.tanaka@gmail.com

Abstract

In this paper, we introduce sandboxer, an application isolation mechanism for Moblin based Mobile Internet Devices (MIDs). MIDs are expected to support the open but secure device model where end users are expected to download applications from potentially malicious sources. Sandboxer allows us to safely construct a system that is similar to the conventional `*NIX` desktop, but with the assumption that applications are malicious. Sandboxer uses a combination of filesystem namespace isolation, which provides a secure `chroot()` like jail; `UID/GID` separation, which provides IPC isolation; and `cgroups` based resource controllers, which provides access control to devices as well as dynamic limits on resources. By combining these facilities, we are able to provide sufficient protection to the user and system from both compromised applications that have been subverted as well as malicious applications while maintaining a very similar environment to the traditional `*NIX` desktop. The mechanism also provides facility for applications to hide the local data from rest of the applications running in their own sandboxes.

1 Introduction

Mobile internet devices (MIDs) have become an increasingly popular choice of device that people use every day as part of their daily routines. The recent released of Intel Atom processor which targets computer systems with small form factor such as MIDs and comes with the capability to deliver full internet experiences to mobile devices; further adds to a roadmap of more powerful processors powering MIDs in the near future. User will thus be able to enjoy high quality entertainment such as game or working towards their business related tasks on their mobile devices. With the increase in the computational power, more complex software applications will be developed to run in mobile devices. This could potentially lead to an increase in the security exploit of the device due to bugs and other possible software design flaws. Malicious software that has successfully penetrated the device will have the available resources to tap into user's privacy, which could be in the form of personal data (e.g., phone number) or sensitive phone conversations.

The majority of the user groups will not be necessarily equipped with sufficient knowledge to identify a possible malicious website or application. Therefore, designing and managing a strict security measure for mobile device is a necessary first step to ensure a safe operating environment. We have thus proposed sandboxer, the security tool that will provide a mechanism to protect mobile device in the event of malicious attacks. The basic sandboxing technique provides a concealed environment in which an application can be run, and in the event of malicious attack, damage to the system is greatly minimized. There have been similar works in the sandboxing design by several researchers. Nevertheless, their respective work has been focusing more on delivering a complete and efficient sandboxing solution that intends to minimize the possibility of an exploit in a vulnerable desktop/server like system rather than targeting specifically on mobile devices. Savitha and Kolar have proposed the use of hardware base solution to create the fine grained sandboxing by utilizing the privilege level adjustment that is available in today's processors [1]. West and Gloudon have proposed to monitor

system calls that required the modification of the kernel codes [2]. Yee *et al.* have developed a novel approach that utilizes the system interaction in terms of software fault isolation and controlling the runtime environment securely [11]. On the other hand, Chang *et al.* have implemented user level sandbox that uses resource monitoring and restrictions on applications specifically on Windows platform [10].

Our implementation differs in that we specifically focus on implementing the application sandboxing in a Linux platform, as part of the Moblin.org open source project [15]. Moblin is an open source Linux based operating system specifically targeted for MIDs. The unique features of our designs are as follows:

- The use of available and simple yet robust filesystem and privilege isolation techniques that are available as part of the Linux platform.
- User level implementation that does not require any modification to the Linux kernel only relying on the existence of a stable kernel and system.
- The ability to further extend the functionality of the sandbox through the use of plugins.
- The use of `cgroups` as a plugin to further enhance the sandboxing capability to include a tool that capable of enforcing a policy base resource control mechanism on the system.

With this, we will begin our discussion of the overall sandbox architecture design. We will then proceed on how we isolate the filesystem and privileges. Finally, we will proceed with the brief discussion of `cgroups` and specifically which features of `cgroups` that is currently included in our overall sandbox design.

2 Design and Implementation

Our design principle is based on the following key objectives:

1. To guarantee that a compromised application could not take ownership of the whole system. In other words, an attacker will not be able to use a possible vulnerable application as a springboard to launch a premeditated attack.
2. To provide the ability to hide information or data associated with an application from the rest of the applications running on the platform.
3. To provide the way to restrict access to the part of the system that an application does not require to accomplish its task.
4. To provide the ability to customize extended functionality of the sandbox by providing software hooks that could be developed and installed as a plugin.

Based on the above objectives, we have the overall high level system architecture of our design as shown in Figure 1.

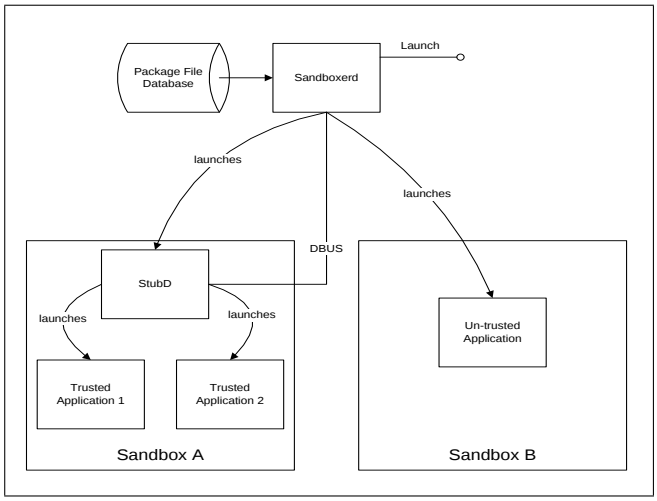

Figure 1: Architecture Overview

Our design consists of three functional components: Package File Database, Sandboxer Daemon, and Stub Daemon, as shown in Figure 1 above. The roles of these three functional components are as follows:

- Package file database decides whether to create a new sandbox or to use the existing one for the newly invoked application.
- Sandboxerd responds to request from the new application to execute.
- Stub Daemon is a daemon that only launches within a sandbox that contains multiple trusted applications. It specifically handles the request from the Sandboxerd where a new trusted application needs to run in the existing sandbox.

Trusted Domains
- Packages are installed in the `/usr` hierarchy as per Filesystem Hierarchy Standard (FHS) recommendations - Binary files/directories are owned by (root, root) - Binaries are run as <unique_uid>, <unique_gid> - Multiple binaries may be run in the same sandbox
Untrusted Domains
- Packages are installed in `/opt/<package-name>` as per FHS recommendations - Files/directories are owned by <unique_uid>, <unique_gid> - Binaries are run as <unique_uid>, <unique_gid>

Table 1: Assumptions based on trusted and untrusted domains

Notice that we emphasize the notion of trusted applications. Trusted applications are verified as safe applications and from trusted domains. The assumptions between trusted and untrusted domains are summarized in Table 1.

The package file database's primary role is to provide a mapping between trusted binaries to sandbox in the form of configuration file or database. All applications to be run in a sandbox are configured here, both trusted and untrusted. The distinction from trusted and untrusted operation is the configuration of the sandboxes, rather than the flag in the database. Care must be exercised during the creation of such entries. The format is illustrated in Table 2 below.

```
[Sandbox]
SandboxName=shared_sandbox
PackageName=firefox
Users=firefox
ExecPaths=/usr/lib/firefox-3.0.8/firefox
```

```
[Sandbox]
SandboxName=shared_sandbox
PackageName=gcalctool
Users=gcalctool
ExecPaths=/usr/bin/gcalctool
```

```
[Sandbox]
SandboxName=xterm_sandbox
PackageName=xterm
Users=xterm
ExecPaths=/usr/bin/xterm.bin
```

Table 2: Example .package files

The application `firefox` and `gcalctool` will therefore share the sandbox which will be referred to as `'shared sandbox'`, while `xterm` will use the new sandbox referred to as `'xterm_sandbox'`.

2.1 Filesystem and privilege isolation method

By filesystem isolation we mean that the sandboxed application runs in the pre-defined subset of filesystem that it cannot escape from. This is commonly referred to as "jail" and is most commonly accomplished with `chroot()`. Exploits that will compromise a simple `chroot()` are well known [14]. Our implementation does not use `chroot()` directly. Instead, we uses the `CLONE_NEWNS` flag introduced in the Linux 2.4.19 as a flag to create a new filesystem namespace with the `unshare()` system call. Once a process has entered its own namespace, `mount()` and `umount()` only affect the namespace of the current process and not the parent. Thus, manipulation to the root filesystem is possible that is specific to a certain process. Bind mounts (Linux 2.4 onwards) allow a sub-tree of the filesystem to be mounted as though it were a filesystem on a path. Using this mechanism, one can, for example, bind mount `/foo/bar` to `/baz` with `mount("/foo/bar", "/baz", NULL, MS_BIND, NULL)`. With these two tools, a secure jail can be constructed simply with:

```
chdir("/jail");
unshare(CLONE_NEWNS);
mount("/jail","/jail",0,MS_BIND,0);
pivot_root("/jail","/jail/old_root");
chdir("/");
mount("/old_root/bin","bin",0,MS_BIND,0);
mount("/old_root/usr","usr",0,MS_BIND,0);
mount("/old_root/lib","lib",0,MS_BIND,0);
umount2("/old_root",MNT_DETACH);
/* drop privilege omitted */
exec(application);
```

For privilege isolation, we use conventional `UNIX` users and groups. It is expected that individual applications will run with individual `UID` and `GIDs`. This allows

traditional isolation among users, which UNIX systems provide to keep applications distinct from each other. Several unprivileged applications will likely be put in their own sandbox. Certain applications will remain outside of sandboxes. These generally include privileged applications and daemons as well as applications that need unfettered access to the whole filesystem to work correctly (such as the IDS system). Of course, X is outside of a sandbox.

It may be desirable to put a PIM application and the web browser in separate sandboxes because both process' considerable input from the outside. It would be undesirable if an arbitrary code execution's flaw in the web browser exposed all of user's email. Likewise, the damage the compromised web browser will do should be limited.

2.2 Sandoxerd and Stub Daemon

Sandboxerd uses D-Bus as the communication medium among client applications. Sandboxerd exists as a daemon that manages the creation of sandbox and verify the policy that comes with that particular sandbox. It exposes an interface that is roughly analogous to `vfork()` and `wait()` usages. Note that this is API usage. A helper application is provided that can be directly `vfork`'d and `wait`ed. The helper utility is called sandbox and the usage of such utility would be:

```
sandbox <cmd> [args]
```

The calling application can directly use `vfork()/exec()` and `waitpid()` on this helper utility. With such a utility, enabling the use of sandboxing to applications is a mere configuration change rather than a code change.

The Stub Daemon exists for the case where two or more trusted applications are to share a sandbox. This can be determined in advance by cross-indexing the application name to the sandbox name in the configuration database during the mapping procedure. Because a sandbox is a premised on a filesystem namespace (CLONE_NEWNS), the only way to add a process to an existing sandbox is with `fork()`. For this scenario, the Sandboxerd `vfork()` the Stub Daemon. The Stub Daemon does the `unshare()` and bind mounts filesystems available to the sandbox and waits for commands from Sandboxerd. When all the children have exited, the daemon exits, thus destroying the sandbox. The following pseudocode further illustrates the mechanism.

```
loop {
  wait for command from Sandboxerd() {
    pid= fork;
    if (pid) {
       children << pid;
       send child pid to Sandboxerd;
    } else {
       setgid();
       setruid();
       exec();
    }
  }

  if children is empty
  exit;
}
```

To better understand how these blocks function together, two flow examples are provided side by side. Note that although we use the UML sequence notation, each lifeline represents a process rather than an object. Referring to Figure 2, in the first use case (the left figure), one application in a sandbox (App1), requests for the launching of a second application in a sandbox (App2), and each application exists in its own sandbox.

2.3 Plugins architecture

Our implementation of sandbox provides strategically placed hooks. Referring to our method of creating a secure "jail" in the previous 2.1 section, the following hooks in order:

- `INIT` – initialization state; run as *sandboxer user*
- `PRE_FORK` – state before `fork()` system call; run as *sandboxer user*
- `PRE_UNSHARE` – state before `unshare()` system call; run as *root*
- `PRE_PIVOT_ROOT` – state before `pivot_root()` system call; run as *root*
- `PRE_UMOUNT` – state before unmounting `old_root`; run as *root*
- `PRE_SETUID` – state before dropping privileges; run as *root*
- `PRE_EXEC` – state before `exec()` system call; run with privilege specified in `.package` file
- `FINALIZE` – final state; run as *sandboxer user*

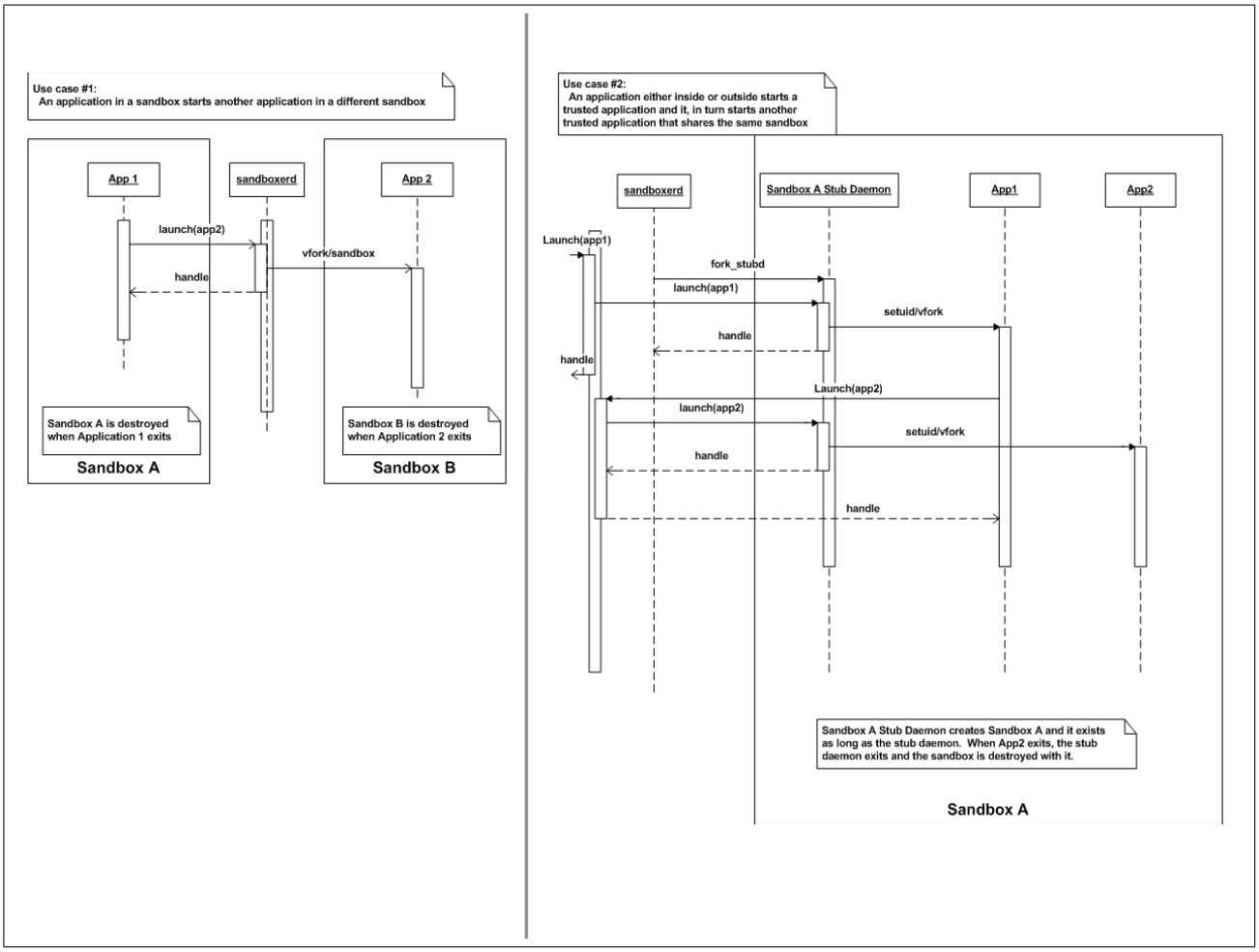

Figure 2: One application run on one sandbox (**left**), and semantics of Stub Daemon with two applications run inside the same sandbox (**right**)

Various plugins will be instantiated and registered within the Sandboxerd and Stub Daemon through configuration files. When new process is to be run, the Sandboxerd and Stub Daemon will invoke a particular function provided by the plugins, i.e. before `unshare()` a filesystem, the `PRE_UNSHARE` plugins function will be called to accomplish the necessary task related to that particular process. The hook function will provide information of the respective parent of particular process and the sandbox environment, i.e. its confined filesystem. Notice that at each of the hook, uid `(UID)` and gid `(GID)` will be different depending on which stage of the sandbox creation process a particular plugin function is invoked.

With the adaption of the plugins architecture towards our sandbox design, it enables the flexibility in extending beyond the simple "jail" mechanism that the basic sandbox provides. The need to expand the security features of the sandbox will be available to a developer simply by implementing a plugin. We have chosen to demonstrate the use of plugin to enhance our sandbox by integrating a resource control mechanism `(cgroups)` via a plugin.

2.4 Resource control

Resource control enables us to establish policy in regards to memory usage or devices available in the system. We have integrated `cgroups` into our sandbox to achieve this goal. `Cgroups` also known as control groups is Linux kernel mechanism that is currently a work in progress, which provides a way to partition tasks and their respective children into a hierarchical groups [6]. It was originally developed with the intention to become a Linux container. `Cgroups` by itself provides a simple job tracking mechanism available inside the kernel. It comes with several subsystems

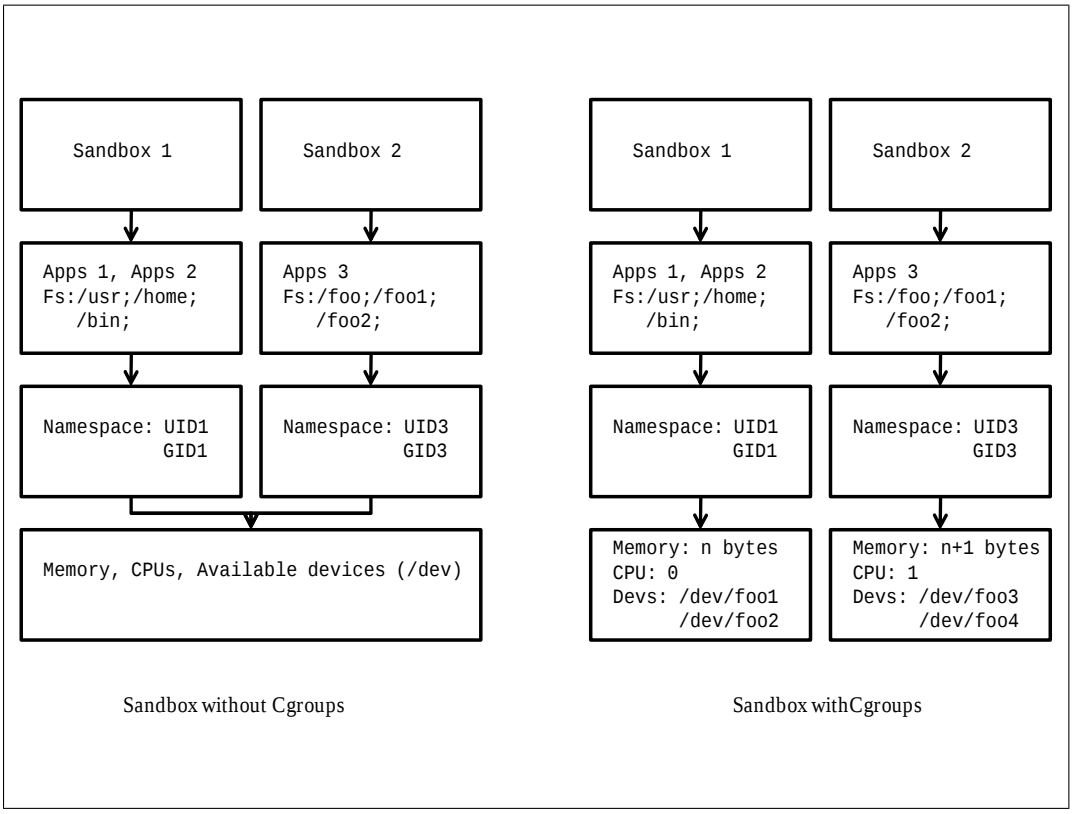

Figure 3: High level comparison on sandbox design with and without `cgroups`

which uses this basic functionality to extend the ability to provide resource controls. Currently there are ten or more `cgroups` subsystems being developed and experimented. Since this is a work in progress, more stable and new subsystems will be available in the near future. By enabling `cgroups` subsystems into our sandbox design, we are able to provide policies that capable of directly controlling the limit on the usage of system resources. Currently, we have included only a total of three subsystems (`memory, memrlimit` and `devices`) to be included as part of our sandbox design.

`Memory` subsystem provides the availability to limit the available memory per sandbox [8]. `Memrlimit` subsystem provides the same functionality as `memrlimit()` system call and this limit applies in per sandbox context (regardless of the number of processes residing inside particular sandbox) as oppose to per process context as in the original system call. `Device` subsystem provides the ability to restrict access to devices available in the system as available in `/dev` filesystem. It provides ability to track and enforce `open` and `mknod` restrictions on device files [7].

The `memrlimit` capability provides the sandbox an ability to automatically keep tracks of the total memory limit (available address space or number of bytes allocated via `malloc(), sbrk(), mmap()`). It will automatically fail any attempt to allocate dynamic memory beyond the specified allocated limit as per sandbox configuration policy. `Device` subsystem provides a way to enable and disable the available devices (device whitelists) to a particular sandbox. Customize policy that distinguished between the availability of device between a trusted sandbox and those which serves untrusted/possible malicious application.

With these three subsystems implemented in our sandbox, we could further provide a more targeted policy control for specific applications. One example is the ability to enforce a strict policy to target sandbox usage for a potential un-trusted/malicious application. Although the filesystem and privilege isolation is a defense mechanism in the event of an attack; the system resources are still available to be exploited, specifically memory limit. As depicted in Figure 3, system's memory and cpu(s) are shared among the sandbox in the absence of `cgroups`. With `cgroups`, memory and

cpu(s) usage and access policy is enforced. The need to always ensure sufficient memory availability is a necessity for a device that has telephonic capability. In the event of an emergency call such as 911, we have to provide a high level of assurance that the call will not be interrupted in the possible event of depleted memory availability consumed by some malicious processes. With resource control functionality, thus we are able to create a policy that will pre-allocate sufficient system resources to ensure the emergency call will proceed uninterrupted.

2.5 Cgroups as a plugin

The first prerequisite in enabling the cgroups resource capability is to compile the Linux kernel with required cgroups subsystem enabled. The configuration for each sandbox object comes with the lists of cgroups variable of interest. The running system is also required by default to create a virtual filesystem that cgroups will use to operate. The hooks for the cgroups functionality need only to be called during the PRE_FORK and PRE_SETUID state. Initialization and setting up necessary parameters such as memory limits and lists of allowable devices are accomplished at the PRE_FORK state by reading available package files. At the PRE_SETUID, the pid of the running process in a particular sandbox will be registered with the cgroups system. Cgroups will then perform an accounting and monitoring activity.

3 Sandbox confines

For most applications, some shared variable is required as well as some Inter Process Communication (IPC). To the end user, the sandbox should be transparent and they should not see individual fields of data that cannot be merged. For most applications to work correctly, several environment variables must be set up in the sandbox. Examples of such environment variables are PATH, USER, HOME, HOSTNAME, and GTK+ themes. However, manipulation of environment variables by a nefarious caller can lead to compromise. As such, the environment variables used inside the sandbox must be taken from a source other than the caller. Since the environment of the Stub Daemon is trusted, it serves as the source for environment variables to be used inside sandboxes.

Since the sandboxes are file-system based, most forms of IPC are possible across sandboxes. Most IPC semantics require some handle to be present for one application to know the IPC method and path to be used to connect to. For example, D-Bus uses the environment variable DBUS_SESSION_ADDRESS. For these IPC mechanisms to be readily available inside the sandbox, they need to be copied over. At present, Xauth cookies and D-Bus handles are copied over. Plugin-type architecture will allow for flexible manipulation of the sandbox environment at creation. Mechanics for sharing files between sandboxed applications can be done simply for trusted applications. Standard POSIX permissions and groups offer an appropriate method. For example, suppose we wish for all trusted applications to be able to read and write files in the directory /usr/share/foo. If all trusted applications are in the "trusted" group and /usr/share/foo is set GID foo with mode 770, then all trusted applications can read, write, create, and execute files out of /usr/share/foo.

Although the sandbox environment provides almost all of the same functionality as a normal Linux programming environment, certain exceptions and caveats are present. The most notable difference for the programmer is the absence of /proc and /sys inside the sandboxes. The removal of proc and sys effectively limit the visibility of sandboxed applications to see the true environment. Most end-user applications function without proc being present; however application writers should be cognizant of this omission. Additionally, the /dev and /etc directories are present, but are not true copies of the respective system directories. They are "shims" with only the relevant files or sections of files present. Unlike the bind mounts for the other root directories, the shims are selective copies that are created on demand, although they can be cached. In /etc, for example, the full passwd database will not be present. However, an abridged version will be created on demand that only contains the relevant user and group information for the application(s) that run in that sandbox. Similarly, most host information available in /etc are not copied over. In /dev only devices that need to be present need to be created. For most applications, only non-hardware devices will be present, such as random, null, and zero. For trusted applications, some devices may need to be present as specified in the package file database.

Additionally, top level directories cannot be removed from within sandboxes. For example if there was a top level directory `/foo` and it was mode `777` with a bind-mounted directory, then it could not be removed from either inside or outside the sandbox. Although files can be removed from `/foo`, the directory `/foo` itself is unremovable. There are two examples where expected results may occur if the developer is not aware of the sandboxed environments. The first is the unintentional launching of an application within the same sandbox. For example, if the browser wishes to launch the email client (to handle a mailto: url), and it uses `vfork/exec` of the email executable directly instead of the convenience wrapper, it will inadvertently start the email client within the same sandbox. This will most likely lead to a non-functional email client, but could be an exploitable condition. Care should therefore be exercised. Similarly, care must be exercised with `D-Bus` activation. Since the session bus is shared amongst all sandboxes outside of their respective sandbox. An application that wishes to use `D-Bus` activation will use the convenience wrappers in its activation procedure to allow proper functionality. Failure to do so will result in activation failing. With these sandboxing approaches, we feel that we can limit the damage of application subversion, lessen the risks of disclosure of sensitive data, as well as reduce opportunities for privilege escalation.

4 Related work

Many implementations of sandboxing technologies focused on almost many different approaches, both software and hardware utilization. One of the hardware implementations from the work of Sahita and Kolar that proposed the use of Virtual Machine Monitor (VMM) is included in the hardware virtualization support inside Intel's CPU [1]. The implementation consisted of creating a monitoring scheme that utilized a VMM and a kernel service that will particularly monitor the request for memory usage. The monitoring agent will allocate a range of linear addresses for the new application. Memory access needs to be requested via a communication with the monitoring agent that runs as VMM. Trusted and untrusted application will run on separate isolation of memory addressing that will be determined by a policy access of a particular sandbox. The similarity in our implementation is in the policy that we have enforced for a particular sandbox which has lists of variables such as memory limit and devices white list. Our implementation is only restricting the amount of memory instead of restricting access to a range of linear addresses. With a fine grained control of range of accessible linear addresses, malicious application will not be able to gain access to the protected memory regions thus reducing the damage caused to the system. However, the tradeoff will be the complexity in the integration with the system hardware. Our implementation focused on providing a simple yet robust solution that provides the sandbox properties that will work on all platforms capable of running Linux operating system regardless of the underlying hardware design.

The work of Chang *et al.* in the implementation of user-level resource constrained sandbox is closely related to our work [10]. The implementation covered the ability to constrain the system usage of CPU, memory and network. The focus of their implementation was specifically on the Windows NT platform. Although Linux platform was included in the experimental testing, the paper lacks the discussion on the details of the implementation. CPU resource constrained is accomplished by implementing a monitoring scheme that scheduled processes based on the priority level. A process that exceeded the limit will be penalized by lowering its priority level. Memory constrained is accomplished with the way of sampling the memory usage by intercepting memory allocation API. Those applications that exceeded its limit will be penalized by having the extra memory pages marked, such that access to the marked pages will result in page fault. The difference in our implementation is that we have used the resource constrained capability-`cgroups` as part of the kernel. `Cgroups` provide a simple yet robust framework for resource control capability within the Linux kernel. Instead of penalizing a process that exceeded its limit, i.e. `malloc()` request, `cgroups` will simply return a fail for any attempt to request for resource usage beyond its preset sandbox limit. Instead of per process restriction, our `cgroups` implementation provides per sandbox policy restriction. Without the complexity of monitoring each and every resources allocation API, `cgroups` keep a simple accounting routine that will check if the policy limit has been exceeded and thus requires less processing overhead. It therefore translated into a lower total power consumption. The needs for more complex solution that involves kernel functionality could always be extended as part of the `cgroups` subsystem.

West and Gloudon proposed a user level sandbox to

provide protection for extensible system [2]. Their approach modifies a process address space to contain one or more shared pages. The extension codes will then be mapped into this shared page that comes with access privilege as a protection mechanism during transition from the user to kernel space. Through changing the access privilege of the shared page, kernel is able to maintain the integrity over the newly implemented extension codes. Though not particularly focusing on an extensible system, our design provides the capability of running a completely new or existing sandbox setup, in which a new extension of the system is desire. The sets of policy could be reused or recreated to accommodate the changes. It may contain a total set of filesystems or various sandbox variables to better accommodate and provide security isolation to the newly modified application.

Yee *et al.* introduced Native Client, the protection mechanism to run un-trusted native code specifically on the x86 based system [11]. Native Client provides a secure runtime protection and software fault isolation. A two layer of sandbox is introduced, with the inner layer provides memory reference constraint through the use of x86 segmented memory capability. The outer layer compares each request of a system call with the database of trusted system call. Our implementation does not intercept against any system call made by an application. However, we restrict the namespace of the particular application, so that it could only have the visibility of a sufficient set of filesystems to accomplish its task. Filesystems that contain system information such as `proc, sys, etc.` may not be available to the application. The application will execute with sufficient privilege to accomplish its task. In the event that software fault that could trigger an attack, such as buffer overflow, the compromised application will be confined to its own sandbox environment with its default privilege, thus gaining a root access privilege will be difficult.

Most of the isolation solutions that we have seen so far tend towards the use of restriction against access to specific memory range to protect sensitive data. With such a fine grained control of memory access, it will be able to prevent any unintended access towards a particular memory area; however, a complex modification in both hardware and software is almost a major requirement. The cost of implementing and maintaining will increase correspondingly with respect to the the complexity of the system design. System call interception is also another common approach. Our main goal of creating the sandbox solution is to use the simple approach that already existed in the system especially in the Linux platform and by integrating a kernel level control that is provided by a framework such as `cgroups` that has low system overhead. By going with a simplistic approach we are not sacrificing any security measure, since we are using a Linux system mechanism that has been thoroughly used and tested overtime in terms of its stability and security. Additionally, our goal is to enable the porting of our implementation across the many different hardware platforms with minimal difficulty.

5 Future work

We hope to get insightful feedback and contribution from the open source community and integrate it into our future design. With the inclusion of `cgroups`, we are always capable of improving and adding the resource control functionalities by including the new `cgroups` subsystem. The plugin capability also allows the user of our sandboxer to add an extra functionality as required.

6 Conclusion

This paper provides an overview of our design of the user level sandboxing design which provides filesystem namespace isolation, flexibility to extend sandbox capability through innovative design of plugins architecture, and utilization of a resource control capability through Linux kernel `cgroups`. We have described in this paper how we achieve filesystem and namespace isolation, and how we utilized our sandbox plugin capability by making `cgroups` a plugin. With the ability to contain and run predefined sets of policy, we are thus preventing a compromised application to invoke a significant damage in a system such as MIDs. This project is also part of the moblin.org open source project initiative for Linux based operating system specifically targeted for MIDs.

7 References

[1] R. Sahita and D. Kolar, *Beyond Ring-3: Fine Grained Application Sandboxing.* World Wide Web Consortium (W3C), December 2008.

[2] R. West and J. Gouldon, *User-Level Sandboxing: a Safe and Efficient Mechanism for Exensibility.* Technical Report, 2003-014, Boston University, June 2003.

[3] R. West and J. Gouldon, *QoS Safe' kernel extensions for real-time resource management.* The 14th EuroMicro International Conference on Real-Time Systems, June 2002.

[4] I. Goldberg, D. Wagner, R. Thomas and E. Brewer, *A secure environment for untrusted helper applications.* In Proceedings of 6th USENIX Security Symposium, July 1996.

[5] S. Miwa, T. Miyachi, M. Eto, M. Yoshizumi and Y. Shinoda, *Design Issues of an Isolated Sandbox Used to Analyze Malwares.* In Lecture Notes in Computer Science: Advances in Information and Computer Security, Heidelberg, 2007.

[6] `Cgroups` documentation. `/linux-2.6.X/Documentation/cgroups/cgroups.txt`

[7] `Cgroups` devices documentation. `/linux-2.6.X/Documentation/controllers/devices.txt`

[8] `Cgroups` memory documentation. `/linux-2.6.X/Documentation/controllers/memory.txt`

[9] Robert N. M. Watson, *Exploiting concurrency vulnerabilites in system call wrappers.* In 1st USENIX Workshop on Offensive Technologies, August 2007.

[10] F. Chang, A. Itzkovitz and V. Karamcheti, *User-level Resource-constrained Sandboxing.* USENIX Windows Systems Symposium, August 2000.

[11] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula and N. Fullagar, *Native Client: A Sandbox for Portable, Untrusted x86 Native Code.* Technical paper Google Inc, 2008.

[12] S. Santhanam, P. Elango, A. A-Dusseau and M. Linvy, *Deploying Virtual Machines as Sandboxes for the Grid.* Proceedings of the 2nd conference on Real, Large Distributed Systems-Volume 2, San Francisco 2005.

[13] B. Ford and R. Cox, *Vx32:Ligthweight User-level Sandboxing on the x86.* 2008 USENIX Annual Technical Conference, June 2008.

[14] Using `chroot()` Securely. `http://linuxsecurity.com/content/view/117632/49/`

[15] Moblin. `http://moblin.org`

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*