A capability inspired low level security model based on modern Linux kernels

Stephan Soller, Computer Science and Media
Stuttgart Media University
ss312@hdm-stuttgart.de

Abstract

The Linux kernel is evolving constantly as APIs change or new APIs are added. This paper explores how a Linux 3.5 kernel can be used to implement some aspects of a capability security model. It proposes an implementation and discusses its problems and limitations.

1. Introduction

Current security software relies on several approaches to handle malicious software:

- Signatures are used to identify malicious programs.
- Virus scanner heuristics analyze the structure and behavior of programs to identify yet unknown threads.
- Access control lists (ACLs) are used to configure access privileges to critical data.

Signatures and heuristics suffer from occasional false positives. ACLs can become very complex and are subject to misconfiguration.

In the 1980s another security mechanism gained traction: capability based security [1] [2]. The main concept of capability based security is the principal of least authority: Every process has only access to the data required for the current task. Access to this data is granted by whoever triggered the action. Access rights are passed along with the data whenever a new task is created. The access rights "flow" with the data.

Early efforts focused on hardware based capabilities [3], usually by associating capabilities with memory addresses. Passing the pointer to a function would grant the called function the permission to dereference the pointer and work with that memory. More modern efforts defined and implemented capability safe subsets of object oriented programming languages [4], associating capabilities with objects. Passing an object to a method grants the called method the privilege to interact with that object. Unfortunately these concepts never became popular among main stream programmers.

Thanks to several recent developments renewed effort is put into sandboxing single processes from the rest of the system. Modern browsers try to isolate tabs from one another as well as isolating insecure subsystems like plugins from the rest of the browser [5]. On Mac OS X, for example, video codecs are isolated from the rest of the video player [6]. Those trends also led to new features in the Linux kernel and this paper explores how a modern Linux kernel can be used to implement a simplified capability based security model.

2. Basic approach

The design uses two APIs of the Linux kernel. seccomp-bpf [7] is an interface to limit the system calls a process can perform. It was introduced in Linux 3.5 [8]. UNIX domain sockets are used to pass open file descriptors between processes [9].

One master process is responsible for starting new sandboxed child processes. This happens via the `fork()` and `exec()` functions. Once the new child process is forked seccomp-bpf is used to install a filter into the new child that denies most system calls. Only a minimal set of safe system calls is whitelisted [10]. System calls that can open new files or network connections are *not* among them. A sandboxed child process without any resources granted to it can't affect the system in any way except consuming CPU time.

For the child to do useful work a secure way to exchange data with the sandbox is required. With UNIX domain sockets a process can send and receive file descriptors from another process. For example a process which manages the users files can open a required file and pass the file descriptor to a sandboxed process. This way the

sandboxed process gains access to the file it requires but not to any other resources.

Installing a seccomp-bpf filter

seccomp-bpf builds on the original secure computing mode. With the `prctl()` system call the process could be put into a mode where only the system calls `read()`, `write()`, `exit()` and `sigreturn()` could be used [11]. Every other system call triggers a SIGKILL and terminates the process. Once the seccomp mode is entered there is no way to leave it since a call to `prctl()` would kill the process as well.

The original seccomp mode only allows a fixed and very minimalistic set of system calls. This limited seccomp to pure compute tasks since a sandboxed process usually requires an extended set of system calls. A way to customize the set of allowed system calls is provided by seccomp-bpf. It extends the original seccomp mode with a user defined simple filter program that decides which system call is allowed. For that it repurposes parts of the network packet filtering system. Because of that the filter rules are defined as a Berkeley Packet Filter. This somewhat unusual approach allows seccomp-bpf to use well tested and maintained code [12].

Installation of a seccomp-bpf filter also works via the `prctl()` system call. But an array of filter rules is passed along as an extra parameter [10]. In the simplest case these filter rules contain a list of allowed system calls. But should the need arise these filter rules can also access the parameters of the system call in question (but not dereference pointers). This can be used to allow some system calls only on predefined file descriptors.

Exchanging data with the sandbox

To send and receive file descriptors with a UNIX domain socket the `recvmsg()` and `sendmsg()` system calls are used. With them multiple file descriptors can be send along as an "ancillary messages" [9]. Therefore the seccomp-bpf filter needs to allow these system calls.

The UNIX domain socket connection is initiated by the master process before forking. It creates a new pair of socket file descriptors with the `socketpair()` system call. After forking the unnecessary ends are closed and the seccomp-bpf filter is installed. Via this initial connection the master can grant the child access to the resources it needs. It can also pass along connections to other (sandboxed) processes the client needs to interact with.

If the seccomp-bpf filter of a sandbox allows the `clone()`, `exec()` and `socketpair()` system calls the child can even fork new sandboxes by itself. If `prctl()` with the PR_SET_SECCOMP option is allowed the child can restrict the available system calls even further. seccomp-bpf filters are inherited by child processes and new filters are stacked on top of the previous filters. This ensures that a sandboxed process can't break out by forking itself or by setting a new filter.

3. Passing file descriptors and restricting access

Usually processes cooperate with one another. For example a file browser might display all files in a directory. To display the contents of a file it employs another process, e.g. an image viewer. In a sandboxed environment the image viewer should only be granted *read-only* access to the files it is supposed to display. This chapter describes several problems with file descriptor (fd) passing and suggests various solutions.

In Linux most state of opened files isn't part of the file descriptor but of the corresponding kernel object [13]. This includes the access mode, status flags (O_NONBLOCK, etc.) and the file offset. As a consequence all file descriptors referring to the same kernel object share this state. Whether they're duplicated with `dup()` or send to another process via a UNIX domain socket.

Shared status flags and file offset

The shared file offset is used by the `read()` and `write()` system calls. One process reads data and advances the file offset for all other processes, too. So if multiple processes read from the same fd each process will skip the data read by the others. This can lead some unexpected behavior if the processes are not aware of each other.

This can be solved by using the `pread()` and `pwrite()` system calls for accessing files (or any seekable fd). These system calls require to explicitly pass the file offset and allow each process to keep track of its own private position within the file.

The problem is also avoided by using `mmap()` to map the entire file into memory. When reading from the memory map the pointer already specifies the exact position. Therefore the file offset of the kernel object is not used.

Nonblocking I/O can suffer a similar problem. If one process sets the `O_NONBLOCK` flag all other fds referring to the same kernel object are also effected. If other processes aren't aware of that it can easily lead to unexpected behavior. As a workaround the `poll()` system call can be used. With a timeout of 0 it returns immediately even if a file descriptor isn't ready [ref internet article].

Shared access mode

Linux provides no way to narrow the access mode of an individual file descriptor. It's not even possible to change the access mode of the associated kernel object once it has been opened. But to grant as little access as possible it is desirable to create a read-only version of a read-write file descriptor.

One way to solve this limitation is to let the original power box open the same file with the narrowed access rights. Thus creating a new kernel object with the desired access mode. This requires a protocol so a sandboxed process can request a "reopen". But it's only possible when a connection to the original power box is available. And even then only with file descriptors that can be opened again (e.g. files and POSIX shared memory). Access to network connections can't be restricted in such a way. Alternatively a read-only file descriptor can be passed along with each read-write file descriptor. If a process wishes to delegate read-only access it can send the read-only file descriptor. But the same "reopen" limitation applies here.

Another way to only grant restricted access is to build a "proxy pipe". The owner of the fd does not send the fd to the delegate process directly. Instead it creates a new pipe with the `pipe()` system call and sends the read-only end of the pipe. It can then monitor the original fd for changes via the `poll()` system call. Once data is available it can be `splice()`ed into the pipe where the delegate process will receive it. This approach adds overhead to the data transfer and the fd received by the delegate process neither be `mmap()`ed into its memory nor supports seeking. But the same applies to all scenarios where traditional shell piping is used. It also allows to implement write-only delegation, e.g. of a network connection by swapping the pipe ends.

Revoking access to a granted fd

In some situations it can be desirable to revoke access to an fd once granted to another process. Linux doesn't allow that but it can be implemented with the same "proxy pipe" approach mentioned before.

A pipe is created and the read end is shared with the delegate process. Revoking access then becomes a simple matter of closing the write end of the pipe and stopping to feed data into it. This leaves the delegate process no way to access the data. The same pipe-approach can be used for revocable write-only access. In that case the write end is passed to the delegate process.

For revocable read *and* write access a bidirectional pipe is required: a UNIX domain socket. Instead of creating a pipe the `socketpair()` system call is used to create an unnamed socket. One end of the socket pair can then be sent to the delegate process. Again, to revoke access closing our end of the socket pair is sufficient.

As with the "proxy pipe" above the file descriptor the delegate process receives does not support seeking nor `mmap()`ing.

4. Limitations and further work

While the proposed mechanisms fulfill basic capability functions some severe limitations remain.

The `open()` system call of sandboxed processes is blocked to prevent them from affecting the system. But loading of shared libraries involves reading the shared object via the `open()` system call. Therefore sandboxed process can't load new shared libraries. This affects any kind of shared library loading. Shared libraries loaded by the `exec()` system call and libraries manually loaded with `dlopen()`. To work around this issue all dependencies have to be statically linked into the executable. This will hurt performance and increase the overall memory footprint of applications. It can even reduce overall system security since common shared libraries can't be updated without recompiling (or at least relinking) all applications that use it.

Alternatively a `chroot()` cage can be used to limit file system access of sandboxed processes. Required shared libraries and other files can then be hard linked into that chroot cage. The fanotify API [14] could also be used to deny most `open()` calls and only allow those the application has access to.

Delegation of file descriptors with a reduced access mode also remains troublesome. If file seeking or memory mapping is required the file has to be "reopened" by the power box that has direct access to the file. Other more nestable approaches require pipes and thus break seeking and memory mapping. This can have a severe impact on application code. There is no obvious solution to this problem yet.

5. References

[1] What is a Capability, Anyway?
http://www.eros-os.org/essays/capintro.html
Retrieved 2013-02-02

[2] Wikipedia: Capability-based security
http://en.wikipedia.org/wiki/Capability-based_security
Retrieved 2013-02-02

[3] The Cambridge CAP Computer, Capability-Based Computer Systems
http://www.cs.washington.edu/homes/levy/capabook/Chapter5.pdf
Levy, Henry M. (1984). Digital Press.

[4] joe-e, Capability-secure subset of Java
http://code.google.com/p/joe-e/
Retrieved 2013-02-02

[5] Chromium Blog: Multi-process Architecture
http://blog.chromium.org/2008/09/multi-process-architecture.html
Retrieved 2013-02-02

[6] Privilege separation, Mac OS X 10.7 Lion: the Ars Technica review
http://arstechnica.com/apple/2011/07/mac-os-x-10-7/9/
Retrieved 2013-02-02

[7] SECure COMPuting with filters
http://git.kernel.org/?p=linux/kernel/git/torvalds/linux.git;a=blob;f=Documentation/prctl/seccomp_filter.txt
Retrieved 2013-02-02

[8] Introducing Chrome's next-generation Linux sandbox
http://blog.cr0.org/2012/09/introducing-chromes-next-generation.html
Retrieved 2013-02-02

[9] Ancillary Messages, Man page unix(7), Linux Programmer's Manual
http://man7.org/linux/man-pages/man7/unix.7.html
Retrieved 2013-02-02

[10] Using simple seccomp filters
http://outflux.net/teach-seccomp/
Retrieved 2013-02-02

[11] PR_SET_SECCOMP option with SECCOMP_MODE_STRICT, Man page prctl(2), Linux Programmer's Manual
http://man7.org/linux/man-pages/man2/prctl.2.html
Retrieved 2013-02-02

[12] Yet another new approach to seccomp
http://lwn.net/Articles/475043/
Retrieved 2013-02-02

[13] Man page open(2), Linux Programmer's Manual
http://man7.org/linux/man-pages/man2/open.2.html
Retrieved 2013-02-03

[14] Manpages for the fanotify API
http://git.xypron.de/?p=fanotify-manpages.git
Retrieved 2013-02-02