CapExec: Towards Transparently-Sandboxed Services (Short Version)

Mahya Soleimani Jadidi*, Mariusz Zaborski†, Brian Kidney*, Jonathan Anderson*

* *Department of Electrical and Computer Engineering*
Memorial University of Newfoundland
{msoleimanija,brian.kidney,jonathan.anderson}@mun.ca

† *Research and Development*
Fudo Security Inc.
{oshogbo@FreeBSD.org}

***Abstract*—Network services are among the riskiest programs executed by production systems. Such services execute large quantities of complex code and process data from arbitrary — and untrusted — network sources, often with high levels of system privilege. It is desirable to confine system services to a least-privileged environment so that the potential damage from a malicious attacker can be limited, but existing mechanisms for *sandboxing* services require invasive and system-specific code changes and are insufficient to confine broad classes of network services.**

Rather than sandboxing one service at a time, we propose that the best place to add sandboxing to network services is in the *service manager* that starts those services. As a first step towards this vision, we propose CapExec, a process supervisor that can execute a single service within a sandbox based on a service declaration file in which, required resources whose limited access to are supported by Caper services, are specified. Using the Capsicum compartmentalization framework and its Casper service framework, CapExec provides robust application sandboxing without requiring any modifications to the application itself. We believe that this is the first step towards ubiquitous sandboxing of network services without the costs of virtualization.

***Index Terms*—application security, sandboxing, service manager, Capsicum, compartmentalization**

I. Introduction

Network services and applications have always been attractive targets for remote attackers. Network services typically incorporate complex protocol parsing code, often written in low-level languages, that is exposed to arbitrary content from the network. Since these services commonly execute with system privilege, they are at high risk for remote exploitation as a gateway to other system resources. Owing to both the risk and the consequence of potential compromises, there is a need to confine network applications and limit the damage that can be inflicted by a successful attack.

One broad class of techniques that seem applicable to the problem of securing network services is *sandboxing*: restricting software's access to system resources such that the application has the least privilege required to fulfill its function. However, applying such limitations is challenging. Many sandboxing frameworks require invasive code modifications, some of which require a great deal of security expertise to apply correctly. Incorrectly applied mechanisms may lead to a false sense of security without additional effectual protection. What is needed, in addition to the development of effective sandboxing techniques, is the development of tools to apply those techniques.

Sandboxing is applicable through different techniques at various levels within operating systems such as the system call `chroot(2)` [1], sandboxing features in service managers such as `systemd` [2], or application containers such as `jail(8)` [3] or `docker` [4]. We see *service managers* as key to securing systems with network-facing services. Securing systems at this level, eliminates the need for securing every service separately.

In this paper, we have combined process supervision and FreeBSD's capability-oriented compartmentalization framework [5], Capsicum [6], to introduce CapExec as a sandboxing process supervisor (section II), which runs applications in a restricted *capability* mode. CapExec uses Capsicum to restrict the privileges of services and to limit access to resources through Capsicum-based Casper daemon framework [7], [8]. To use CapExec, services' required run-time resources should be described in service declaration files, as an initial step to unify security and functionality descriptions. When a service's run-time requirements can be described in terms of Casper services, CapExec provides sandboxing without any modifications to source code. The mechanism and its various evaluation results are described in section II and section III respectively. This project is a significant advance beyond the current state of the art described in section IV.

II. CapExec: A Sandboxing Service Supervisor

We have designed and developed CapExec to be a security-focused service supervisor that executes a service in a sandbox transparently, which means no modification to the service's source code is required. CapExec creates sandboxes relying on Capsicum and Casper services. Capsicum is a sandboxing framework developed for FreeBSD [6]. The fundamental idea for the framework comes from the concept of capabilities in capability-based systems. Capabilities are unforgeable tokens

978-3-903176-24-9

of authority carried by processes to authorize access to the system's resources [9]. Using Capsicum's API, once a process enters to the capability mode, all system calls trying to access global namespaces, such as the root filesystem or the PID[1] namespace, will fail. Capsicum's sandboxing has changed the regular flow of the system call mechanism in FreeBSD. In addition to Capsicum's general API, Casper has extended Capsicum's features by defining capability channels to allow access to some riskier but widely-used services [7].

Using Capsicum and Casper, CapExec creates and loads required Casper services to provide access to white-listed runtime resources. These services and their limits are specified in the application's declaration files as the service configuration such as what has been shown in the listing 1. CapExec executes one application at the time in a sandbox made with required Casper's capability channels. Hence, limited accesses to the subsetted namespaces are proxied for the application, but any unprivileged behavior or requests beyond defined limits will fail due to capabilities violations. The details about this mechanism are explained in [10].

A. Service Declaration: Unifying Application Functionality with Security Requirements

CapExec employes `libucl` [11] to parse the service declaration file conforming to JSON format [12]. The file should describes the service's binary and its *requirements*. Requirements are those system resources that are disallowed in capability mode, but their alternative restricted services are supported by Casper [8]. An example configuration for the `traceroute` utility is given in Listing 1.

Listing 1. An example of the content of `traceroute(8)`'s declaration file

```
{
        binary: "/usr/sbin/traceroute"
        "system.fileargs" : {
                operations: "OPEN",
                flags: "RDONLY",
                cap_rights: "READ",
                cap_rights: "FCNTL",
                cap_rights: "FSTAT",
                filename: "/etc/protocols",
                filename: "/dev/null"
        }
        "system.dns" : {
                family: AF_INET
        }
        "system.net" : {
                host: "example.com",
                family: AF_INET
        }
        "system.sysctl" : {
                "vm.overcommit": {
                        type: "mib",
                        flag: "CAP_SYSCTL_READ"
                }
        }
}
```

[1]Process ID

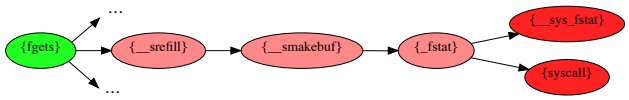

Fig. 1. Reduced call graph from `traceroute(8)` showing how a call to `fgets(3)` can result in system calls disallowed in Capsicum.

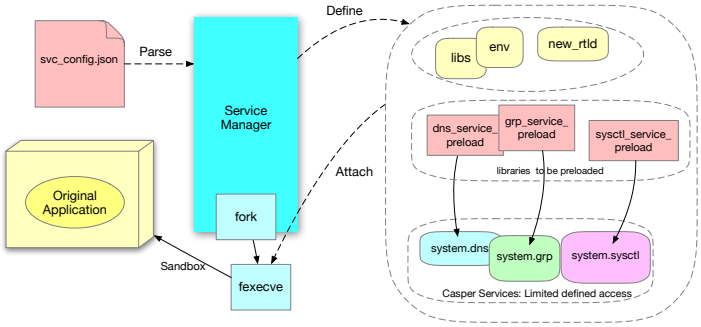

Fig. 2. CapExec's approach to run a service in a sandbox

Writing service declarations requires knowledge of system calls used within the applications code. The problem is complicated by system calls used indirectly by libraries. Recognizing this issue, we developed `CapCheck`, a tool for highlighting library calls that use system calls not allowed in a Capsicum sandbox. As can be seen in figure 1, even a call to `fgets(1)` can result in unexpected system calls that are disallowed in Capsicum sandboxes.

`CapCheck` uses `readelf` and `ldd` to determine the calls made to external libraries and the libraries that provide them. It then builds a full call graph for the application and searches it to find paths that result in disallowed system calls. This information is provided to the end user to develop service declaration files.

B. Sandboxed Execution

As the first step, CapExec parses declaration files and creates the corresponding Casper services. In addition to supporting all Casper services, CapExec needed a capability channel for simple network communications. Therefore, we developed an experimental networking Casper service, `system.net`, supporting `bind(2)` and `connect(2)`. After parsing service declaration files, CapExec forks and executes the binary sandboxed in a child process, creating a new environment or context for the process, including arguments, libraries, environment variables, the new runtime linker, shared memory mappings and required libraries to make the process sandboxed. CapExec replaces disallowed `libc` system calls with other functions that redirect requests to the existing Casper services. As a result, the source code remains unmodified. Figure 2 shows this mechanism.

III. Evaluation and Comparisons

We have evaluated CapExec in various ways [10]. In this section, we describe evaluation results and our observations

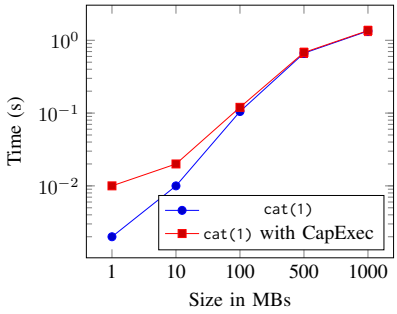

Fig. 3. Time to open single file with original `cat(1)` in comparison with `cat(1)` running in CapExec's sandbox (with negligible uncertainty)

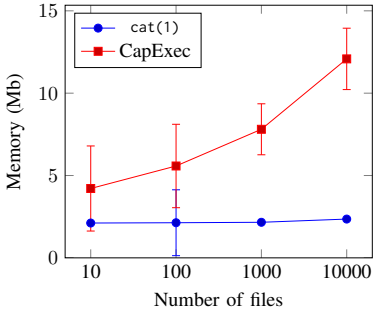

Fig. 5. Memory utilization for various number of files opened by `cat(1)`

about runtime performance, correctness of our approach and other similar virtualization-based solutions, by investigating `cat(1)` and `traceroute(8)` in CapExec's sandbox. As these two small utilities were previously sandboxed using Capsicum, examining these application allows us to have a direct comparison to traditional applications of Capsicum requiring source code modification. So, we have shown the simplicity of having a declaration file and consequently sandboxed applications, rather than modifying the source code, even for older approaches benefiting from Capsicum. Here is a summary of each evaluation approach.

1) Runtime Performance: To investigate running `cat(1)` with CapExec, which only needs one Casper service (`system.fileargs`), we examined `cat(1)` and its sandboxed version with two test scenarios. In the first scenario, we examined invocations of `cat` with files of various sizes from 1 MB to 1 GB, shown in figure 3.

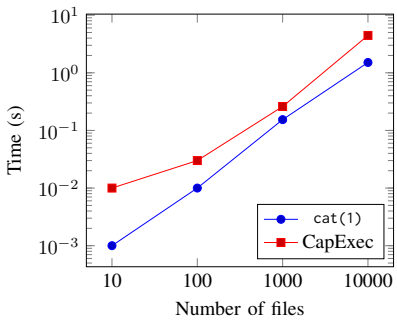

Fig. 4. Time to open multiple files with `cat(1)`(with negligible uncertainty)

As can be seen in figure 3, executing `cat(1)` in our sandbox adds overhead to the execution time. We find this delay more tolerable as the number of inputs grows. The majority of the cost of using CapExec is spent in setting up the sandbox. This includes the reading of configuration files, the creating of Casper services and `fork(2)` calls to load and open them. Since a large portion of CapExec's overhead is spent pre-opening and holding handles to file descriptors, in the second test scenario, we ran our tests with a varied number of *empty* files as the input set, from 10 to 10000. Figure 4 shows the expected delay. All measurements were performed ten times per test case.

CapExec's approach includes the additional cost of spawning Casper services and CapExec's preloaded libraries. In addition to the structures keeping services configurations and limits, new processes have to be spawned by the supervisor program. We can see the impact of this design in figure 5, which shows a broader impact on memory overhead.

2) Completeness and Correctness: We have investigated the correctness of our sandboxed application through `ktrace(1)` output, which provides us detailed information about invoked system calls and their returned values. In Capsicum, when an application tries to access a global namespace, a capability error is returned, and the intended system call fails. Listing 2 is demonstrating the capability error returned from the system call `open(2)` in capability mode.

Listing 2. `ktrace(1)`'s output showing a failure in capability mode

```
cat    CALL  cap_enter
cat    RET   cap_enter 0
cat    CALL  openat(AT_FDCWD,0x7fffffffe990,0<O_RDONLY>)
cat    CAP   restricted VFS lookup
cat    RET   openat -1 errno 94 Not permitted in capability
      mode
```

CapExec delegates tasks to Casper services for disallowed replaced system calls. Requests are sent to Casper services through a capability channel which passes commands through UNIX domain sockets to Casper services. Since in our examined applications, no system call failed and no related capability error was observed and they worked normally, we can infer that the corresponding configuration file sufficed for the applications.

3) Comparisons with Virtualization-based Solutions: We have also compared CapExec with equivalent virtualization-based solutions. There are various aspects to compare these systems such as required time to set up the sandboxed or virtualized environment, the complexity of configurations, required storage for each approach, memory utilization for each tool and finally the latency of running an application using each tool. To investigate CapExec against most widely-used containers, we examined running `traceroute(8)` under five situations. First, we examined the utility on the native system. For the rest of the cases, we started the container, ran `traceroute(8)` on it and then stopped it. We practiced this

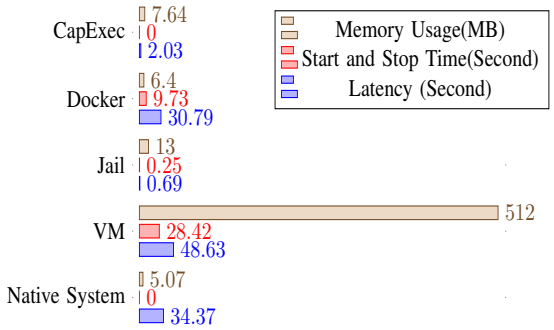

Fig. 6. Comparison of running `traceroute(8)` in different environments

procedure on a virtual machine, a FreeBSD `jail`, a `docker`, and finally with CapExec. Figure 6 shows the latency and memory usage of running `traceroute(8)` using each technology, giving an average of five test runs, and also the time spent to start up and to stop each of them.

Along with setup difficulties, there is a need for additional storage for virtualization-based solutions to keep images of containers. As an example, the size of the image for a virtual machine, a `docker` image, and a `jail` image in our tests, were 2.6 GB, 204 MB, and 800 MB respectively. These all show that how much isolating a small application might cost. This issue makes a light user-level sandboxing application like CapExec distinguished from other sandboxing solutions.

IV. Related Work

To design CapExec's scheme, we have studied several sandboxing mechanisms and service managers, focusing on their security options. As the most important options, user permissions and privileges are essential for most of contemporary `init` systems. Configuring `uid` and `gid` in `systemd` [2], `s6` [13] or other service managers, are examples of these options. However, they are still susceptible to complex attacks such as those utilizing privilege escalation.

`launchd` [14], the service manager on macOS, was the first system that expanded `inetd`'s [15] socket activation, and was adopted because of its performance. `launchd` provides security options that are mostly focused on permission features such as `username`, `groupname`, `initgroup` and `umask`. The only option concerning sandboxing schemes is the ability to set the `root` directory. The short list of security options in `launchd` originated from the internal security scheme of macOS, which is consistent and well-designed.

`systemd` [2] is another widely-used service manager in which various security options are supported. In contrast with `launchd`,`systemd` supports very fine-grained security controls such as `uid`/`gid` control and isolation options such as inaccessible or read-only paths, root, and `tmp` directories. Benefiting from `seccomp` [16], system call filtering is also provided. However, with all of these features, it is left to the developer to verify the configuration's compatibility with the use of the application, which can result in inappropriate policies.

There are also other service managers with similar security mechanisms such as `relaunchd`, also known as `jobd` [17], `nosh` [18], and `s6` [13], which are quite different in design. For example, `relaunchd` runs services in jails [3], and its options are configurable for the user, while most of `nosh`'s security features are internal. Also, `nosh` uses the concept of capabilities in design. `nosh`'s design and mechanism are based on `daemontools`, which is a package of tools for UNIX service management [19]. There are also other service management tools inspired by `deamontool` such as `runit` [20] and `s6` that are much more than an init system. `s6` is a package of tools including various security features such as access control management on client connections, supporting `uid`-less privileges, ability to define `sudo` family, etc [21]. Most of `deamontool`-based service managers benefit security applied in their internal design.

In addition to service managers, we also have studied widely-used existing sandboxing tools such as `chroot(2)` as a sandboxing system call, FreeBSD's `jail` and `docker` that were examined as containers in section III, `seccomp(2)` [16] as a framework that applies system call filtering, and`CloudABI` [22] which provides process-level sandboxing benefiting from capabilities. An interesting point about all of these applications is the different level at which each of them is providing sandboxing.

V. Future Work

CapExec is a service supervisor that executes one application at a time. That application executes in isolation, a sandbox provided by Capsicum with limited access to global resources mediated by Casper. However, CapExec is not a complete service manager. Our intent is to use CapExec as a foundation for a service manager that handles interdependencies and sandboxes groups of network services. To aid in this goal, we are also developing more network-oriented Casper services. Additionally, we are investigating ways to reduce the amount of security-specific knowledge required to use CapExec. We aim to make defining security policies as simple as specifying white-listed resources required by services, so that CapExec decides which Casper services, with what configuration, should start.

VI. Conclusion

In this paper, we introduced CapExec, a prototype sandboxing supervisor that facilitates service sandboxing both on local and network services. Using Capsicum and Casper, along with a simple configuration file, we transparently provide this isolation at run time without any modification on the application's source code. The system requires to be configured based on essential Casper services, which is challenging for the user. Facilitating this procedure, we provide `CapCheck`, a tool to discover system calls that require wrapping. This demonstrates that sandboxing itself can be a service, a key foundation for building security-aware service managers in the future.

Acknowledgement

The authors gratefully acknowledge the support of the NSERC Discovery program (contract RGPIN-2015-06048).

REFERENCES

[1] (Retrieved at: June 6, 2019) Freebsd manual pages: chroot(8). [Online]. Available: "https://www.freebsd.org/cgi/man.cgi?chroot(8)"

[2] (Last edited: May 23, 2019) systemd system and service manager. [Online]. Available: "https://www.freedesktop.org/wiki/Software/systemd/"

[3] (2018) Freebsd's mannual page: jail(8). [Online]. Available: "https://www.freebsd.org/cgi/man.cgi?query=jail&sektion =8&apropos=0&manpath=FreeBSD+12.0-RELEASE+and+Ports"

[4] (Retrieved at: June 14, 2019) Docker security. [Online]. Available: https://docs.docker.aom/engine/security/security/

[5] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson, *The design and implementation of the FreeBSD operating system*. Pearson Education, 2014.

[6] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for unix," in *Proceedings of the 19th USENIX Conference on Security*, ser. USENIX Security'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1929820.1929824

[7] P. J. Dawidek and M. Zaborski, "Sandboxing with capsicum," *; login:: the magazine of USENIX & SAGE*, vol. 39, no. 6, pp. 12–17, 2014.

[8] Z. Dawidek, Pawel Jakub. (2016, 2) Freebsd manual pages: libcasper'. [Online]. Available: "https://www.freebsd.org/cgi/man.cgi?query=libcaspe r&sektion=3&apropos=0&manpath=FreeBSD+11.0-RELEASE+and+Ports"

[9] R. S. Fabry, "Capability-based addressing," *Commun. ACM*, vol. 17, no. 7, pp. 403–412, Jul. 1974. [Online]. Available: http://doi.acm.org/10.1145/361011.361070

[10] M. S. Jadidi, M. Zaborski, B. Kidney, and J. Anderson, "Capexec: Towards transparently-sandboxed services (extended version)," 2019, arXiv:1909.12282.

[11] (Retrieved at: June 14, 2019) Libucl. [Online]. Available: http://rodrigo.ebrmx.com/github_/vstakhov/libucl

[12] D. Crockford. (2000) Json's website. [Online]. Available: "https://json.org"

[13] (Last update: June 2019) Gentoo linux's article: s6. [Online]. Available: "https://wiki.gentoo.org/wiki/S6"

[14] (Retrieved at: June 6, 2019) Official introduction page: Launchd. [Online]. Available: "https://www.launchd.info"

[15] (Retreived at: June 6, 2019) inetd daemon. [Online]. Available: "https://www.ibm.com/support/ knowledgecenter/en/ssw_aix_72/com.ibm.aix.cmds3/inetd.htm"

[16] (Retrieved at: June 6, 2019) Seccomp bpf (secure computing with filters). [Online]. Available: "https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html"

[17] M. Heily. (Retrieved at: June 6, 2019) relaunchd. [Online]. Available: "https://github.com/csjayp/relaunchd"

[18] (Retrieved at: June 6, 2019) The nosh package. [Online]. Available: "https://jdebp.eu/Softwares/nosh/"

[19] (Last update: 2019) Gentoo linux's article: daemontools. [Online]. Available: "https://wiki.gentoo.org/wiki/Daemontools"

[20] (Last update: June 2019) Gentoo linux's article: runit. [Online]. Available: "https://wiki.gentoo.org/wiki/Runit"

[21] (Retrieved at: June 6, 2019) An overview of s6. [Online]. Available: "https://skarnet.org/software/s6/overview.html"

[22] (Retrieved at: June 14, 2019) Introducing cloudabi. [Online]. Available: https://cloudabi.org