SYSTEMD HARDENING MADE EASY WITH SHH

Written by Maxime Desbrus - 07/11/2023 - in

Introducing SHH, Systemd Hardening Helper, a tool written in Rust to automatically build a set of hardening options for a service using runtime profiling.

HARDENING IS... HARD

Most security experts will tell you that part of securing a system is not only avoiding vulnerabilities, but also preparing for them, by mitigating their consequences and limiting their exploitation. Hardening is often the last resort defense, but if an attacker finds a vulnerability and attempts to exploit it, it can be the difference between a program crashing and restarting without many consequences, and a fully compromised system.

Hardening, however, can be a tricky thing. If you set a very strict policy, you increase the risk of breaking programs attempting normal operations. On the opposite, setting relaxed rules may simply render hardening useless, if an exploited vulnerability can work around it.

Here we will focus on hardening using the systemd init system, which is the default choice of init system on most Linux distributions, and thus is ubiquitous today. Among other things, its role is to start the system, and the various programs that are a part of it. systemd configurable components are called *units*, and units that handle the life cycle of programs needed on a Linux system are called *services*.

systemd supports many options and mechanisms for hardening service units. For example, they allow running a service while preventing it from writing any data on the file system, making it unable to access the network, or disallowing specific operations via their associated system call.

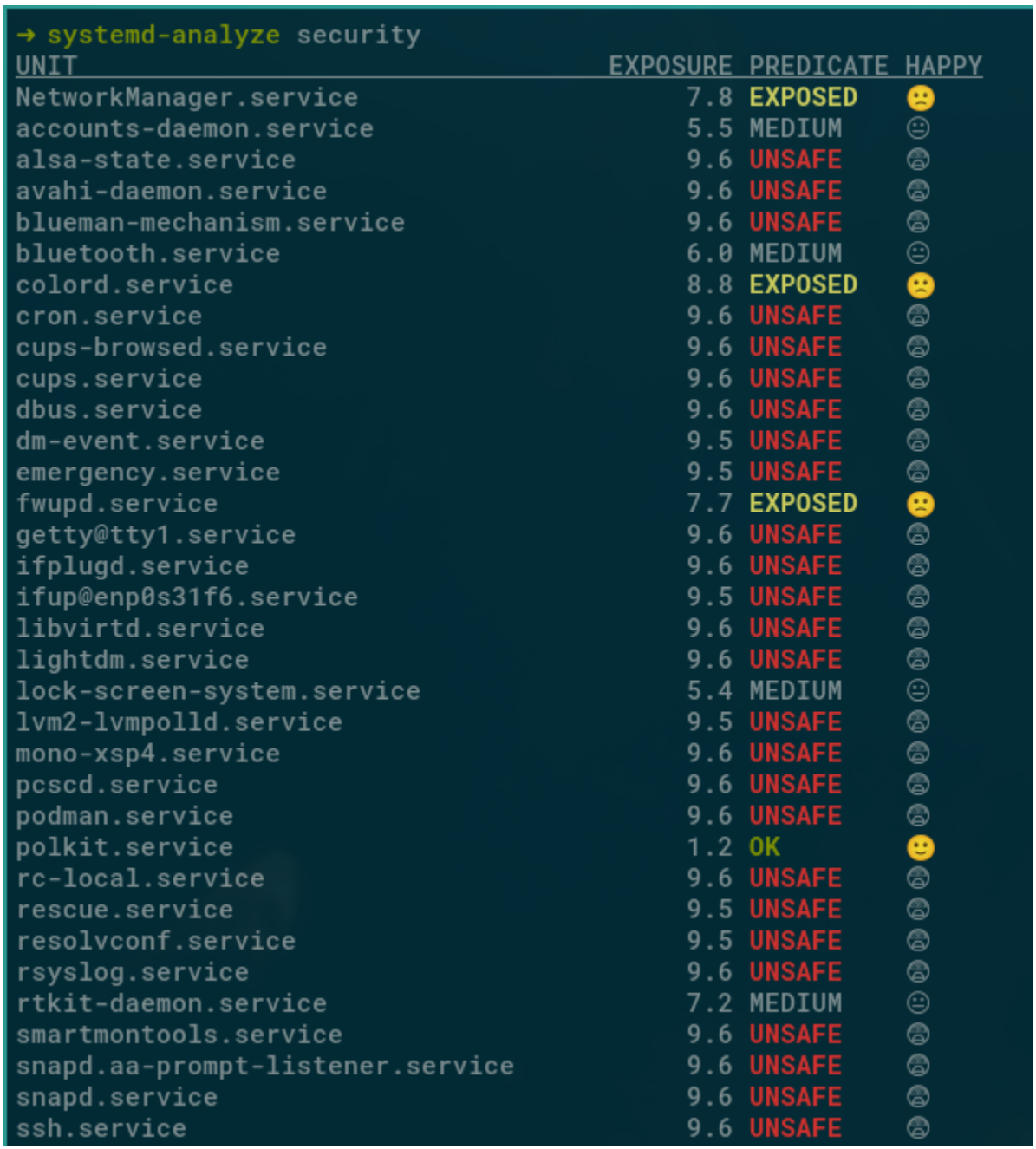
Under the hood they are most often implemented using two classes of kernel technologies. The first one is namespacing. This allows running a program while *unsharing* (`unshare` is in fact the name of system call to do this), a part of its environment with the host system. For example this allows running it with a different file system hierarchy, with different permissions and content. There are several types of namespaces to similarly define a different view of the system for the network, the system users, time, cgroups, etc. The other main hardening Linux mechanism used by systemd is *seccomp*. This is a low level facility giving access to fine grained filtering of the various system calls made by a program.

As we can see, systemd offers a set of options for hardening services using powerful kernel facilities, however using them in practice brings a new set of challenges.

Linux users usually install systemd services from their distribution's package managers, which sometimes include some hardening options. However by definition the given package is generic, and will most likely allow many actions that the service is not supposed to do, facilitating vulnerability exploitation. For example, the common Nginx web server can be used as a simple HTTP proxy, or as a caching proxy, the difference being that the latter requires write access to a part of the file system. Similarly it may bind a privileged TCP port, or it may not even be using IP sockets if using Unix sockets to forward requests to another service. If we define the optimal, most secure, hardening options as the ones that allow normal service operation, and disallow other actions, following the least privilege principle, then this optimal set cannot be coming from the default service configuration, which is written to be generic.

This point can be illustrated by running `systemd-analyze security`, a systemd utility to provide a high level view of the hardening options of services on a running system. On most systems, this will display many `UNSAFE` warnings, proving that in

practice, systemd hardening is vastly underused.



'systemd-analyze security' alarming output on Debian Bookworm

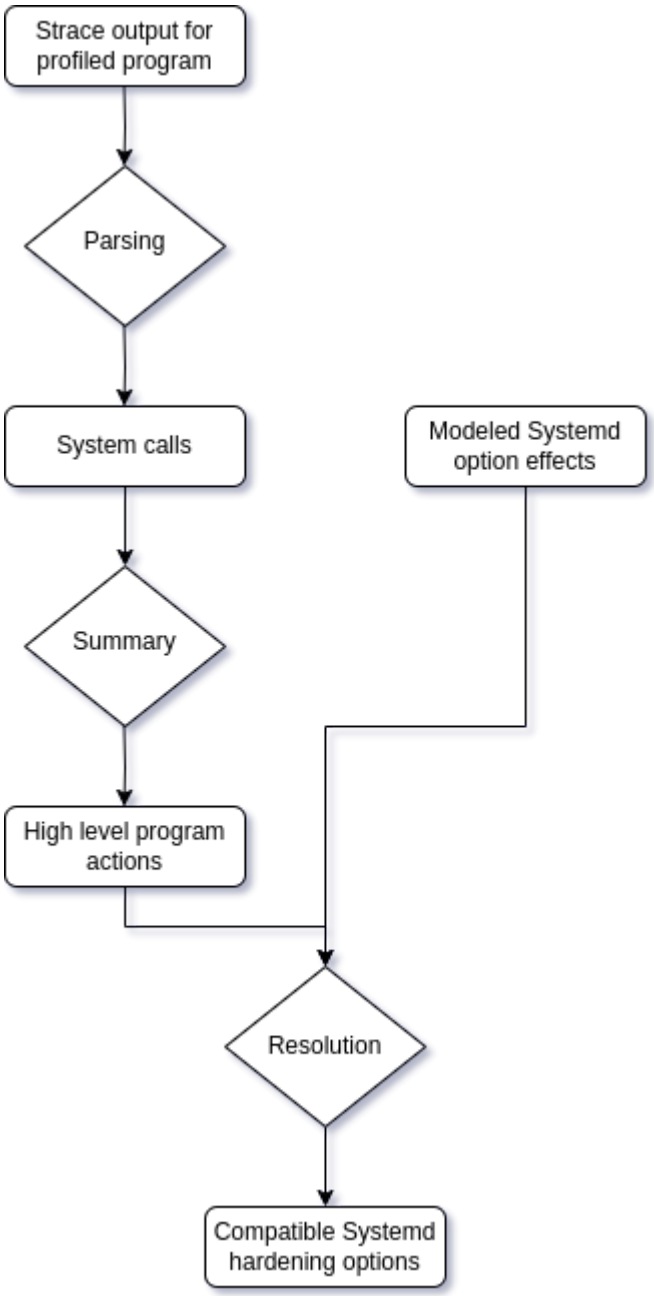
Additionally, if one wants to push hardening restrictions to a high level, it requires an in depth knowledge not only about what a service does, but also how. For example one may want to deny system calls that we know are not used by the program, to reduce kernel attack surface. However it is not apparent exactly what system calls are used by a program. The program itself may typically be unaware of it, because it relies on libraries, the lowest level one being the C library, which chooses what system call to use, and may call a different one if the libc version changes, or if it is a different C library.

Here we will present a new tool we have developed using the Rust language, SHH (Systemd Hardening Helper), that aims at making hardening systemd units both easy and reliable, by leveraging runtime profiling.

SHH: CONCEPTS AND ARCHITECTURE

The goal of SHH is to automatically generate a set of optimal hardening options for a given service. To do that, SHH must run on the same system as the service, as it relies on runtime profiling. By running the service in normal conditions, we can build a profile of what the program does, which we can use to know what it does not do, and build a hardening configuration to prevent it from doing it, by the principle of the least privilege.

The high level program flow of SHH is the following:



SHH processing flow

In the rest of the article, we will detail the main parts of the processing.

PROFILING

In order to observe what a service does, and more precisely how it does interact with the Linux kernel, we use the strace tool. Strace fills exactly our need since it provides a detailed and highly configurable log output of each system call the program does, which is the interface between kernel and user space, where permissions checks are done by the kernel, and where systemd hardening have a direct effect. We also only consider successful system calls, the rationale being that we want to set a hardening configuration as restricted as possible to permit those, but we don't need to consider the calls that failed anyway during normal program operation.

We chose to read and parse strace output, and to then run the next summary step in a direct processing pipeline (using Rust lazy iterator under the hood), rather than to generate a profiling log file to process afterwards. This has several advantages:

- for services that may need to be profiled for an extended period of time
  - this avoids the creation of huge (several GB) log files to parse
  - this spreads the SHH processing load across time, consequently at the end of profiling, the service can be almost immediately restarted with its new hardened configuration
- since SHH by design runs directly alongside the profiled service, it can probe its environment (for example to resolve file paths by following symbolic links), that may be different outside the service unit because of mount namespace, or simply because it changed since then

SHH starts strace which then starts the profiled program, and outputs its logs into a named pipe, which SHH reads at the other end, line by line. For example a strace output line when the program opens a file may read:

```
998518      0.000033 openat(AT_FDCWD<\x2f\x68\x6f\x6d\x65\x2f\x6d\x64\x65\x2f\x73\x72\x63\x2f\x73\x68\x68
>, "\x2e\x2e", O_RDONLY|O_NONBLOCK|O_CLOEXEC|O_DIRECTORY) = 3<\x2f\x68\x6f\x6d\x65\x2f\x6d\x64\x65\x2f\x73
\x72\x63>"
```

Theses lines are then parsed using a custom regex based parser. The strace output can include complex data, like C structures, or'd flags, raw buffers, but in our experience and for our current use they are not complex enough to require a grammar based parser.

After the parsing stage, we have structured data describing the system calls and their arguments, for example for the previous `openat` example:

```
Syscall {
    pid: 998518,
    rel_ts: 3.3e-5,
    name: "openat",
    args: [
        Integer {
            value: NamedConst(
                "AT_FDCWD",
            ),
            metadata: Some([47, 104, 111, 109, 101, 47, 109, 100, 101, 47, 115, 114, 99, 47, 115, 104, 10
4]),
        },
        Buffer {
            value: [46, 46],
            type_: Unknown,
        },
        Integer {
            value: BinaryOr(
                [
                    NamedConst(
                        "O_RDONLY",
                    ),
                    NamedConst(
                        "O_NONBLOCK",
                    ),
                    NamedConst(
                        "O_CLOEXEC",
                    ),
```

```
                    NamedConst(
                        "O_DIRECTORY",
                    ),
                ],
            ),
            metadata: None,
        },
    ],
    ret_val: 3,
}
```

MODELING SYSTEM CALL EFFECTS

The goal of the summary step is to transform a large number of system calls into a smaller number of objects modeling their effect, which we call *actions*.

In our profiling use case, most of these system calls are not of interest for our needs. For example we don't necessarily need to know that the program called the `write` system call on file descriptor `3`, with specific buffer data. However we are very much interested in knowing that a `open` call was made just before with the flag `O_RDWR`, which indicates that the program may write to a path. This is a common theme in this step of the processing: since there are many different system calls (more than 300 in total), we try to deduce the high level program actions from a limited number of system calls to simplify the processing.

For file system operations, we model three effects: `Read`, `Write` and `Create`. Despite their name, they do not exactly refer to their file operation system call counterparts. For example listing the directory content of the directory `/home` with the `ls` program will generate `Read("/home")`, because it generates a logical read of the file system of that path.

This part requires full understanding of the system calls and their effect, often deeply affected by the various flags passed to them. For example one can naively think that the renameat2 system call will only write to the path of its third `newpath` argument, but this is not the case if the flag `RENAME_EXCHANGE` is set, in this case the paths are atomically exchanged, which corresponds to both the logical `Read` and `Write` actions, on both `oldpath` and `newpath` paths.

This step is called summary for a good reason, a typical service will make many thousands of system calls, but these will be summarized in only a few dozens of high level actions.

MODELING SYSTEMD OPTIONS

If we want to decide which systemd option is most relevant for hardening a given service, we first need to model these. This means knowing which possible values are supported for a given option, what is their exact effect, which is the most restrictive to apply if we can, etc.

As an example, the `PrivateDevices` systemd options can be described with the following data structure:

```
OptionDescription {
    name: "PrivateDevices",
    possible_values: [
        OptionValueDescription {
            value: Boolean(
                true,
            ),
            desc: Simple(
                Multiple(
                    [
                        Hide(
                            Base {
                                base: "/dev/",
                                exceptions: [
                                    "/dev/null",
                                    "/dev/zero",
```

```
                                "/dev/full",
                                "/dev/random",
                                "/dev/urandom",
                                "/dev/tty",
                                "/dev/pts/",
                                "/dev/ptmx",
                                "/dev/shm/",
                                "/dev/mqueue/",
                                "/dev/hugepages/",
                                "/dev/log",
                            ],
                        },
                    ),
                    DenySyscall {
                        class: "@raw-io",
                    },
                ],
            ),
        ),
    },
],
}
```

Which roughly means:

- this option is named "PrivateDevices"
- it supports a single value, "true" (of course in practice it supports the "false" value, but since this is the default which sets no hardening restriction, we can ignore that)
- it has the following cumulative effects
  - create a mount namespace which:
    - empties the `/dev/` directory
    - remount a few files and directories inside it, like `/dev/null` and such
  - denies all system calls in the class "raw-io"

RESOLVING OPTIONS

Once we have both modeled the supported systemd options, and the high level actions the program does, what is left to do is to combine both to resolve the optimal hardening options. This is actually the simplest step because it relies on the structured data from both inputs. We can check, for each program action, if it would be blocked by a systemd hardening option. If that is the case, then this option cannot be retained, because it would prevent the program from running properly. We can iterate over all options following this logic, excluding those that are not compatible, and keeping the most restrictive ones that are.

There are a few special cases to consider though, for example let's look at the option `PrivateTmp=true`. This option sets up a new empty private *tmpfs* file system under `/tmp` and `/var/tmp` for the service, ensuring that its temporary files cannot be accessed by other programs, and also preventing it from accessing files created by other programs. If after the summary step, SHH finds out that a program does a `Read("/tmp/file")` action, it is apparently incompatible with the option, because the file at `/tmp/file` needs to exist and be readable, and `/tmp` is made empty by the effects of the option. Of course there is an exception to this if the file is created by the program itself before reading it. So for the proper handling of this case, we need to search all previous actions for a `Create("/tmp/file")`, and not just consider the current action.

PUTTING IT ALL TOGETHER

`shh` is a command line program, that can directly alter systemd service configuration to inject the profiling wrapper, and apply the generated option set at the end of the processing.

As an example, we will consider the service for the Caddy HTTP server. In its default configuration on Debian, Caddy serves a placeholder static site from `/usr/share/caddy` on the TCP port 80.

Its default service is considered "exposed" by the `systemd-analyze` tool:

```
$ systemd-analyze security caddy | tail -n 1
→ Overall exposure level for caddy.service: 8.8 EXPOSED 🥲
```

To harden its configuration, we do the following steps:

1. `shh service start-profile caddy`
2. Then we should build a representative and realistic runtime profile. Since we know the service only serves static files, we can simply fetch the static page on `http://127.0.0.1:80/`. For more complex services it is important to ensure all features and code paths have been covered, and it may involve running the profiling step for an extended period of time.
3. To finish profiling, and apply the new hardening options: `shh service finish-profile -a caddy`

We can see the following options have been generated:

```
$ cat /etc/systemd/system/caddy.service.d/zz_shh-harden.conf
# This file has been autogenerated by shh
[Service]
ProtectSystem=full
ProtectHome=tmpfs
PrivateTmp=true
PrivateDevices=true
ProtectKernelTunables=true
ProtectKernelModules=true
ProtectKernelLogs=true
ProtectControlGroups=true
MemoryDenyWriteExecute=true
RestrictAddressFamilies=AF_INET AF_NETLINK AF_UNIX
SocketBindDeny=ipv4:udp
SocketBindDeny=ipv6:udp
LockPersonality=true
RestrictRealtime=true
ProtectClock=true
SystemCallFilter=~@aio:EPERM @chown:EPERM @clock:EPERM @cpu-emulation:EPERM @debug:EPERM @keyring:EPERM @m
emlock:EPERM @module:EPERM @mount:EPERM @obsolete:EPERM @pkey:EPERM @privileged:EPERM @raw-io:EPERM @reboo
t:EPERM @resources:EPERM @sandbox:EPERM @setuid:EPERM @swap:EPERM @sync:EPERM @timer:EPERM
```

The effect of each of these is best understood from reading the systemd documentation, however we can already see that these settings, especially the `SystemCallFilter` one, have been created with detailed knowledge of the program runtime actions, which would have been difficult to guess for a system administrator, even familiar with the service.

Now let's see what `systemd-analyze` reports with the new configuration:

```
$ systemd-analyze security caddy | tail -n 1
→ Overall exposure level for caddy.service: 4.7 OK 🙂
```

The reported exposure level has been lowered, a nice improvement reflecting the raised level of hardening due to the options automatically added by SHH. Note that we have only scratched the surface of the possible areas for hardening with systemd. We expect the exposure level in this example to be even lower once support for more system calls and systemd options are added to SHH.

CONCLUSION

As we have seen, even if SHH does not yet have a complete coverage of all the systemd options or existing system calls, it can nonetheless prove to be a very useful base to further harden commonly exposed services.

SHH is released under GPLv3 license at https://github.com/synacktiv/shh.