Securing (legacy) services

Jailing with systemd

Anton Lundin <glance@ac2.se>

What is “jailing”?

The act of locking things down.

Limiting access to certain things, devices, files, folders…

Why “jailing”?

Separation of impact.

Control who might get access to what

Different from regular “access control” by not being the primary line, but rather the auxiliary defenses.

When the shit hits the fan, contain the spraying.

How “jailing”?

In the beginning you where luckey if your service dropped root privileges, and `setuid()` to a non root user after startup. Some did, some didn’t.

In the 70’s we’ve got concepts like `chroot()`, but now we have more than you can shake a stick at.

We’ve had SELinux for a long time, grsec hardened chroot, Apparmor came along, cgroups made resource control better, linux capabilities, Secure Computing Mode 2 and so on.

“Containers” made namespaces popular.

The new kid on the block is BPF. You can “solve” everything nowadays with BPF

High level jailing technologies

SELinux - Put labels on everything and rules on how those labels are allowed to interact with each other. Eg. If you’re a process spawned from a network listening service, you’re not allowed to load kernel modules.

Apparmor - Not armoring the “app” from the world, but rather “jail” the app to protect the rests of the system from this “app”.

Linux capabilities - Giving or redacting root-like privileges from processes. Disallowing a root process from eg. using raw network sockets. Allowing a non root process to `bind()` to port <1024

High level jailing technologies - continuation

cgroups - Allow a group of processes to run on a certain cpu, consume a certain amount of memory, do a certain amount of I/O.

Secure Computing Mode 2 (seccomp) - Install filters on which syscalls a process is allowed to perform.

Namespaces - Re-map uid’s, mounts, pids, IPC, Time, for a group of processes. Eg. uid 65535 is now uid 0 for this group of processes. /home is now read only for this group of processes. Shmid X is now shmid Y.

Low level jailing technologies

BPF. Berkeley packet Filter. Used to be a technology to inform the kernel about which packets you where interested in capturing.

Now run in a VM in the Linux kernel, and BPF programs can be attached to everything and the kitchen sink. (`BPF_PROG_ATTACH`)

Hints:

```
src/core/bpf-devices.c
src/core/bpf-firewall.c
src/core/bpf-foreign.c
src/core/bpf-restrict-fs.c
src/core/bpf-restrict-ifaces.c
src/core/bpf-socket-bind.c
```

Why doing it with systemd - example?

```
static int drop_capabilities(void)
{
    /* The posix/linux low-level API is ugly, but doesn't require libcap */
    struct __user_cap_header_struct cap_header_data = {0};
    cap_user_header_t cap_header = &cap_header_data;
    struct __user_cap_data_struct cap_data_data = {0};
    cap_user_data_t cap_data = &cap_data_data;
    cap_header->pid = getpid();
    cap_header->version = _LINUX_CAPABILITY_VERSION_3;
    if (capget(cap_header, cap_data) < 0) {
        perror("capget() failed");
        return -1;
    }
    if (!(cap_data->inheritable & CAP_TO_MASK(CAP_DAC_OVERRIDE)) &&
        !(cap_data->permitted   & CAP_TO_MASK(CAP_DAC_OVERRIDE)) &&
        !(cap_data->effective   & CAP_TO_MASK(CAP_DAC_OVERRIDE))) {
        // fprintf(stderr, "No need to drop CAP_DAC_OVERRIDE from main sets\n");
    } else {
        /* Drop the capability from the inheritable and effective sets.
         * This stops subthreads and forked processes from using it. */
        cap_data->inheritable &= ~CAP_TO_MASK(CAP_DAC_OVERRIDE);
        cap_data->effective   &= ~CAP_TO_MASK(CAP_DAC_OVERRIDE);
        if (capset(cap_header, cap_data) < 0) {
            perror("capset() failed");
            return -1
        }
        /* Also drop the capability from the bounding set */
        int res = prctl(PR_CAPBSET_READ, CAP_DAC_OVERRIDE, 0, 0, 0, 0);
        if (res == -1) {
            perror("prctl(PB_CAPBSET_READ,...) failed");
        } else if (res == 0) {
            // fprintf(stderr, "No need to drop CAP_DAC_OVERRIDE from bounding set.\n");
        } else if (prctl(PR_CAPBSET_DROP, CAP_DAC_OVERRIDE, 0, 0, 0)) {
            perror("prctl(PB_CAPBSET_DROP,...) failed");
        }
    }
    return 0;
}
```

```
[Service]
# Don’t allow root service to write to
read-only things
CapabilityBoundingSet=~CAP_DAC_OVERRIDE
```

Discovery, what? - `systemd-analyze security`

There's a simple tool to help you spot your most exposed services:

```
$ systemd-analyze security
UNIT                                   EXPOSURE PREDICATE
HAPPY
ModemManager.service                        6.3 MEDIUM    😐
NetworkManager.service                      7.8 EXPOSED   🙁
accounts-daemon.service                     5.5 MEDIUM    😐
acpid.service                               9.6 UNSAFE    😨
systemd-resolved.service                    2.1 OK        🙂
…
```

Discovery, how? - `systemd-analyze security <service>`

```
systemd-analyze security acpid.service
  NAME                 DESCRIPTION                                    EXPOSURE
✗ PrivateNetwork=      Service has access to the host's network            0.5
✗ User=/DynamicUser=   Service runs as root user                           0.4
✗ DeviceAllow=         Service has no device ACL                           0.2
✗ IPAddressDeny=       Service does not define an IP address allow list    0.2
✓ KeyringMode=         Service doesn't share key material with other services
✗ NoNewPrivileges=     Service processes may acquire new privileges        0.2
✓ NotifyAccess=        Service child processes cannot alter service state
…

<total of 83 suggested jailings>

…

→ Overall exposure level for acpid.service: 9.6 UNSAFE 😨
```

Systemd-resolved.service - Example

```
[Unit]
Description=Network Name Resolution

[Service]
AmbientCapabilities=CAP_SETPCAP
CAP_NET_RAW CAP_NET_BIND_SERVICE
CapabilityBoundingSet=CAP_SETPCAP
CAP_NET_RAW CAP_NET_BIND_SERVICE
LockPersonality=yes
MemoryDenyWriteExecute=yes
NoNewPrivileges=yes
PrivateDevices=yes
PrivateTmp=yes
ProtectProc=invisible
ProtectClock=yes
ProtectControlGroups=yes
ProtectHome=yes
ProtectKernelLogs=yes
ProtectKernelModules=yes
ProtectKernelTunables=yes
ProtectSystem=strict
RestrictAddressFamilies=AF_UNIX
AF_NETLINK AF_INET AF_INET6
RestrictNamespaces=yes
RestrictRealtime=yes
RestrictSUIDSGID=yes
RuntimeDirectoryPreserve=yes
SystemCallArchitectures=native
SystemCallErrorNumber=EPERM
SystemCallFilter=@system-service
User=systemd-resolve

…
```

Yes, it’s messy

- But it’s way less messy than the alternatives
- You collect all the whole “mess” in one place
- Discovery, clever people have already been there, follow their steps
- Unifies access to a whole slew of technologies in a single place

Code is never run with privileges

Quite a few of the interfaces needs elevated privileges to drop it’s privileges

By letting systemd to all the work, no privileges are needed in your code

Can reasonably be retrofitted into old/3pp applications

By letting systemd do the jailing and then start the application inside the jail, old, 3rd party, binary only, whatever applications can be placed inside jails with modern techniques.

Usually you find a couple of old sins.

Buuut, Apparmor? SELinux? Smack?

A AppArmor profile can easily be applied to your service to, just name it with:

`AppArmorProfile=`

A Service can be started in a specific SELinux context to:

`SELinuxContext=`

A Smack label can also be assigned to your service:

`SmackProcessLabel=`

You don’t need to throw away your old world view, but they might not always mix...

But, I have some parts which needs to be privileged?

Quite common that you have some parts, callouts, actions, which needs to have more/different privileges.

A good technique for this is to split it to a separate socket activated service.

Socket activated callout

```
foo.socket:
[Socket]
ListenStream=/run/%N/socket
SocketMode=660
Accept=true
SocketGroup=foo_callers
```

```
foo@service:
[Service]
Type=simple
ExecStart=/usr/local/bin/foo
StandardInput=socket
StandardError=journal
<Jailing>...
```

Will create a socket, for which a certain group can connect to and get some command run in a different service, and thus with a different set of privileges.

Any data can be send to service over the socket, and service will get it on stdin/stdout pipe.

```
socat ECHO:"engage" UNIX:/run/foo/socket
```

Networked services without network access

Socket activation can be used to run a networked service without any network access to. Systemd will either pass the “accepted” connection to a per-connection instance of the service or systemd will pass the “listen” socket to the service on first connection.

Thus the service can be run in a jailed context without access to the network, except for the connection systemd have set up for it and then handed over.

Constrained network access for app

One nice trick is to run a app with limited filtered network access.

```
RestrictAddressFamilies=AF_INET
Environment=http_proxy=http://127.0.0.1:8888/
Environment=https_proxy=http://127.0.0.1:8888/
IPAddressAllow=localhost
IPAddressDeny=any
```

This way you can have a “web application firewall” but in reverse. Limits injections.

honeypot.socket

```
honeypot.socket:
[Socket]
ListenStream=:23
Accept=true
```

One can pretty simply create a honeypot to see what someone who connects to it tries to do…

```
honeypot@service:
[Service]
Type=simple
ExecStart=/usr/local/bin/shell_with_logginng
StandardInput=socket
ProtectSystem=strict
PrivateNetwork=true
<Jailing>...
```

I used to do this with VMS, just to see the frustration when script kiddies tries to use DCL… Those were the days.

There’s lots more…

systemd.unit(5), systemd.service(5), systemd.socket(5), systemd.kill(5), systemd.exec(5), systemd.resource-control(5), systemd.directives(7)

But that’s left as a exercise for the interested.

Questions?