Linux Hardening Guide

Last edited: March 19th, 2022

Linux is not a secure operating system. However, there are steps you can take to improve it. This guide aims to explain how to harden Linux as much as possible for security and privacy. This guide attempts to be distribution-agnostic and is not tied to any specific one.

DISCLAIMER: Do not attempt to apply anything in this article if you do not know exactly what you are doing. This guide is focused purely on security and privacy — not performance, usability or anything else. This guide is not intended to be followed exactly — readers must examine their own threat model and decide which steps to apply. This guide is also not meant to attack a particular group of people or software. Certain software is recommended against in this guide due to security concerns, but this is not out of disdain. At its core, hardening is reducing the ways in which your system can be attacked. Under some threat models, the attack surface presented by a specific program may be too large to be acceptable. Whether or not this is applicable to you depends on your personal threat model.

All commands listed in this guide will require root privileges. Words beginning with "$" sign indicate a variable that may differ between users as to suit their setup.

Contents

1. Choosing the right Linux distribution

There are many factors that go into choosing a good Linux distribution.

- Avoid distributions that freeze packages, as they are often quite behind on security updates.

- Use a distribution with an init system other than systemd. systemd contains a lot of unnecessary attack surface and inserts a considerable amount of complexity into the most privileged user space component; it attempts to do far more things than necessary and goes beyond what an init system should do. An init system should not need many lines of code to function properly. While a common argument in favour of systemd is its ability to sandbox system services, this can be replicated on other init systems through sandboxing utilities like bubblewrap, as documented below.

- Use musl as the default C library. musl is heavily focused on minimality, which results in very small attack surface, whereas other C libraries such as glibc are overly complex and prone to vulnerabilities. For example, over a hundred vulnerabilities in glibc have been publicly disclosed, compared to the very few in musl. While counting CVEs by itself is often an inaccurate statistic, in this case, it represents an overarching issue and is symptomatic of underlying security issues. musl also has invested in decent exploit mitigations, particularly its hardened memory allocator, heavily inspired by GrapheneOS' hardened_malloc.

- Preferably use a distribution that utilises LibreSSL by default rather than OpenSSL. OpenSSL contains tremendous amounts of totally unnecessary attack surface and follows poor security practices. For example, it still maintains OS/2 and VMS support — ancient operating systems that are multiple decades old. These abhorrent security practices are what led to the dreaded Heartbleed vulnerability. LibreSSL is a fork of OpenSSL by the OpenBSD team that applies superior programming practices and eradicates a lot of attack surface. Within LibreSSL's first year, it mitigated a large number of vulnerabilities, including a few high severity ones.

The best distribution to use as a base for your hardened operating system would be Gentoo Linux, as it allows you to configure your system exactly how you want it to be, which will be extremely useful, especially when we come to more secure compilation flags later in the guide.

However, Gentoo may not be feasible for many people due to its significant usability

pitfalls. In this case, Void Linux's musl build or Alpine Linux would be a good compromise.

2. Kernels

The kernel is the core of the operating system and is unfortunately very prone to attacks. As Brad Spengler has once said, it can be thought of as the largest, most vulnerable setuid root binary on the system. Thus, it is very important for the kernel to be hardened as much as possible.

2.1 Stable vs. LTS kernels

The Linux kernel is released under two main forms: stable and long-term support (LTS). Stable releases are more recent, whereas LTS releases are an older stable release that is being supported for a long time. There are many consequences to choosing either of the aforementioned releases.

The Linux kernel does not use CVEs to identify security vulnerabilities properly. This means that fixes for most security vulnerabilities are not able to be backported to LTS kernels; but stable releases contain all security fixes made so far.

With those fixes, however, a stable kernel includes a lot more new features, therefore vastly increasing the attack surface of the kernel and introducing a large amount of new bugs. On the contrary, LTS kernels have less attack surface because these features are not being constantly added.

Additionally, stable kernels include newer hardening features to mitigate certain exploits which LTS kernels do not. A few examples of such features are the Lockdown LSM and STACKLEAK GCC plugin.

To conclude, there is a trade-off when choosing a stable or LTS kernel. LTS kernels have less hardening features and not all public bug fixes at that point have been backported, but it generally has less attack surface and potentially a smaller chance of introducing unknown bugs. Stable kernels have more hardening features and all known bug fixes are included, but it also has more attack surface and greater chances

of introducing more unknown bugs. In the end though, a more recent LTS branch, such as the 4.19 kernel, would be preferred due to the much smaller attack surface.

2.2 Sysctl

Sysctl is a tool that allows the user to configure certain kernel settings and enable various security features or disable dangerous features to reduce attack surface. To change settings temporarily you can execute:

```
sysctl -w $tunable = $value
```

To change sysctls permanently, you can add the one you want to change to `/etc/sysctl.conf` or the corresponding files within `/etc/sysctl.d`, depending on your Linux distribution.

Since Linux 5.8, sysctls can also be set via the `sysctl.$tunable=$value` boot parameter. This may be better, as it is set at the beginning of the boot process, without depending on a user space service to read the values from configuration files.

The following are the recommended sysctl settings that you should change.

2.2.1 Kernel self-protection

```
kernel.kptr_restrict=2
```

A kernel pointer points to a specific location in kernel memory. These can be very useful in exploiting the kernel, but kernel pointers are not hidden by default — it is easy to uncover them by, for example, reading the contents of `/proc/kallsyms`. This setting aims to mitigate kernel pointer leaks. Alternatively, you can set `kernel.kptr_restrict=1` to only hide kernel pointers from processes without the `CAP_SYSLOG` capability.

```
kernel.dmesg_restrict=1
```

dmesg is the kernel log. It exposes a large amount of useful kernel debugging information, but this can often leak sensitive information, such as kernel pointers. Changing the above sysctl restricts the kernel log to the `CAP_SYSLOG` capability.

```
kernel.printk=3 3 3 3
```

Despite the value of `dmesg_restrict`, the kernel log will still be displayed in the console during boot. Malware that is able to record the screen during boot may be able to abuse this to gain higher privileges. This option prevents those information leaks. This must be used in combination with certain boot parameters described below to be fully effective.

```
kernel.unprivileged_bpf_disabled=1
net.core.bpf_jit_harden=2
```

eBPF exposes quite large attack surface. As such, it must be restricted. These sysctls restrict eBPF to the `CAP_BPF` capability (`CAP_SYS_ADMIN` on kernel versions prior to 5.8) and enable JIT hardening techniques, such as constant blinding.

```
dev.tty.ldisc_autoload=0
```

This restricts loading TTY line disciplines to the `CAP_SYS_MODULE` capability to prevent unprivileged attackers from loading vulnerable line disciplines with the `TIOCSETD` ioctl, which has been abused in a number of exploits before.

```
vm.unprivileged_userfaultfd=0
```

The `userfaultfd()` syscall is often abused to exploit use-after-free flaws. Due to this, this sysctl is used to restrict this syscall to the `CAP_SYS_PTRACE` capability.

```
kernel.kexec_load_disabled=1
```

kexec is a system call that is used to boot another kernel during runtime. This functionality can be abused to load a malicious kernel and gain arbitrary code execution in kernel mode, so this sysctl disables it.

```
kernel.sysrq=4
```

The SysRq key exposes a lot of potentially dangerous debugging functionality to unprivileged users. Contrary to common assumptions, SysRq is not only an issue for physical attacks, as it can also be triggered remotely. The value of this sysctl makes it so that a user can only use the secure attention key, which will be necessary for accessing root securely. Alternatively, you can simply set the value to `0` to disable SysRq completely.

```
kernel.unprivileged_userns_clone=0
```

User namespaces are a feature in the kernel which aim to improve sandboxing and make it easily accessible for unprivileged users. However, this feature exposes significant kernel attack surface for privilege escalation, so this sysctl restricts the usage of user namespaces to the `CAP_SYS_ADMIN` capability. For unprivileged sandboxing, it is instead recommended to use a setuid binary with little attack surface to minimise the potential for privilege escalation. This topic is covered further in the

sandboxing section.

Be aware though that this sysctl only exists on certain Linux distributions, as it requires a kernel patch. If your kernel does not include this patch, you can alternatively disable user namespaces completely (including for root) by setting `user.max_user_namespaces=0`.

```
kernel.perf_event_paranoid=3
```

Performance events add considerable kernel attack surface and have caused abundant vulnerabilities. This sysctl restricts all usage of performance events to the `CAP_PERFMON` capability (`CAP_SYS_ADMIN` on kernel versions prior to 5.8).

Be aware that this sysctl also requires a kernel patch that is only available on certain distributions. Otherwise, this setting is equivalent to `kernel.perf_event_paranoid=2`, which only restricts a subset of this functionality.

2.2.2 Network

```
net.ipv4.tcp_syncookies=1
```

This helps protect against SYN flood attacks, which are a form of denial-of-service attack, in which an attacker sends a large amount of bogus SYN requests in an attempt to consume enough resources to make the system unresponsive to legitimate traffic.

```
net.ipv4.tcp_rfc1337=1
```

This protects against time-wait assassination by dropping RST packets for sockets in the time-wait state.

```
net.ipv4.conf.all.rp_filter=1
net.ipv4.conf.default.rp_filter=1
```

These enable source validation of packets received from all interfaces of the machine. This protects against IP spoofing, in which an attacker sends a packet with a fraudulent IP address.

```
net.ipv4.conf.all.accept_redirects=0
net.ipv4.conf.default.accept_redirects=0
net.ipv4.conf.all.secure_redirects=0
net.ipv4.conf.default.secure_redirects=0
net.ipv6.conf.all.accept_redirects=0
net.ipv6.conf.default.accept_redirects=0
net.ipv4.conf.all.send_redirects=0
net.ipv4.conf.default.send_redirects=0
```

These disable ICMP redirect acceptance and sending to prevent man-in-the-middle attacks and minimise information disclosure.

```
net.ipv4.icmp_echo_ignore_all=1
```

This setting makes your system ignore all ICMP requests to avoid Smurf attacks, make the device more difficult to enumerate on the network and prevent clock fingerprinting through ICMP timestamps.

```
net.ipv4.conf.all.accept_source_route=0
net.ipv4.conf.default.accept_source_route=0
net.ipv6.conf.all.accept_source_route=0
net.ipv6.conf.default.accept_source_route=0
```

Source routing is a mechanism that allows users to redirect network traffic. As this can be used to perform man-in-the-middle attacks in which the traffic is redirected for nefarious purposes, the above settings disable this functionality.

```
net.ipv6.conf.all.accept_ra=0
net.ipv6.conf.default.accept_ra=0
```

Malicious IPv6 router advertisements can result in a man-in-the-middle attack, so they should be disabled.

```
net.ipv4.tcp_sack=0
net.ipv4.tcp_dsack=0
net.ipv4.tcp_fack=0
```

This disables TCP SACK. SACK is commonly exploited and unnecessary in many circumstances, so it should be disabled if it is not required.

2.2.3 User space

```
kernel.yama.ptrace_scope=2
```

ptrace is a system call that allows a program to alter and inspect another running process, which allows attackers to trivially modify the memory of other running programs. This restricts usage of ptrace to only processes with the `CAP_SYS_PTRACE` capability. Alternatively, set the sysctl to `3` to disable ptrace entirely.

```
vm.mmap_rnd_bits=32
vm.mmap_rnd_compat_bits=16
```

ASLR is a common exploit mitigation which randomises the position of critical parts of a process in memory. This can make a wide variety of exploits harder to pull off, as they first require an information leak. The above settings increase the bits of entropy used for mmap ASLR, improving its effectiveness.

The values of these sysctls must be set in relation to the CPU architecture. The above values are compatible with x86, but other architectures may differ.

```
fs.protected_symlinks=1
fs.protected_hardlinks=1
```

This only permits symlinks to be followed when outside of a world-writable sticky directory, when the owner of the symlink and follower match or when the directory owner matches the symlink's owner. This also prevents hardlinks from being created by users that do not have read/write access to the source file. Both of these prevent many common TOCTOU races.

```
fs.protected_fifos=2
fs.protected_regular=2
```

These prevent creating files in potentially attacker-controlled environments, such as world-writable directories, to make data spoofing attacks more difficult.

2.3 Boot parameters

Boot parameters pass settings to the kernel at boot using your bootloader. Some settings can be used to increase security, similar to sysctl. Bootloaders often differ in how boot parameters are set. A few examples are listed below, but you should research the required steps for your specific bootloader.

- If using GRUB as your bootloader, edit `/etc/default/grub`, and add your parameters to the `GRUB_CMDLINE_LINUX_DEFAULT=` line.
- If using Syslinux, edit `/boot/syslinux/syslinux.cfg`, and add them to the `APPEND` line.
- If using systemd-boot, edit your loader entry, and append them to the end of the `linux` line.

The following settings are recommended to increase security.

This section originally recommended to apply various `slub_debug` options; however, due to Linux deciding to implicitly disable kernel pointer hashing when using this option, in addition to several other issues with these features, they can no longer be recommended. Users are instead advised to use `init_on_free` as a replacement for memory poisoning and linux-hardened's slab canaries in place of redzoning. If `slub_debug` is in use for anything other than debugging, it is highly recommended to remove it immediately.

2.3.1 Kernel self-protection

```
slab_nomerge
```

This disables slab merging, which significantly increases the difficulty of heap exploitation by preventing overwriting objects from merged caches and by making it harder to influence slab cache layout.

```
init_on_alloc=1 init_on_free=1
```

This enables zeroing of memory during allocation and free time, which can help mitigate use-after-free vulnerabilities and erase sensitive information in memory.

```
page_alloc.shuffle=1
```

This option randomises page allocator freelists, improving security by making page allocations less predictable. This also *improves* performance.

```
pti=on
```

This enables Kernel Page Table Isolation, which mitigates Meltdown and prevents some KASLR bypasses.

```
randomize_kstack_offset=on
```

This option randomises the kernel stack offset on each syscall, which makes attacks that rely on deterministic kernel stack layout significantly more difficult, such as the exploitation of CVE-2019-18683.

```
vsyscall=none
```

This disables vsyscalls, as they are obsolete and have been replaced with vDSO. vsyscalls are also at fixed addresses in memory, making them a potential target for ROP attacks.

```
debugfs=off
```

This disables debugfs, which exposes a lot of sensitive information about the kernel.

```
oops=panic
```

Sometimes certain kernel exploits will cause what is known as an "oops". This parameter will cause the kernel to panic on such oopses, thereby preventing those exploits. However, sometimes bad drivers cause harmless oopses which would result in your system crashing, meaning this boot parameter can only be used on certain hardware.

```
module.sig_enforce=1
```

This only allows kernel modules that have been signed with a valid key to be loaded, which increases security by making it much harder to load a malicious kernel module. This prevents all out-of-tree kernel modules, including DKMS modules from being loaded unless you have signed them, meaning that modules such as the VirtualBox or Nvidia drivers may not be usable, although that may not be important, depending on your setup.

```
lockdown=confidentiality
```

The kernel lockdown LSM can eliminate many methods that user space code could abuse to escalate to kernel privileges and extract sensitive information. This LSM is necessary to implement a clear security boundary between user space and the kernel. The above option enables this feature in confidentiality mode, the strictest option. This implies `module.sig_enforce=1`.

```
mce=0
```

This causes the kernel to panic on uncorrectable errors in ECC memory which could be exploited. This is unnecessary for systems without ECC memory.

```
quiet loglevel=0
```

These parameters prevent information leaks during boot and must be used in combination with the `kernel.printk` sysctl documented above.

2.3.2 CPU mitigations

It is best to enable all CPU mitigations that are applicable to your CPU as to ensure that you are not affected by known vulnerabilities. This is a list that enables all built-in mitigations:

```
spectre_v2=on spec_store_bypass_disable=on tsx=off tsx_async_abort=full,nosmt
mds=full,nosmt l1tf=full,force nosmt=force kvm.nx_huge_pages=force
```

You must research the CPU vulnerabilities that your system is affected by and apply a selection of the above mitigations accordingly. Keep in mind that you will need to install microcode updates to be fully protected from these vulnerabilities. All of these may cause a significant performance decrease.

2.3.3 Result

If you have followed all of the above recommendations, excluding your specific CPU mitigations, you will have:

```
slab_nomerge init_on_alloc=1 init_on_free=1 page_alloc.shuffle=1 pti=on
vsyscall=none debugfs=off oops=panic module.sig_enforce=1
```

```
lockdown=confidentiality mce=0 quiet loglevel=0
```

You may need to regenerate your GRUB configuration file to apply these if using GRUB as your bootloader.

2.4 hidepid

`/proc` is a pseudo-filesystem that contains information about all processes currently running on the system. By default, this is accessible to all users, which can allow an attacker to spy on other processes. To permit users to only see their own processes and not those of other users, you must mount `/proc` with the `hidepid=2,gid=proc` mount options. `gid=proc` exempts the `proc` group from this feature so you can whitelist specific users or processes. One way to add these mount options is to edit `/etc/fstab` and add:

```
proc /proc proc nosuid,nodev,noexec,hidepid=2,gid=proc 0 0
```

systemd-logind still needs to see other users' processes, so for user sessions to work correctly on a systemd system, you must create `/etc/systemd/system/systemd-logind.service.d/hidepid.conf` and add:

```
[Service]
SupplementaryGroups=proc
```

2.5 Kernel attack surface reduction

It is best to disable any functionality that is not absolutely required as to minimise potential kernel attack surface. These features do not necessarily have to be dangerous; they could simply be benign code that is removed to reduce attack

surface. Never disable random things that you don't understand. The following are some examples that may be of use, depending on your setup.

2.5.1 Boot parameters

Boot parameters can often be used to reduce attack surface. One such example is:

```
ipv6.disable=1
```

This disables the entire IPv6 stack which may not be required if you have not migrated to it. Do not use this boot parameter if you are using IPv6.

2.5.2 Blacklisting kernel modules

The kernel allows unprivileged users to indirectly cause certain modules to be loaded via module auto-loading. This allows an attacker to auto-load a vulnerable module which is then exploited. One such example is CVE-2017-6074, in which an attacker could trigger the DCCP kernel module to be loaded by initiating a DCCP connection and then exploit a vulnerability in said kernel module.

Specific kernel modules can be blacklisted by inserting files into `/etc/modprobe.d` with instructions on which kernel modules to blacklist.

The `install` parameter tells `modprobe` to run a specific command instead of loading the module as normal. `/bin/false` is a command that simply returns `1`, which will essentially do nothing. Both of these together tells the kernel to run `/bin/false` instead of loading the module, which will prevent the module from being exploited by attackers. The following are kernel modules that are most likely to be unnecessary:

```
install dccp /bin/false
install sctp /bin/false
install rds /bin/false
```

```
install tipc /bin/false
install n-hdlc /bin/false
install ax25 /bin/false
install netrom /bin/false
install x25 /bin/false
install rose /bin/false
install decnet /bin/false
install econet /bin/false
install af_802154 /bin/false
install ipx /bin/false
install appletalk /bin/false
install psnap /bin/false
install p8023 /bin/false
install p8022 /bin/false
install can /bin/false
install atm /bin/false
```

Obscure networking protocols in particular add considerable remote attack surface. This blacklists:

- DCCP — Datagram Congestion Control Protocol
- SCTP — Stream Control Transmission Protocol
- RDS — Reliable Datagram Sockets
- TIPC — Transparent Inter-process Communication
- HDLC — High-Level Data Link Control
- AX25 — Amateur X.25
- NetRom
- X25
- ROSE
- DECnet
- Econet
- af_802154 — IEEE 802.15.4
- IPX — Internetwork Packet Exchange
- AppleTalk
- PSNAP — Subnetwork Access Protocol
- p8023 — Novell raw IEEE 802.3

- p8022 — IEEE 802.2
- CAN — Controller Area Network
- ATM

```
install cramfs /bin/false
install freevxfs /bin/false
install jffs2 /bin/false
install hfs /bin/false
install hfsplus /bin/false
install squashfs /bin/false
install udf /bin/false
```

This blacklists various rare filesystems.

```
install cifs /bin/true
install nfs /bin/true
install nfsv3 /bin/true
install nfsv4 /bin/true
install ksmbd /bin/true
install gfs2 /bin/true
```

Network filesystems can also be blacklisted if not in use.

```
install vivid /bin/false
```

The vivid driver is only useful for testing purposes and has been the cause of privilege escalation vulnerabilities, so it should be disabled.

```
install bluetooth /bin/false
install btusb /bin/false
```

This disables Bluetooth, which has a history of security issues.

```
install uvcvideo /bin/false
```

This disables the webcam to prevent it from being used to spy on you.

You can also blacklist the microphone module; however, this can differ from system to system. To find the name of the module, look in `/proc/asound/modules` and blacklist it. For example, one such module is `snd_hda_intel`.

Be aware though that sometimes the kernel module for the microphone is the same as the module for the speaker. This means that disabling the microphone like this may also inadvertently disable any speakers. Although speakers can potentially be turned into microphones too, so this isn't necessarily a negative outcome.

It would be preferred to physically remove these devices or, at the very least, disable them in the BIOS / UEFI. Disabling the kernel modules is not as effective.

2.5.3 rfkill

Wireless devices can be blacklisted through `rfkill` to reduce remote attack surface further. To blacklist all wireless devices, execute:

```
rfkill block all
```

WiFi can be unblocked with:

```
rfkill unblock wifi
```

On systems using systemd, rfkill persists across sessions. However, on systems using a different init system, you may have to create an init script to execute these commands upon boot.

2.6 Other kernel pointer leaks

Previous sections have prevented some kernel pointer leaks, but there are still a few more.

On the filesystem, there exists the kernel images and System.map files in `/boot`. There is also other sensitive kernel information in the `/usr/src` and `/{,usr/}lib/modules` directories. You should restrict the file permissions of these directories to make them only readable by the root user. You should also delete the System.map files, as they are not required for anything except advanced debugging.

Additionally, certain logging daemons, such as systemd's `journalctl`, include the kernel logs which can be used to bypass the above `dmesg_restrict` protection. Removing the user from the `adm` group is often sufficient to revoke access to these logs:

```
gpasswd -d $user adm
```

2.7 Restricting access to sysfs

sysfs is a pseudo-filesystem which provides large quantities of kernel and hardware information. It is commonly mounted at `/sys`. sysfs has been the cause of numerous information leaks, particularly of kernel pointers. Whonix's security-misc package includes the hide-hardware-info script, which restricts access to this directory as well as a few in `/proc` in an attempt to hide potential hardware identifiers and prevent kernel pointer leaks. This script is configurable and allows whitelisting specific

applications based on groups. It is recommended to apply this and make it execute on boot with an init script. For example, this is a systemd service to do so.

For basic functionality to work on systems using systemd, you must whitelist a few system services. This can be done by creating `/etc/systemd/system/user@.service.d/sysfs.conf` and adding:

```
[Service]
SupplementaryGroups=sysfs
```

This will not fix everything though. Many applications may still break and it is up to you to whitelist them properly.

2.8 linux-hardened

Certain distributions, such as Arch Linux, include a hardened kernel package. This contains many hardening patches and a more security-conscious kernel configuration. It is recommended to install this if possible.

2.9 Grsecurity

Grsecurity is a set of kernel patches that can massively increase kernel security. These patches used to be freely available, but they are now commercial and must be purchased. If it is available, then it is highly recommended that you get it. Grsecurity offers state-of-the-art kernel and user space protections.

2.10 Linux Kernel Runtime Guard

Linux Kernel Runtime Guard (LKRG) is a kernel module which ensures kernel integrity at runtime and detects exploits. It can kill entire classes of kernel exploits; but it is not a perfect mitigation, as LKRG is bypassable by design. It is only suitable for

off-the-shelf malware. However, while it is unlikely, LKRG may in itself expose new vulnerabilities like any additional kernel module.

2.11 Kernel self-compilation

It is recommended to compile your own kernel whilst enabling as little kernel modules as possible and as many security features as possible to keep the attack surface of the kernel at an absolute minimum.

In addition, apply kernel hardening patches, such as linux-hardened or grsecurity, as mentioned above.

Distribution-compiled kernels also have public kernel pointers / symbols which are very useful for exploits. Compiling your own kernel will give you unique kernel symbols, which, along with `kptr_restrict`, `dmesg_restrict` and other hardening against kernel pointer leaks, will make it considerably harder for attackers to create exploits that rely on kernel pointer knowledge.

You can take inspiration from or use Whonix's hardened-kernel once it is complete to develop your kernel configuration.

3. Mandatory access control

Mandatory access control (MAC) systems give fine-grained control over what programs can access. This means that your browser won't have access to your entire home directory or similarly.

The most used MAC systems are SELinux and AppArmor. SELinux is a lot more secure than AppArmor, as it is more fine-grained. For example, it's inode-based rather than path-based, allows enforcing significantly stronger restrictions, can filter kernel ioctls and much more. Unfortunately, this comes at the cost of being much more difficult to use and harder to learn, so AppArmor may be preferred by some.

To enable AppArmor in the kernel, you must set the following boot parameters:

```
apparmor=1 security=apparmor
```

To enable SELinux instead, set these parameters:

```
selinux=1 security=selinux
```

Keep in mind that simply enabling a MAC system won't by itself magically increase security. You must develop strict policies to fully utilise it. For example, to create AppArmor profiles, execute:

```
aa-genprof $path_to_program
```

Open the program and start using it as you normally would. AppArmor will detect what files it needs to access and will add them to the profile if you choose. This alone will not be sufficient for high quality profiles though; seek the AppArmor documentation for more details.

If you want to take it a step further, you can setup a full system MAC policy that confines every single user space process by implementing an initramfs hook which enforces a MAC policy for the init system. This is how Android uses SELinux and how Whonix will use AppArmor in the future. This is necessary for enforcing a strong security model implementing the principle of least privilege.

4. Sandboxing

4.1 Application sandboxing

A sandbox allow you to run a program in an isolated environment that has either limited access or none at all to the rest of your system. You can use these to secure

applications or run untrusted programs.

It is recommended to use bubblewrap in a separate user account along with AppArmor or SELinux to sandbox programs. You could also consider using gVisor instead, which has the advantage of providing each guest with its own kernel.

Either of these can be used to create a very powerful sandbox with minimal attack surface exposed. If you do not wish to create sandboxes yourself, consider using Whonix's sandbox-app-launcher once it is complete. You should not use Firejail.

Container solutions, such as Docker and LXC, are often used as a misguided form of sandboxing. These are too permissive as to support a wide variety of applications, so they cannot be considered a strong application sandbox.

4.2 Common sandbox escapes

4.2.1 PulseAudio

PulseAudio is a common sound server, but it was not written with isolation or sandboxing in mind, making it a recurring sandbox escape vulnerability. To prevent this, it is recommended to block access to PulseAudio from within your sandbox or uninstall it from your system entirely. You can use the standard ALSA utilities or PipeWire instead.

4.2.2 D-Bus

D-Bus is the most prevalent form of inter-process communication on desktop Linux, but it is also another common avenue for sandbox escapes since it allows freely interacting with services. One such example of these vulnerabilities was in Firejail. You should either block access to D-Bus from within the sandbox or mediate it via MAC with fine-grained rules.

4.2.3 GUI isolation

Any Xorg window can access another window. This allows trivial keylogging or screenshot programs that can even record things such as the root password. You can sandbox Xorg windows with a nested X11 server, such as Xpra or Xephyr and bubblewrap.

Wayland isolates windows from each other by default and would be a much better choice than Xorg, although Wayland may not be as generally usable as Xorg since it is earlier in development.

4.2.4 ptrace

As discussed earlier, ptrace is a system call that could be abused to trivially compromise processes running outside of the sandbox. To prevent this, you can enable the kernel's YAMA ptrace restrictions via sysctl, or you can blacklist the ptrace syscall in your seccomp filter.

4.2.5 TIOCSTI

TIOCSTI is an ioctl which allows injecting terminal commands and provides an attacker with an easy mechanism to move laterally among other processes within the same user's session. This attack can be mitigated by blacklisting the ioctl in your seccomp filter or by using bubblewrap's `--new-session` argument.

4.3 Systemd service sandboxing

systemd is unrecommended, but some may be unable to switch. These people can at the very least, sandbox services so they can only access what they need. Here is an example of a sandboxed systemd service:

```
[Service]
CapabilityBoundingSet=CAP_NET_BIND_SERVICE
ProtectSystem=strict
ProtectHome=true
ProtectKernelTunables=true
```

```
ProtectKernelModules=true
ProtectControlGroups=true
ProtectKernelLogs=true
ProtectHostname=true
ProtectClock=true
ProtectProc=invisible
ProcSubset=pid
PrivateTmp=true
PrivateUsers=true
PrivateDevices=true
PrivateIPC=true
MemoryDenyWriteExecute=true
NoNewPrivileges=true
LockPersonality=true
RestrictRealtime=true
RestrictSUIDSGID=true
RestrictAddressFamilies=AF_INET
RestrictNamespaces=true
SystemCallFilter=write read openat close brk fstat lseek mmap mprotect munmap
rt_sigaction rt_sigprocmask ioctl nanosleep select access execve getuid
arch_prctl set_tid_address set_robust_list prlimit64 pread64 getrandom
SystemCallArchitectures=native
UMask=0077
IPAddressDeny=any
AppArmorProfile=/etc/apparmor.d/usr.bin.example
```

Explanations of all the options:

- `CapabilityBoundingSet=` — Specifies the capabilities the process is given.
- `ProtectHome=true` — Makes all home directories inaccessible.
- `ProtectKernelTunables=true` — Mounts kernel tunables, such as those modified through `sysctl`, as read-only.
- `ProtectKernelModules=true` — Denies module loading and unloading.
- `ProtectControlGroups=true` — Mounts all control group hierarchies as read-only.
- `ProtectKernelLogs=true` — Prevents accessing the kernel logs.

- `ProtectHostname=true` — Prevents changes to the system hostname.
- `ProtectClock` — Prevents changes to the system clock.
- `ProtectProc=invisible` — Hides all outside processes.
- `ProcSubset=pid` — Permits access to only the pid subset of `/proc`.
- `PrivateTmp=true` — Mounts an empty tmpfs over `/tmp` and `/var/tmp`, therefore hiding their previous contents.
- `PrivateUsers=true` — Sets up an empty user namespace to hide other user accounts on the system.
- `PrivateDevices=true` — Creates a new `/dev` mount with minimal devices present.
- `PrivateIPC=true` — Sets up an IPC namespace to isolate IPC resources.
- `MemoryDenyWriteExecute=true` — Enforces a memory W^X policy.
- `NoNewPrivileges=true` — Prevents escalating privileges.
- `LockPersonality=true` — Locks down the `personality()` syscall to prevent switching execution domains.
- `RestrictRealtime=true` — Prevents attempts to enable realtime scheduling.
- `RestrictSUIDSGID=true` — Prevents executing setuid or setgid binaries.
- `RestrictAddressFamilies=AF_INET` — Restricts the usable socket address families to IPv4 only (`AF_INET`).
- `RestrictNamespaces=true` — Prevents creating any new namespaces.
- `SystemCallFilter=...` — Restricts the allowed syscalls to the absolute minimum. If you aren't willing to maintain your own custom seccomp filter, then systemd provides many predefined system call sets that you can use. `@system-service` will be suitable for many use cases.
- `SystemCallArchitectures=native` — Prevents executing syscalls from other CPU architectures.
- `UMask=0077` — Sets the umask to a more restrictive value.
- `IPAddressDeny=any` — Blocks all incoming and outgoing traffic to/from any IP address. Set `IPAddressAllow=` to configure a whitelist. Alternatively, setup a network namespace with `PrivateNetwork=true`.
- `AppArmorProfile=...` — Runs the process under the specified AppArmor profile.

You cannot just copy this example configuration into yours. Each service's requirements differ, and the sandbox has to be fine-tuned for each of them specifically. To learn more about all of the options you can set, read the systemd.exec manpage.

If you use an init system other than systemd, then all of these options can be easily replicated with bubblewrap.

4.4 gVisor

Ordinary sandboxes inherently share the same kernel as the host. You are trusting the kernel, which we have already evaluated to be insecure, to confine these programs properly. A kernel exploit from within the sandbox can bypass any restrictions, as the host kernel's entire attack surface is completely exposed. There have been efforts to limit the attack surface with seccomp; however, it is not enough to fully fix this issue. gVisor is a solution to this problem. It provides each application with its own kernel that reimplements a large portion of the Linux kernel's syscalls but in a memory safe language, therefore providing significantly stronger isolation.

4.5 Virtual machines

While not a traditional "sandbox", virtual machines separate processes by virtualising an entirely new system, thereby providing very strong isolation. KVM is a kernel module that allows the kernel to function as a hypervisor, and QEMU is an emulator that utilises KVM. Virt-manager and GNOME Boxes are both good and easy-to-use GUIs to manage KVM/QEMU virtual machines. Virtualbox is unrecommended for several reasons.

5. Hardened memory allocator

hardened_malloc is a hardened memory allocator that provides substantial protection from heap memory corruption vulnerabilities. It is heavily based on OpenBSD's malloc design but with numerous improvements.

hardened_malloc can be used per-application via the `LD_PRELOAD` environment variable. For example, assuming the library you have compiled is located at `/usr/lib/libhardened_malloc.so`, you can execute:

```
LD_PRELOAD="/usr/lib/libhardened_malloc.so" $program
```

It can also be used system-wide by globally preloading the library, which is the recommended way of using it. To do so, edit `/etc/ld.so.preload` and insert:

```
/usr/lib/libhardened_malloc.so
```

hardened_malloc may break some applications, although the majority will work fine. If issues are experienced, it is recommended to compile hardened_malloc in its "light" configuration via the following build option in order to minimise breakage:

```
VARIANT=light
```

You should also set the following with sysctl to accomodate the large number of guard pages created by hardened_malloc:

```
vm.max_map_count=1048576
```

The Whonix project provides a hardened_malloc package for Debian-based distributions.

6. Hardened compilation flags

Compiling your own programs can offer numerous benefits, as it gives you the ability to optimise your programs for security. However, it is easy to do the exact opposite and worsen security — if you are unsure about what you are doing, skip this section. This will be easiest on a source-based distribution, such as Gentoo, but it is possible

to do this on others.

Certain compilation options can be used to add additional exploit mitigations that eliminate entire classes of common vulnerabilities. You have likely heard of regular protections, such as Position Independent Executables, Stack Smashing Protector, immediate binding, read-only relocations and FORTIFY_SOURCE, but this section will not go over those, as they have already been widely adopted. Instead, it will discuss *modern* exploit mitigations like Control Flow Integrity and shadow stacks.

This section refers to native programs, written primarily in C or C++. You must be using the Clang compiler, as most of these features are not available on GCC. Keep in mind that due to the lack of widespread adoption of these mitigations, certain applications may fail to function with them enabled.

- Control Flow Integrity (CFI) is an exploit mitigation that aims to prevent code reuse attacks like ROP or JOP. A large portion of vulnerabilities are exploited using these techniques due to more widely adopted mitigations, such as NX, making older exploit techniques obsolete. Clang supports fine-grained, forward-edge CFI, meaning that it effectively mitigates *JOP* attacks. Clang's CFI does not mitigate *ROP* by itself; you must also use a separate mechanism as documented below. To enable this, you must apply the following compilation flags:

```
-flto -fvisibility=hidden -fsanitize=cfi
```

- Shadow stacks protect a program's return address by replicating it in a different, hidden stack. The return addresses in the main stack and the shadow stack are then compared in the function epilogue to see if either differ. If so, this would indicate an attack and the program will abort, therefore mitigating ROP attacks. Clang has a feature known as ShadowCallStack which accomplishes this, although it is only available on ARM64. To enable this, you must apply the following compilation flag:

```
-fsanitize=shadow-call-stack
```

- If the aforementioned ShadowCallStack is not an option, you can alternatively use SafeStack which has a similar goal. Unfortunately though, this feature suffers from many vulnerabilities, so it isn't nearly as effective. If you still wish to enable this, you must apply the following compilation flag:

```
-fsanitize=safe-stack
```

- One of the most common memory corruption vulnerabilities is uninitialised memory. Clang and GCC both have options to automatically initialise variables with either zero or a specific pattern. It is recommended to initialise variables to zero, as using other patterns is more suitable for bug finding than exploit mitigation. To enable this, you must apply the following compilation flag:

```
-ftrivial-auto-var-init=zero
```

  If on Clang, you must also apply:

```
-enable-trivial-auto-var-init-zero-knowing-it-will-be-removed-from-clang
```

- An integer overflow is a class of vulnerabilities in which an arithmetic operation causes an integer value to exceed its maximum range, resulting in either the value wrapping around (in the case of unsigned integers) or causing undefined behaviour (in the case of signed integers). Clang's UndefinedBehaviorSanitizer (UBSan) is designed to prevent many forms of undefined behaviour, including integer overflows. To enable Clang's integer sanitisation for both signed and unsigned integers, apply the following compilation flags:

```
-fsanitize=signed-integer-overflow,unsigned-integer-overflow -fsanitize-trap=signed-integer-overflow,unsigned-integer-overflow
```

Note that these options can cause a substantial performance regression. An alternative is to use the `-fwrapv` flag, which causes signed overflows to wrap rather than causing undefined behaviour, although this isn't as comprehensive as UBSan.

- The Stack Clash is an attack in which the stack grows too large, allowing an attacker to jump over the guard page and arbitrarily read/write to another memory region such as the heap. Both Clang and GCC introduced mitigations against this, which can be enabled with the following compilation flag:

```
-fstack-clash-protection
```

7. Memory safe languages

Programs written in memory safe languages are automatically protected from various security vulnerabilities, including buffer overflows, uninitialised variables, use-after-free and more. The majority of vulnerabilities that have been discovered were memory safety issues, as proven by research conducted by security researchers at Microsoft and Google. Examples of such memory **safe** languages include Rust, Swift and Java, whereas examples of memory **unsafe** languages include C and C++. If practical, you should replace as many programs as possible with memory safe alternatives.

8. The root account

Root can do anything and has access to your entire system. Thus, it should be locked down as much as possible so attackers cannot easily gain root access.

8.1 /etc/securetty

The file, `/etc/securetty` specifies where you are allowed to login as root from. This file should be kept empty so that nobody can do so from a terminal.

8.2 Restricting su

`su` lets you switch users from a terminal. By default, it tries to login as root. To restrict the use of `su` to users within the `wheel` group, edit `/etc/pam.d/su` and `/etc/pam.d/su-l` and add:

```
auth required pam_wheel.so use_uid
```

You should have as little users in the `wheel` group as possible.

8.3 Locking the root account

To lock the root account to prevent anyone from ever logging in as root, execute:

```
passwd -l root
```

Make sure that you have an alternative method of gaining root (such as booting from a live USB and chrooting into the filesystem) before doing this so you do not inadvertently lock yourself out of the system.

8.4 Denying root login via SSH

To prevent someone from logging in as root via SSH, edit `/etc/ssh/sshd_config` and add:

```
PermitRootLogin no
```

8.5 Increasing the number of hashing rounds

You can increase the number of hashing rounds that shadow uses, thereby increasing the security of your hashed passwords by forcing an attacker to compute substantially more hashes to crack your password. By default, shadow uses 5000 rounds, but you can increase this to as many as you want. Although the more rounds you configure, the slower it will be to login. Edit `/etc/pam.d/passwd` and add the rounds option. For example:

```
password required pam_unix.so sha512 shadow nullok rounds=65536
```

This makes shadow perform 65536 rounds.

Your passwords are not automatically rehashed after applying this setting, so you need to reset the password with:

```
passwd $username
```

8.6 Restricting Xorg root access

Certain distributions run Xorg as the root user by default. This is an issue because Xorg contains a massive amount of ancient, complicated code, which adds huge attack surface and makes it more likely to have exploits that can gain root privileges. To stop it from being executed as root, edit `/etc/X11/Xwrapper.config` and add:

```
needs_root_rights = no
```

Alternatively, just switch to Wayland.

8.7 Accessing root securely

There are a wide range of methods that malware can use to sniff the password of the root account. As such, traditional ways of accessing the root account are insecure. Preferably, root would not be accessed at all, but this isn't really feasible. This section details the safest way possible of accessing the root account. These instructions should be applied as soon as the OS is installed to make sure it is free of malware.

You must not use your ordinary user account to access root, as it may have been compromised. You also must not log directly into the root account. Create a separate "admin" user account that is used solely for accessing root and nothing else by executing:

```
useradd admin
```

Set a very strong password by executing:

```
passwd admin
```

Allow *only* this account to use your preferred mechanism of escalating privileges. For example, if using `sudo`, add a sudoers exception by executing:

```
visudo -f /etc/sudoers.d/admin-account
```

Now enter:

```
admin ALL=(ALL) ALL
```

Make sure that no other account has access to `sudo` (or your preferred mechanism).

Now, to actually login to this account, reboot first — this prevents, for example, a compromised window manager from performing login spoofing. When provided with a login prompt, activate the secure attention key by pressing the following combination of keys on your keyboard:

```
Alt + SysRq + k
```

This will kill all applications on the current virtual console, therefore defeating login spoofing attacks. Now, you can safely login to your admin account and perform tasks using root. Once you are finished, log out of the admin account and log back in to your unprivileged user account.

9. Firewalls

Firewalls can control incoming and outgoing network traffic and can be used to block or allow certain types of traffic. You should always block all incoming traffic unless you have a specific reason not to. It is recommended to set up a strict iptables or nftables firewall. Firewalls must be fine-tuned for your system, and there is not one ruleset that can fit all of them. It is recommended to get familiar with creating firewall rules. The Arch Wiki and man page are both good resources for this.

This is an example of a basic iptables configuration that disallows all incoming network traffic:

```
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
```

```
:OUTPUT ACCEPT [0:0]
:TCP - [0:0]
:UDP - [0:0]
-A INPUT -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A INPUT -i lo -j ACCEPT
-A INPUT -m conntrack --ctstate INVALID -j DROP
-A INPUT -p udp -m conntrack --ctstate NEW -j UDP
-A INPUT -p tcp --tcp-flags FIN,SYN,RST,ACK SYN -m conntrack --ctstate NEW -j
TCP
-A INPUT -p udp -j REJECT --reject-with icmp-port-unreachable
-A INPUT -p tcp -j REJECT --reject-with tcp-reset
-A INPUT -j REJECT --reject-with icmp-proto-unreachable
COMMIT
```

You should not attempt to use this example on your actual system though. It is only suitable for certain desktop systems.

10. Identifiers

For privacy, it is best to minimise the amount of information that can be traced back to you.

10.1 Hostnames and usernames

Do not put anything uniquely indentifying in your hostname or username. Keep them as generic names, such as "host" and "user", so you cannot be identified by them.

10.2 Timezones / Locales / Keymaps

If possible, your timezone should be set to "UTC" and your locale and keymap to "US".

10.3 Machine ID

A unique Machine ID is stored in `/var/lib/dbus/machine-id` and on systemd systems, `/etc/machine-id` also. These should be edited to something generic, such as the Whonix ID:

```
b08dfa6083e7567a1921a715000001fb
```

10.4 MAC address spoofing

MAC addresses are unique identifiers assigned to network interface controllers (NICs). Every time you connect to a network (e.g. WiFi or ethernet), your MAC address is exposed. This allows people to use it to track you and uniquely identify you on the local network.

You should **not** completely randomise the MAC address. Having a completely random MAC address is obvious and will have the adverse effect of making you stand out. The OUI (Organizationally Unique Identifier) part of the MAC address identifies the chipset's manufacturer. Randomising this part of the MAC address may give you an OUI that has never been used before, hasn't been used in decades or is extremely rare in your area, therefore making you stand out and making it obvious that you are spoofing your MAC address.

The end of the MAC address identifies your specific device and is what can be used to track you. Randomising only this part of the MAC address prevents you from being tracked whilst still making the MAC address seem believable.

To spoof these addresses, first find out your network interface name by executing:

```
ip a
```

Next, install macchanger and execute:

```
macchanger -e $network_interface
```

To randomise the MAC address upon each boot, you should create an init script for your particular init system. This is an example of one for systemd:

```
[Unit]
Description=macchanger on eth0
Wants=network-pre.target
Before=network-pre.target
BindsTo=sys-subsystem-net-devices-eth0.device
After=sys-subsystem-net-devices-eth0.device

[Service]
ExecStart=/usr/bin/macchanger -e eth0
Type=oneshot

[Install]
WantedBy=multi-user.target
```

The above example spoofs the MAC address of the `eth0` interface at boot. Replace `eth0` with your network interface.

10.5 Time attacks

Nearly every system has a different time; this can be used for clock skew fingerprinting attacks. Differences as small as a few milliseconds are sufficient for deanonymizing users.

10.5.1 ICMP timestamps

ICMP timestamps leak the system time in query replies. The easiest way to block these is by blocking incoming connections with a firewall or by making the kernel

ignore ICMP requests.

10.5.2 TCP timestamps

TCP timestamps also leak the system time. The kernel attempted to fix this by using a random offset for each connection, but this is not enough to fix the issue. Thus, TCP timestamps should be disabled. This can be done by setting the following with sysctl:

```
net.ipv4.tcp_timestamps=0
```

10.5.3 TCP initial sequence numbers

TCP initial sequence numbers (ISNs) are another method of leaking the system time. To mitigate this, you must install the tirdad kernel module, which generates random ISNs for connections.

10.5.4 Time synchronisation

Time synchronisation is vital for anonymity and security. A wrong system clock can expose you to clock skew fingerprinting attacks or can be used to feed you outdated HTTPS certificates, bypassing certficate expiry or revocation.

The most popular time synchronisation method, NTP, is insecure, as it is unencrypted and unauthenticated, allowing an attacker to trivially intercept and modify requests. NTP also leaks your local system time in NTP timestamp format, which can be used for clock skew fingerprinting, as briefly mentioned before.

Thus, you should uninstall any NTP clients and disable systemd-timesyncd if it is in use. Instead of NTP, you can connect to a trusted website over a secure connection (HTTPS or, preferably, a Tor onion service) and extract the current time from the HTTP header. Tools that accomplish this are sdwdate or my own secure-time-sync.

10.6 Keystroke fingerprinting

It is possible to fingerprint a person by the manner in which they enter keys on the keyboard. You can be uniquely fingerprinted via your typing speed, pauses in between key presses, the exact time at which each key is pressed and released and so on. It is possible to test this online with KeyTrac.

kloak is a tool that aims to defeat this method of tracking by obfuscating the time intervals between key press and release events. When a key is pressed, it introduces a random delay before it is picked up by the application. Although this may be frustrating for certain individuals and unsuitable for them.

This form of tracking must not be confused with stylometry.

11. File permissions

By default, the permissions of files are quite permissive. You should search across your system for files and directories with improper permissions and restrict them. For example, on some distributions, such as Debian, users' home directories are world-readable. This can be restricted by executing:

```
chmod 700 /home/$user
```

A few more examples are `/boot`, `/usr/src` and `/{,usr/}lib/modules` — these contain the kernel image, System.map and various other files, all of which can leak sensitive information about the kernel. To restrict access to these, execute:

```
chmod 700 /boot /usr/src /lib/modules /usr/lib/modules
```

On Debian-based distributions, the file permissions must be reserved with dpkg-statoverride. Otherwise, they will be overwritten during an update.

Whonix's SUID Disabler and Permission Hardener applies the steps detailed in this section automatically.

11.1 setuid / setgid

setuid / SUID allows a user to execute a binary with the privileges of the binary's owner. This is often used to allow unprivileged users to utilise certain functionality that is normally only reserved for the root user. As such, many SUID binaries have a history of privilege escalation security vulnerabilities. setgid / SGID is similar but for groups rather than users. To find all binaries on the system with the setuid or setgid bit, execute:

```
find / -type f \( -perm -4000 -o -perm -2000 \)
```

You should then remove any unnecessary setuid / setgid bits on programs you don't use, or replace them with capabilities.

To remove the setuid bit, execute:

```
chmod u-s $path_to_program
```

To remove the setgid bit, execute:

```
chmod g-s $path_to_program
```

To add a capability to the file instead, execute:

```
setcap $capability+ep $path_to_program
```

To remove an unnecessary capability, execute:

```
setcap -r $path_to_program
```

11.2 umask

umask sets the default file permissions for newly created files. The default umask is `0022`, which is not very secure, as this gives read access to every user on the system for newly created files. To make new files unreadable by anyone other than the owner, edit `/etc/profile` and add:

```
umask 0077
```

12. Core dumps

Core dumps contain the recorded memory of a program at a specific time, usually when that program has crashed. These can contain sensitive information, such as passwords and encryption keys, so these must be disabled.

There are three main ways to disable them: sysctl, systemd and ulimit.

12.1 sysctl

Set the following setting via sysctl:

```
kernel.core_pattern=|/bin/false
```

12.2 systemd

Create `/etc/systemd/coredump.conf.d/disable.conf` and add:

```
[Coredump]
Storage=none
```

12.3 ulimit

Edit `/etc/security/limits.conf` and add:

```
* hard core 0
```

12.4 setuid processes

Process that run with elevated privileges may still dump their memory even after these settings. To prevent them from doing so, set the following via sysctl:

```
fs.suid_dumpable=0
```

13. Swap

Similar to core dumps, swapping or paging copies parts of memory to disk, which can contain sensitive information. The kernel should be configured to only swap if absolutely necessary with this sysctl:

```
vm.swappiness=1
```

14. PAM

PAM is a framework for user authentication — it's what you use when you login. You can make it more secure by requiring strong passwords or enforcing delays upon failed login attempts.

To enforce strong passwords, you can use pam_pwquality. It enforces a configurable policy for passwords. For example, if you want passwords to contain a minimum of 16 characters (minlen), at least 6 different characters from the old password (difok), at least 3 digits (dcredit), at least 2 uppercase (ucredit), at least 2 lowercase (lcredit) and at least 3 other characters (ocredit), then edit `/etc/pam.d/passwd` and add:

```
password required pam_pwquality.so retry=2 minlen=16 difok=6 dcredit=-3
ucredit=-2 lcredit=-2 ocredit=-3 enforce_for_root
password required pam_unix.so use_authtok sha512 shadow
```

To enforce delays, you can use pam_faildelay. To add a delay of at least 4 seconds between failed login attempts to deter bruteforcing attempts, edit `/etc/pam.d/system-login` and add:

```
auth optional pam_faildelay.so delay=4000000
```

"4000000" being 4 seconds in microseconds.

15. Microcode updates

Microcode updates are essential to fix critical CPU vulnerabilities, such as Meltdown and Spectre, among numerous others. Most distributions include these in their software repositories, such as Arch Linux and Debian.

16. IPv6 privacy extensions

IPv6 addresses are generated from your computer's MAC address, making your IPv6 address unique and tied directly to your computer. Privacy extensions generate a random IPv6 address to mitigate this form of tracking. Note that these steps are unnecessary if you have spoofed your MAC address or have disabled IPv6.

To enable these, set the following settings via sysctl:

```
net.ipv6.conf.all.use_tempaddr=2
net.ipv6.conf.default.use_tempaddr=2
```

16.1 NetworkManager

To enable privacy extensions for NetworkManager, edit `/etc/NetworkManager/NetworkManager.conf` and add:

```
[connection]
ipv6.ip6-privacy=2
```

16.2 systemd-networkd

To enable privacy extensions for systemd-networkd, create `/etc/systemd/network/ipv6-privacy.conf` and add:

```
[Network]
IPv6PrivacyExtensions=kernel
```

17. Partitioning and mount options

File systems should be separated into various partitions to gain fine-grained control over their permissions. Different mount options can be added to restrict what can be done:

- nodev — Disallow devices.
- nosuid — Disallow setuid or setgid bits.
- noexec — Disallow execution of any binaries.

These mount options should be set wherever possible in `/etc/fstab`. If you cannot use separate partitions, then create bind mounts. An example of a more secure `/etc/fstab`:

```
/        /          ext4    defaults                          1 1
/home    /home      ext4    defaults,nosuid,noexec,nodev      1 2
/tmp     /tmp       ext4    defaults,bind,nosuid,noexec,nodev 1 2
/var     /var       ext4    defaults,bind,nosuid              1 2
/boot    /boot      ext4    defaults,nosuid,noexec,nodev      1 2
```

Be aware that `noexec` can be bypassed via shell scripts.

18. Entropy

18.1 Additional entropy sources

Entropy is basically the randomness collected by an operating system and is crucial for things such as encryption. Hence, it is best to gather as much entropy as possible from a variety of sources by installing additional random number generators like haveged and jitterentropy.

For jitterentropy to work properly, the kernel module must be loaded as early as possible by creating `/usr/lib/modules-load.d/jitterentropy.conf` and adding:

```
jitterentropy_rng
```

18.2 RDRAND

RDRAND is a CPU instruction for providing random numbers. It is automatically used by the kernel as an entropy source if it is available; but since it is proprietary and part of the CPU itself, it is impossible to audit and verify its security properties. You are not even able to reverse engineer the code if you wish. This RNG has suffered from vulnerabilities before and often has a weak implementation. It is possible to distrust this feature by setting the following boot parameter:

```
random.trust_cpu=off
```

19. Editing files as root

It is unrecommended to run ordinary text editors as root. Most text editors can do much more than simply edit text files, and this can be exploited. For example, open `vi` as root and enter `:sh`. You now have a root shell with access to your entire system, which an attacker can easily exploit.

A solution to this is using `sudoedit`. This copies the file to a temporary location, opens the text editor as an ordinary user, edits the temporary file and overwrites the original file as root. This way, the actual editor doesn't run as root. To use `sudoedit`, execute:

```
sudoedit $path_to_file
```

By default, it uses `vi`, but the default editor can be switched via the `EDITOR` or `SUDO_EDITOR` environment variables. For example, to use `nano`, execute:

```
EDITOR=nano sudoedit $path_to_file
```

This environment variable can be set globally in `/etc/environment`.

20. Distribution-specific hardening

20.1 HTTPS package manager mirrors

Linux distributions often use HTTP or a mixture of HTTP and HTTPS mirrors by default to download packages from their software repositories. People assume this is fine because package managers verify the signatures of packages before installation. However, historically, there have been multiple bypasses of this. You should configure your package manager to exclusively download from HTTPS mirrors for defence-in-depth.

20.2 APT seccomp-bpf

Since Debian Buster, the package manager, APT has supported optional seccomp-bpf filtering. This restricts the syscalls that APT is allowed to execute, which can severely limit an attacker's ability to do harm to the system if they attempt to exploit a vulnerability in APT. To enable this, create `/etc/apt/apt.conf.d/40sandbox` and add:

```
APT::Sandbox::Seccomp "true";
```

21. Physical security

21.1 Encryption

Full-disk encryption ensures that all data on your drive is encrypted and cannot be read by a physical attacker. Most distributions support enabling encryption during

installation. Make sure you set a strong password. You can also encrypt your drive manually with dm-crypt.

Be aware that full-disk encryption does not cover `/boot`. As such, it is still possible to modify the kernel, bootloader and other critical files. To fully protect against tampering, you must also implement verified boot.

21.2 BIOS / UEFI hardening

If you are still using legacy BIOS, then you should migrate to UEFI as to take advantage of newer security features.

Most BIOS or UEFI implementations support setting a password. It is best to enable this and set a very strong password. This is a weak protection though, as it is trivial to reset the password. It is often stored in volatile memory, so an attacker just needs to be able to remove the CMOS battery for a few seconds, or they can reset it with a jumper on certain motherboards.

You should also disable all unused devices and boot options, such as USB booting, to reduce attack surface.

Updating the BIOS or UEFI is often neglected — make sure you keep them updated. Treat it as important as regular operating system updates.

Additionally, refer to the NSA's Hardware and Firmware Security Guidance.

21.3 Bootloader passwords

The bootloader executes very early in the boot process and is responsible for loading your operating system. It is very important that you protect your bootloader. Otherwise, it can be tampered with — for example, a local attacker can easily gain a root shell by using `init=/bin/bash` as a kernel parameter at boot, which tells the kernel to execute `/bin/bash` instead of your normal init system. You can prevent this by setting a password for your bootloader.

Setting a bootloader password alone is not enough to fully protect it. You must also setup verified boot as documented below.

21.3.1 GRUB

To set a password for GRUB, execute:

```
grub-mkpasswd-pbkdf2
```

Enter your password and a string will be generated from that password. It will be something like "grub.pbkdf2.sha512.10000.C4009...". Create `/etc/grub.d/40_password` and add:

```
set superusers="$username"
password_pbkdf2 $username $password
```

Replace "$password" with the string generated by `grub-mkpasswd-pbkdf2`. "$username" will be for the superusers that are permitted to use the GRUB command line, edit menu entries, and execute any menu entry. For most people, this will just be "root".

Regenerate your configuration file and GRUB will now be password protected.

To restrict only editing the boot parameters and accessing the GRUB console whilst still allowing you to boot, edit `/boot/grub/grub.cfg` and next to "menuentry '$OSName'", add the "--unrestricted" parameter. For example:

```
menuentry 'Arch Linux' --unrestricted
```

You will need to regenerate your configuration file again to apply this change.

21.3.2 Syslinux

Syslinux can either set a master password or a menu password. A master password is required for booting any entry, while a menu password is only required for booting a specific entry.

To set a master password for Syslinux, edit `/boot/syslinux/syslinux.cfg` and add:

```
MENU MASTER PASSWD $password
```

To set a menu password, edit `/boot/syslinux/syslinux.cfg` and within a label that has the item you want to password protect, add:

```
MENU PASSWD $password
```

Replace "$password" with the password you wish to set.

These passwords can either be plaintext or hashed with MD5, SHA-1, SHA-256 or SHA-512. It is recommended that you hash your password with a strong hashing algorithim like SHA-256 or SHA-512 first to avoid storing it in plaintext.

21.3.3 systemd-boot

systemd-boot has the option to prevent editing the kernel parameters at boot. In the `loader.conf` file, add:

```
editor no
```

systemd-boot does not officially support password protecting the kernel parameters editor, but you can achieve this with systemd-boot-password.

21.4 Verified boot

Verified boot ensures the integrity of the boot chain and base system by cryptographically verifying them. This can be used to ensure that a physical attacker cannot modify the software on the device. Without verified boot, all of the precautions mentioned above could be bypassed with ease once physical access is gained. Contrary to common assumptions, verified boot is not just important for physical security — it prevents the persistence of **any** tampering with your system, be it from a physical attacker or a malicious application that has managed to hook itself into the operating system. For example, if a remote attacker has managed to exploit the system and gain high privileges, verified boot would revert their changes upon reboot and ensure that they cannot persist.

The most common verified boot implementation is UEFI Secure Boot. However, this by itself is not a complete implementation, as this only verifies the bootloader and kernel, meaning there are ways to bypass this:

- UEFI secure boot alone lacks an immutable root of trust, so a physical attacker can still reflash the firmware of the device. To mitigate this, use UEFI secure boot in combination with Intel Boot Guard or AMD Secure Boot.
- A remote attacker (or a physical attacker when not using encryption) can simply modify any other privileged part of the operating system. For example, if they have the privileges to modify the kernel, then they'd also be able to modify `/sbin/init` to achieve effectively the same result. Thus, only verifying the kernel and bootloader will not do anything against a remote attacker. To mitigate this, you must verify the base operating system with dm-verity, although this is very difficult and cumbersome due to the layout of traditional Linux distributions.

In general, it's hard to achieve a respectable verified boot implementation on traditional Linux.

21.5 USBs

USB devices present significant attack surface for physical attacks. Example of such attacks are BadUSB and Stuxnet. It is good practice to block all newly connected USBs and only whitelist trusted devices. USBGuard is great for this.

You could also use `nousb` as a kernel boot parameter to disable all USB support in the kernel. If using linux-hardened, you can set the `kernel.deny_new_usb=1` sysctl.

21.6 DMA attacks

Direct memory access (DMA) attacks involve gaining complete access to all of system memory by inserting certain physical devices. This can be mitigated via an IOMMU, which controls the areas of memory accessible to devices, or by blacklisting particularly vulnerable kernel modules.

To enable the IOMMU, set the following kernel boot parameters:

```
intel_iommu=on amd_iommu=on
```

You only need to enable the option for your specific CPU manufacturer, but there are no issues with enabling both options.

```
efi=disable_early_pci_dma
```

This option fixes a hole in the above IOMMU by disabling the busmaster bit on all PCI bridges during very early boot.

Furthermore, Thunderbolt and FireWire are often vulnerable to DMA attacks. To disable them, blacklist these kernel modules:

```
install firewire-core /bin/false
install thunderbolt /bin/false
```

21.7 Cold boot attacks

A cold boot attack occurs when an attacker analyses the data in RAM before it is erased. When using modern RAM, cold boot attacks aren't very practical, as RAM usually clears within a few seconds or minutes unless it has been placed inside a cooling solution, such as liquid nitrogen or a freezer. An attacker would have to rip out the RAM sticks from your device and expose it to liquid nitrogen all within a few seconds and without the user noticing.

If cold boot attacks are part of your threat model, then guard your computer for a few minutes after shutdown to ensure that nobody has access to your RAM sticks. You could also solder the RAM sticks into your motherboard to make it harder for them to be seized. If using a laptop, take out the battery and run directly off the charging cable. Pull out the cable after shutdown to ensure that the RAM has no access to more power to stay alive.

In the kernel self-protection boot parameters section, the zeroing of memory at free time option will overwrite sensitive data in memory with zeroes. Furthermore, the hardened memory allocator can purge sensitive data within user space heap memory via the CONFIG_ZERO_ON_FREE configuration option. Despite these though, some data may still remain in memory.

Additionally, modern kernels include a reset attack mitigation, which commands the firmware to erase data upon shutdown, although this requires firmware support.

Make sure that you shutdown your computer normally, so the mitigations explained above can kick in.

If none of the above are adequate for your threat model, you can implement Tails' memory erasure process, which erases the majority of memory (with the exception of video memory) and has been proven to be effective.

22. Best practices

Once you have hardened the system as much as you can, you should follow good privacy and security practices:

1. Disable or remove things you don't need to minimise attack surface.
2. Stay updated. Configure a cron job or init script to update your system daily.
3. Don't leak any information about you or your system, no matter how minor it may seem.
4. Follow general security and privacy advice.

Despite the hardening you have done, you must remember that Linux is still a fundamentally flawed operating system, and no amount of hardening can ever fix it fully.

Other guides

You should perform as much varied research as possible and not rely on a single source of information. One of the largest security problems is the user. These are links to other guides that I find valuable:

- Arch Linux Security wiki page
- Whonix Documentation
- NSA RHEL 5 Hardening Guide (slightly outdated but contains useful information nevertheless)
- KSPP recommended kernel settings
- kconfig-hardened-check

Glossary

Regenerate GRUB configuration

You may need to regenerate your GRUB configuration to apply certain changes you have made to the bootloader. The steps to do this can sometimes differ between

different distributions. For example, on distributions such as Arch Linux, you are expected to regenerate your configuration file by executing:

```
grub-mkconfig -o $path_to_grub_config
```

"$path_to_grub_config" depends on how you have setup your system. It is often either `/boot/grub/grub.cfg` or `/boot/EFI/grub/grub.cfg`, but you should make sure before executing this command.

Alternatively, on distributions like Debian or Ubuntu, you should execute:

```
update-grub
```

Capabilities

In the Linux kernel, "root privileges" are split up into various different capabilities. This is helpful in applying the principle of least privilege — instead of giving a process total root privileges, you can grant them only a specific subset instead. For example, if a program simply needs to set your system time, then it only needs `CAP_SYS_TIME` rather than total root. This could limit the potential damage that can be done; however, you must still be cautious with granting capabilities, as many of them can be abused to gain full root privileges anyway.

Go back