Examining OpenSSH Sandboxing and Privilege Separation – Attack Surface Analysis

Examining the effectiveness of OpenSSH's security mechanisms

By Yair Mizrahi, Senior Security Researcher | March 14, 2023

18 min read

The recent OpenSSH double-free vulnerability – CVE-2023-25136, created a lot of interest and confusion regarding OpenSSH's custom security mechanisms – Sandbox and Privilege Separation. Until now, both of these security mechanisms were somewhat unnoticed and only partially documented. The double-free vulnerability raised interest for those who were affected and those controlling servers that use OpenSSH.

This blog post provides an in-depth analysis of OpenSSH's attack surface and security measures.

How does OpenSSH implement Privilege Separation?

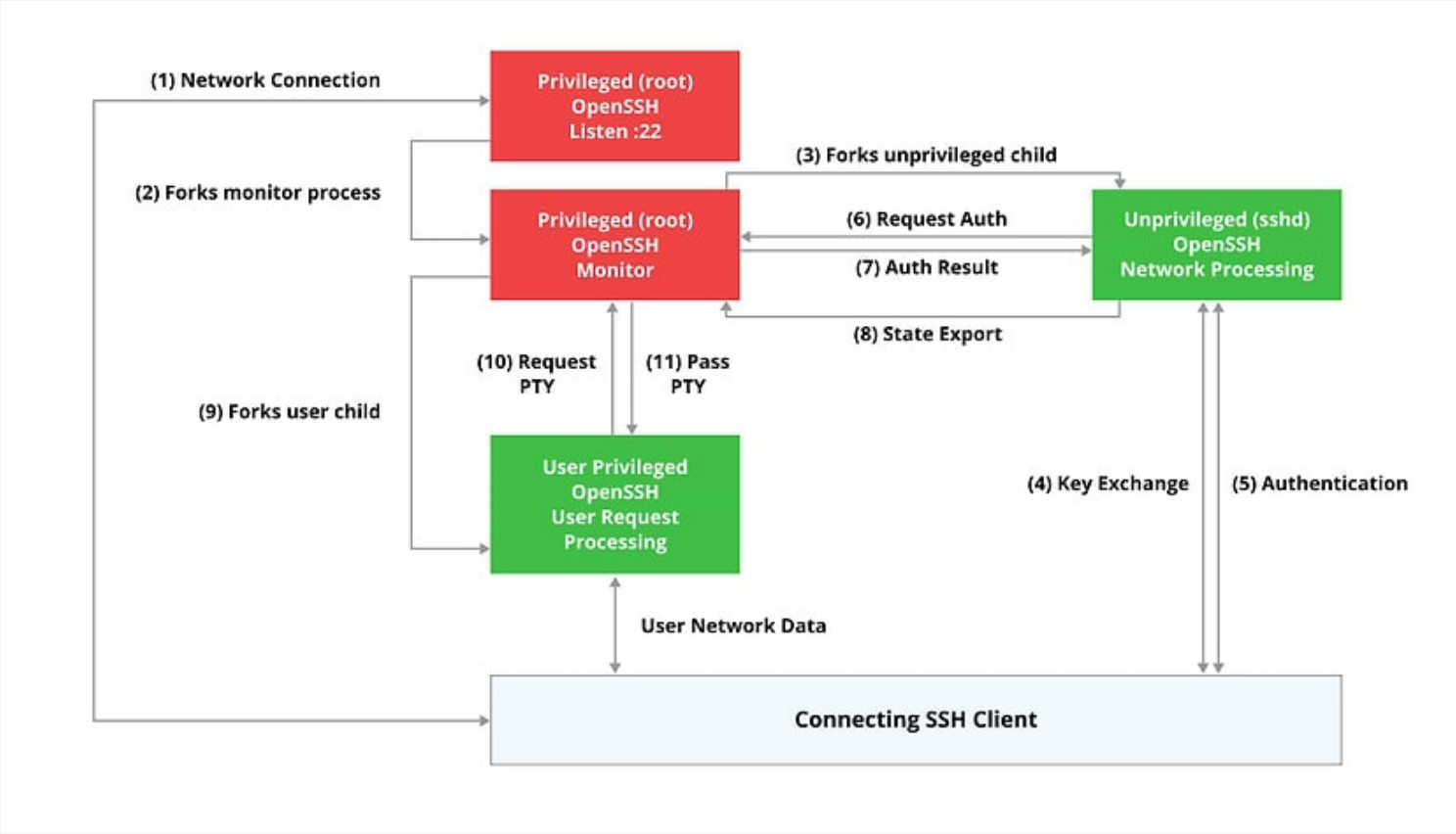
OpenSSH's privilege separation mechanism has been around since March 2002, implemented more than 20 years ago.

The feature is designed to enhance the security of SSH servers by limiting the privileges of the SSH server process and separating it from the user's authentication and session processes.

The goal of privilege separation is to make sure pre-authentication attacks cannot compromise the root account even though other parts of OpenSSH do run with root privileges.

Prior to the introduction of Privilege Separation, the OpenSSH server process had to run with elevated privileges to access system resources required for authentication and session management. This elevated privilege level made the server process a high-value target for attackers, who could potentially gain full control over a system by exploiting any vulnerability in the server process.

Any remote code execution vulnerability in the OpenSSH server process (sshd) could lead to an immediate remote root compromise if it happened before authentication, subsequently giving the attacker full control over the machine running OpenSSH.



Privilege Separation Mechanism

With Privilege Separation, the OpenSSH server process is split into two separate processes: one process that runs with elevated privileges to handle system-level tasks such as network I/O, and another process that runs with reduced privileges to handle user authentication.

When a user initiates an SSH connection to an OpenSSH server with Privilege Separation enabled, the server spawns two separate processes to handle the incoming connection.

The first process, known as the privileged process, runs with elevated privileges and is responsible for handling network I/O, such as listening for incoming connections, managing network sockets, and managing pseudo-terminals.

The second process, known as the unprivileged process, runs with reduced privileges and is responsible for handling user authentication. It is isolated from the privileged process and has limited access to system resources, such as file systems and network interfaces.

OpenSSH Privilege Separation – In-Depth Analysis

Privilege separation uses two processes: A privileged parent process monitors the progress of an unprivileged child process.

The child process is unprivileged. This is achieved by changing its uid/gid to an unused user (usually `sshd`) which has no login shell, and restricting its file system access via `chroot()` to `/var/empty`. ItThe child process is the only process that handles network data.

The parent process determines whether the child process performed the authentication successfully.

Communication between the privileged and the unprivileged process is achieved via pipes. Shared memory stores state that can not be otherwise exported and the child has to ask the privileged parent to determine whether authentication was successful.

If the child process gets corrupted and believes that the remote user has been authenticated, access will only be granted if the parent has reached the same decision.

During authentication, the child process communicates with the user and the authentication agent to obtain the necessary credentials for authentication. Once the child process has obtained the credentials, it sends them to the parent process for validation.

The parent process then performs the actual authentication by using the credentials provided by the child process to authenticate the user. If the authentication is successful, the parent process sends a message to the child process indicating that authentication has succeeded. If the authentication fails, the parent process sends a message to the child process indicating that authentication has failed.

The communication between the parent and child processes is done using Unix domain sockets, which are a form of inter-process communication (IPC) mechanism.

The parent and child processes each have their own Unix domain socket, and they use these sockets to communicate with each other.

By performing the authentication in the parent process, OpenSSH is able to ensure that sensitive authentication data never leaves the privileged process, which provides an additional layer of security. Additionally, by using IPC to communicate between the parent and child processes, OpenSSH is able to maintain separation between the two processes and prevent the child process from interfering with the critical operation performed by the parent process.

During the pre-authentication phase, `sshd` will `chroot()` to `/var/empty` and change its privileges to the `sshd` user and its primary group. `sshd` is a pseudo-account that is locked, is not used by other daemons, and does not contain a valid shell.

Given the following process listing:

UID	PID	PPID	C	STIME	TTY	TIME	COMMAND
root	957	9	0	09:14	?	00:00:00	/usr/sbin/sshd -D [listener] 0 of 10-100 startups
root	1015	957	0	09:14	?	00:00:00	sshd: [accepted]
sshd	1016	1015	0	09:14	?	00:00:00	sshd: [net]

- Process 957 is the sshd process listening for new connections.
- Process 1015 is the privileged monitor process.
- Process 1016 is the unprivileged `authenticator-handler` process.

The Privilege Separation mechanism is controlled by the `UsePrivilegeSeparation` configuration key. By default, the key is set to the most restrictive `sandbox` setting (even when the key is not specified) which means the pre-authentication unprivileged process is subject to additional restrictions, which we will cover in the next section. The default location for this configuration file is `/etc/sshd_config`.

A sample `sshd_config` configuration file:

```
# Connection
Port 22
Protocol 2
UseDNS no
Compression no

# Authentication:
PubkeyAuthentication yes
PermitEmptyPasswords no
UsePAM yes
ChallengeResponseAuthentication yes
LoginGraceTime 60
UsePrivilegeSeparation sandbox # The relevant Privilege Separation config key
…
```

What is the OpenSSH Sandbox?

The OpenSSH pre-authentication sandbox is a security mechanism first introduced in OpenSSH version 5.9 that aims to prevent attackers from fully compromising a system after exploiting vulnerabilities during the pre-authentication phase. It creates a restricted environment that limits the scope of potential vulnerabilities during the authentication phase of SSH connections.

It operates by launching an isolated environment using a combination of kernel security mechanisms, such as seccomp filtering and namespace isolation – essentially restricting its capabilities to only a few pre-approved system calls.

When a user initiates an SSH connection to an OpenSSH server with the sandbox feature enabled, the server spawns a new process that runs in a restricted environment, also known as the sandbox. The sandboxed process is created with limited privileges and restricted access to system resources, including file systems and network interfaces.

OpenSSH Sandbox – In-Depth Analysis

OpenSSH has 7(!) different sandbox styles that are determined by the platform you compile it for and its kernel capabilities.

All of the different sandbox styles are centered around the concept of system call restriction – meaning that the sandboxed process cannot use most of the system's services, like opening files, communicating over the network, etc.

Linux Sandbox

The OpenSSH configuration step (that runs just before the compilation step) checks for seccomp compatibility by checking whether the kernel is configured with the `SECCOMP_MODE_FILTER` option.

This is what it looks like when configuring:

```
checking whether SECCOMP_MODE_FILTER is declared... yes
checking kernel for seccomp_filter support... yes
```

Checking the Ubuntu 22.04 LTS sshd daemon binary for the sandbox type, we see that it uses a seccomp filter, so we'll focus on that:

```
> strings ./sshd | grep preparing
%s: preparing seccomp filter sandbox
```

Seccomp stands for secure computing mode and has been a feature of the Linux kernel since version 2.6.12, released in 2005. It is used to **filter and restrict the available system calls** to userland processes, thus reducing the kernel surface exposed which limits the attack surface for a privilege escalation.

This is done by having only the essential system calls needed for the application to function properly. The filter is expressed as a Berkeley Packet Filter (BPF), as with socket filters, except that the data operated on is related to the system call being made: system call number and the system call arguments.

We examine the `sandbox-seccomp-filter.c` file, and find the attachment of the Seccomp filter to the program in `ssh_sandbox_child()` function, the seccomp profile used by OpenSSH is –

```
    /* Syscall filtering set for preauth. */
static const struct sock_filter preauth_insns[] = {
.....
    /* Syscalls to non-fatally deny */
#ifdef __NR_lstat
    SC_DENY(__NR_lstat, EACCES),
#endif
#ifdef __NR_lstat64
    SC_DENY(__NR_lstat64, EACCES),
#endif
#ifdef __NR_fstat
    SC_DENY(__NR_fstat, EACCES),
#endif
.....

    /* Syscalls to permit */
#ifdef __NR_brk
    SC_ALLOW(__NR_brk),
#endif
#ifdef __NR_clock_gettime
    SC_ALLOW(__NR_clock_gettime),
#endif
#ifdef __NR_clock_gettime64
    SC_ALLOW(__NR_clock_gettime64),
#endif
#ifdef __NR_close
    SC_ALLOW(__NR_close),
#endif
#ifdef __NR_exit
    SC_ALLOW(__NR_exit),
#endif

#ifdef __NR_mmap
    SC_ALLOW_ARG_MASK(__NR_mmap, 2, PROT_READ|PROT_WRITE|PROT_NONE),
#endif
#ifdef __NR_mmap2
    SC_ALLOW_ARG_MASK(__NR_mmap2, 2, PROT_READ|PROT_WRITE|PROT_NONE),
#endif
#ifdef __NR_mprotect
    SC_ALLOW_ARG_MASK(__NR_mprotect, 2, PROT_READ|PROT_WRITE|PROT_NONE),
#endif

/* Default deny */
BPF_STMT(BPF_RET+BPF_K, SECCOMP_FILTER_FAIL),
```

It uses macros (SC_DENY/SC_ALLOW/SC_ALLOW_ARG_MASK) to create the BPF filters that are used as the Seccomp filter.

For example, we see that lstat() is explicitly denied to fail silently, close() is explicitly allowed, and mprotect() is allowed but must pass an argument mask that denies unwanted arguments.

We can also see that the default for non-detailed syscalls is SECCOMP_FILTER_FAIL:

```
/* Linux seccomp_filter sandbox */
#define SECCOMP_FILTER_FAIL SECCOMP_RET_KILL
```

SECCOMP_RET_KILL results in the process exiting immediately without executing the system call.
This was our result in the previous blog post when trying to trigger the vulnerability, the seccomp sandbox would fail and exit the process because writev() is not defined, and automatically leads to SECCOMP_RET_KILL.

Some of the major silently-denied syscalls [1]:

`open` – used to open a file and obtain a file descriptor that can be used to read from or write to the file. Removing this syscall heavily decreases the attack surface since attackers won't be able to open arbitrary files.

`openat` – similar to open, but works relative to a given directory.

Some of the major explicitly-allowed syscalls that check for arguments [2]:

`mmap` – used to map a region of memory into the calling process's address space. Denying certain arguments will prevent attackers from creating dangerous memory maps, and block some ROP shellcodes (see next section).

`mprotect` – used to modify the access permissions for a range of memory pages.

Some of the major explicitly-allowed syscalls without arguments checking [3]:

`close` – used to release a file descriptor previously obtained by opening a file using the `open` or `openat` system calls.

`madvise` – used to advise the kernel about the intended usage of a range of memory. Allows a program to communicate to the operating system how it plans to use a particular region of memory, which can help the kernel optimize its management of that memory.

`mremap` – used to change the size or location of an existing memory mapping.

`munmap` – used to remove a memory mapping that was previously established using the mmap syscall. When a program no longer needs to access a memory mapping, it should call the `munmap` syscall to release the associated memory and remove the mapping.

`write` – Used to write data from a buffer to a file descriptor.

We'll first dive into the restriction for mmap:

```
#ifdef __NR_mprotect
    SC_ALLOW_ARG_MASK(__NR_mprotect, 2, PROT_READ|PROT_WRITE|PROT_NONE),
#endif
```

```
/* Allow if syscall argument contains only values in mask */
#define SC_ALLOW_ARG_MASK(_nr, _arg_nr, _arg_mask) \
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, (_nr), 0, 8), \
    /* load, mask and test syscall argument, low word */ \
    BPF_STMT(BPF_LD+BPF_W+BPF_ABS, \
        offsetof(struct seccomp_data, args[(_arg_nr)]) + ARG_LO_OFFSET), \
    BPF_STMT(BPF_ALU+BPF_AND+BPF_K, ~((_arg_mask) & 0xFFFFFFFF)), \
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 0, 4), \
    /* load, mask and test syscall argument, high word */ \
    BPF_STMT(BPF_LD+BPF_W+BPF_ABS, \
        offsetof(struct seccomp_data, args[(_arg_nr)]) + ARG_HI_OFFSET), \
    BPF_STMT(BPF_ALU+BPF_AND+BPF_K, \
        ~(((uint32_t)((uint64_t)(_arg_mask) >> 32)) & 0xFFFFFFFF)), \
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 0, 1), \
    BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW), \
    /* reload syscall number; all rules expect it in accumulator */ \
    BPF_STMT(BPF_LD+BPF_W+BPF_ABS, \
        offsetof(struct seccomp_data, nr))
```

This verifies that the second argument to `mmap` is one of the following:

`PROT_READ|PROT_WRITE|PROT_NONE`

This prevents an attacker from creating an executable memory segment (`PROT_EXEC`) which makes it harder for an attacker to bypass the `DEP/NX` exploit mitigation.
Also, most shellcodes use the `open()` syscall to open new file descriptors, but the seccomp filter denies it, effectively minimizing a lot of local privilege escalation possibilities since an attacker would have to find already-open file descriptors in order to exploit a privilege escalation.

macOS Sandbox

`Seccomp` is a Linux Kernel feature, meaning it does not exist on machines running macOS

Let's inspect the `sandbox-darwin.c` file (Darwin is the core Unix system of macOS):

```
void
ssh_sandbox_child(struct ssh_sandbox *box)
{
    char *errmsg;
    struct rlimit rl_zero;


    debug3("%s: starting Darwin sandbox", __func__);
    if (sandbox_init(kSBXProfilePureComputation, SANDBOX_NAMED,
        &errmsg) == -1)
        fatal("%s: sandbox_init: %s", __func__, errmsg);


    /*
     * The kSBXProfilePureComputation still allows sockets, so
     * we must disable these using rlimit.
     */
    rl_zero.rlim_cur = rl_zero.rlim_max = 0;
    if (setrlimit(RLIMIT_FSIZE, &rl_zero) == -1)
        fatal("%s: setrlimit(RLIMIT_FSIZE, { 0, 0 }): %s",
            __func__, strerror(errno));
    if (setrlimit(RLIMIT_NOFILE, &rl_zero) == -1)
        fatal("%s: setrlimit(RLIMIT_NOFILE, { 0, 0 }): %s",
            __func__, strerror(errno));
    if (setrlimit(RLIMIT_NPROC, &rl_zero) == -1)
        fatal("%s: setrlimit(RLIMIT_NPROC, { 0, 0 }): %s",
            __func__, strerror(errno));
}
```

OS X has a feature called Seatbelt- its own sandbox kernel extension.

There are 5 documented profiles:

`kSBXProfileNoInternet` – TCP/IP networking is prohibited.

`kSBXProfileNoNetwork` – All sockets-based networking is prohibited.

`kSBXProfileNoWrite` – File system writes are prohibited.

`kSBXProfileNoWriteExceptTemporary` – File system writes are restricted to the temporary folder /var/tmp and the folder specified by the confstr(3) configuration variable _CS_DARWIN_USER_TEMP_DIR.

`kSBXProfilePureComputation` – All operating system services are prohibited.

`kSBXProfilePureComputation` is the most restrictive mode.

When an application is launched with this profile, it is limited to accessing only the following resources:

- The application's own code and resources
- System libraries and frameworks required for computation
- Shared memory
- Unix signals
- The network loopback interface

The application is prevented from accessing the file system, other network interfaces, user data, hardware peripherals, or any other resources that could potentially be used to modify the system or interfere with other applications.

We can see OpenSSH's sandbox uses this profile – essentially restricting all OS services and thus minimizing OpenSSH's attack surface.

OpenBSD Sandbox

The OpenBSD operating system also has sandbox styles that are specific to it and cannot be used on other platforms.

The first one is `systrace`, which monitors and controls an application's access to the system by enforcing access policies for system calls, much like a primitive `seccomp`.

It uses a pseudo-device, `/dev/systrace`, which allows userland processes to control the behavior of `systrace` through an `ioctl` interface.

It was deprecated in favor of `pledge` (originally named `tame`) that was released in 2015.

It has a concept named Promises, which are sets of permissions that a process can request in order to perform its operations.

A promise is a declaration made by a process to the system that it will only use a specific list of system calls, thus restricting its syscall access to a predefined set of operations.

Promises can be requested by a process using the pledge() system call, and each promise is identified by a string that represents a specific category of syscalls that the process is allowed to use.

Some examples of promises include `rpath` (which allows the process to read its own executable and linked shared libraries) and `inet` (which permits network communication).

`pledge` can't filter file system paths or internet addresses. For example, if you enable a category like `inet`, your process will be able to talk to any internet address.

We'll inspect the `sandbox-pledge.c` file:

```
void
ssh_sandbox_child(struct ssh_sandbox *box)
{
    if (pledge("stdio", NULL) == -1)
        fatal_f("pledge()");
}
```

We can see that OpenSSH uses the promise stdio.

This promise grants access to standard input/output, threads, and benign system calls.

Some of the major explicitly-allowed syscalls [4]:

`close` – used to release a file descriptor previously obtained by opening a file using the `open` or openat system calls.

`madvise` – used to advise the kernel about the intended usage of a range of memory. Allows a program to communicate to the operating system how it plans to use a particular region of memory, which can help the kernel optimize its management of that memory.

`mmap` – used to map a region of memory into the calling process's address space.

`mprotect` – used to modify the access permissions for a range of memory pages.

`munmap` -used to remove a memory mapping that was previously established using the mmap syscall. When a program no longer needs to access a memory mapping, it should call the munmap syscall to release the associated memory and remove the mapping.

`pipe` – used to create an inter-process communication channel, or "pipe", between two related processes.

`read` – used to read data from a file or input stream.

`recv` – used to receive data from a connected socket.

`send` – used to send data over a connected socket.

`write` – used to write data from a buffer to a file descriptor.

This filter, much like the seccomp one, is very restrictive and denies any `PROT_EXEC` mappings or invoking `open()`.

Conclusion – Don't mess with the defaults!

OpenSSH's security mechanisms, namely `Privilege Separation` and `Sandboxing`, provide a robust and effective solution for enhancing the security of the OpenSSH server. These mechanisms work together to minimize the attack surface and prevent privilege escalation attacks by isolating and restricting access to critical system resources.

The one point of failure of those security mechanisms is the user configuration.

OpenSSH will enable all the restrictions by default (on a supported system), but a user can also choose to partially enable the restriction mechanisms or disable them completely:

1. To only use the privilege separation (`UsePrivilegeSeparation=yes`) without sandboxing. This may allow network attackers to fully compromise a system with privilege escalation.
2. Or to disable both the sandbox and privilege separation (`UsePrivilegeSeparation=no`).

This leads to an insecure system. Once code execution is achieved in the pre-authentication phase, attackers may fully compromise the system.

By running parts of the SSH daemon in a separate, unprivileged process and by confining it to a sandboxed environment, OpenSSH can prevent attackers from exploiting vulnerabilities in the SSH server (like CVE-2023-25136 Double-Free) to gain privileged access to the system.

Following the research above, it is safe to say that at this time, organizations can deploy OpenSSH with confidence, knowing that the risk of code execution and privilege escalation attacks has been considerably mitigated. However, be sure to stay up-to-date with the latest version of OpenSSH and all of the latest security findings.

Stay up-to-date with JFrog Security Research

The security research team's findings and research play an important role in improving the JFrog Software Supply Chain Platform's software security capabilities. This manifests in the form of enhanced CVE metadata and remediation advice for developers, DevOps and security teams in the JFrog Xray vulnerability database. And also as new security scanning capabilities used by JFrog

Xray.

Follow the latest discoveries and technical updates from the JFrog Security Research team in our research website, security research blog posts and on Twitter at @JFrogSecurity.

Appendix A – OpenSSH sandbox full syscall lists

[1] Linux Sandbox – silently-denied syscalls: `lstat, lstat64, fstat, fstat64, fstatat64, open, openat, newfstatat, stat, stat64, shmget, shmat, shmdt, ipc, statx`

[2] Linux Sandbox – explicitly-allowed syscalls that check for arguments: `mmap, mmap2, mprotect, socketcall, ioctl (only on s390 architecture)`

[3] Linux Sandbox – explicitly-allowed syscalls without arguments checking: `brk, clock_gettime, clock_gettime64, close, exit, exit_group, futext, futext_time64, geteuid, geteuid32, getpgid, getpid, getrandom, gettid, gettimeofday, getuid, getuid32, madvise, mremap, munmap, nanosleep, clock_nanosleep, clock_nanosleep_time64, clock_gettime64, newselect, ppoll, ppoll_time64, poll, pselect6, pselect6_time64, read, rt_sigprocmask, select, shutdown, sigprocmask, time, write`

[4] OpenBSD Sandbox – explicitly-allowed syscalls: `exit_group, close, dup, dup2, dup3, fchdir, fstat, fsync, fdatasync, ftruncate, getdents, getegid, getrandom, geteuid, getgid, getgroups, getitimer, getpgid, getpgrp, getpid, getppid, getresgid, getresuid, getrlimit, getsid, wait4, gettimeofday, getuid, lseek, madvise, brk, arch_prctl, uname, set_tid_address, clock_getres, clock_gettime, clock_nanosleep, mmap (PROT_EXEC and weird flags aren't allowed), mprotect (PROT_EXEC isn't allowed), msync, munmap, nanosleep, pipe, pipe2, read, readv, pread, recv, poll, recvfrom, preadv, write, writev, pwrite, pwritev, select, send, sendto (only if addr is null), setitimer, shutdown, sigaction (but SIGSYS is forbidden), sigaltstack, sigprocmask, sigreturn, sigsuspend, umask, socketpair, ioctl(FIONREAD), ioctl(FIONBIO), ioctl(FIOCLEX), ioctl(FIONCLEX), fcntl(F_GETFD), fcntl(F_SETFD), fcntl(F_GETFL), fcntl(F_SETFL)`

Tags: openssh | security-research |

Cookies Policy

Cookies Settings

Accessibility Mode