The missing guide to the security of filesystems and file APIs

Gergely Kalman

v1, 2024

Intro

These are the technical slides that I always have to cut from my presentations. I try to sprinkle them in, but it’s just always too much. So I decided that it’s big enough to be it’s own thing:

The missing guide to the security of filesystems and file APIs.

(a braindump of everything I know)

I will publish this on https://gergelykalman.com as well, with any potential revisions/additions based on your feedback.

I hope you find it useful.

Gergely Kalman

A quick riddle

- on an **HFS+** volume on **macOS**
- in a directory called **ours** owned by the **attacker** user
- we can trigger a **file creation**
  - by a system daemon running as **root**
  - **ours**/**secret** can be created as **root:wheel**, perms **“rwx------”**
    - a **POSIX “read” extended ACL** will be created for **attacker**
    - and an **extended attribute** called “**com.apple.quarantine**” will be placed by the system
    - content will be written to the file by the daemon
- **Question**: can **attacker** read the contents of “**secret**”?

If you think **“How the *@!# should I know?”**

You are not alone

The question can’t be answered.

Why not?

- how is the mount situation?
  - we don’t know how **HFS+** is mounted
    - is **noowners** on?
    - can we turn it on?
    - do we have access to the backing image?
  - is there anything mounted on top of **ours**?
- is the **secret** “file” a **regular file**, or we just mean “file” in the general sense?
- what about **ACL**s?
  - is the **ACL** an **allow** or **deny**?
  - are there any other **ACL**s on the file?
- what is the **value** of the quarantine **extended attribute**?

Why not?

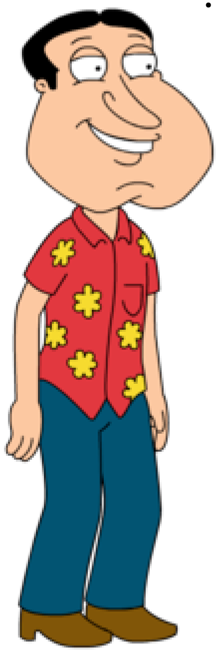
- **I’m not done...**
- how is **secret** created?
  - do we control the path or is it fixed?
  - would **open()** follow symlinks?
    - is it **open()** that gets called at all!?
  - would umask be honored?
  - who sets the permissions (is there a **chmod()** call)?
  - is there a race between the file creation and
    - application of the **ACL**?
    - application of the **extended attribute** (quarantine flag)?
      - who places this anyway?

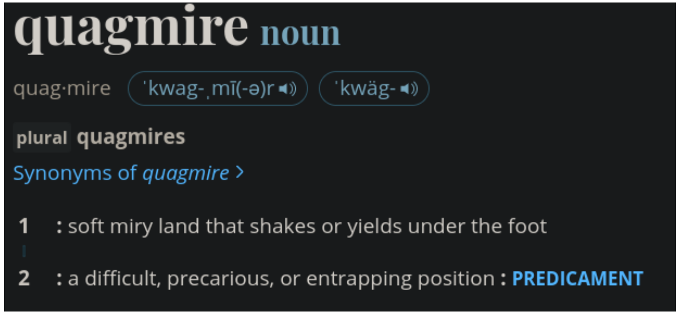
Why not?

- **STILL not done...**
- is the **write()** done securely?
  - meaning it **write()**s to the file that it just opened
    - or is this a **creat()/open()** race
- can **attacker** use **sudo**?
  - cheeky, I know
- is there a **SIP** rule on **macOS** that prevents any of this for **attacker?**

Well...

- **this is a Quagmire**
  - **easy in theory** but **shockingly difficult in practice**
  - not just on **macOS** either: variants of these exist on **Linux** as well
    - Windows is different, but it has similar issues
      - I’m not a Windows guy so I won’t speak on it
      - but I suspect most of the concepts translate





Well...

- **file ops are extremely difficult to get right**
  - and this is a **HUGE problem**
    - if **we** (security researchers) can’t reason about them then how can regular developers?

Let’s learn some basics

- **show of hands**
- who knows about:
  - POSIX standard file permissions (rwxrwxrwx)?
  - POSIX file APIs (open, read, chmod, unlink, mkdir, rename, ...)?
  - Filesystem object types (file / dir / symlink / hardlink)?

POSIX standard file permissions

- POSIX standard file permissions (rwxrwxrwx)?
- **Everyone should be familiar with this**
- To note:
  - suid, sgid, sticky bit
    - sgid for example inherits dir ownership on mkdir on Linux
      - on BSD this is what happens by default (without sgid)
        - **FML**

POSIX file APIs

- **POSIX** file APIs (open, read, chmod, unlink, mkdir, rename, ...)?
  - most of you should know at least a few of these syscalls
  - defined in **IEEE Std 1003.1-2024**
    - https://pubs.opengroup.org/onlinepubs/9799919799/
  - despite the massive standard, **OSes still had to augment it**:
    - for example: **renameat2**() on Linux, **renameatx_np**() on macOS
      - new features:
        - prevents symlinks everywhere in the path
        - swap file inodes atomically
    - sometimes regular **POSIX**-standard syscalls can **take extra, non-POSIX flags**, like **O_DIRECT** on Linux

POSIX file APIs

- some APIs fell hilariously short
  - just a few examples:
    - **rename(src, dst) → no way to prevent symlinks** from being followed
    - **open()**’s **O_NOFOLLOW** prevents resolving **only the last path component**
  - bad enough that OSes rolled their own versions
    - sometimes these made it back into **POSIX**, sometimes they didn’t
    - if you want portability you miss out on these (mostly security) features

Advanced filesystem stuff

- who knows about:
  - ~~POSIX standard file permissions (rwxrwxrwx)?~~
  - ~~POSIX file APIs (open, read, chmod, unlink, mkdir, rename, ...)?~~
  - POSIX extended ACLs?
  - Filesystem object types (file / dir / symlink / **hardlink**)?
  - Filesystem internals?
  - POSIX pitfalls?
  - Filesystem extended attributes?

POSIX extended ACLs

- not a lot of people know that this is even a thing (I didn’t)
- **IEEE 1003.1e draft 17**
- A revoked (abandoned) **POSIX** standard
- Got implemented anyway
  - different implementations (**Linux ACL** != **BSD/macOS ACL**)

    → useless for portability
  - great for security researchers
- **Creates edge cases that no program/library expects**
  - especially portable ones

POSIX extended ACLs

- for example on **macOS** I can use:
  - **file_inherit → Inherits the directory's ACL to files created in them**
  - **root** creates a file with "**rwx------**" perms in a directory I control
    - **without ACL**s:
      - best I can do is remove the file and recreate it
        - but this often doesn't help
    - **with ACL**s:
      - I can give myself **any permission** on the file
        - that also stays on the file if it moves

POSIX extended ACLs

- **extended ACLs are very backdoor-like**
  - **they're "hidden"**
    - invisible unless you look for it
      - traditional **POSIX** calls like **stat()** won't show them
    - most hackers and most programmers don't even know they exist
  - **they tamper with important security functionality**
    - differently on each OS
  - **they are available to unprivileged users**

Filesystem object types

- You **definitely** have to know these
  - file (reg), directory, symlink, fifo, blockdev, chardev, socket
  - of course OS-es sometimes have others:
    - **whiteout** on macOS
    - **door** on Solaris
  - Notice how **hardlink** is not here...
    - because it's not a "file type"
    - **it's an organizational quirk**

Hardlinks

- **the same file under two different names**
  - **only within a single filesystem**
    - can’t cross filesystems like symlinks can
  - **two names → one inode**
    - not a clone, literally the **same** thing
    - **one object from two separate viewpoints**
  - **lots of stuff can be hardlinked**
    - symlink, socket, etc...
    - **but not a directory**
      - well, at least not officially

Hardlinks

- **directory hardlinks**
  - these are **everywhere**, but not like you think:
    - **“.”** is a hardlink to self
    - **“..”** is a hardlink to parent
    - **./a/b → b** is a hardlink in dir **a** to the inode of b
  - “actual” hardlinks between ./x/a/ and ./x/b/ **are strictly forbidden**
    - in theory, we’ll talk about it later...

Three layers of attack surface

- **API layer**
  - bugs in userspace applications
  - example: **open()** done insecurely
- **VFS layer**
  - these bugs are in the kernel
  - example: **VFS** removes a **directory** even though **unlink()** was called
- **FS layer**
  - user or kernelspace - depending on where the **FS driver** runs from
  - example: **FAT32** driver can be raced to return an error unnecessarily

POSIX file APIs

- some APIs fell hilariously short
  - just a few examples:
    - **rename(src, dst) → no way to prevent symlinks** from being followed
    - **open()**’s **O_NOFOLLOW** prevents resolving **only** the last path component
  - bad enough that OSes rolled their own versions
    - sometimes these made it back into **POSIX**, sometimes they didn’t
    - if you want portability you miss out on these (mostly security) features

POSIX compatiblity

- **in case you were wondering:**
  - Linux is not fully **POSIX**-compatible
  - neither is FreeBSD
  - and **definitely not macOS**
    - since the **VFS** comes from **FreeBSD…**
- **they are very close though**
- so when I say **POSIX:**
  - think: everything except Windows
    - I know, **WSL**, I don’t have time

VFS

- **VFS** - Virtual Filesystem Switch
  - open() **syscall** → **VFS** open → **FS** open
    - **VFS** translates between the user and the underlying **FS** driver
    - great idea, but abstractions are always leaky
  - **VFS** abstracts a **HUGE** attack surface – easy to forget
    - **every mountable filesystem driver is exposed via the VFS**
  - **VFS** also takes care of some things itself
    - caching
    - **lots of global filesystem magic**
      - union mounts, resource forks, AppleDouble handling, firmlinks, etc...

VFS attack surface

- **VFS** has to “translate” things
  - not all filesystems support everything
  - sometimes **FS** drivers are just plain stupid
  - sometimes they just don’t support things that are “required”
- for example:
  - **macOS** purges **AppleDouble** files from an otherwise empty directory on **rmdir()** when it would fail with **ENOTEMPTY**
    - this is done **everywhere**, in **VFS**, even if the volume **does** support xattrs and has no use for **AppleDouble**
      - yes, horrific. Thank you

FS driver attack surface

- **FS** code is often old/dumb/bad
- **FS** code is sometimes modified to support weird shit, usually for compatibility
  - for example: **on macOS there are symlinks on FAT32 volumes**
    - they are “emulated” using regular files with magic sizes and content
      - **yeah :|**
  - every OS has tons of compatibility code like this
    - that is rarely exercised or tested...

FS driver attack surface

- **FS** drivers are particularly vulnerable to malicious images
  - since they are in large part just elaborate file format parsers
  - so you can **create impossible, forbidden structures**
    - hexedit / custom drivers / userspace drivers
    - create **hardlinked directories**
    - create an **infinite directory loop**
    - create **files with 2 hardlinks but linkcount of 1**
    - **endless possibilities...**
  - traditionally users can’t mount disk images for exactly this reason
    - except on **macOS**
    - and some **Linux** distros

Path resolution

- the process by which a user-supplied name can be turned into the kernel representation of an inode
- **two types of paths**
  - **absolute** “/etc/passwd”
  - **relative** “./hello.txt”
    - this depends on the **CWD** (Current Working Directory)
- this is in-band signaling: “does the file start with /”?

Path resolution

- path resolution is really unintuitive sometimes...
- **since the filesystem is a hallucination**
    - you **always** see a snapshot of the filesystem structure
    - which might be out of date by the time the kernel returns
    - **which is interesting, but is it important?**

Path resolution

- **oh yes!**
- **consider this:**

```
$ echo hi > secret.txt
$ mkdir -p a/b/c/d/e/f/g/h/i/j/k/l/
$ cat a/b/c/d/e/f/g/h/i/j/k/l/../secret.txt
cat: a/b/c/d/e/f/g/h/i/j/k/l/../secret.txt: No such file or directory
```

  - this obviously failed...
  - **but what if I move “l” at just the right time?**

Path resolution

- process 1 loop:

  $ cat a/b/c/d/e/f/g/h/i/j/k/l/../secret.txt
- process 2:
  - $ mkdir ./x
  - $ **switchdirs** ./x ./a/b/c/d/e/f/g/h/i/j/k/l
    - **switchdirs** implements atomic rename swap in a loop

Path resolution

- after a while the race is won
  - between the lookup of **l** and the lookup of “**..**” (in **l**) - **l will have moved**
    - if this happens, “..” no longer points to **k** but to the (old) parent of **x**
    - and here, there **is** a file called **secret.txt**
- this race could be optimized a lot more, but you get my point
  - **you can’t trust anything once someone else has access to it**

POSIX pitfalls

- The **POSIX** filesystem API was **never meant to handle concurrent access**
  - any concurrent access across privilege boundaries is **disastrous**
- **POSIX** had some bad API choices:
  - **open()**’s **O_NOFOLLOW** prevents resolving **only the last path component**
    - fun fact: this was not even part of **POSIX** until **POSIX.1-2008**
  - **open()** originally had no **O_CLOEXEC** → only since **POSIX.1-2008**
    - if you executed any other program it got access to all your currently opened **fd**s
  - **rename()** always follows symlinks (well, it’s complicated)
  - there are many others

POSIX pitfalls

- **access()**/**open()** race
  - the most classic **TOCTOU** (Time of Check Time of Use)
  - proven to be impossible to secure
- **symlinks**
  - a great feature
  - but **has to be explicitly handled by every program**
- **in-band signaling**
  - special meaning of “/” at the start of a path signals absolute path
    - **this becomes an issue if you can have the path truncated**
      - which is a super common bug that no-one cares about

POSIX pitfalls

- **no copy() system call**
  - so every program has to implement their own file copy routines
  - and they usually do it badly
- **no recursive unlink() or rmdir() either**
  - good luck hand-rolling these
    - this is impossible to do correctly, for a multitude of reasons
- **too barebones**
  - every program has to implement tons of boilerplate
    - so libraries usually provide this

Well-known pitfalls

- **symlinks are nasty**
- **tempfiles are a nightmare**
- **file descriptor names are hardcoded (stderr closing trick)**
  - close stderr before running the victim program
  - victim opens a file for writing
    - will be at **fd** #2, since that's the lowest available **fd**
  - victim writes an error message to the file it just opened since stderr == fd #2
- only useful with programs that start at a higher privilege than you
  - suids (kernel mitigates these)
  - entitled binaries on **macOS**
    - Oops...

Filesystem extended attributes

- Most filesystems support “extra” stuff
  - extended attributes
  - special mount flags
- example:
  - **ext2/3/4**:
    - append-only/immutable/undeletable files that **override ALL permission checks**
  - **HFS+**:
    - attributes, **resource forks**, compression, etc…

Resource fork rant

- **macOS resource forks are insane:**

  ```
  $ rm a; echo hi>a; echo wat>a/..namedfork/rsrc; cat a/..namedfork/rsrc
  wat
  ```

  - let’s add this insanity into the **path lookup**
    - **WHY NOT!?**
    - who needs consistency anyway?

Resource fork rant

- if the meaning of special markers (“..” and “/”) is not consistent, multiple interpretations will exist (duh)
- what does this look like: **“./a/..namedfork/rsrc”**?
  - **everyone:**
    - **rsrc** in the “**..namedfork**” directory of directory “**a**”
  - **macOS:**
    - the resource fork named “**rsrc**” of file “**a**”

mount pitfalls

- **mountpoints can move**
  - if you can **rename()** their parents
- **the same disk can be mounted multiple times** (not on **macOS**)
- **bind mounts**
  - the same **FS** is in two different locations at the same time
    - can overlap for added hilarity
- **union (macOS) / overlay (Linux) mounts**
  - lookups traverse to the **FS** under the current one if a file is not found

macOS-specific pitfalls

- I have done a **lot** of **macOS/iOS** research recently
  - these most likely won't translate to **Linux**
  - **but I included them to give you some ideas**

macOS-specific pitfalls

- **mkdir(path)** creates a directory through a dangling link if **path** ends in “/”
  - a completely undocumented quirk of **macOS**
- **/.vol/ supports accessing files by fsid + inodenum:**

```
$ stat /etc/passwd

16777225 40077649 -rw-r--r-- 1 root wheel 0 8542 "Aug 12 13:45:20 2024" "May  7 09:01:44 2024" "May 14 12:02:37 2024" "May
 7 09:01:44 2024" 4096 8 0x20 /etc/passwd

$ stat /.vol/16777225/40077649

16777225 40077649 -rw-r--r-- 1 root wheel 0 8542 "Aug 12 13:45:20 2024" "May  7 09:01:44 2024" "May 14 12:02:37 2024" "May
 7 09:01:44 2024" 4096 8 0x20 /.vol/16777225/40077649
```

  - not a security issue, but really convenient for exploitation
    - inodenum is monotonically increasing
- **/.file is similar to /.vol**
  - I think, help me out here

macOS-specific pitfalls

- **unprivileged users can mount any image they want**
  - no comment
- **macOS relies on extended attributes (xattrs) for security**
  - you can just mount a filesystem that doesn’t support them...
- **filesystem is case-insensitive by default** (**macOS** only, **iOS** is not)
  - good edge cases like: **rename(“./a” “./A”)**
  - random filenames are considerably less random...
- **union mounts are available**
  - specially handled by the **VFS** everywhere

macOS-specific pitfalls

- **firmlinks**
  - **Apple’s magical bind-mounts**
    - also specially handled by the **VFS** everywhere
    - doesn’t physically exist on disk
- **hardlinked directories**
  - these are **permitted(!)** on some filesystems
    - like **HFS+**
    - creating them from the host OS is pretty restricted though
      - they no longer seem to work on the latest version
    - **but you can always just create them on Linux or with a hex editor**

macOS-specific pitfalls

- **has AppleSingle/AppleDouble files**
    - only **AppleDouble** matters for us (**AppleSingle** is legacy)
    - if a **FS** doesn’t support xattrs **macOS will emulate them**
        - by creating another file of the same name and prefix **“._”**
        - and storing the xattr value there
        - **a nightmare of a “solution”**
    - the **VFS** is responsible for this
        - anything you do on the lower levels **can** clash with it

macOS-specific pitfalls

- **kernel crash time!**

```
$ mkdir mnt
$ touch mnt/._a
$ hdiutil create -size 128m -fs MS-DOS disk.dmg  # create disk
$ hdiutil attach disk.dmg -owners off -nomount   # mount disk
$ mount_msdos -o union /dev/disk4s1 mnt          # remount as union
$ touch mnt/a
```

  - **this used to panic the kernel :)**
    - it got fixed recently (after two years)
  - **source: https://github.com/gergelykalman/macos-crasher**

Learn more

- **You can get more information about all of this by using**
  - **man pages**
    - “man ls” is a good place to start
  - **standards**
    - good to find interesting things
    - not authoritative enough
      - standard is broken surprisingly often
  - **kernel source code**
    - best source of information
    - not as intimidating as you think

Thank You

- **Please reach out if you have questions:**
  - **https://gergelykalman.com**
  - **gergely [AT] gergelykalman.com**
  - **@gergely_kalman** on Twitter (X)

- **Please tell me what you think about this!**
  - **any suggestions / corrections?**