OS レベル セキュリティの基本

田浦

Contents

序章

(サイバー) セキュリティの基本 3 要件

- **秘匿性 (Confidentiality)** ≈ 情報 (データ) を勝手に読まれない
- **完全性 (Integrity)** ≈ 情報 (データ) を勝手に変更されない, 壊されない (書かれない)
- **可用性 (Availability)** ≈ 必要なときに使える

侵害される要因は様々…

- ユーザの誤操作 (宛先間違い)
- 管理者誤設定
- パスワードクラッキング
- フィッシング (偽サイトへの誘導)
- アプリケーション脆弱性
- OS 脆弱性
- …

セキュリティと OS

- OS がデータへのアクセスをどう制御**しているか・いないか**を知ることはセキュリティの基本
  - OS だけでアプリケーションを安全にできるわけではない
  - OS を知るだけでアプリケーションを安全に作れるようになるわけではない
- セキュリティ保証の方法としては暗号がすぐに思い浮かぶ
  - 暗号は多くの場合, **システムを信用せずに**秘匿性や完全性を保証する手段になる
  - 一方どんな暗号もコンピュータ上で実現されている限り, **鍵の書かれたファイル, 暗号化・復号プロセス, などの保護が前提**であり, それには OS の理解が必須
  - 暗号化しなくても得られる保護の理解も実践的には欠かせない

本講義の範囲

- セキュリティ全般を扱うことは無理 (別講義があります)
- 本講義の範囲: **1 台の計算機 (1 OS 管理下)** に複数のユーザがいるとき, データの (秘匿性, 完全性) がどう保護されるか?
  - 焦点: **ファイルのアクセス制御, プロセスが持つ権限**に関する理解
  - ネットワーク (e.g., web) アプリケーションのセキュリティを理解する基礎
  - ネットワークアプリケーションでは中心となる技術 (暗号, 認証, etc.) は扱わない

(復習) OS がアプリケーションを保護する基本枠組み

- プロセスの**アドレス空間 (≈ メモリ上のデータ)** をお互いに分離
  - プロセスがシステムコール無しにアクセスできるのはプロセス**内**のデータのみ(*)
  - ⇒ システムコール経由の**ファイルへのアクセス(*)を制御**することが基本
- (*) 「ファイル」以外にも制御すべきもの (ネットワークなど) があり個別に議論が必要だがまずはファイルに焦点

ファイルに対するアクセス制御

ファイルに対するアクセス制御の基本

- アクセス制御 ≈ open, exec 系など, ファイルをアクセスするシステムコールの成否
- 成否を決める 2 要素

1. 「**誰**」がファイルをアクセスしているか
   - → open を実行したプロセスの
     - **実効ユーザ ID (Effective User ID, EUID)**
     - **実効グループ ID (Effective Group ID, EGID)**
     - **補助グループ (supplementary groups)**
2. そのファイルは「**誰に対してどのようなアクセスを許可**」しているか
   - → ファイルの**アクセス許可 (Access Permission) 属性**

Unix における「ユーザ」の実体

- OS (カーネル) が認識している (≈ システムコールで使われる) 「ユーザ」の正体 — ユーザ ID — は単なる番号
  - `uid_t` 型 (実際は整数型)
  - 実はカーネルはシステムにユーザが何人いるかも「知らない」
- ログイン時などに使われる見慣れたユーザ名は, ユーザ番号との対応を OS 外の仕組みで決めた物に過ぎない
  - 対応を決める仕組み: `/etc/passwd`, `LDAP`, `NIS` 等
  - 単独で動作している環境では `/etc/passwd`, ネットワーク環境では `LDAP` が普通

Unix における「グループ」の実体

- グループ ≈ ユーザの集合
- ユーザ同様, グループの正体 — グループ ID — も単なる番号
  - `gid_t` 型 (実際は整数型)
- グループ名 (文字列) も, グループ番号との対応を OS 外の仕組みで決めた物に過ぎない
  - 対応を決める仕組み: `/etc/group`, `LDAP`, `NIS` 等
  - 単独で動作している環境では `/etc/group`, ネットワーク環境では `LDAP` が普通

特権ユーザ, スーパーユーザ, ルート

- ユーザ番号 0 のユーザはシステムコールレベルで特別扱い
  - **特権ユーザ (privileged user), スーパーユーザ (superuser), ルート (root)** などと呼ばれる
  - いわゆるシステム管理者でありほとんどのシステムコールが許可される
- 「特権ユーザ」と (CPU の)「特権モード」を混同しないように
  - root が実行しているプロセスだからといって, 普段から特権モードで動いているわけではない

アクセス許可関連の属性 基本編

ファイルにつくアクセス許可関連の属性

1. **ユーザ (User)** ≈ そのファイルの「所有者」のようなイメージ
2. **グループ (Group)** ≈ 上記のグループ版
3. **アクセス許可 (なぜか** mode **と呼ばれる)**
   - 「誰」に対する
   - 「どのような」アクセス
   - を許可{する・しない}

- **「ユーザ」「グループ」**属性に関する言葉の注
  - 「ユーザ」属性はそのファイルの「所有者」というイメージがわかりやすい
  - 一方「グループ」属性はそのファイルを「所有するグループ」というイメージは破綻しておりとむしろ混乱する
    – そのグループの属するユーザが皆, 「所有している」というわけではない
  - 実際は両者とも単に, その他大勢のユーザと違う (通常, より緩い) アクセス権限を与える対象を指定しているだけ

mode — 「誰」に対する「どのような」アクセスの許可 — の実体

- 「誰」は以下の 3 区分
  1. **User** (ファイルの「ユーザ」属性で指定されたユーザ)
  2. **Group** (ファイルの「グループ」属性で指定されたグループ)
  3. **Other** (それ以外のユーザ)
- 「どのような」アクセスかは以下の 3 区分
  1. **読み出し (Read)**
  2. **書き込み (Write)**
  3. **実行 (eXecute)**
- mode の正体 : 合計 $3 \times 3 = 9$ 個の 2 値 (許可 or 不許可) の属性

アクセス許可関連属性を見る — (お馴染みの) ls

```
$ ls -l
-rw-r--r-- 1 tau taulab   823 Dec 16 12:55  foo.txt
-rwxr-xr-x 1 tau taulab   199 Dec 16 12:55 watch.sh
```

アクセス許可関連属性を見るシステムコール — stat

```
struct stat sb;
int err = stat(path, &sb);
```

- *path* (ファイルやディレクトリ名) の属性を *sb* に格納
- *sb*.`st_uid` : ユーザ
- *sb*.`st_gid` : グループ
- *sb*.`st_mode` : 前述の 9 通りのアクセス許可 (9 bit) を含む整数

st_mode 中の 9 つのアクセス許可 bit

上位 bit から順に

数値 (8 進数)	「誰」による	「どんな」アクセス	C 言語での記号
0400	ユーザ	読み出し	S_IRUSR
0200	ユーザ	書き込み	S_IWUSR
0100	ユーザ	実行	S_IXUSR
0040	グループ	読み出し	S_IRGRP
0020	グループ	書き込み	S_IWGRP
0010	グループ	実行	S_IXGRP
0004	その他	読み出し	S_IROTH
0002	その他	書き込み	S_IWOTH
0001	その他	実行	S_IXOTH

- st_mode の当該 bit が 1 ⇔ その操作が許可されている
  - 例: (st_mode & S_IROTH) ≠ 0 ⇔ その他のユーザによる読み出しが許可されている
- ls -l の出力 (-rwxr-xr-x etc.) も上記の通り並んでいる

stat 中のその他の属性

- stat 構造体にはその他にも多くのファイル属性が格納されている
  - *sb*.st_size : サイズ
  - *sb*.st_mtime : 最後の更新時刻
  - etc.
- st_mode にも前述の 9 つ以外の情報が格納されている (後述)
  - set-user-ID
  - set-group-ID
  - sticky
- つまり, stat ≈ ファイルに関する情報のうち中身以外の情報

stat コマンド

```
$ stat 10_security.typ
  File: 10_security.typ
  Size: 13668       Blocks: 32         IO Block: 4096   regular file
Device: 252,0 Inode: 178538093   Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/     tau)   Gid: ( 1000/     taulab)
Access: 2024-09-29 02:55:24.449105666 +0900
Modify: 2024-09-28 12:58:37.151968205 +0900
Change: 2024-09-29 02:55:24.449105666 +0900
 Birth: 2024-09-29 02:55:24.449105666 +0900
```

open (読み出し) の成否

- プロセスPによるファイルFの**読み出し** open (`O_RDONLY` や `O_RDWR`) の成否

0. Pの実効 UID = 0
   $\Rightarrow$ OK
1. Pの実効 UID = Fのユーザ
   $\Rightarrow$ (F`.st_mode & S_IRUSR`) $\neq 0$ なら OK
2. Pの実効 GID または Pの補助グループ のどれか = Fのグループ
   $\Rightarrow$ (F`.st_mode & S_IRGRP`) $\neq 0$ なら OK
3. 上記に該当しない
   $\Rightarrow$ (F`.st_mode & S_IROTH`) $\neq 0$ なら OK

- 実効 UID/GID, 補助グループがどう決まるかは後述
  - さしあたり「実効 UID $\approx$ プロセスを起動したユーザ」と思っておけば良い

open (書き込み) の成否

- 読み出しのケースとほぼ同様
- 前スライドの `S_IR`*xxx* を `S_IW`*xxx* にするだけ

exec 系システムコールの成否

- open とほぼ同様 (前ページの `S_IR`*xxx* を `S_IX`*xxx* にするだけ)
- シェルスクリプトを実行する前に

```
chmod +x script.sh
```

としなくてはいけないのはこれが理由 (chmod コマンドについては後述)

- gcc などコンパイラが出力する実行可能ファイルの場合はコンパイラが上記相当を行っている

ディレクトリに対する R, W, X の意味

- R : 読み出し ≈ そのディレクトリに対する open システムコールの成否に影響
  - 実際問題としてはそのディレクトリのファイル列挙
- W : 書き込み ≈ そのディレクトリ内のファイルに対する作成 (open)・削除 (unlink)・名前変更の成否に影響
- X : **移動・通過権限**
  - そのディレクトリへの移動 (chdir システムコール) の成否に影響
  - そのディレクトリを含むパスを使ったシステムコールの成否に影響

アクセス許可を変更するシステムコール — chmod

```
int err = chmod(path, mode);
```

- *path* で指定されるディレクトリのアクセス許可 (st_mode) を *mode* に設定
- *mode* の指定方法
  1. 直接数値 (8 進数 3 桁が便利) で指定
  2. S_I{R,W,X}{USR,GRP,OTH} を ビット和 | で結んで指定

chmod コマンド

`chmod` *mode path*

- システムコールと引数の順番が逆, 動作はほぼ同じ
- *mode* の指定方法
  - 数値 (8 進数) で直接指定
  - 記号で指定 *who*`{-,+,=}`*what*
    - *who* は `u`, `g`, `o` の任意個を指定 (空は `ugo` (全て) と同義)
    - *what* は `r`, `w`, `x` の任意個を指定 (空は文字通り空)
    - `+`は権限追加, `-` は削除, `=`はセット
- 例:
  - `chmod 644` *path* (`rw-r--r--` にする)
  - `chmod g+w` *path* (グループの読み出し権限を追加)
  - `chmod  +x` *path* (全てに実行権限を追加)

chmod の成否

- プロセスPによるファイルFの chmod の成否 (≈ 管理者または持ち主が変更可)

0. Pの実効 UID が 0
   ⇒ OK
1. Pの実効 UID = Fのユーザ
   ⇒ OK
2. 上記に該当せず
   ⇒ NG

ファイルの{ユーザ, グループ}属性を変更するシステムコール — chown

- システムコール

```
int err = chown(path, user, group);
```

- 同名のコマンド

```
chown user:group path
```

- どちらも *path* のユーザ属性を *user* に, グループ属性を *group* に変更する

chown の成否

- プロセスPによるファイルFの chown の成否 ($\approx$ 管理者だけが変更可)
- ユーザ属性が変更される ($user \neq F$のユーザ属性) 場合

0. Pの実効 UID が 0
   $\Rightarrow$ OK
1. それ以外
   $\Rightarrow$ NG

- ユーザ属性が変更されない ($user = F$のユーザ属性) 場合

0. Pの実効 UID が 0
   $\Rightarrow$ OK
1. それ以外 ($\approx$ 自分が所属するグループへの変更が可能)
   $\Rightarrow$ $user \in$ {Pの実効 GID} + Pの補助グループ ならば OK

ファイルに対して, できることの平易な言語でのまとめ

- 所有者以外のユーザ ⇒
  - mode で許可された open/exec
- 所有者 ⇒
  - 上記
  - mode の変更
  - 自分が所属するグループへのグループ属性の変更
- root ⇒
  - 上記
  - 所有者の変更

プロセスの実効 UID

プロセスの実効 UID

- **プロセスの実効 UID (Effective User ID)** ≈ そのプロセスが「誰」として動いているかを表すプロセスの属性
  - `open`, `exec` などの成否を決める一要素
  - (再掲)
    0. Pの実効 UID が 0
       ⇒ OK
    1. Pの実効 UID = Fのユーザ
       ⇒ (F`.st_mode & S_IRUSR`) $\neq 0$ なら OK
    2. …
    3. …
- 実践的には「自分が立ち上げたプロセスの実効 UID は自分」という当たり前が殆ど成り立ち, それ以上知らなくてもあまり問題はない …

プロセスの実効 UID

- … だがまさか OS がキーボードを打っている人の顔や指紋を見て「誰」を判断しているわけではないので, プロセスの実効 UID がどういう仕組み (規則) で決まっているかを理解することは必要
- 実践的には,
  - `ssh`, `jupyterhub` (入力されたユーザ名に応じて適切に他のユーザに成り代わる)
  - `sudo` (管理者権限で実行する)

などがシステムコールレベルではどういう仕組みで成り立っているかを理解する鍵

プロセスに付随する 3 つの UID

- 実はプロセスには 3 つの UID がついている (理由は後述)
  1. **実** UID (**Real** User ID, RUID または単に UID)
  2. **実効** UID (**Effective** User ID, EUID)
  3. **保存** UID (**Saved** User ID, SUID)
- 基本は親プロセスの UID を全て継承するが, 変更するシステムコールがある

プロセスの {実・実効・保存} UID を{取得・変更}するシステムコール

- 取得
  - `uid_t` r `= getuid();` — rに実 UID を返す
  - `uid_t` e `= geteuid();` — eに実効 UID を返す
  - `int` err `= getresuid(&`r`, &`e`, &`s`);` — {実,実効,保存}UID をそれぞれr, e, sに返す
- 変更
  - `int` err `= setuid(`r`);` ≈ 実 UID をrにする
  - `int` err `= seteuid(`e`);` ≈ 実効 UID をeにする
  - `int` err `= setresuid(`r, e, s`);` ≈ {実,実効,保存}UID をそれぞれr, e, sにする

set?uid それぞれの正確な効果と成否

呼び出したプロセスの{実・実効・保存} UID を (R, E, S) と書くことにする

名前	成功条件	効果
setresuid(r, e, s)	$E = 0$ または $\{r, e, s\} \in \{R, E, S\}$	$(R, E, S) \to (r, e, s)$
seteuid(e)	$E = 0$ または $e \in \{R, E, S\}$	$(R, E, S) \to (R, e, S)$
setuid(e)	$E = 0$	$(R, E, S) \to (e, e, e)$
	$e \in \{R, E, S\}$	$(R, E, S) \to (R, e, S)$

- すぐにわかる通り, 後者 2 つは setresuid の特殊形に過ぎない
  - seteuid$(e) \equiv$ setresuid(R, e, E)
  - setuid$(e) \equiv E$ == 0 ? setresuid(e, e, e) : setresuid(R, e, S)
- 3 つ存在するのは冗長で, 歴史的な事情
  - setresuid は後からでき, かつ POSIX (Unix 標準) ではない

ポイント 1

- 実効 UID (E) の変更 $\Rightarrow$ 権限の変更であり, 明らかに無条件に許してはならない
  - 基本は管理者プロセス ($E = 0$) だけに許される
  - ただし通常ユーザのプロセス ($E \neq 0$) も, {実・実効・保存} UID 中のどれかになることは可能
- `ssh`, `jupyterhub` などが色々なユーザとして動くプロセスを起動する仕組み:
  - 先祖プロセスは root として ($E = 0$ で) 起動される
  - ユーザuとして認証が済むと, `setuid(`u`)` を用いてユーザuに「成り代わる」
    – $(R, E, S) \to (u, u, u)$ になる
  - この状態から他のユーザになり代わる(UID を変更する)ことは不可能

ポイント 2

- 実効 UID = 0 ($E = 0$) であれば無条件で, 任意のユーザに「なりすまし」(UID を変更すること)が可能
  - システムコールレベルではそのユーザの認証 (パスワード要求など) は行わない
  - `set?uid` を発行するプログラムが適切な方針・認証を実装する必要がある

ポイント 3

- 通常ユーザ ($E \neq 0$) のプロセスも, {実・実効・保存}UID 中のどれかになることは可能
- $\Rightarrow$ 非 root ($E \neq 0$) から root ($E = 0$) に「昇格」が可能な場合 (Rまたは$S = 0$) もある!
- 例:

$$(0,0,0) \xrightarrow{\text{seteuid}(100)} (0,100,0) \xrightarrow{\text{seteuid}(0)} (0,0,0)$$

- 降格して必要なときにまた昇格するアプリケーションを可能にする
- 例えば普段は一般ユーザで走行, ユーザuとして認証が済んだら$E = 0$に昇格してuに成り代わる

$$(0,100,0) \xrightarrow{\text{seteuid}(0)} (0,0,0) \xrightarrow{\text{setuid}(u)} (u,u,u)$$

- 「最小限の権限で動く」という鉄則に沿った設計

実行可能ファイルの set-user-ID 属性

- (再掲) `st_mode` に格納されている, アクセス許可 9 bit 以外の情報
  - set-user-ID (setuid bit, suid bit)
  - set-group-ID (setgid bit, sgid bit)
  - sticky
- set-user-ID bit の効果: プロセスPが, set-user-ID 属性が 1 の実行可能ファイルFを (exec 系システムコールで) 実行すると, Pの{実効, 保存}UID をFのユーザ(uとする)にする
  - $(R, E, S) \to (R, u, u)$

set-user-ID 属性のポイント

- set-user-ID 属性がセットされたファイルを実行するのは, $(R, E, S) = (u, u, u)$ $(u \neq 0)$ の状態からでも実効 UID を変更できる唯一の手段
- sudo, su など (root ではない) 普通のユーザであっても管理者や他のユーザに「成り代わる」コマンドにとって必須の仕組み
- それらのコマンドの実行可能ファイルは
  - ユーザ属性 (所有者) = 0 (root)
  - set-user-ID 属性をセット
- root に限らず, あるユーザの権限を他のユーザに与えるための一般的手段

set-user-ID 属性の使用場面例

- 複数ユーザで共有したいデータベース D (ファイル)
  - 複数ユーザがDに書き込める必要があるが直接ファイルへの書き込み権限を与えるわけにはいかない (壊される危険)
- 例えばどのユーザにも D に追記 (行を追加) する権限を与えたい

$\Rightarrow$

- D の
  - ユーザ属性を u
  - アクセス権限は u 以外には与えない (`st_mode = 0600`)
- D にデータを追記するコマンド (実行可能ファイル) の
  - ユーザ属性を u
  - set-user-ID 属性をセット
- このコマンドで意図しないデータベース操作 (e.g., シェルの起動) を行えないようにするのはコマンド作者の責任

グループについて

グループの存在意義

- グループ = ユーザの集合
- 柔軟なアクセス制御のためのツール
  - グループを作る
  - グループに対するアクセス権限を設定する
- クラウドのファイル共有と異なり, 原始的な Unix では「複数だが全員ではない」人に対するアクセス権限を出す唯一の方法
- 注: 実は Linux ではもう少し柔軟な仕組み (ACL) があるが省略

コマンドレベルでの操作例 (シス管目線)

1. グループ `student` を作り, ユーザ `ohtake, mimura, degawa` をメンバーにする
   - ⇒ `/etc/group` にグループ名, グループ番号, 所属するユーザを記述
   - 注: 設定の仕方は `/etc/group` 以外にもある (LDAP, NIS など)
2. `ohtake` のファイル `hello.txt` を `student` グループのメンバーに読み書き許可する
   - `chown :student hello.txt` `# グループ属性を student に`
   - `chmod g+rw hello.txt` `# グループに対する rw 権限を出す`
3. これで `student` グループ所属のユーザ `ohtake, mimura, degawa` が `hello.txt` を読み書きできるようになる

以下これを OS (システムコール) レベルの動作として理解する

プロセスの実効 GID と補助グループ ≈ 実効 UID のグループ版

- open の成否 (一部再掲)
  0. …
  1. …
  2. Pの実効 GID または Pの補助グループ のどれか = Fのグループ
     ⇒ (F`.st_mode & S_IRGRP`) $\neq 0$ なら OK
  3. …
- ⇒ あとは実効 GID, 補助グループがいかにして決まるかを理解すれば良い
- 実効 GID の決まり方は実効 UID のそれとそっくり

プロセスに付随する 3 つの GID

- UID 同様, プロセスには 3 つの GID がついている
  1. **実** GID (**Real** Group ID, RGID または単に GID)
  2. **実効** GID (**Effective** GROUP ID, EGID)
  3. **保存** GID (**Saved** Group ID, SGID)
- UID 同様, 親プロセスの GID を全て継承 + 変更するシステムコールがある

プロセスの {実・実効・保存} GID を{取得・変更}するシステムコール

- UID と全く同じ
- 取得
  - `gid_t` r `= getgid();` — rに実 GID を返す
  - `gid_t` e `= getegid();` — eに実効 GID を返す
  - `int` err `= getresgid(&`r`, &`e`, &`s`);` — {実,実効,保存}GID をそれぞれr, e, sに返す
- 変更
  - `int` err `= setgid(`r`);` ≈ 実 GID をrにする
  - `int` err `= setegid(`e`);` ≈ 実効 GID をeにする
  - `int` err `= setresgid(`r, e, s`);` ≈ {実,実効,保存}GID をそれぞれr, e, sにする

set?gid それぞれの正確な効果と成否

(これも UID とほぼ同じ)

- 呼び出したプロセスの{実・実効・保存} GID を (R, E, S)
- プロセスの実効 UID をFと書く (Eが使えないので苦肉の記号)

名前	成功条件	効果
setresgid(r, e, s)	$F = 0$ または $\{r, e, s\} \in \{R, E, S\}$	$(R, E, S) \to (r, e, s)$
setegid(e)	$F = 0$ または $e \in \{R, E, S\}$	$(R, E, S) \to (R, e, S)$
setgid(e)	$F = 0$	$(R, E, S) \to (e, e, e)$
	$r \in \{R, E, S\}$	$(R, E, S) \to (R, e, S)$

実行可能ファイルの set-group-ID 属性

- これも set-user-ID 属性とほぼ同じ
- (再掲) `st_mode` に格納されている, アクセス許可 9 bit 以外の情報
  - set-user-ID (setuid bit, suid bit)
  - set-group-ID (setgid bit, sgid bit)
  - sticky
- set-group-ID bit の効果: プロセスPが, set-group-ID 属性が 1 の実行可能ファイルFを (exec 系システムコールで) 実行すると, Pの{実効, 保存}GID をFのグループ(gとする)にする
  - $(R, E, S) \to (R, g, g)$

補助グループの必要性

- 実効 GID はグループに対するアクセス権限を享受するのに使われる
  1. …
  2. …
  3. Pの実効 GID または Pの補助グループ のどれか = Fのグループ
     $\Rightarrow$ (F`.st_mode & S_IRGRP`) $\neq 0$ なら OK
  4. …
- 実効 GID $\approx$ それを実行しているユーザが所属しているグループの一つ
- だが実効 GID ひとつだけでは一人のユーザが複数のグループに所属しているという状態を表現できない $\Rightarrow$ 補助グループ
- 補助グループには任意個のグループを指定できる

補助グループを{取得・変更}するシステムコール

- 取得

```
gid_t groups[n];
int err = getgroups(n, groups);
```

- 変更

```
gid_t groups[n] = { ... };
int err = setgroups(n, groups);
```

- 変更ができるのは root ($E = 0$) だけ
- 補助グループは, ssh サーバや login サーバなどがユーザ認証後に `/etc/group` などを参照して, そのユーザが所属する全グループに設定され, 以降は変わらない

補助グループがあるなら実効 GID は不要では?

- アクセス権限の設定のためには実際, 実効 GID 不要であろう
- 実効 GID は,
  - ファイル作成時のファイルのグループ属性を決める
  - set?gid や set-gid-ID bit つきのファイルを使って変更が可能

という違い (逆に言うとそれだけ)

伝統的 Unix のアクセス権限設定の不自由さ

- 「全員ではない複数のユーザ」にアクセス権限を出す唯一の手段がグループ
- その OS (システムコール)レベルの仕組みはプロセスに実効 GID と補助グループという属性を持たせることだった
- だが補助グループは root ($E = 0$) プロセスだけに設定可能
- `/etc/group` ファイルも一般ユーザに編集できるわけではない
- $\Rightarrow$ 事実上, グループの設定も root 権限がないとできない
- $\Rightarrow$ 結局, (偶然既存のグループと一致していない限り) 「全員ではない複数のユーザ」でアクセスできるファイルは, root 権限がなければ作れない
- クラウドのファイル共有で複数のユーザを指定して共有ができるのと比べて不自由

この続きの高度な話題

- アクセス制御リスト (Access Control List; ACL)
  - 前述した問題を解決し, 自由に複数のユーザに権限を出せる仕組み
  - “posix acl”, “getfacl”, “setfacl” などで調べてみてください
- Capability (権限分割)
  - 多くのシステムコールの成否が実効 ID = 0 (root) か否かで左右される
    - chown, seteuid, …
  - ⇒ ひとつでも「特権」が必要なプロセスは実効 ID = 0となり「全権委任状態」になるしかない (危険)
  - システムコールごとに必要な権限を capability という小さい権限に分割
  - ユーザやプロセスに一部の capability だけを持たせることが可能
  - “Linux capability”, “setcap” などで調べてみてください