[9-4.] 特権処理の局所化

機密情報やハードウェアを直接扱うプログラムは管理者権限など特別な権限「特権」で動作する必要がある。万一セキュリティホールがあると，プログラムは悪用されシステム管理権限が奪われる。こうしたプログラムでは特権処理を局所化し，またその他の処理では特権を放棄することで，セキュリティホールによる被害を最小限に抑える設計が必要だ。

● ● ●

特権処理

パスワードなどの機密情報を格納したファイルやデータベースへのアクセスは，厳しく制限すべきである。しかしその機密情報を扱うプログラムからのアクセスは許可しなければならない。このようなプログラムは管理者権限などの特別な権限で動作するように設計される。

また近代オペレーティングシステムはネットワークデバイスなどのハードウェアへのアクセスを厳しく制限している。ハードウェアへアクセスするプログラムも管理者権限などの特別な権限で動作するように設計される。

システムに大きな影響を及ぼす変更操作は一般にシステム管理者のみに許されるようになっており，こうした操作を行う特別な権限を「特権」と呼ぶ。Unix や Linux では root アカウント，Windows 2000 や NT では Administrator および SYSTEM アカウントが特権を持っている。これらのアカウントを使い，特権が与えられて動作しているプログラムの状態を「特権モード」と呼び，特権が与えられなければ達成できない処理を「特権処理」と呼ぶ。先ほどのパスワードファイルへのアクセスやハードウェアへのアクセスが特権処理の例である。

特権処理を行うプログラムに一つでもセキュリティホールが存在すると，システム全体のセキュリティを著しく低下させてしまう。たとえばバッファオーバーランなどの問題があれば，それを悪用して攻撃者が自分に都合のよいプログラムを特権モードで実行させることが可能で，その結果システム管理者アカウントが奪取されることもあるのだ。

プログラムミス

ソフトウェアのセキュリティホールは設計ミスやプログラムミスにより生じる。我々はミスを根絶するように努力しなければならないが，どうしてもミスが入り込んでしまうものである。特権処理を行うプログラムは十分注意して設計・プログラムされるものであるが，しばしばセキュリティホールが報告されるのも事実である。

次節以降では，重大な影響を及ぼす特権処理を小さな範囲に閉じこめることで，ミスによりセキュリティホールが生じてしまった場合でも被害を最小限に食い止める予防策を紹介する。

最小権限の原則

特権処理を行うプログラムといえども，プログラムが開始してから終了するまでの間，終始，特権が必要というわけではない。特権処理はプログラム全体の一部の処理だけであるのが普通である。

しかしながら，多くのプログラムは**図１**(a)のように終始特権モードで動作するよう設計されている。特権が不要な処理でさえも特権モードで実行される。このためプログラム全体にわたって，一つでもセキュリティホールが存在すれば，システム管理者権限が奪取されてしまう。

安全で信頼性の高いプログラムを設計する際に守るべき原則の一つに，「最小権限の原則」(The least privilege principle）というものがある。

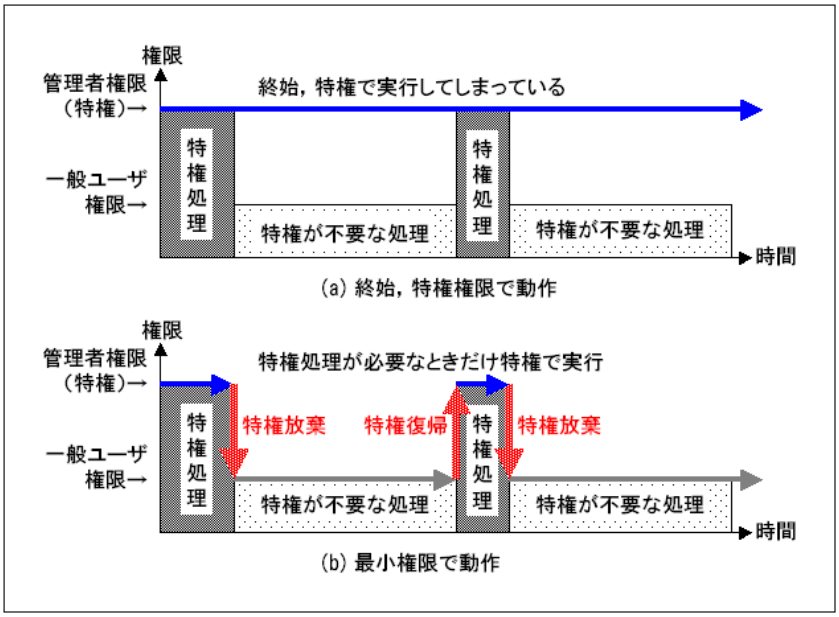
最小権限の原則

> **すべてのプログラムおよびユーザは目的のジョブを達成するために**
> **必要な最小限の権限において処理を実行すべきだ。**

この原則にしたがうことで，事故やエラー，攻撃から受ける被害を最小限にとどめることが可能だ。**図１**(b)はこの原則にしたがい，権限を処理内容に応じて動的に変化させるようにプログラムを再設計したものだ。特権処理が終了すると特権を放棄し，一般ユーザ権限で特権が不要な処理を実行する。再び特権処理の直前になると特権を復帰し特権モードで特権処理を実行する。特権放棄と特権復帰を繰り返しながら，プログラムは処理に必要とされる最低限の権限で常に動作する。

もしセキュリティホールが「特権が不要な処理」の部分で生じたとしても，攻撃者は一般ユーザ権限までしか奪取できない。システム管理者権限は保護される。特権処理部分で生じるセキュリティホールは依然としてシステム管理者権限の奪取につながるが，一般的に特権処理部分はプログラム全体に対して占める割合が小さいため，最小権限の原則の効果が発揮される。また特権処理部分のプログラムに集中して注意を注ぐことで，重大なセキュリティホールにつながるミスを防ぎやすくなる。

図１　最小権限の原則



プログラム開始時の特権処理局所化

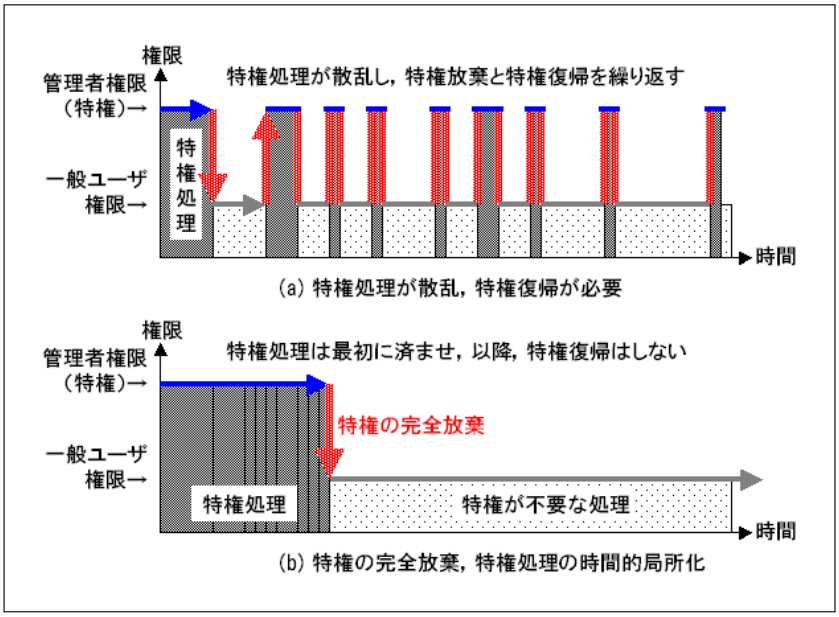
最小権限の原則に沿った**図１**(b）のプログラム設計例では，プログラムの開始から終了まで，必要性に応じて特権放棄と特権復帰を繰り返しながら特権処理を実行していた。しかし特権処理の大部分は工夫することによりプログラム開始時に完了させることができる場合がある。

例えばUnixやLinuxにおいてTCP/IPサーバプログラムが1023以下のポート番号をbind( )するときには，プログラムが特権モードで動作している必要があるが，一度bind( )が完了すると，そこで得られたソケットに対しては一般ユーザ権限でアクセス可能だ。

また管理者権限でしか読み書きできないファイルであっても，一度，特権モードでオープンしてしまえば，その後は得られたファイルディスクリプタを経由して一般ユーザ権限で読み書きできる。

図２(a)のように特権処理がプログラムの進行に伴い時間的に散乱しているプログラムであっても，工夫次第で**図２**(b)のようにプログラム開始時に大半の特権処理を完了できることが多い。プログラム開始時に特権処理を完了してしまえば，特権を《完全放棄》することができる。もう特権を復帰する必要はない。その後はプログラムが終了するまで一般ユーザ権限で動作するため，セキュリティホールがあったとしても，システム管理者権限を奪取されることはなくなる。

図２　プログラム開始時の特権処理局所化



特権の完全放棄

Unix および Linux システムの場合は root 権限を完全に放棄する手段があり，プログラムは終了するまで二度と root 権限を復帰できなくなる。この手段を活用することで，バッファオーバーラン攻撃などにより特権復帰するプログラムを送り込まれたとしても，特権復帰は失敗しシステム管理者権限は保護される。

Windows システムだと残念ながらこのようなことはできない。Windows 2000 や NT ではそのアカウントにあらかじめ許されている特権操作（システムのシャットダウンや仮想記憶ページのロックなど）を一つずつオン／オフする API が提供されてはいるが，誤動作対策には役立っても攻撃者にアカウントそのものを奪われてしまえば元も子もない。もう一つ，方式はかなり異なるが Windows 2000 や NT には「インパーソネーション」という機能がある。高い権限のサーバプログラムの一部のスレッドを低い権限のクライアントのアカウントで実行するといったことを実現する技術である。インパーソネーションについては関連記事『8-2. プロセス間通信とバックドア』で少し触れている。

プログラム内局所化

特権処理部分のプログラムにミスがあると重大なセキュリティホールにつながる。特権処理がプログラム全体のあちこちに分散していると，安全確認のチェック漏れが生じやすく，安全対策が手薄になりがちである。

特権処理はソースコードレベルで一般処理から分離し，特権処理専用のモジュールに閉じ込めてしまうべきだ。特権モードにより実行されるソースコードを局所化することで，小さな対象に対して手厚い安全対策が可能となり，同時にミスの排除も容易になる。

リスト１と**リスト２**はUnixあるいはLinux用のサンプルプログラムで，root権限でしか読み書きできないファイル"/var/mydata/secretfile.dat"を特権モードで読み書きする機能を提供するモジュールだ。パスワードなどの機密情報をこのファイルに格納することができる。

特権モードの取り扱いに関する設計方針は**図２**(b）に沿っている。プログラム開始時に本体プログラムから

リスト１　特権処理用モジュールのインタフェースファイル，secretfile.h

```
 1  /*
 2   * secretfile.h
 3   */
 4
 5  #ifndef __SECRETFILE_H__
 6  #define __SECRETFILE_H__
 7
 8  /* root 権限が必要な初期化処理 */
 9
10  extern void secretfile_init_in_privileged_mode();
11
12  /* 一般ユーザ権限でアクセスすべき処理 */
13
14  extern int  secretfile_read(int offset, int bytes, char *outbuffer);
15  extern int  secretfile_write(int offset, int bytes, const char* inbuffer);
16
17  /* root 権限でも一般ユーザ権限でもアクセスできなければならない処理 */
18
19  extern void secretfile_finish();
20
21  #endif
```

リスト２　特権処理用モジュールの実装ファイル，secretfile.c

```
 1  /*
 2   * secretfile.c
 3   */
 4
 5  #include "secretfile.h"
 6  #include <sys/types.h>
 7  #include <sys/stat.h>
 8  #include <fcntl.h>
 9  #include <unistd.h>
10  #include <stdlib.h>
11  #include <errno.h>
12
13  #define SECRETFILE "/var/mydata/secretfile.dat"
14
15  /* 条件が満たされなければ abort()するマクロ */
16  #define MAKESURE(expr) ((void)((expr)?0:abort()))
17
18  /* secretfile モジュール内部のスタティック変数 */
19  static int fd = -1;     /* ファイルディスクリプタ */
20
21
22  /* root 権限が必要な初期化処理 */
23
24  void secretfile_init_in_privileged_mode()
25  {
26      /* root 権限で動作していることを確認 */
27      MAKESURE(geteuid()==0);
28
29      /* root 権限でデータファイルのオープン */
30      fd = open(SECRETFILE, O_RDWR|O_EXCL);
31
32      /* データファイルが正しくオープンできていることを確認 */
33      MAKESURE(fd!=-1);
34  }
35
36  /* 一般ユーザ権限でアクセスすべき処理 */
37
38  int secretfile_read(int offset, int bytes, char *outbuffer)
39  {
40      int actual_readbytes = 0;
41
42      /* 一般ユーザ権限で動作していることを確認 */
43      MAKESURE(geteuid()!=0);
44
45      /* データファイルが正しくオープンされていることを確認 */
46      MAKESURE(fd!=-1);
47
```

リスト２（つづき）

```
 48      /* ファイルポインタを指定オフセット位置へ移動 */
 49      MAKESURE(lseek(fd, offset, SEEK_SET)!=-1);
 50
 51      /* ファイル内容を読み出す */
 52      {
 53          ssize_t readbytes;
 54          while(bytes>0) {
 55              /* 読み出してみる */
 56              readbytes = read(fd, outbuffer, bytes);
 57
 58              /* シグナルにより read()が中断することは正常なケース */
 59              if(readbytes==-1 && errno==EINTR) continue;
 60
 61              /* エラーが発生していないか確認 */
 62              MAKESURE(readbytes!=-1);
 63
 64              /* EOF に到達したか確認 */
 65              if(readbytes==0) break;
 66
 67              /* 実際に読み込んだバイト数によりカウンタなどの変数を加減算 */
 68              bytes -= readbytes;
 69              actual_readbytes += readbytes;
 70              outbuffer += readbytes;
 71          }
 72      }
 73
 74      /* 実際に読み込んだバイト数を返す */
 75      return actual_readbytes;
 76  }
 77
 78  int secretfile_write(int offset, int bytes, const char* inbuffer)
 79  {
 80      int actual_writtenbytes = 0;
 81
 82      /* 一般ユーザ権限で動作していることを確認 */
 83      MAKESURE(geteuid()!=0);
 84
 85      /* データファイルが正しくオープンされていることを確認 */
 86      MAKESURE(fd!=-1);
 87
 88      /* ファイルポインタを指定オフセット位置へ移動 */
 89      MAKESURE(lseek(fd, offset, SEEK_SET)!=-1);
 90
 91      /* ファイルへ書き込む */
 92      {
 93          ssize_t writtenbytes;
 94          while(bytes>0) {
 95              /* 書き込んでみる */
 96              writtenbytes = write(fd, inbuffer, bytes);
 97
 98              /* シグナルにより write()が中断することは正常なケース */
 99              if(writtenbytes==-1 && errno==EINTR) continue;
100
101              /* エラーが発生していないか確認 */
102              MAKESURE(writtenbytes!=-1);
103
104              /* EOF に到達したか確認 */
105              if(writtenbytes==0) break;
106
107              /* 実際に書き込んだバイト数によりカウンタなどの変数を加減算 */
108              bytes -= writtenbytes;
109              actual_writtenbytes += writtenbytes;
110              inbuffer += writtenbytes;
111          }
112      }
113
114      /* 実際に書き込んだバイト数を返す */
115      return actual_writtenbytes;
116  }
117
118  /* root 権限でも一般ユーザ権限でもアクセスできなければならない処理 */
119
120  void secretfile_finish()
121  {
122      /* ファイルがまだオープンされているならばクローズ */
123      if(fd!=-1) {  /* ← この関数が２回以上呼ばれても安全 */
124          close(fd);
125          fd = -1;
126      }
127  }
```

リスト３　パスワードを暗号化し保存するプログラム，savecrypt.c

```
 1  /*
 2   * savecrypt.c
 3   */
 4
 5  #include <stdio.h>
 6  #define __USE_XOPEN
 7  #include <unistd.h>
 8  #include <sys/types.h>
 9  #include <time.h>
10  #include <stdlib.h>
11  #include "secretfile.h"
12
13  /* 条件が満たされなければabort()するマクロ */
14  #define MAKESURE(expr) ((void)((expr)?0:abort()))
15
16  int main(void);
17
18  /* salt char table (64bytes+NULL byte) */
19  static const char salt_char[] =
20      "AaB7bCc4DdEe.FfG6gHhIi2JjKkLlMm3N8nO/oPpQq0RrSs1TtUuVv9WwX5xYyZz";
21
22  int main(void)
23  {
24      const char* passwd;
25      const char* crypted;
26      char salt[2];
27      int unixtime;
28      int offset;
29
30      /*
31       * 特権モードで初期化
32       */
33
34      secretfile_init_in_privileged_mode();
35
36      /* 特権の放棄 → プログラムを実行させた元ユーザ権限へ権限を落とす */
37      MAKESURE(setuid(getuid())==0);
38
39      /*
40       * 一般ユーザ権限でサービス提供
41       */
42
43      /* パスワードをユーザに入力してもらう */
44      passwd = getpass("Input your passwd:");
45
46      /* パスワードを暗号化する */
47      unixtime = (int)time(NULL);
48      salt[0] = salt_char[unixtime&63];
49      salt[1] = salt_char[(unixtime/7)&63];
50      crypted = crypt(passwd, salt);
51
52      /* パスワードを書き込む */
53      offset = getuid() * 13;
54      secretfile_write(offset, 13, crypted);
55
56      /* 終了処理 */
57      secretfile_finish();
58
59      return 0;
60  }
```

特権モードでsecretfile_init_in_privileged_mode( )関数を呼び出してもらう設計である。この関数で特権処理を完了させてしまう。その後，本体プログラムには特権放棄後の一般ユーザ権限で，secretfile_read( )関数やsecretfile_write( )関数を呼び出してもらい，ファイル入出力機能を提供する設計だ。

secretfile.datファイルの読み書きの処理はこの特権処理用モジュールsecretfileに局所化されている。安全対策はこのモジュールに集中すればよい。**リスト２**を見ていただきたい。MAKESHUREマクロ（注1）をふんだんに使用し，さまざまな異常チェックを盛り込んである。異常が発見された場合はプログラムを即座に終了させることで，特権プログラムが悪用されることを防いでいる。

リスト３は上記のsecretfileモジュールを利用したサンプルプログラムsavecryptで，プログラムを起動したユー

ザにパスワード入力を求め，それを暗号化（注2）して上記モジュールが管理するファイルへ書き込むというものだ。

上記モジュールが管理するファイル"/var/mydata/secretfile.dat"はroot権限がなければ読み書きできないので，**リスト３**のsavecryptプログラムは特権モードで動作しなければならない。そのためこのプログラムはUnix（およびLinux）のsetuidビット（注3）がセットされている。setuidビットにより，このプログラムは一般ユーザが起動してもroot権限で実行される（**リスト４**）。

リスト４　savecryptプログラムはsetuidビットがセットされている

```
1  $ ls -l savecrypt
2  -rwsrwxr-x    1 root     root        14724 Feb 15 00:10 savecrypt
     ↑
   setuidビットがセットされている
```

プログラムが起動するとroot権限で（すなわち特権モードで）実行を開始する。22行目のmain( )関数からの処理が始まった直後，34行目で**リスト１**，**リスト２**のsecretfileモジュールの中の特権モードで実行すべき初期化関数secretfile_init_in_privileged_mode( )を呼び出している。**リスト３**はサンプルプログラムであるのでプログラム開始直後の特権処理はこれだけであるが，本格的なアプリケーションの場合，特権モードで実行すべき複数の初期化処理が続くこともあるだろう。

特権処理が完了した後，37行目でroot権限を放棄する（特権放棄）。それ以降，savecryptプログラムはこのプログラムを起動した一般ユーザの権限で動作する。38行目以降の処理でセキュリティホールが存在したとしてもシステム管理者権限は保護される。

リスト３全体をもう一度見ていただきたい。特権処理は34行目と370行目のみ関係するが，単純な関数呼び出し程度の処理であるため，セキュリティホールにつながるようなミスは発生しにくい。それ以外の個所については特権モードで動作しないため，問題が発生したときの影響は特権処理ほど大きくない。このように特権処理をモジュールに閉じ込め局所化することで，プログラム本体部分の安全を高めることができた。

リスト３で示したsavecryptプログラムは簡単なものであるので，特権処理を実装した**リスト１**，**リスト２**のほうが一般処理を実装した**リスト３**より大きくなっている。しかし実際のアプリケーションの場合，プログラム全体に対して特権処理部分の占める割合は小さいものであるのが普通だ。

（**注１**・MAKESUREマクロ（**リスト２**，16行目）は与えられた条件式を評価し，それが偽である場合にabort( )関数を呼び出してプログラムを即座に終了させる。特権処理を伴うプログラムがプログラマの予期しない状態に陥ると，大きなセキュリティ障害へとつながる危険性が生じる。サンプルプログラムでは，プログラムの各所にMAKESUREマクロを盛り込むことにより，常にプログラムが「正しい状態」で動作していることを確認している。）

（**注２・リスト３**における暗号化処理には簡単のためcrypt( )関数を使用しているが，この関数が提供する暗号の強度はそれほど高くない。実用に供するアプリケーションには専門のベンダが提供している暗号パッケージソフトウェアの使用をお奨めする。）

（**注３**・Unix（およびLinux）の実行形式ファイルの属性（ファイルモード）にはsetuidビットというものがある。setuidビットがセットされている場合，その実行ファイルを実行させると，プログラムを起動するユーザ権限ではなく，ファイルのオーナの権限でプログラムが実行される。例えばファイルのオーナがrootであるならば，一般ユーザがこのプログラムを起動したときでもプログラムはroot権限で動作する。）

外部プログラム化

上記では特権処理を同じプログラムの中の別のモジュールに閉じ込め局所化する例を示したが，特権処理部分を別プログラムとして分離してしまう手法もある。この場合本体プログラムは特権モードで動作する必要がなくなる。本体プログラムと特権処理用プログラムとの間のインタフェースが十分安全に設計されている必要があるが，インタフェースを監視するモニタ機能やログ機能を組み込むことで，さらに安全性を高めることも可能だ。

まとめ

特権処理は一つ誤るとシステムに重大な被害をもたらす。プログラムの設計を工夫することで特権処理を小さな部分にくくり出せることが多い。プラットフォームによっては，プログラム開始時に特権処理を済ませてしまい以降の実行では特権を完全に放棄する，といったやり方ができる。高度な安全対策が求められる箇所を狭い範囲に限定し，プログラムの他の部分からの影響を受けないよう隔離することが重要である。

関連記事

『7-3. setuid は慎重に』

『8-3. NTFS のセキュリティ機能と落とし穴』

『10-1. 設計段階からのセキュリティ』

参考文献

『Secure UNIX Programming FAQ』（英文），Thamer Al-Herbish，1999 年
http://www.whitefang.com/sup/secure-faq.html

修正履歴

2002 年 3 月 23 日　MAKESURE マクロを導入するなど，**リスト１**，**リスト２**，**リスト３**を修正した。