# Linux Kernelをハックしてみた

作者 : Hiroki @ 3J

## 1　何があった

私は前々からOSやLinuxに興味があったのだが、実際にそのOSをいじるということは経験したことがなかった。ソースファイルだけ手元にあっても、OSをいじることはできない。Linuxの Kernelだけで900〜1000万行あるとさえ言われる膨大なソースコードを前に、目的意識なしに突っ込むことは容易にできない。

話は変わって、IPA(情報処理推進機構:http://www.ipa.go.jp/)が毎年主催する「セキュリティ＆プログラミングキャンプ」というイベントが、8月の12日(木)〜16日(月)の4泊5日で行われた。このキャンプでは、全国から書類選考を経て集められた22歳以下の60名近い学生が、プログラミングとセキュリティという2コース各3組の合計6組に分かれ、それぞれを専門で扱う講師の下技術を学んだ。

私はプログラミングコース、Linuxカーネル組に参加し、Linuxを如何にしてハックするかという基礎知識を学んだ。せっかくなので、少しでも多くの人に興味を持ってもらうべく、基礎部分をまとめることにした。

## 2　取り扱う内容

基本的に以下のことについて触れる。

- ソースの入手とバージョン管理
- コンパイル
- ソースの読み方
- LKMLへの参加方法

記事の分量上、かなり基礎的な部分から扱うことになる。

## 3　必要技能・環境

### 3.1　環境

今回取り扱うLinux環境は**Fedora 13**とする。
なぜUbuntuじゃないかと言えば、私がFedoraとかCentOSの方が好きだから。

### 3.2 必要な環境

- Linux のターミナル操作に嫌悪感を持っていない
- root 権限がある
- vi(vim) が使える
- ちゃんとしたインターネット環境がある

vi は趣味なので emacs の拡張については触れていない。emacs でもソースコードの改変は十分に可能である。

## 4 ソースの入手とバージョン管理

まず Linux Kernel をハックするに当たり、必要なものはソースコードである。ソースコードは普通にネット上からダウンロードできるが、それはある種安定版で開発者向けではない。不安定だが最新のソースコードを入手しなければ、まともな開発はできない。

そこで、Linux のバージョンを管理するために作られた Git と呼ばれるバージョン管理ソフトを導入し、ソースコードを入手する。

### 4.1 Git の導入

Git は Linux のバージョン管理を行うために、Linux の開発者である Linus 氏がわずか数日で作った。特徴として、高速に動作することや、メーリングリストに投げることを考えて作られた機能等がある。

Fedora では root 権限で `yum install global`(Debian や Ubuntu は `apt-get install git-core`)でインストール可能であるが、今回はソースファイルからインストールしてみる。

以下インストールまでのコマンドを記す。コマンドを実行するのは`$`の行である。

```
//ソースファイルのダウンロード。
$ wget http://tamacom.com/global/global-5.9.2.tar.gz
//解凍。
$ tar zxvf global-5.9.2.tar.gz
//コンパイルのため解凍後に生成されたディレクトリに移動。
$ cd global-5.9.2
//設定を行う。
$ ./configure
//コンパイル。
$ make
//インストール。
$ make install
```

ソースファイルに関しては `http://www.gnu.org/software/global/` の Download のページでダウンロードしても良い。

これで Git の導入は完了したが、エディタである vi を連動できるように拡張しておくと、この後非常に便利なのでプラグインを追加する。

```
//vim のプラグインを入れるディレクトリを、hiroki ユーザーの中に作る。
$ mkdir -p /home/hiroki/.vim/plugin
//global のソースに混じって存在する gtags.vim というプラグインをコピーする。
$ cp gtags.vim /home/hiroki/.vim/plugin/
```

ユーザー名に関しては自分のものを使うこと。これにより、vi のコマンドが新しく追加されるが、それに関しては後ほど詳しく説明するのでここでは扱わない。

## 4.2 Linux Kernel のソースコードを入手する

いよいよ Linux のソースコードをダウンロードする。注意しておくことがいくつかあり、まずソースコード自体はそれほどでもないが、コンパイルなどをすると中間生成ファイルなどの関係で、5〜7GB 程度の空き容量が必要になるので、そこはちゃんと容量を空けておくこと。

Linux のソースコードは、バージョン管理できるツリーという状態でダウンロードする。

```
//hiroki ユーザーディレクトリ内に src ディレクトリを作る。
$ mkdir /home/hiroki/src
//カレントディレクトリの移動。
$ cd /home/hiroki/src/
//ツリーの取得。linux-2.6 と言うディレクトリが生成されその中にソースファイル
  が入る。
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torval
ds/linux-2.6.git
```

git clone アドレスというコマンドで、開発ツリーのダウンロードができる。今回ダウンロードしたのは最も有名なもので、他にも linux-next 等のツリーもダウンロードできる。

> Column - 開発
> Linux Kernelの開発はどのように行われているのか。
> Linuxは各開発者がGitによってソースをダウンロードし、開発を個別に行う。その結晶をパッチという形でメーリングリストに投げ、世界中の人に見てもらう。改良などがあればそれを行った上で大元のソースに適用できる人が作業を行う。このメーリングリストに関しては最後で扱う。

## 4.3 クロスリファレンスの作成

さて、これで開発の準備は整ったが、良く考えてほしい。1000 万行あるかというソースコードの中で、目的の関数や宣言をどうやって探すのか。検索しても良いが、検索は多少時間がかかるので、あらかじめ辞書を作って、簡単にジャンプできるように準備しておく。

```
//ツリーのルートディレクトリに移動。
$ cd linux-2.6
//クロスリファレンスの作成。
$ gtags -v
```

gtags はクロスリファレンスの作成を行うコマンドで、-v でどのファイルを読んでいるかが分かる。初めは時間がかかるのでこれをしておくと安心できるだろう。次回以降クロスリファレンスを更新したい場合は-i をつけることで最低限の作業で終了する。

# 5　コンパイル

さて、Linux のソースコードを入手したので、さっそくコンパイルをしてみる。初めに書いておくが、私のように Atom 等の低スペック CPU(1.3GHz くらい) でコンパイルを行うと、カスタマイズなしの場合 5 時間かかる。ハイスペックでも仮想マシン上でコンパイルすれば、やはりそれなりに時間がかかるので、そこら辺は覚悟してコンパイルを始めること。私は寝る前にコンパイルを開始して翌日の朝結果を確認していた。

```
//ツリーのカレントディレクトリへ移動。
$ cd /home/hiroki/src/linux-2.6
//Kernel の設定を行う。
$ make menuconfig
//コンパイル開始。一応時間計測もしてみる。
$ time make
```

どの程度の時間がかかるか分からないので、今回は `time` コマンドを用いて、コンパイル時間の計測を行う。コンパイルが終わったら、Kernel をインストールしてみる。#はルート権限での作業となる。

```
//モジュールのインストール。
# make modules_install
//イメージのインストール。
# make install
```

本来であれば、コンパイルをホスト OS(仮想マシン上じゃない OS) で行い、この Kernel インストール等はゲスト OS (仮想マシン上の OS) でやることが望ましい。何かあったら取り返しのつかないことになるので。

これで再起動してインストールした Kernel で起動すれば、晴れて自分でコンパイルした Linux を使うことができる。

……本当にこれだけ？と思うかもしれないが、これだけである。あっけないかもしれないが、Makefile(複数のソースで構成されるプログラムをどのようにコンパイルするかなどを記述したファイル。make コマンドで実行可能) があるので、コンパイル自体は非常に簡単である。

# 6　ソースの読み方

ようこそいらっしゃいました。ここからが本編。Linux Kernel の森の中へようこそ。

Linux のソースコードの量は膨大であると書いたが、なぜこれほど膨大なのか。まず、CPU の違いによるアセンブリ言語の違いが上げられる。いくつかの CPU に対応した OS を作るとなると、最終的に CPU にとって個別の処理が必要な部分も出てくる。そのようなソースコードがあるため、使わないソースファイルも多々存在する。必ずしも全てコンパイルするわけではない。

そして、モジュール (拡張機能) やドライバの割合もかなり多い。特にドライバは世界中で量産され、OS はその中でも良く使われるものをあらかじめ持っておいて使うので、その量がかなり多くなっている。(最近世界標準規格が生まれつつある。Web カメラに関しては共通規格の UVC が生まれ徐々に広がっている。詳細は本誌「WebCam を作ってみた」にて。)

量が多い多いと言っても分からないので、先ほど Linux のコンパイルに 5 時間かかった Atom を例にとると、モジュールやドライバの多くをコンパイルしない最小構成の設定を行った上でコンパイルすると、15 分で終了する場合もある。本当の核の部分はそれほど大きくはない (動かなかったけど)。

そう考えると、私たちがいじる部分は、かなり限定的ということになる。それを知った上で、少しソースコードを読み、改造してコンパイルを行う。

## 6.1 システムコールを探せ

システムコールを探してみる。この課題はセキュリティ＆プログラミングキャンプの事前課題で出され、多くの学生が頭を悩ませた部分である。

システムコールとは、Linux における権限を越えた命令の実行を代行する命令のことである。例えばファイルの読み書きは、実はユーザー権限では許されておらず、Kernel が厳重に守っている。しかしそれではまともにプログラムも実行できないので、システムコールという形で Kernel にその作業を代行してもらうことで、セキュリティ等を高めている。

今回は symlink(ショートカットを作るシステムコール) を探してみる。ソースファイルの検索はコマンドが用意されているとは思うが、今回は global を使ってみる。

```
//すでに linux-2.6(クロスリファレンスと作ったところ) にいるとする。
//global コマンドで、main という関数を検索する。
$ global main
```

たくさんのファイル名が出てきただろう。これらのファイルの中に、main() 関数が書かれているという意味である。

では、さっそく symlink の場所を調べる。システムコールも関数として定義されているので検索できるはずである。

```
//global コマンドで、symlink という関数を検索する。
$ global symlink
```

何かエラーが返されただろうか。このエラーは検索したオブジェクト (関数や定数等) が見つからなかったという意味である。そんな馬鹿な！検索して見つからないってどういうことだ！と思うだろう。実は別の名前なのかとも疑いたくなる。

こんな装備で森の中に迷い込んでしまっては、探索も進まない。ここで秘密兵器である、vi と global を連携させて symlink の捜索を行う。

## 6.2 vim+Gtags

ここで symlink が見つからなかった理由を書いておく。ずばり、クロスリファレンスを作った時に、シンボルが作成されなかったということである。global で関数検索を行うと、シンボルを検索するので、無い場合にはエラーとなる。ここで symlink は特殊な書き方をされているということが何となく分かる。

さて、まずは vi を起動する。(恐らく vim だけど。) すでにプラグインで拡張しているので、global コマンドと同等のことが行える。まずは main() 関数を検索してみる。

起動時にはコマンドモードなので、:Gtags main というコマンドを実行する。いきなり上下の 2 画面モードとなり、上にソースファイル。下にファイル一覧が出てきたはずだ。このように、vim の Gtags を使えばファイル一覧とソースの中身を見たりジャンプすることが可能だ。global コマンドよりははるかに捜索効率が上がる。ここでいつものように:q コマンドで脱出すれば、ファイル一覧のみとなり、上下で移動してエンターキーを押すと、そのファイルの main() 関数が記述されている部分を読むことができる。

主に使うことになる vi のコマンドを以下にまとめる。

| コマンド | コマンド説明 |
|---|---|
| :Gtags シンボル | 関数やマクロなどのシンボルを検索する。 |
| :Gtags -g 文字列 | 文字列を検索する。 |
| :GtagsCursor | 現在カーソル位置にある文字列の定義場所にジャンプする。 |
| /文字列 | vi で表示している文中から文字列を検索しジャンプする。 |

これらのコマンドを駆使して、今から symlink を見つけ出す。

## 6.3 symlink を探せ

まずは無難に:Gtags symlink でシンボル検索を行う。見つからない。ここで全文検索を行う。:Gtags -g symlink のコマンドを実行してしばらく待つと、結果が返ってくる。結果は次のような形式で返ってくるので、非常に分かりやすい。

ファイルパス | 行数 | その行の記述

大量のファイルが出てきただろう。これで symlink の使われている場所、宣言されている場所が全て分かる。だが少し待とう。上から全部調べればいつか定義にたどりつくだろうが、量がおかしい。システムコールとなれば、他のソースでも多々使われる。ここから探し出すのは困難である。

そこで、システムコールは恐らく C 言語で書かれているだろうという予測の元、アセンブリ言語で書かれたファイルを無視することも考えられる。それでも量が多い。宣言らしきものや#define がたくさん見えることだろう。

ここで、少し落ち着こう。ここからは Linux やパソコンに対する基本知識を総動員する必要がある。簡単なところから考えてみると、ファイルへのショートカットを作るのだから、symlink という関数は恐らくファイル関連の場所に集められているはずだ。その考えから ls してディレクトリ構造を調べると、fs というディレクトリが見つかるだろう。この中にはファイルシステム絡みのソースコードが入っているので、そこに移動してみる。

ここにも大量のファイルがあるので、良く考える。ショートカットを作ると言うことは、ファイルへの別名をつけるとも考えられる。ファイルシステムについて記述した ext3 とか fat 等も怪しいが、ファイルシステムごとに作っていたら大変である。別名というところからファイル名を斜め読みすると、namei.c というファイルが見つかるはずだ。少し覗いてみる。

vi でファイルを開き、/symlink で検索してみる。いくつか候補が見つかるので、n キーを押して次に進む。

しばらく進むと、vfs_symlink 等の似たような関数の宣言が出てくる。これが本体か？でも引数にディレクトリのようなものがあるので違うだろう。

もう少しnを押して次に進むと、少し英語の読める人なら「あれ？」と感じる部分が出てくる。

```
SYSCALL_DEFINE2(symlink,const char __user *,oldname,const char __
user *,newname)
```

SYSCALL……システムコールのこと？第一引数はsymlinkだ。第二引数はconst char __user *で型宣言だとしたら、第三引数はoldname。古い名前という変数名。じゃぁ次は同じ文字列型で、新しい名前という変数名だ。どう見てもsymlinkのシステムコールを呼び出すのに必要な変数名が書かれている。

ここでようやく、symlinkの正体にたどりついた。

```
SYSCALL_DEFINE2(symlink,const char __user *,oldname,const char __
user *,newname)
{
        return sys_symlinkat(oldname, AT_FDCWD, newname);
}
```

これが、symlinkシステムコールのソースコードである。sys_symlinkatも実はシステムコールなので普通に探すと見つからない。

さて、ここで分かったことがある。システムコールは、SYSCALL_DEFINE[引数の数]というマクロで定義されていた。だからクロスリファレンス作成時に、シンボルとして登録されなかったのである(関数定義には見えないため)。

これはLinux独特の書き方で、少し前からシステムコールはこのようなマクロで定義するようになったらしい。この前提知識を使えば、viでGtags -g SYSCALL_DEFINEで全文検索を行った後、viの検索でシステムコール名を探せば、それなりに早く見つけることができる。

他にも独特な書き方として、条件分岐後goto文を使って後ろの方に飛ばしていくつかの処理を視覚的に見やすく書いている部分があったりする(goto文はスパゲッティコードになるので使わないと言う人も多くいるが、比較的この書き方は多用されている)。

## 6.4 システムログへの書き込み

ここで、システムログ(dmesgコマンドで読める)にsymlinkシステムコールが呼ばれたら通知する改良をKernelに施す。

ここで一度バージョン管理ソフトであるGitを使って、作業の差分をとるようにする。

```
//すでにlinux-2.6にいるとする。
//チェックアウト。ここで作業を分岐させる。名前はmydebugだがある程度好きにするこ
//とができる。
$ git checkout mydebug
```

ここから作業を行うことになる。

ここで一つ注意したいのが、C言語で使えた命令文が使えない可能性が多々あることを意識することである。例えば、メモリ領域を確保するmalloc()関数は存在せず、その代わりとなるLinux Kernel内部でのみ使えるメモリ確保の命令(kmallocとか)があったりする。

さて、システムログに関してだが、これに関してもファイルをわざわざ開くなどの必要はなく、専用の関数が用意されている。

```
printk( ログレベル "整形文", 代入する変数など );
```

ログレベル以外は普通の printf() 関数と同じものである。ここのログレベルは省略可能なもので、ここの内容によってはLinuxを停止させることも可能である。

| ログレベル | マクロ | 意味 |
|---|---|---|
| 0 | KERN EMERG | システムが使用不能 |
| 1 | KERN ALERT | 直ちに対処が必要 |
| 2 | KERN CRIT | 致命的な状態 |
| 3 | KERN ERR | エラー状態 |
| 4 | KERN WARNING | 警告状態 |
| 5 | KERN NOTICE | 通常状態だが大事な情報 |
| 6 | KERN INFO | 通知 |
| 7 | KERN DEBUG | デバッグレベル情報 |

この中で、例えば警告状態等のログレベルにすると、Fedoraのデスクトップにポップアップウィンドウが出て、警告文を出力するようになる。このようにこの値を読み取って視覚的に訴える処置などが施される場合がある。

これを使って以下のように改造する。

```
SYSCALL_DEFINE2(symlink,const char __user *,oldname,const char __
user *,newname)
{
        printk(KERN_DEBUG "symlink( %s, %s )¥n",oldname,newname);
        return sys_symlinkat(oldname, AT_FDCWD, newname);
}
```

この改良を施してKernelをコンパイルすると、symlinkが呼ばれるたびに、元のファイル名と新しいファイル名の情報をシステムログに出力するようになった。ログレベルも値が大きいので、警告が出るようなこともない。

これができたら、ひとまずKernelハック第一歩である。

**Column - カーネルパニックを起こす関数**

Linux Kernelには、BUG()と呼ばれる関数が用意されている。この関数は、実行された瞬間カーネルパニックを起こし、パソコンの電源が落ちる。これを使うことで致命的なエラーに関してはそれ以上の動作を止めることができる。このように便利な関数がKernelに用意されている。

例えば、上のソースのprintk()関数をBUG()に書き換えてみよう。ログイン画面を見ることなくパソコンの電源が落ちるだろう。

ここまでの作業をパッチファイル(変更点を記したファイル)として出力してみる。

```
//変更に名前をつけて更新する。これがパッチファイルの名前となる。スペースは-に置
  換される。
$ git commit -m "symlink systemlog write"
//master(本体)との差分をとる。
$ git format-patch -o p1 master..mydebug
```

2つ目のコマンドで、p1というディレクトリを作り、その中にパッチファイルを出力するようにしている。これによりp1の中に変更箇所の詰まったいくつかのパッチファイルが吐

かれる。今回はファイルを 1 ついじっただけなので、1 つのファイルが存在しているはずである。

今回の命名はかなり下手な例である。ちゃんとした修正を行う場合は、どの個所を変更したかが分かるような名前の付け方にする必要がある。

**Column - コンパイルエラーを起こすマクロ**

Linux Kernel側で用意された関数に、不適切な値を渡されるとコンパイルエラーを起こすマクロが用意されている。これをBUILD_BUG_ON()といい、条件式が真の場合コンパイルエラーとなる。内容は次のようになっている。

```
#define BUILD_BUG_ON(condition) ((void)sizeof(char[1 - 2*!!(condition)]))
```

条件式の2重否定は条件を真偽値の0と1に固定する効果がある。それを知った上で条件が真と偽の時の様子を見てみる。

```
#define BUILD_BUG_ON(真) ((void)sizeof(char[1 - 2*1]))
#define BUILD_BUG_ON(偽) ((void)sizeof(char[1 - 2*0]))
```

面白いのは、コンパイルエラーの起こし方である。char型配列のサイズを調べるだけなのだが、その配列の数が-1か1となる。条件が真の場合、配列の数が負となり、そこでコンパイルエラーが発生する。Linuxではこのようにして不正な固定値の与え方に対してコンパイルを阻止している。

# 7 LKML への参加方法

致命的なバグを見つけ、それが自分の確認できる範囲内で修正されたことを確認し、Git を使ってパッチファイルを作ることにも成功した。後はこれを今の Linux のソースコードに反映させるだけである。しかし、どのようにして大元のソースコードに修正を反映させるのか。

Linux の場合はメーリングリストを使っている。メーリングリストに修正パッチファイルを送り、それをいろんな人が見て、修正箇所を議論し、修正を反映できる人がとりこんだ時初めてそれが全体に反映される。この議論を行うメーリングリストが、Linux Kernel Mailing List。略して LKML である。

こちらにもソースコード同様独特の文化があり、適当に送っても採用されるどころか無視されることもある。ここではメーリングリストへの登録方法と、簡単な注意事項をまとめる。

## 7.1 LKML の登録

まずはメーリングリストに参加する。`http://vger.kernel.org/vger-lists.html` に Linux 絡みのメーリングリストがたくさんあり、その中から自分の参加するメーリングリストを選ぶことになる。

今回は Kernel の開発に参加すると言うことで、`linux-kernel` というメーリングリストに参加する。リンクをクリックし、詳細の上の方にある `subscribe` をクリックする。ここに参加するためのメールの定型文や送り先が書かれている。メールの新規作成ウィンドウが出てくるので、必要な部分をコピーして LKML 用のメールアドレスで送ることになる。LKML に登録するメールアドレスはどこのものでも良いが、できれば本名を連想しやすいとか、関係のある文字列であることが好ましい。

メールを送ると、英語のメールが送られてくる。内容的には「登録したいってメール送ったけど大丈夫？本当に登録するなら以下の内容が本文のメールを送ってね。」という感じの文章が書かれている。内部構造は次のようになっている。

```
auth 何かよく分からん文字列 subscribe メーリス名 登録メールアドレス
```

これだけを本文にして送り返すと登録が完了する。

### 7.2 注意事項

まず注意事項として、メールの量が半端ない。多い日には 300 通来ることになる。Gmail で見たところ、テキストベースなので容量は小さいが、それでも日に 4MB ずつ消費されることを覚悟すること。

次に、LKML にも文化はあるので、少し不安な人は様子見をすると良い。例えばメールの返信の仕方もある程度決まりがあり、普通少人数のやり取りでは会話が混乱しないので自分の文章を書いた後返信元文章を書くことが多いが、LKML は量が多く会話も入り乱れるので、簡単な挨拶、返信元文章、自分の文章という順番でメールが書かれることが多い。こうでもしないと何について書いたメールかが判別できない。

パッチの送り方だが、添付ファイルではなくメールの本文に直接貼り付けることになる。複数ある場合はタイトルに (n/N) のように全部でいくつ送るのか、今はいくつめのメールなのかを明記し、1 つずつ送る。これはテキストベースでしかメールのやり取りができない人を考慮するためだとか、解凍や添付ファイルのコピー等の面倒な作業ではなく単純なコピペで済ませるためなどいろいろ言われている。パッチに関しても、1 つの機能や 1 つの修正に対して 1 つのパッチにしないと、修正箇所が良く分からなくなるので分割方法についても注意が必要である。

他にも、相手は人間である。そのため、過去にあった質問や過去に議論されたような機能追加等に対しては (相手も人間なので、何度も何度も同じ議論をしたくはないため) 冷たく対応されることがある。自分で過去ログを検索することも重要である (よくググれカスなどという表現がネット上で見られるが、自分でできる限り調べることは最低限のマナーである。自分が面倒だからと言って面倒な作業を他人に押し付けるべきではない)。

最後に、英語のメーリングリストは怖いかもしれないが、何となく伝わる内容であれば十分である。みんながみんな正しい英語できっちりとしたやりとりをするのではなく、時には誰かがぶちぎれて汚い言葉を喚き散らすこともあるメーリングリストである。良くも悪くも、ちゃんとコミュニケーションをとろうと努力すれば何とかなるものなので、そこら辺は勇気を振り絞って頑張ってほしい。

## 8 まとめ

まず、今回の記事は非常にアバウトである。本来 Git など分厚い本が書ける多機能なバージョン管理ソフトであるので、そこら辺は自分で調べてほしい。パッチを吐いたり、作業用ブランチを作っていくつも分岐させたり、統合させることも非常に簡単にできるようになっている。

他にも、ちゃんと Kernel をハックするならまだまだまだまだ書かなければいけないことが山のようにあるのだが、私の気力も知識もついていかないので、この記事は導入編と捉え、足りない部分は自分で調べてほしい。いくつか注意事項があるとすれば、モジュール開発ならともかく、Kernel 内部をいじるとなるとやはり仮想環境も必要なので、それなりのスペックのパソコンがないと、まともに開発ができない。決して、私のようにネットブック並みのスペックのパソコンで開発を行おうなどという無謀なことはしないでほしい。心が折れるので。

# 9　最後に

今回はあまり x68 の会誌内容としてはあまり出てこない分野について記事を書きました。(ソフトウェアの中でも特に重い OS という分野の、しかも内部改造入門。) Linux と言えば、初め名前を聞いた時は Linux という OS があるものだと思っていましたが、実際 Linux と呼ばれる OS はたくさんあって、あれ? これってどういうこと?と混乱しました。Linux は Windows とは全く異なる歴史、構造を持った OS であるがゆえに、どこかに日本語の入門的な文章がないとなかなか手を出せないのが現実です。

Linux Kernel も同じで、ただただ膨大で、何をすればよいのか分からない。ちょっといじってみたいと言う人がいても、なかなかまとまった文章が無ければ挫折しがちであるので、私の記憶が新しいうちに、私の備忘録的な意味も込めて、今回の記事を書きました。

そういう意味で、Linux Kernel を学ぶ機会であるセキュリティ＆プログラミングキャンプを開催してくださった IPA、スパルタ方式でビシバシ鍛えてくださった、講師である日本のカーネルハッカーの方々、一緒にメーリングリストで事前課題を出し合い、当日も協力し合った仲間のみなさんに感謝しています。

最後に、ハッカーは誰が手助けをせずとも勝手にハッカーになる。だが、そのハッカーの卵が孵化するのを助けることは可能かもしれない。できるならばその手助けをしたい。そういう宣言をキャンプ中に聞きました。この記事もそこまでとは行かずとも、Linux や Kernel への興味を持つ人が増えれば嬉しい限りです。

# 10　参考

## 10.1　セキュリティ&プログラミングキャンプ関係

- そこはコンピュータ版「精神と時の部屋」
  - http://www.atmarkit.co.jp/flinux/special/camp2010/01a.html
- セキュリティ&プログラミングキャンプ 2010
  - http://www.ipa.go.jp/jinzai/renkei/spcamp2010/

## 10.2　Linux 関係

- JM Project
  - http://www.linux.or.jp/JM/
- Git 入門
  - http://www8.atwiki.jp/git_jp/
- Git - SVN Crash Course(in Japanese)
  - http://www.tempus.org/n-miyo/git-course-trans-ja/svn.ja.html

この記事の HTML 版と PDF のダウンロードは X680x0 同好会の公式サイト (http://www.x68uec.org/) のその他の作品＞デジタル版会誌＞ 2010 年度（Vol.16）にあります。