==========

セキュリティ・キャンプ２０１５ 解析トラック TOMOYO / AKARI / CaitSith ハンズオン（事前学習資料）

熊猫さくら

==========

■１■このハンズオンについて

Linux カーネルの中でシステムコールの監視を行う TOMOYO / AKARI / CaitSith を用いて、 Linux システムの挙動を解析する方法と、その挙動を制限する方法について学びます。

キーワード： TOMOYO / AKARI / CaitSith / アクセス解析 / アクセス制限

参考： http://tomoyo.osdn.jp/ http://akari.osdn.jp/ http://caitsith.osdn.jp/

■２■このハンズオンで使用する Linux 環境について

Windows 上のノートＰＣに Oracle VM VirtualBox をインストールし、その中で Linux 環境を稼働させます。なお、 GUI 環境を必要としないため、大部分は Tera Term から ssh 接続して操作します。キャンプ本番までに、 CUI 環境で Linux システムの管理を行えるように練習しておいてください。

Linux 環境としては、 CentOS 6 / CentOS 7 / Ubuntu 14.04 という３種類の仮想マシンイメージを使用します。この事前学習資料では、 VirtualBox および Tera Term の操作に慣れていただくための練習も兼ねて、 PXE でブートして sl コマンドが走るだけの Linux 環境の作成に挑戦していただきたいと思います。２つのゲスト（メモリ 1024MB と 128MB ）を同時に実行させることができる環境を用意してください。

■３■ x86_32 環境で試す場合の注意点について

キャンプ本番では x86_64 環境向け Windows 上で x86_64 環境向け Linux を動作させることを想定しているため、この事前学習資料は x86_64 環境向けに作成しています。 CentOS 6 または Ubuntu 14.04 を x86_32 環境で試す場合には、以下の点に注意してください。

・パス名やファイル名が若干異なります。

ubuntu-14.04.2-server-amd64.iso → ubuntu-14.04.2-server-i386.iso

CentOS-6.6-x86_64-minimal.iso → CentOS-6.6-i386-minimal.iso

ld-linux-x86-64.so.2 → ld-linux.so.2

x86_64 → i386

lib64 → lib

・カーネルコンパイル時に「 64-bit kernel 」を選択しないでください。

CentOS 7 は x86_32 環境向けが存在しないので、 x86_64 環境向け Linux を動作させる環境を用意できない場合には、試していただく必要はありません。

■４■ネットワーク環境について

ホスト（ Windows ）とゲスト（ Linux ）との間の通信にブリッジアダプターを使えない環境の場合、各ゲストに対して、インターネットとの通信およびゲスト同士の通信を行うために「 NAT ネットワーク」と、ホストとゲストとの間の通信を行うための「ホストオンリーアダプター」という２つのネットワークアダプターを使うように設定してください。設定方法は Web を探せば見つかります。

この事前学習資料では、アダプター１に「 NAT ネットワーク」を、アダプター２に「ホストオンリーアダプター」を割り当てる構成で使用し、ＤＨＣＰを用いてアダプター１に 10.0.2.4 / 255.255.255.0 というアドレスが、アダプター２に 192.168.56.101 / 255.255.255.0 というアドレスが割り当てられる場合の例で説明しています。環境によりアドレスが異なりますので、コピー＆ペーストをする際には注意してください。

■５■ CentOS 7 環境での作成手順

(1) メモリ 1024MB でゲストを新規作成し、 CentOS-7-x86_64-Minimal-1503-01.iso を用いて最小構成でインストールし、ネットワークアダプターを有効にしてから最新のパッケージにアップデートする。

```
# ifup enp0s3

# ifup enp0s8

# yum -y update
```

(2) ＩＰアドレスを確認し、 Tera Term から ssh 接続を行う。

```
# ip addr list
```

問題なく接続できれば、ここから先は Tera Term から操作できる。

(3) 必要なパッケージを追加でインストールする。

```
# yum -y install wget make gcc ncurses-devel bc xz      (カーネルのコンパイルに必要なパッケージ)

# yum -y install syslinux tftp-server dhcp              ( PXE ブートに必要なパッケージ)

# yum -y install epel-release                           ( sl パッケージをインストールするのに必要なパッケージ)

# yum -y install sl                                     ( sl コマンドを使うのに必要なパッケージ)
```

(4) initramfs 内のファイルを保持するためのディレクトリを作成し、その中に sl コマンドを繰り返し実行するだけの init コマンドを配置する。

```
$ mkdir -m 700 /var/tmp/slboot

$ cd /var/tmp/slboot/

$ cc -Wall -O2 -o init -x c - << "EOF"

#include <unistd.h>

#include <sys/wait.h>

extern char **environ;

int main(int argc, char *argv[])

{

    char *args[2] = { "/bin/sl", NULL };

    while (1) {

        switch (fork()) {

        case 0:

            execve(args[0], args, environ);

            _exit(1);

        case -1:

            break;

        }

        wait(NULL);

        sleep(1);
```

```
    }
    return 0;
}
EOF
```

(5) init コマンドと sl コマンドの実行に必要なファイルをコピーする。

```
$ cd /var/tmp/slboot/
$ mkdir -p bin dev lib64 usr/share/terminfo/l
$ cp -a /usr/bin/sl bin/
$ su -c 'mknod -m 600 dev/console c 5 1'
$ cp -a -L /lib64/libncurses.so.5 lib64/
$ cp -a -L /lib64/libtinfo.so.5 lib64/
$ cp -a -L /lib64/libc.so.6 lib64/
$ cp -a -L /lib64/libdl.so.2 lib64/
$ cp -a -L /lib64/ld-linux-x86-64.so.2 lib64/
$ cp -a -L /usr/share/terminfo/l/linux usr/share/terminfo/l/
```

(6) 動作テストを行う。

```
$ su -c 'TERM=linux chroot /var/tmp/slboot/ /init'
```

正常に動作していることを確認できたら、 Ctrl-C で強制終了させる。強制終了させると端末設定が乱れるので、 reset コマンドで元に戻す。

```
$ reset
```

(7) 必要最小限の機能を持つカーネルを作成する。

```
$ cd
$ wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.1.3.tar.xz
$ echo '96c2c77b1c54ba01cfd8fc2d13fbf828  linux-4.1.3.tar.xz' | md5sum -c -
$ tar -Jxf linux-4.1.3.tar.xz
$ cd linux-4.1.3
$ make -s allnoconfig
$ make -s menuconfig
```

menuconfig では以下のオプションを選択する。

「 64-bit kernel 」

「 General setup 」→「 Kernel compression mode (XZ) 」

「 General setup 」→「 Initial RAM filesystem and RAM disk (initramfs/initrd) support 」

「 General setup 」→「 Optimize for size 」

「 Executable file formats / Emulations 」→「 Kernel support for ELF binaries 」

「 Device Drivers 」→「 Character devices 」→「 Enable TTY 」

選択した際に自動的に選択された他のオプションは、原則としてそのまま残しておく。ただし「 General setup 」→ 「 Support initial ramdisks compressed using 〜 」は全て選択解除する。

「 General setup 」→「 Initramfs source file(s) 」には、カーネル内に埋め込む initramfs の内容が含まれるディレクトリ（ /var/tmp/slboot/ ）を指定する。

変更内容を保存して menuconfig を終了し、カーネルをビルドする。

```
$ make -s
```

ビルドにより作成された bzImage を TFTP のダウンロードディレクトリへコピーする。 initramfs の内容は bzImage 内に埋め込まれているので、 bzImage のみでよい。

```
$ su -c 'cp arch/x86/boot/bzImage /var/lib/tftpboot/'
```

(8) PXE でブートするのに必要な DHCP と TFTP の設定を行う。

```
# cp -p /usr/share/syslinux/pxelinux.0 /var/lib/tftpboot/

# mkdir /var/lib/tftpboot/pxelinux.cfg

# echo 'default bzImage' > /var/lib/tftpboot/pxelinux.cfg/default

# sed -i -e 's/disable.*/disable = no/' /etc/xinetd.d/tftp

# cat > /etc/dhcp/dhcpd.conf << EOF

subnet  10.0.2.0 netmask 255.255.255.0 {

    range dynamic-bootp  10.0.2.128 10.0.2.254;

    next-server 10.0.2.4;

    filename "/pxelinux.0";

}

EOF
```

```
# firewall-cmd --permanent --add-service=tftp

# firewall-cmd --reload

# systemctl start xinetd

# systemctl start dhcpd
```

(9) メモリ 128MB でゲストを新規作成し、 PXE でゲストをブートさせる。

成功すると、いつまでも sl コマンドが繰り返し実行される。シャットダウンさせるには、メニューの「仮想マシン」→「閉じる」から「仮想マシンの電源オフ」を選択する。

■６■ CentOS 6 環境での作成手順

(1) メモリ 1024MB でゲストを新規作成し、 CentOS-6.6-x86_64-minimal.iso を用いて最小構成でインストールし、ネットワークアダプターを有効にしてから最新のパッケージにアップデートする。

```
# ifup eth0

# ifup eth1

# yum -y update
```

(2) ＩＰアドレスを確認し、 Tera Term から ssh 接続を行う。

（■５■の (2) と同じ内容であるため省略。）

(3) 必要なパッケージを追加でインストールする。

（■５■の (3) と同じ内容であるため省略。）

(4) initramfs 内のファイルを保持するためのディレクトリを作成し、その中に sl コマンドを繰り返し実行するだけの init コマンドを配置する。

（■５■の (4) と同じ内容であるため省略。）

(5) init コマンドと sl コマンドの実行に必要なファイルをコピーする。

（■５■の (5) と同じ内容であるため省略。）

(6) 動作テストを行う。

（■５■の (6) と同じ内容であるため省略。）

(7) 必要最小限の機能を持つカーネルを作成する。

（■５■の (7) と同じ内容であるため省略。）

(8) PXE でブートするのに必要な DHCP と TFTP の設定を行う。

```
# cp -p /usr/share/syslinux/pxelinux.0 /var/lib/tftpboot/
# mkdir /var/lib/tftpboot/pxelinux.cfg
# echo 'default bzImage' > /var/lib/tftpboot/pxelinux.cfg/default
# sed -i -e 's/disable.*/disable = no/' /etc/xinetd.d/tftp
# cat > /etc/dhcp/dhcpd.conf << EOF
subnet  10.0.2.0 netmask 255.255.255.0 {
    range dynamic-bootp  10.0.2.128 10.0.2.254;
    next-server 10.0.2.4;
    filename "/pxelinux.0";
}
EOF
# iptables -I INPUT -p udp --dport 69 -s 10.0.2.0/24 -j ACCEPT
# service xinetd start
# service dhcpd start
```

(9) メモリ 128MB でゲストを新規作成し、 PXE でゲストをブートさせる。

（■５■の (9) と同じ内容であるため省略。）

■７■ Ubuntu 14.04 環境での作成手順

(1) メモリ 1024MB でゲストを新規作成し、 ubuntu-14.04.2-server-amd64.iso を用いて最小構成でインストールし、ネットワークアダプターを有効にしてから最新のパッケージにアップデートする。

```
$ sudo sh -c 'echo "iface eth1 inet dhcp" >> /etc/network/interfaces'
$ sudo ifup eth0
$ sudo ifup eth1
$ sudo apt-get update
$ sudo apt-get -y dist-upgrade
```

(2) ssh 接続に必要なパッケージをインストールし、ＩＰアドレスを確認し、 Tera Term から ssh 接続を行う。

```
$ sudo apt-get -y install openssh-server
```

```
$ ip addr list
```

問題なく接続できれば、ここから先は Tera Term から操作できる。

(3) 必要なパッケージを追加でインストールする。

```
$ sudo apt-get -y install wget make gcc libncurses-dev bc xz-utils      (カーネルのコンパイルに必要なパッケージ)
```

```
$ sudo apt-get -y install syslinux tftpd-hpa isc-dhcp-server            ( PXE ブートに必要なパッケージ)
```

```
$ sudo apt-get -y install sl                                            ( sl コマンドを使うのに必要なパッケージ)
```

(4) initramfs 内のファイルを保持するためのディレクトリを作成し、その中に sl コマンドを繰り返し実行するだけの init コマンドを配置する。

（■５■の (4) と同じ内容であるため省略。）

(5) init コマンドと sl コマンドの実行に必要なファイルをコピーする。

```
$ cd /var/tmp/slboot/
```

```
$ mkdir -p bin dev lib64 lib/x86_64-linux-gnu/ lib/terminfo/l
```

```
$ cp -a /usr/games/sl bin/
```

```
$ sudo mknod -m 600 dev/console c 5 1
```

```
$ cp -a -L /lib/x86_64-linux-gnu/libncurses.so.5 lib/x86_64-linux-gnu/
```

```
$ cp -a -L /lib/x86_64-linux-gnu/libc.so.6 lib/x86_64-linux-gnu/
```

```
$ cp -a -L /lib/x86_64-linux-gnu/libdl.so.2 lib/x86_64-linux-gnu/
```

```
$ cp -a -L /lib/x86_64-linux-gnu/libtinfo.so.5 lib/x86_64-linux-gnu/
```

```
$ cp -a -L /lib64/ld-linux-x86-64.so.2 lib64/
```

```
$ cp -p /lib/terminfo/l/linux lib/terminfo/l/
```

(6) 動作テストを行う。

（以下の１か所を除いて■５■の (6) と同じ内容であるため省略。）

CentOS 7 の場合→　　$ su -c 'TERM=linux chroot /var/tmp/slboot/ /init'

Ubuntu 14.04 の場合→　$ sudo TERM=linux chroot /var/tmp/slboot/ /init

(7) 必要最小限の機能を持つカーネルを作成する。

（以下の１か所を除いて■５■の (7) と同じ内容であるため省略。）

CentOS 7 の場合→　$ su -c 'cp arch/x86/boot/bzImage /var/lib/tftpboot/'

Ubuntu 14.04 の場合→　$ sudo cp arch/x86/boot/bzImage /var/lib/tftpboot/

(8) PXE でブートするのに必要な DHCP と TFTP の設定を行う。

```
$ sudo cp -p /usr/lib/syslinux/pxelinux.0 /var/lib/tftpboot/
$ sudo mkdir /var/lib/tftpboot/pxelinux.cfg
$ sudo sh -c 'cat > /var/lib/tftpboot/pxelinux.cfg/default' << EOF
default sl-linux
label sl-linux
kernel /bzImage
EOF
$ sudo sh -c 'cat > /etc/dhcp/dhcpd.conf' << EOF
subnet  10.0.2.0 netmask 255.255.255.0 {
    range dynamic-bootp  10.0.2.128 10.0.2.254;
    next-server 10.0.2.4;
    filename "/pxelinux.0";
}
EOF
$ sudo ufw allow proto udp from 10.0.2.0/24 to 10.0.2.4 port 69
$ sudo ufw allow proto tcp from 192.168.56.0/24 to 192.168.56.101 port 22
$ sudo ufw enable
$ sudo service isc-dhcp-server start
```

(9) メモリ 128MB でゲストを新規作成し、 PXE でゲストをブートさせる。

（■５■の (9) と同じ内容であるため省略。）

■８■うまくいきましたか？

さて、ここでクイズです。■５■の中でさりげなく「 (5) init コマンドと sl コマンドの実行に必要なファイルをコピーする。」と書かれていますが、これらのプログラムの動作に必要なファイルはどのようにすれば知ることができるでしょうか？

最初に思いつくのは、オープンされるファイルを strace コマンドで追跡する方法だと思います。ということで、準備体操として、入門書レベルのＣ言語のプログラムを幾つか作成し、 strace コマンドで調査していただきたいと思います。

■９■プログラム例１：メッセージを表示する。

```
--------- my_printf.c　ここから ---------
#include <stdio.h>
extern char **environ;

int main(int argc, char *argv[])
{
    while (*environ)
        printf("%s¥n", *environ++);
    return 0;
}
--------- my_printf.c　ここまで ---------

$ gcc -Wall -O3 -o my_printf my_printf.c
$ ./my_printf
$ strace -o my_printf.log ./my_printf
$ cat my_printf.log
```

■１０■プログラム例２：他のプログラムを実行する。

--------- my_system.c　ここから ---------

```
#include <stdlib.h>

int main(int argc, char *argv[])
{
    system("/usr/bin/env");
    return 0;
}
```

--------- my_system.c　ここまで ---------

```
$ gcc -Wall -O3 -o my_system my_system.c
$ ./my_system
$ strace -o my_system.log -f ./my_system
$ cat my_system.log
```

■１１■プログラム例３：子プロセスを作る。

--------- my_fork.c　ここから ---------

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    switch(fork()) {
    case 0:
```

```
        printf("I am child.¥n");
        fflush(stdout);
        _exit(0);
    case EOF:
        printf("fork() failed.¥n");
        break;
    default:
        wait(NULL);
        printf("I am parent.¥n");
    }
    return 0;
}
```

--------- my_fork.c　ここまで ---------

```
$ gcc -Wall -O3 -o my_fork my_fork.c
$ ./my_fork
$ strace -o my_fork.log -f ./my_fork
$ cat my_fork.log
```

■１２■プログラム例４：異なる方法で他のプログラムを実行する。

--------- my_execve.c　ここから ---------

```
#include <unistd.h>
extern char **environ;

int main(int argc, char *argv[])
{
```

```
    char *args[2] = { "/usr/bin/env", NULL };
    execve(args[0], args, environ);
    return 0;
}
```

--------- my_execve.c　ここまで ---------

```
$ gcc -Wall -O3 -o my_execve my_execve.c
$ ./my_execve
$ strace -o my_execve.log ./my_execve
$ cat my_execve.log
```

■１３■プログラム例５：ファイルの中身を表示する。

--------- my_cat.c　ここから ---------

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    char buffer[4096];
    int fd = open(argv[1], O_RDONLY);
    while (1) {
        int len = read(fd, buffer, sizeof(buffer));
        if (len <= 0 || write(1, buffer, len) != len)
            break;
```

```
    }

    return 0;

}

---------- my_cat.c  ここまで ----------


$ gcc -Wall -O3 -o my_cat my_cat.c

$ ./my_cat /etc/issue

$ strace -o my_cat.log ./my_cat /etc/issue

$ cat my_cat.log
```

■１４■ログの内容を比較する。

my_system.log と my_fork.log と my_execve.log の内容を比較することで、どのような共通点があるかを考えてみましょう。

system() ライブラリ関数のマニュアルページ（ http://linuxjm.osdn.jp/html/LDP_man-pages/man3/system.3.html ）も参考に比較すると、 system() ライブラリ関数は fork() システムコール（ http://linuxjm.osdn.jp/html/LDP_man-pages/man2/fork.2.html ）と execve() システムコール（ http://linuxjm.osdn.jp/html/LDP_man-pages/man2/execve.2.html ）を組み合わせて実現しているという点に気付く筈です。

この fork() と execve() というのが、 Linux などの Unix 系ＯＳの根幹をなす機能です。シェルのコマンドラインから何かのプログラムを実行する場合にも、 fork() と execve() が使われています。

■１５■「ちょっと不便だなぁ」と思うことはありませんか？

strace コマンドはシステムコール呼び出しを追跡するのに対して、 ltrace コマンドはライブラリ関数呼び出しを追跡します。

```
$ strace -o strace.log -f ./my_system

$ cat strace.log

$ ltrace -o ltrace.log -f ./my_system
```

```
$ cat ltrace.log
```

でも、どちらの方法にしても、「動作に必要なファイル」を知りたいときに、絶対パス名で取得できないというのは不便ですよね？また、 strace や ltrace は、 setuid されたプログラムの挙動に影響を与えてしまうなど、そもそも調査に使えないというケースもあります。

■１６■だから、このハンズオンがあるのです。

TOMOYO / AKARI / CaitSith では、パス名を絶対パス名で取得するため、どのようなファイルにアクセスしているかを調査する際に重宝します。カーネル内で動作するため、プログラムの動作に影響を与えることもありません。さらに、 fork() と execve() を活用してプログラムがどのような経緯で実行されたのかを分類するため、 Linux システムの挙動を理解する際に重宝します。

TOMOYO / AKARI / CaitSith 以外の方法としては、 System Call Auditing 機能を使って追跡する autrace とか TaskTracker とか、任意の箇所にフックを挿入して追跡する SystemTap とか、いろいろ紹介したい方法はありますが、このハンズオンでは触れない予定です。

http://www.intellilink.co.jp/article/column/oss/index.html

■１７■このハンズオンで取り扱う範囲について

Linux カーネルが関知できる範囲（≒システムコールの範囲）です。

・ HTML 内のスクリプティングバグは扱いません。

・ファームウェアレベルのマルウェアも扱いません。

システムの中身を理解することは難しいです。

・アプリケーションの視点とＯＳ（システムコール）の視点という違い、あるいは、利用する側の視点と提供する側の視点という違いがあるためです。

なぜ理解することが必要なのでしょうか？

・セキュリティ／非セキュリティどちらにも関係するからです。

・システム構築の自動化が注目される中、あえて手作りによるオーダーメイドにこだわります。

このハンズオンの前半では「セキュリティとは関係ないトラブルの予防のためにシステムの挙動を解析する方法」を扱い、後半では「セキュリティに関係するトラブルの予防のためにシステムの挙動を制限する方法」を扱う予定です。

■１８■ Shell Shock 脆弱性（ CVE-2014-6271 ）を覚えていますか？

とっても怖かったですよね？環境変数がコマンドラインとして解釈されるだなんて予想していましたか？

オーダーメイドにこだわる TOMOYO / AKARI / CaitSith は、ＯＳコマンドインジェクション対策にも威力を発揮します。２０１１年度と２０１２年度のキャンプで使用した資料が以下の場所にありますので、事前に目を通していただけると理解が深まると思います。

http://I-love.SAKURA.ne.jp/tomoyo/spc2011-kumaneko.pdf

http://I-love.SAKURA.ne.jp/tomoyo/CaitSith-ja.pdf

TOMOYO に関する様々な資料は以下の場所で公開されています。

http://osdn.jp/projects/tomoyo/docs/?category_id=532&language_id=2 （日本語）

http://osdn.jp/projects/tomoyo/docs/?category_id=532&language_id=1 （英語）

http://osdn.jp/projects/tomoyo/docs/?category_id=531&language_id=2 （日本語）

http://osdn.jp/projects/tomoyo/docs/?category_id=531&language_id=1 （英語）

以上で、事前学習資料は終わりです。

==========

セキュリティ・キャンプ２０１５ 解析トラック TOMOYO / AKARI / CaitSith ハンズオン（当日学習資料）

熊猫さくら

==========

■１９■このハンズオンで使用する環境について

アダプター１に「 NAT 」を割り当てる構成で作成し、ホストのＩＰアドレスのポート２２２２番にＴＣＰで接続するとゲストのポート２２番に転送されるようにホスト側のポートフォワーディング設定を行った状態の仮想マシンイメージを使用します。（仮想マシンイメージを複製元ＰＣ側に仕込むことにより複製しているため、同じＭＡＣアドレスを持つゲストが複数存在している状態ですので、そのままではホスト側のＤＨＣＰネットワークに接続することができないという問題を回避するためです。）ユーザは root と user の２つを作成してあり、ログインパスワードは sc2015 です。なお、■２９■では他のＰＣからのアクセスを受け付けますので、ホスト側のＩＰアドレスを確認しておいてください。

TOMOYO / AKARI / CaitSith は Linux カーネルの中で動作するため、カーネルをコンパイルする必要があります。しかし、コンパイルする時間が勿体ないため、事前にコンパイル済みのものを使用します。

CUI での操作やコンパイル手順などは事前学習資料で習得済みであるという前提で話が進みますのでご了承ください。

■２０■ AKARI による簡易プロファイル方法について習得する。

TOMOYO とは、 Linux システムの挙動を実測して SELinux 向けのアクセス制御設定を生成しようとして失敗したことから生まれたアクセス制御機能です。 AKARI とは TOMOYO の機能の一部をＬＳＭ（ Linux Security Modules ）を使ったローダブルカーネルモジュールとして切り出したものです。

TOMOYO の動作と AKARI の動作の間には差異が存在していますが、どちらもシステムの挙動を実測に基づき理解する用途で使うことができます。 TOMOYO の話は後で登場しますので、ここでは AKARI を通じて、カーネルモジュールを使った解析方法について体験してください。 http://akari.osdn.jp/1.0/index.html の Chapter 1 から Chapter 4 までが、解析目的で利用する上で必要となる説明です。

今回は、■８■にあるように、動作に必要なファイルが何であるかを把握したいので、普段は省略してしまう部分を省略せずに行ってみようと思います。 Chapter 3 でインストール手順が説明されていますが、今回は

共有ライブラリファイルなども追跡したいので、仮想マシンイメージの中では init_policy コマンドの実行後に以下のコマンドを実行しています。

```
# echo 'aggregator proc:/self/exe /proc/self/exe' > /etc/ccs/exception_policy.conf
# echo 'use_profile 1' >> /etc/ccs/domain_policy.conf
```

それ以外はオンラインマニュアルで説明されている手順を使用して設定を行った状態の仮想マシンを用意しています。

仮想マシン（ CentOS 6 / CentOS 7 / Ubuntu 14.04 いずれでも構いません）を起動すると AKARI が有効な状態で起動しますので、事前学習資料で実行したコマンドを実行してみてください。

解析結果の確認には、ポリシーエディタというプログラムを使用します。使い方の説明は http://akari.osdn.jp/1.0/tool-editpolicy.html にあります。

実行したコマンドのパス名に一致する行を Domain Transition Editor 画面の中から探してください。そして、その行にカーソルを合わせた状態で Domain Policy Editor 画面を開くと、以下のようなキーワードで始まる行が見つかると思います。

```
file read ・・・読み込みモードでのオープン（ライブラリや設定ファイルなど）
file execute ・・・プログラムの実行（処理の引継ぎ）
```

自動学習機能では最初のコマンドライン引数だけしか記録されませんが、アクセスログの方には全てのコマンドライン引数および環境変数の内容についても（合計で８ＫＢを上限として）記録されています。

単純なプログラムの場合、「 file read あるいは file execute で始まる行で指定されているパス名のファイル」がそのまま「プログラムの動作に必要なファイル」になります。事前学習資料で使用した init コマンドや sl コマンドは、そのようなプログラムの例です。（なお、 initramfs 内には /dev/console も含まれていますが、カーネルが init コマンドを実行する前にオープンするファイルであるため、 init コマンドの実行時に開始される AKARI では追跡することができません。また、 file execute 以外はシンボリックリンクを解決したパス名で記録されるため、 cp コマンドに指定するライブラリのパス名とは異なっています。）

順調に進んだ場合には、他のプログラムについても解析してみましょう。例えば、ファイルへの書き込み／追記を行う必要があるプログラムを解析すると、以下のようなキーワードで始まる行が見つかるかもしれません。

file write ・・・書き込みモードでのオープン（データファイルなど）

file append ・・・追記モードでのオープン（ログファイルなど）

また、 TCP/IP ソケットを使ったネットワーク通信をしている場合、以下のようなキーワードで始まる行が見つかるかもしれません。

network inet stream connect ・・・接続要求の送信先ＩＰアドレスおよびポート

network inet stream accept ・・・接続要求の送信元ＩＰアドレスおよびポート

AKARI は、 strace コマンドで追跡可能な情報の内、ファイルの読み書き実行や TCP/IP 通信などのログを、「プログラム単位で分類して取得」してくれるので、システムのおおよその挙動を把握する上で役に立ちます。

■２１■システムの挙動を理解しておくと嬉しい例について考える。

熊猫はＳＩ業界に所属しており、サポートセンターに３年間居たのですが、技術的なセキュリティ以前の問題に振り回され続けていました。

例１：問題発生時にどこをチェックすればよいかを把握しておかないと、障害発生時に対処できない。

例２：どこでどのように使われているかを把握しておかないと、ソフトウェアのエラータ適用の要否や影響範囲について考えることができない。

AKARI によるプロファイルを行うことで、見当がつくようになります。最終的に SELinux を導入する場合であっても、無駄にはなりません。

■２２■ TOMOYO について触れる。

セキュリティ＆プログラミングキャンプ２０１１の資料を用いて、 TOMOYO の歴史の話をします。その中で、TOMOYO によるシステム全体の詳細プロファイル方法についても習得します。

http://I-love.SAKURA.ne.jp/tomoyo/spc2011-kumaneko.pdf

ディストリビューションやバージョンの違いによる差異について知ることができるよう、 CentOS 6 / CentOS 7 / Ubuntu 14.04 という３つの環境で試すことができるようにしてあります。

このハンズオンではフル機能版である TOMOYO 1.8 を使いますが、あなたが Ubuntu のユーザならメインライン版である TOMOYO 2.5 を使うこともできます。また、 CentOS Plus カーネルを容認できるなら、 CentOS 6 で TOMOYO 2.2 を、 CentOS 7 で TOMOYO 2.5 を使うことができます。

■２３■特定の機能だけを使う方法について触れる。

TOMOYO / AKARI は、デフォルトでは全ての機能が使えるプロファイルを作成しますが、必要とする機能は利用者のスキルや用途によって異なる場合があります。全ての機能を使いこなせればそれだけ安全になるのでしょうが、そこまでの労力をかけることを利用者に強要する訳にはいきません。そのため、スキルや用途に応じて特定の機能だけを使えるようになっています。

例えば、 /etc/ccs/profile.conf の

```
1-CONFIG={ mode=learning grant_log=no reject_log=yes }
```

という行を

```
1-CONFIG={ mode=disabled grant_log=no reject_log=yes }
1-CONFIG::file={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::getattr={ mode=disabled grant_log=no reject_log=yes }
1-CONFIG::file::ioctl={ mode=disabled grant_log=no reject_log=yes }
```

あるいは

```
1-CONFIG={ mode=disabled grant_log=no reject_log=yes }
1-CONFIG::file::execute={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::open={ mode=learning grant_log=no reject_log=yes }
```

```
1-CONFIG::file::create={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::unlink={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::getattr={ mode=disabled grant_log=no reject_log=yes }
1-CONFIG::file::mkdir={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::rmdir={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::mkfifo={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::mksock={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::truncate={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::symlink={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::mkblock={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::mkchar={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::link={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::rename={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::chmod={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::chown={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::chgrp={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::ioctl={ mode=disabled grant_log=no reject_log=yes }
1-CONFIG::file::chroot={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::mount={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::unmount={ mode=learning grant_log=no reject_log=yes }
1-CONFIG::file::pivot_root={ mode=learning grant_log=no reject_log=yes }
```

のように書き換えて再読み込みすると、ファイルに関する操作の内、属性の取得（ getattr ）と入出力制御（ ioctl ）以外の操作についてのみ追跡するようになります。また、 /etc/ccs/exception_policy.conf で

```
acl_group 255 file read /
acl_group 255 file read /¥{¥*¥}/
acl_group 255 file read ¥*:/
```

acl_group 255 file read ¥*:/¥{¥**¥}/

という指定を行い、 /etc/ccs/domain_policy.conf の各ドメインに対して

use_group 255

という指定を行ってから再読み込みすることにより、ディレクトリエントリの読み込み（ readdir ）については常に許可することができるようになります。

一気にいろんなキーワードが出てきましたね。 ioctl とか言われても何のことか判りませんよね？これらのキーワードは、 Linux におけるシステムコールを識別するために TOMOYO / AKARI が使用しているキーワードです。どんなシステムコールがあって、どんな操作ができるのかについて勉強すると、何のことか判るようになります。■８■から■１５■にかけて行った、 strace コマンドを用いたシステムコールの追跡とも関係しています。

・・・難しい？ご尤もです。それには、次に登場する話が関係しています。

■２４■カーネル内アクセス制御の難しさについて触れる。

よく知られた攻撃手法であるシェルコードの場合、シェルコード自身は通常の実行可能ファイルのような姿を持たないため、標的の環境に潜り込むためにはバッファオーバーフローというプログラム的な脆弱性を突く必要があります。そして、ＮＸビットやアドレスのランダム化などを用いて攻撃の成功確率を低減する防御手法の研究が進み、簡単には成功しないようになってきています。幸い、プログラムの提供者側が面倒を見てくれるため、利用者側が意識する必要もありません。なんとなく知っていれば導入することができます。

しかし、標的型攻撃などで使われるマルウェアの場合は、マルウェア自身が通常の実行可能ファイルの姿を持つことができるため、ブラウザからのダウンロードやメールの添付ファイルなどの形で標的の環境に潜り込むことができてしまいます。そして、ダブルクリックする（「実行しろ」と利用者が命じる）ことで、シェルコードのようにプログラム的な脆弱性を突かなくても、通常の実行可能ファイルのように何でもできてしまいます。これを防ぐために、カーネル側で「本当に許可して良いの？」という判断を行うカーネル内アクセス制御が研究されており、 Linux においては SELinux や AppArmor などの実装が有名です。

「必要なことだけができて、不要なことは一切させない」というアクセス制御規則を設定することができれば、アクセス制御である程度は防御できるでしょう。しかし、カーネル内アクセス制御の提供者側はアクセス制御の仕組みを提供するところまでしか面倒を見ることができないため、利用者側が「何が必要で何が不要か」という内容を理解して適切に設定しなければ導入することができません。利用者側がシステムの動作をシステムコールレベルで理解することを要求されてしまうため、難しすぎて人気がありません。

実際、アクセス制御は、日本で Linux カーネル向けのアクセス制御機能の開発に関わった人たちが全員（熊猫を含めて）社内失業したという実績があるくらい、人気が無く、商売にもならない大変な領域なのです。この状況を打破できる、「利用者が理解できるセキュリティ」のチャレンジャーを募集中です！

熊猫は TOMOYO / AKARI / CaitSith の開発を通じて、アクセス制御の難しさについて感じてきました。利用者側の理解と判断を必要とするアクセス制御機能は、論文や特許のように新規性を評価軸とする研究向けの実装、コミュニティにとってメリットがあることを評価軸とするメインライン向けの実装、特定の人たちにとってのメリットを評価軸とする実用向けの実装など、１つの物差しでは対応しきれないものなのだと思います。

試行錯誤を続ける中、少しでも敷居を下げられないかと考えて思いっきり路線を転換したのが、次に登場する CaitSith です。

■２５■ CaitSith について触れる。

セキュリティ・キャンプ２０１２の資料を用いて、 CaitSith の話をします。その中で、 CaitSith による特定の操作の記録方法についても習得します。 CaitSith は、デフォルトでは何もしません。そのため、TOMOYO や AKARI は全てのプロセスに対して全ての操作を追跡／制限する用途に適しているのに対し、CaitSith は特定のプロセスや特定の操作だけを追跡／制限する用途に適している筈です。

http://I-love.SAKURA.ne.jp/tomoyo/CaitSith-ja.pdf

例えば、以下のようにネットワーク通信のみを監視するルールを作成することで、 TCP/IP ネットワーク通信の状況をログに取得し、どのプログラムがどのＩＰアドレス／ポート番号と通信をしているのかを把握するという使い方ができます。

--------- /etc/caitsith/policy/current の指定例　ここから ---------

```
POLICY_VERSION=20120401

quota audit[255] allowed=0 unmatched=1024 denied=0
```

0 acl inet_stream_bind

    audit 255

0 acl inet_stream_listen

    audit 255

0 acl inet_stream_connect

    audit 255

0 acl inet_stream_accept

    audit 255

0 acl inet_dgram_bind

    audit 255

0 acl inet_dgram_send

    audit 255

0 acl inet_dgram_recv

    audit 255

0 acl inet_raw_bind

    audit 255

0 acl inet_raw_send

    audit 255

```
0 acl inet_raw_recv

    audit 255
```

---------- /etc/caitsith/policy/current の指定例　ここまで ----------

「 iptables コマンドで netfilter の設定するのと何が違うんだ？」と思われることでしょう。送信元／送信先のＩＰアドレス／ポート番号を制限するだけであれば、 netfilter を使う方が効率的です。しかし、 TOMOYO / AKARI / CaitSith では、「どのプログラムが」という要素を考慮することができます。つまり、特定のポート番号を利用できるプログラムやユーザＩＤなどを、通信を許可する際の条件として指定できるのです。これは、想定外のプログラムが勝手に不審なサイトとの通信を行っていないかどうかを確認する上で役に立つことでしょう。

■２６■特定の操作や資源の制限を行う方法について習得する。

ここから先は、アクセスを制限する方法について学びます。システムがどのような振る舞いをしていて、どのようなアクセス要求を許可する必要があるのかは、現在までに判明している筈ですので、ゴールは近いです。具体的には、 TOMOYO / AKARI ではドメインに対して強制モード用のプロファイルを割り当てて動かし、 CaitSith ではルールの最後に deny 行を追加して動かします。

■２７■シェルセッションの操作制限を行う方法について習得する。

「シェルを与える」ってどういうイメージですか？やりたい放題されてしまうので怖い？即ゲームオーバー？シェルを与えるつもりが無くても、バッファオーバーフローやＯＳコマンドインジェクションによってシェルを奪われてしまうこともありますね？

ですから、普通に考えればシェルを実行されることは怖いです。しかし、カーネル内でアクセス制限をしていればダメージを減らすことができます。そして、オーダーメイドにこだわる TOMOYO / AKARI / CaitSith はＯＳコマンドインジェクション／ Shell Shock （ CVE-2014-6271 ） 対策が得意です。ここでは、応募問題で使用した脆弱ログインシェルを、 TOMOYO / AKARI / CaitSith のいずれかを使用して防御します。（脆弱性を突くことができるように、ここでは CentOS 6.5 の仮想マシンイメージを使用します。）

■２８■「セキュリティ・キャンプ全国大会 2015 応募用紙」の「選択問題 12 」を解く。

CentOS 6.5 リリース時点のパッケージを使用して構築された CentOS 6 サーバがあります。ユーザ test のログインシェルには、特定のディレクトリ内のファイルを scp を用いてダウンロードできるようにすることを意図した、以下に示すログインシェルが使われています。このサーバのホスト名を shell.scamp2015.com とし、 /etc/ssh/sshd_config の内容がデフォルト設定のままであると仮定して、このログインシェルの脆弱性と、このログインシェルから起動されるプログラムの脆弱性の両方を突いて、書き込み可能なディレクトリ（好きな場所を仮定してよい）の中に任意のファイルをアップロードする手順を、コマンドラインを交えながら解説してください。

--------- ログインシェルのソースコード　ここから ---------

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <wordexp.h>

int main(int argc, char *argv[])
{
    int i;
    int err = EINVAL;
    wordexp_t p;
    char *w;
    static char *args[1024] = { };
    if (argc != 3 || strcmp(argv[1], "-c")) {
        fprintf(stderr, "You are not permitted to login.¥n");
        return 1;
    }
    w = argv[2];
    if (strncmp(w, "scp -f ", 7) ||
        strspn(w + 7, "abcdefghijklmnopqrstuvwxyzO123456789./+-_$[{()}]=`?*'") != strlen(w + 7) ||
```

```
        wordexp(w, &p, WRDE_NOCMD) || p.we_wordc < 3 || p.we_wordc > 1023)
        goto out;
    for (i = 0; i < p.we_wordc; i++) {
        w = p.we_wordv[i];
        if (strncmp(w, "/home/", 6) && strncmp(w, "/var/ftp/", 9) &&
            strncmp(w, "/var/www/html/public/", 21) && i >= 2)
            goto out;
        args[i] = w;
    }
    execv("/usr/bin/scp", args);
    err = errno;
out:
    fprintf(stderr, "%s : %s¥n", argv[2], strerror(err));
    return 1;
}
```

--------- ログインシェルのソースコード　ここまで ---------

正答例：

```
  LANG='() { :; }; /bin/echo Hello > /home/test/file' scp
test@shell.scamp2015.com:'/home/test/$((`.`))' .
```

解説：

scp コマンドの隠しオプションである -f と、 $$((``)) を用いて bash や python などの標準入力からコマンドラインを読み込めるインタプリタを起動することで攻撃する（ ssh -y test@shell.scamp2015.com 'scp -f /home/$((1`bash`))' ）という回答が複数ありましたが、これだけでは「両方を突いて」という注文に応えることができません。

実は、 sshd_config がデフォルト設定であることから、 LANG などの環境変数を渡すことができるようになっているので、標準入力からのコマンドラインを読み込めない scp コマンドであっても、注文に応えることができるようになっています。

・ CVE-2014-7817 glibc: command execution in wordexp() with WRDE_NOCMD specified

/home/test/$((`.`)) のようなパス名を指定することで、 /bin/sh を起動できる。

・ CVE-2014-6271 bash: specially-crafted environment variables can be used to inject shell commands

LANG などの細工した環境変数を指定することで、 /bin/sh の起動時に任意のコマンドを実行できる。

このログインシェルにはディレクトリトラバーサル攻撃が可能な脆弱性もあるのですが、これは「ディレクトリトラバーサルを活用して、 root ユーザ権限を奪取できないか」という方向に誘導するための罠でした。この問題を解くのには関係ありません。

では、 TOMOYO / AKARI / CaitSith のいずれかを使用して防御してみましょう。

このログインシェルが /bin/scp-checker というパス名であると仮定すると、 /bin/scp-checker に対しては /usr/bin/scp の実行だけを許可し、 /usr/bin/scp に対しては /home/ または /var/ftp/ または /var/www/html/public/ で始まるパス名を持つファイルを読み込みモードでオープンする許可を与えればよいことになります。

TOMOYO または AKARI の場合、おおよそ以下のような指定をすることになります。

--------- /etc/ccs/domain_policy.conf での指定例　ここから ---------

<kernel> /usr/sbin/sshd /bin/scp-checker

use_profile 3

file execute /usr/bin/scp

<kernel> /usr/sbin/sshd /bin/scp-checker /usr/bin/scp

use_profile 3

file read /home/¥*

file read /home/¥{¥*¥}/¥*

file read /var/ftp/¥*

file read /var/ftp/¥{¥*¥}/¥*

file read /var/www/html/public/¥*

```
file read /var/www/html/public/¥{¥*¥}/¥*
```

---------- /etc/ccs/domain_policy.conf での指定例　ここまで ----------

CaitSith の場合、おおよそ以下のような指定をすることになります。

```
---------- /etc/caitsith/policy/current での指定例　ここから ----------
0 acl execute task.exe="/bin/scp-checker"
  10 allow path="/usr/bin/scp" transition="<scp-checker/scp>"
  20 deny

10000 acl read task.domain="<scp-checker/scp>"
  10 allow path="/home/¥(¥*¥)/¥*"
  10 allow path="/var/ftp/¥(¥*¥)/¥*"
  10 allow path="/var/www/html/public/¥(¥*¥)/¥*"
  20 deny
```

---------- /etc/caitsith/policy/current での指定例　ここまで ----------

「 TOMOYO / AKARI / CaitSith でアクセスできるパス名を制限できるのなら、そもそもこの脆弱ログインシェルを使う必要は無いのでは？」と思われるかもしれません。実際には、上記の他に、共有ライブラリファイルや設定ファイルなどへのアクセスも許可する必要があります。カーネル側ではディレクトリトラバーサル攻撃かどうかを判断することはできませんので、 scp コマンドに対して /etc/passwd へのアクセスを認めていれば、ディレクトリトラバーサル攻撃により /etc/passwd をダウンロードされてしまう可能性は残ります。それを防ぐには、脆弱ログインシェル側でディレクトリトラバーサル攻撃ができないように正しく検査してやる必要があります。

■２９■ケロちゃんチェックについて習得する。

先の脆弱ログインシェルの例では /usr/sbin/sshd と /usr/bin/scp との間に /bin/scp-checker が入ることで、 /bin/scp-checker が「これから実行される scp は、ダウンロード専用ですよ」という意思表示をする役割を担っていました。例えば、ログインシェルとして /bin/bash を使っているユーザが scp コマンド経由で ssh 接続してきた場合、 /usr/sbin/sshd と /usr/bin/scp との間に /bin/bash が入ることで、「これか

ら実行される scp では、アップロードも許可しますよ」という意思表示する役割を担わせることも可能である訳です。

あるプログラムが実行されるまでの間に、どのようなプログラムが実行されたかを fork() / execve() の仕組みを活用して追跡する TOMOYO / AKARI では、それぞれのプログラムに役割を与えることができます。

では、 /usr/sbin/sshd と（任意のコマンドを実行できる）シェルとの間に、他のプログラムが入ったらどういうことが起こるでしょうか？そして、プログラムに、ユーザ認証を行う機能を持たせたら、どんなことが可能になるでしょうか？

ということで、このハンズオンの最後の演習として、シェルセッションの操作制限の応用である、ログイン認証の多重化を体験してみましょう。こちらからプログラムを提示するのでは面白くないので、受講者自らがアイデアを出して実装していただきます。そして、他の受講者とログインパスワードを共有することで、他の受講者からの SSH アクセスを受け入れてもらい、防衛に成功するかどうかを競っていただきます。

参考： http://osdn.jp/projects/tomoyo/docs/winf2005.pdf

http://osdn.jp/projects/tomoyo/docs/PacSec2008-ja.pdf

http://kumaneko-sakura.sblo.jp/article/21959204.html

ccs-tools ソース tar ball 内にあるサンプルプログラムなど

■３０■最後に

このハンズオンでは、カーネル内でのアクセス解析／アクセス制御という領域について扱いました。

コンピュータシステムに限らず、セキュリティが大変なことになってきています。セキュリティ・キャンプ２０１３と２０１４では、アクセス制御の話から離れて、「セキュリティは誰のためにあるのか？」を考えました。あなたが持っているその技術力で何をするのかを考えていただけたら幸いです。

http://I-love.SAKURA.ne.jp/tomoyo/sc2013-kumaneko.pdf

http://I-love.SAKURA.ne.jp/tomoyo/sc2014-kumaneko.pdf