# ワークショップ第1回
# 多種アーキテクチャでの攻撃と防御

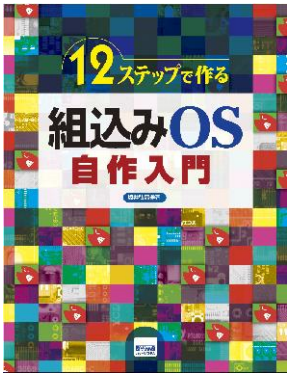
坂井弘亮

# 自己紹介

坂井弘亮(さかい・ひろあき)
http://kozos.jp/
個人でいろんな活動をしています

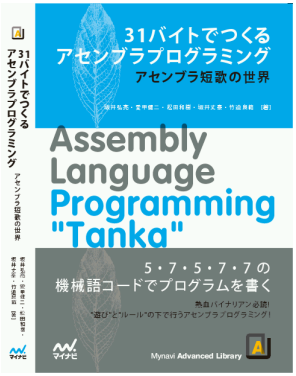
- 組込みOS自作(KOZOSプロジェクト)
- イベントへの出展・セミナーなど
  (オープンソースカンファレンス(OSC)など)
- SECCONへのコミット
- 雑誌記事や書籍執筆など
- アセンブラ短歌・六歌仙のひとり(白樺派)
- 技術士(情報工学部門)

# 書籍の執筆


12ステップで作る
組込みOS
自作入門


リンカ・ローダ
実践開発テクニック


全国津々浦々！
勉強会＆イベント
探訪記　完全版
坂井弘亮
北は北海道から南は
九州まで、各地の
IT系イベントを写真
と記録で振り返る。


31バイトでつくる
アセンブラプログラミング
アセンブラ短歌の世界
Assembly
Language
Programming
"Tanka"
5・7・5・7・7の
機械語コードでプログラムを書く


0と1のコンピュータ世界
バイナリで遊ぼう!


熱血!
アセンブラ入門
やらなければ、
一生
わからん!!


ハロー
"Hello, World"
OSと標準ライブラリのシゴトとしくみ
あなたが最初に書いた
プログラムにはまだ
「知らないこと」がたくさんある。

低レイヤーまわりを中心にいろいろやってます

# セキュリティ・キャンプへの参加の経緯

- 組込み関連
  - 学習向け組込みシステム(KOZOS)の開発
  - 書籍の執筆
  - OSCなどのイベントへの出展・セミナー登壇

- セキュリティ関連
  - セキュリティ・キャンプへの参加
  - SECCONへのコミット

(経緯) 組込みOS開発者としてセキュリティ・キャンプに参加
→ セキュリティ界隈では低レイヤー技術が求められている
(組込み技術者とセキュリティ技術者のスキルセットは似ている)

# この講義について

# 概要

マルチアーキテクチャでのセキュリティを理解するために，多種アーキテクチャでの攻撃と防御の仕組み，エミュレータによる脆弱性検証，防御機能の検討とエミュレータへの追加による実験手法などを学ぶ.

# 前提知識

以下の知識を前提とします. 自身の無いかたは, ぜひ予習をお願いします.

- C言語の基礎(関数呼び出し, 配列, 構造体, ポインタ, 関数へのポインタなどを理解していること)
- シェルによるコマンド操作(講義はLinux環境で, シェルによるコマンドライン操作で説明します. lsによるファイル一覧, cdによるディレクトリ移動, cpによるファイルコピーなどできること)
- テキストエディタの操作
- バイナリエディタの操作
- アセンブラを読み書きします. アセンブラがバリバリ書けることまでは前提としませんが, 「アセンブリ言語」「機械語」「逆アセンブル」などの言葉の意味がわかること

# こんなことをやります

1. 脆弱性のあるサーバプログラムを様々なアーキテクチャ上(実際にはシミュレータ上)で動作させて, それに対する攻撃を検証します.
2. サーバプログラムの動作を解析します. デバッガ(GDB)の使いかたを知っているといいでしょう.
3. 攻撃コードを解析します. バッファオーバーフロー脆弱性の基本知識や, シェルコードの知識があるといいでしょう.
4. システムコール発行時のシミュレータの動作を, シミュレータのソースコード読んで解析します. システムコールの知識(システムコール命令, ABI)があるといいでしょう.

# キーワード

講義中，以下の言葉が出てくると思います．もちろん講義中に説明もしますが，それでも以下のような言葉が講義中にいきなり出てきたらわからなくてあせる!というかたは，事前に調べて，どんなものかなんとなくくらいは事前に知っておくといいでしょう．

- アセンブリ言語，ニーモニック，コンパイル，アセンブル，リンク，逆アセンブル，コンパイラ，アセンブラ，リンカ，逆アセンブラ
- オブジェクトファイル，実行ファイル，ELFフォーマット，標準Cライブラリ
- 機械語，命令セット，オペコード，オペランド，レジスタ，即値，アドレス，スタック，スタックポインタ，リンクレジスタ，メモリマップドI/O
- システムコール，割込み，例外，API，ABI(Application Binary Interface)，システムコール番号，システムコール命令，システムコール・ラッパー
- バッファオーバーフロー脆弱性，シェルコード

# 注意

本日実施する内容は，サーバに対する攻撃を含みます．

他人が管理しているサーバに対して無断で行ったりすると，不正アクセス禁止法に問われる可能性があります．(たとえ善意の検証目的でもNGです)

本講義の内容を復習などする場合には，必ず自身の閉じた環境内で実施してください．

本講義の目的は，本質的な防御のためには攻撃手法を知る必要があり，攻撃に対しての正しい知識を身につけ，脆弱性に対して適切な 調査・検証・検討・判断を行えるようにするという点にあります．不正な攻撃を助長するものではありません．

# 本日やること

本日の内容は，2部構成になっています．

前半 様々なアーキテクチャのアセンブリを読み，慣れる

好きなアーキテクチャをひとつ選び，そのアセンブリを(ドキュメント無しで)フィーリングで読みとく手法を 学んでいただきます (これができるようになると，たいていのアーキテクチャの アセンブリはドキュメント無しでパッと読めるようになります)

後半 様々なアーキテクチャのサーバ脆弱性を検証する

各種アーキテクチャ上で動作するようにコンパイルされた，脆弱性のあるサーバプログラムがエミュレータ上で動作しています．検証用のExploitコードも用意されていますので，脆弱性を検証してみます．

# (準備)
# 演習環境

# 演習環境

## (手持ちPCを利用する場合)

事前学習で説明したクロスコンパイル環境がビルド済みならば，手持ちPCで演習を行うことができます．

## (演習用サーバを利用する場合)

手持ちPCに環境構築していない場合には，演習用サーバにログインして 演習を行うことができます．
以下のサーバにSSHでログインしてください．

サーバ:
ポート番号:

# (準備)
# 必要ファイルのダウンロードと説明

# 必要ファイルのダウンロード

Webサーバからファイルをダウンロードしてください

(サーバ)
http://192.168.1.1/

| ファイル | 内容 |
| --- | --- |
| cross-20130826.zip | 各種アーキテクチャのアセンブリの生成環境とサンプル集 |
| bof-server.zip | 脆弱性のある，各種アーキテクチャ用のサーバプログラム |
| simple-inetd.c | 簡易inetd(検証に利用) |
| exploit.zip | 検証用コードのサンプル |

# (前半)
# 様々なアーキテクチャのアセンブリを読み，慣れる

# サンプルを展開

```
$ cd ~
$ unzip cross-20130826.zip
$ cd cross/sample
$ ls *.x
→ 各種アーキテクチャの実行ファイル
$ ls arm-elf.*
arm-elf.c        arm-elf.o        arm-elf.x
arm-elf.d        arm-elf.s
$ ls sample.c
sample.c
```

# 演習可能なアーキテクチャ一覧

| ファイル | アーキ名 | 概要 |
|---|---|---|
| arm-elf.d | ARM | スマフォでおなじみ |
| arm16-elf.d | Thumb | ARMの16ビット縮小命令 |
| cris-elf.d | CRIS | CRISという謎アーキ |
| frv-elf.d | FR-V | 富士通のマイコン. 複数命令を同時並列で実行する |
| h8300-elf.d | H8 | 高専でよく使われている国産マイコン |
| m32r-elf.d | M32R | 三菱のマイコンで2命令を同時実行したり1命令だけ実行したり |
| mcore-elf.d | M・CORE | 謎マイコン |
| mips-elf.d | MIPS | ワークステーション向けの高速アーキのはずだったが, 現在は組み込み向けが主流 |
| mips16-elf.d | MIPS16 | MIPSの16ビット縮小命令 |
| mn10300-elf.d | MN10300 | 松下のマイコン. DVDドライブとかで使われているという噂 |
| powerpc-elf.d | PowerPC | ゲーム機や旧PowerMACで使われていた |
| sh-elf.d | SH | 国産マイコン. 自動車の制御系でよく使われているとか |
| sh64-elf.d | SH64 | 名前的にはSHの64ビット化みたいだけど命令は全然違う |
| v850-elf.d | V850 | エンジン制御など |

# ファイルの説明

| ファイル | 内容 |
| --- | --- |
| sample.c | 元となるC言語サンプル |
| arm-elf.c | sample.cをコピーしたファイル |
| arm-elf.s | arm-elf.cをコンパイルして生成したアセンブリ |
| arm-elf.o | arm-elf.sをアセンブルして生成したオブジェクトファイル |
| arm-elf.x | arm-elf.oをリンクして生成した実行ファイル |
| arm-elf.d | arm-elf.xを逆アセンブルした出力 |

# サンプルの使いかた

```
$ make clean → 生成したファイルを削除（掃除）
$ make       → ファイルを再生成

(サンプルを修正し，全アーキのファイルを再生成する)
$ vi sample.c
$ make

(ARMのサンプルだけ修正してARMのファイルのみ再生成)
$ vi arm-elf.c
$ make
```

サンプルのCソースコードを自由に変更し,アセンブリを
再生成して いろいろ変化を見てみることができます

# アセンブリ言語を読んでみよう

1. まず, 読んでみたいアーキを選んでください
2. sample.cとそのアーキの*.dを見比べてみてください (左端がアドレス, 中央が機械語コード, 右端がニーモニック)
3. リターン命令を推測してください
4. 戻り値の返しかたを推測してください
5. リターン命令がへんな位置にある場合は, 遅延スロットのアーキです
6. ビット幅の広い値を代入する方法を見てみてください
7. 引数の渡しかたを推測してください
8. 演算(加算)の方法を推測してください
9. メモリの読み書きの方法を推測してください

# シミュレータによる実行

```
$ cd ~/cross/exec
$ make run

(シミュレータを直接実行)
$ /usr/local/cross/bin/arm-elf-gdb arm-elf.x
→ トレースオプションなど指定したい場合に利用
→ オプションはアーキごとに異なるので，適当に調べる
```

# GDBによる動的解析

```
$ cd ~/cross/exec
$ /usr/local/cross/bin/arm-elf-gdb arm-elf.x
(gdb) target sim
(gdb) load
(gdb) break main
→ main()の先頭にブレークポイントを設定する

(gdb) run
→ main()の先頭でブレークする

(gdb) layout asm
→ アセンブリの表示モード

(gdb) stepi
→ ステップ実行(関数呼び出しの先に入っていく)
```

# GDBコマンド

(gdb) nexti → ステップ実行(関数内は一気に実行)
(gdb) continue → 実行継続
(gdb) until → ループ終了まで一気に実行
(gdb) finish → 関数の終端まで一気に実行

(gdb) where → スタックトレースを出力
(gdb) up → 呼び出し元関数に移動
(gdb) down → 呼び出し先関数に移動
(gdb) info registers → レジスタ情報一覧

Ctrl+x - a → アセンブリの表示モードから抜ける

# (後半)
# 様々なアーキテクチャのサーバ脆弱性を検証する

# サーバプログラムを展開

```
$ cd ~
$ unzip bof-server.zip
$ cd bof-server/cross
$ ls *.x
arm-elf.x       m32r-elf.x      powerpc-elf.x
arm16-elf.x     mcore-elf.x     sh-elf.x
cris-elf.x      mips-elf.x      sh64-elf.x
frv-elf.x       mips16-elf.x    v850-elf.x
h8300-elf.x     mn10300-elf.x
→ サーバプログラムの実行ファイル

$ less sample.c
→ サーバプログラムのソース
   (makeするとサーバプログラムを各種アーキ用に
    コンパイルする)
```

# サーバプログラムをシミュレータで実行

```
$ cd ~/bof-server/cross
$ /usr/local/cross/bin/arm-elf-run arm-elf.x
This is arm-elf server.
Push Enter:

OK.
Input name:
sample
OK. Your name: sample
$
```

# 簡易inetd経由でサーバとして起動

```
$ cd ~
$ gcc simple-inetd.c -o simple-inetd -Wall
$ cd ~/bof-server/cross
$ ~/simple-inetd 10000 ¥
  /usr/local/cross/bin/arm-elf-run arm-elf-run ¥
  arm-elf.x
→ ポート10000で待ち受けする

$ telnet localhost 10000
→ ネットワーク経由でサーバプログラムに接続する
```

# 検証コードによる検証

```
$ cd ~
$ unzip exploit.zip
$ cd exploit/cross
$ make
→ 検証コードをコンパイルして生成

$ vi flag.txt
→ サーバプログラムのカレントディレクトリに
   flag.txt を作成(中身は適当に書いておく)

$ ./cat.pl enter.bin arm-elf.dat ¥
  | /usr/local/cross/bin/arm-elf-run ¥
  ~/bof-server/cross/arm-elf.x
→ flag.txtの中身が出力されてしまう！
```

# ネットワーク経由での検証

```
$ cd ~/bof-server/cross
$ vi flag.txt
  → flag.txtを適当に作成して置いておく
$ ~/simple-inetd 10000 ¥
  /usr/local/cross/bin/arm-elf-run arm-elf-run ¥
  arm-elf.x

$ cd exploit/cross
$ ./cat.pl enter.bin arm-elf.dat | nc localhost 10000
→ ネットワーク経由でflag.txtの中身が
   取得できてしまう！
```

# 検証コードを調べる

```
$ cd ~/exploit/cross
$ /usr/local/cross/bin/arm-elf-objdump -d arm-elf.x
```

→ 検証コードを逆アセンブルする

# シミュレータのシステムコール処理

```
$ cd ~/bof-server/cross
$ less lib-arm-elf.S
→ システムコールの呼び出し方法を知るための
   GDBのソースコードの参照先が，先頭にコメント
   として書いてあります

/*
 * Use SWI instruction.
 * See gdb/sim/arm/armemu.c:ARMul_Emulate32(),armos.c:ARMul_OSHandleSWI()
 * (case 0xf0:)
 * See gdb/sim/testsuite/sim/arm/hello.ms
 */

   また read や write システムコールを
   呼ぶための関数があります

        .globl  __read
        .type   __read, %function
__read:
        SWI(SWI_Read)
        mov     pc, lr
```

# システムコール処理(ARMの例)

```
$ tar xvzf gdb-7.3.1.tar.gz
$ cd gdb-7.3.1/sim/arm
$ less armos.c
→ 「ARMul_OSHandleSWI」を検索

  switch (number)
    {
    case SWI_Read:
      if (swi_mask & SWI_MASK_DEMON)
        SWIread (state, state->Reg[0], state->Reg[1], state->Reg[2]);
      else
        unhandled = TRUE;
      break;

    case SWI_Write:
      if (swi_mask & SWI_MASK_DEMON)
        SWIwrite (state, state->Reg[0], state->Reg[1], state->Reg[2]);
      else
        unhandled = TRUE;
      break;
    ...
```

# 考察

材料はすべてそろいました!
あとは自由に解析してみましょう!

- スタック構造はどのようになっているのか?
- どのような流れでシェルコードが実行されてしまっているのか?
- システムコールはどのようにして呼んでいるのか?
- システムコールが呼ばれると, シミュレータはどうするのか?

# 次回予告

- GDB内蔵のシミュレータの処理を深く見てみる
- シミュレータを改造してみる
- シミュレータを改造して，シェルコード実行を防止するための機能を追加してみる

おつかれさまでした！