ワークショップ第2回
多種アーキテクチャでの
攻撃と防御

坂井弘亮

はじめに

資料や各種サンプルは
以下から
ダウンロードできます

http://192.168.1.2:10080/

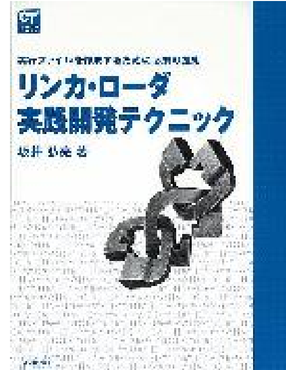
workshop2016-emulator.pdf

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)

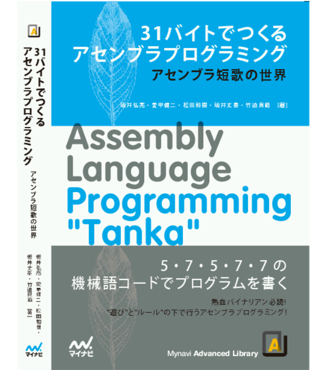
書籍の執筆



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。



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

第1回では，こんなことをやりました

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．

今回は，こんなことをやります

1. ARMのエミュレータにより簡単なプログラムを実行し，システムコールが呼ばれるときの動作を見てみます．

2. エミュレータのソースコードを参照して，内部構造を見てみます．特定の命令の処理部分を探してみます．(代入命令など)

3. エミュレータのシステムコール処理部分を探してみます．システムコールの呼び出し手順を調べます．

4. エミュレータに独自の改造を加えます(独自命令の追加など)．追加した独自命令をサンプルプログラムから呼び出してみて，動作を確認します．

5. 同様のことを他アーキテクチャでやってみます．(V850など)

6. バッファオーバーラン脆弱性への攻撃を防ぐには，どのような機能を追加すれば可能か考え，実装し，検証してみます．

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．

（準備）
演習環境

演習環境

自分のPCと演習用サーバのどちらを利用しても構いません

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，自分のPC内で演習を行うことができます．

自分のPCに環境構築していない場合には，ＰＣから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．

自分のPCを利用する場合

GDBのビルドなど行うため，それなりのCPUスペックがあったほうがいいです．

遅いと思ったら途中で演習用サーバの利用に切替えてもいいです．

演習用サーバを利用する場合

以下のサーバにSSHでログインしてください.

サーバ: 192.168.1.1 (ポート番号: 22)
ユーザ: user01～user40 (パスワードは「user」で共通)

適当なユーザを選んで, ログインしてみてください.

ログインしてホームディレクトリに何もファイルがなければ,
何かファイルを作成してからそのユーザを使ってください.

すでに何かのファイルがあれば, そのユーザは他の誰かが
使っているので そのユーザはやめて, 別のユーザでログイ
ンしなおしてください.

必要ファイルのダウンロード

(配布サーバ)
http://192.168.1.2:10080/

必要なファイルを演習環境にダウンロードして，展開してください

```
$ cd ~
$ wget http://192.168.1.2:10080/cross-20130826.zip
$ wget http://192.168.1.2:10080/gdb-7.3.1.tar.gz
$ wget http://192.168.1.2:10080/bof-server.zip
$ wget http://192.168.1.2:10080/exploit-built.zip
$ wget http://192.168.1.2:10080/patch.zip
$ unzip cross-20130826.zip
$ unzip bof-server.zip
$ unzip exploit-built.zip
$ unzip patch.zip
```

この演習で利用するファイル一覧

ファイル	内容
workshop2016-emulator.pdf	スライド
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プログラム(http://kozos.jp/books/asm/)
gdb-7.3.1.tar.gz	各種アーキテクチャのデバッガ＆エミュレータ
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(http://kozos.jp/seccon/seccon2015.html)
exploit-built.zip	脆弱性の検証コード(http://kozos.jp/seccon/seccon2015.html)
patch.zip	各種パッチ類

GDBのビルド

```
$ cd ~
$ tar xvzf gdb-7.3.1.tar.gz
(64bit環境でARMエミュレータが落ちる問題の修正パッチ)
$ patch -p0 < patch/gdb/avoid-64bit-segfault.diff
$ cd gdb-7.3.1
$ ./configure --target=arm-elf
          --disable-werror --disable-nls
$ make (複数コアでは make -j 2 のようにコア数を指定すると高速)
  → ~/gdb-7.3.1/sim/arm/run が生成される
     (エミュレータの実行ファイル)
```

- GDBがARM向けにビルドされます (ARMのプログラムをデバッグ＆エミュレーション実行できます)
- ビルドに時間がかかるので，早めにやっておきましょう
- 演習環境(FreeBSD)でうまくビルドできない場合は，makeでなくgmakeを使ってみてください
- 他アーキテクチャもいじってみたい人は，他アーキ用のGDBもビルドしておきましょう (./configureの--targetオプションでターゲットのアーキを指定します．指定できるターゲットは次ページ参照)
  → ~/gdb-7.3.1/sim/<アーキ名>/run が生成されます(エミュレータ)

演習可能なアーキテクチャ一覧

ターゲット	アーキ名	概要
arm-elf	ARM	スマフォでおなじみ
ARMと共通	Thumb	ARMの16ビット縮小命令 (cross-20130826.zipの環境ではarm16-elf.*というファイルを生成)
cris-elf	CRIS	CRISという謎アーキ
frv-elf	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	H8	高専でよく使われている国産マイコン
m32r-elf	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	M･CORE	謎マイコン
mips-elf	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	MIPS16	MIPSの16ビット縮小命令
mn10300-elf	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	PowerPC	ゲーム機や旧PowerMACで使われていた
sh-elf	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	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	V850	エンジン制御など

(やることその1)

ARMのエミュレータにより簡単なプログラムを実行し，システムコールが呼ばれるときの動作を見てみます．

サンプルプログラムを見てみる

```
$ cd ~/cross/exec
$ 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.sot
arm-elf.d        arm-elf.s        arm-elf.x
$ ls sample.c
sample.c
$
```

サンプルプログラムの使いかた

```
$ cd ~/cross/exec
$ make clean → 生成したファイルを削除（掃除）
$ make       → 実行ファイルを再生成
$ make run   → GDBのエミュレータで実行
```

（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用のファイルのみ再生成）

```
$ vi arm-elf.c
$ make arm-elf.d
```

サンプルのCソースコードを自由に変更し，実行ファイルを再生成して，動作の変化をいろいろ見てみることができます

ファイルの説明(ARMの場合)

ファイル	内容
sample.c	ハロー・ワールドのC言語サンプル
lib-arm-elf.S	スタートアップとシステムコール・ラッパー
ld.scr	リンカ・スクリプト
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
arm-elf.sot	arm-elf.xをエミュレータで実行させた結果

エミュレータによる実行

(エミュレータで実行ファイルを実行)
```
$ make arm-elf.sot  (ARMの場合)
$ make v850-elf.sot (V850の場合)
```

(エミュレータを直接指定して実行)
```
$ /usr/local/cross/bin/arm-elf-run arm-elf.x   (ARMの場合)
$ /usr/local/cross/bin/v850-elf-run v850-elf.x (V850の場合)
```
→ 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

(本演習のメイン操作)

(ビルドしたGDBのエミュレータで実行)
```
$ ~/gdb-7.3.1/sim/arm/run arm-elf.x   (ARMの場合)
$ ~/gdb-7.3.1/sim/v850/run v850-elf.x (V850の場合)
```
→ エミュレータに改造を入れて実行したい場合に利用
→ エミュレータがダウンする場合は，「GDBのビルド」
　で説明したパッチを当てているか確認してください

GDBによる動的解析

```
$ /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
```
→ 実行開始する．main()の先頭でブレークするはず

```
(gdb) layout src
```
→ ソースコードの表示モード

```
(gdb) step
```
→ 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) next → 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) break <関数名> → ブレークポイントを設定
(gdb) info breakpoints → ブレークポイント一覧
(gdb) disable <ブレークポイント番号>
→ ブレークポイントを無効化 (enableで有効化)

[Ctrl]+[x] - [a] → ソースコードの表示モードから抜ける

アセンブリベースでの解析

```
$ /usr/local/cross/bin/arm-elf-gdb arm-elf.x
(gdb) layout asm
→ アセンブリの表示モード

(gdb) stepi → アセンブリ単位でのステップ実行
(gdb) nexti → アセンブリ単位でのステップ実行
(gdb) info registers → レジスタ情報一覧
```

演習1

1. GDBのステップ実行を使って，文字列が出力される瞬間の命令を探ってください
2. その命令が実行される直前のレジスタ情報を調べてください
3. 命令が実行された直後のレジスタ情報を調べて，直前の状態と比較してください
4. このARM実行環境での，システムコール発行の「仕様」を推測してください
5. ARM以外のアーキテクチャでもやってみてください(V850など)

(やることその2)

エミュレータのソースコードを参照して，内部構造を見てみます．特定の命令の処理部分を探してみます．(代入命令など)

エミュレータのソースコード

$ cd ~/gdb-7.3.1
$ ls
→ simというディレクトリがあります
$ cd sim
$ ls
→ 各種アーキテクチャごとのディレクトリがあります
$ cd arm
$ ls
→ ARM向けエミュレータのソースコード

メインループを探してみる

$ ls *.c
→ 各ファイルのファイル名を見て，
　「メインループがありそうなファイル」を探す

$ less XXX.c
→ メインループを探す

メインループを探す際のコツ

- grepでそれっぽいキーワードで検索する
- まずファイル名を見る
  デバイス名っぽいのとか機種名っぽい名前は，おそらくそれらの特殊処理用のコードなので対象から外せます
- #ifdef 〜 #endif でくくられている部分は読み飛ばす
  コンパイルオプションによって有効になったり無効になったりするような，要するに本筋でないオプション的な処理なので，読み飛ばしてOKなことが多いです

メインループを探す際のコツその2

- エミュレータのメインループの多くは，以下のような構造になっています

  - CPUの多数の命令に応じた処理をするための，巨大な switch～case
  - 上記 switch は，命令コードの特定ビット(オペコード)を見てcaseで 分岐している
  - プログラムカウンタを更新して命令をロードし，上記 switch～case を 実行するだけの while ループ

  これを想定して，caseがひたすら連続しているような部分を探します．(grep case *.c してみるなど)

- 「難しそう．自分は読めるかなあ」と思っちゃったときは，「自分は天才」と思って読みます (重要)

エミュレータ用語

エミュレータのソースコードには，以下の用語が多く出てきます． 知っているとソースコードを読む際の助けになります

- insn ... instruction(命令)の略． なぜかinsnと略すことが多い
- arch ... architecture(アーキテクチャ)の略
- op,ope ... 命令のオペコードなど
- load/store ... メモリの読み込み(ロード)と書き込み(ストア)
- reg ... レジスタ(register)
- SP ... スタックポインタ

- PC ... プログラム･カウンタ(Program Counter)． 現在実行中(の，次)の命令の アドレスを保持する． x86用語ではIP(Instruction Pointer)と呼ぶので x86寄りの人はIPと言ったりするが，PCと言うほうが一般的

- B,H,L,W ... Bはbyteで１バイト，Hは Half Word で２バイト， LはLongで４バイト，WはWordで(32ビットアーキでは)４バイトを 指す． 命令がバイトアクセスだったりワードアクセスだったり するときの指定などに多用される

エミュレータのメインループ(ARMの例)

```
$ cd ~/gdb-7.3.1/sim/arm
$ less armemu.c → 「ARMul_Emulate32」を検索

ARMword
#ifdef MODE32
ARMul_Emulate32 (ARMul_State * state)
...
  do
    {
...
          switch ((int) BITS (20, 27))
            {
...
            case 0x1a:          /* MOV reg */
...
              dest = DPRegRHS;
              WRITEDEST (dest);
              break;
```

演習2

1. ARMのエミュレータ用ソースコード内の，メインループ部分を探してください

2. 適当な命令の実行処理を見てみてください(MOV命令，ADD命令など．ARMの命令はcross/sample/arm-elf.dを参照するといいでしょう)

3. ARM以外のアーキテクチャでもやってみてください(V850など)

(やることその3)

エミュレータのシステムコール処理部分を探してみます．システムコールの呼び出し手順を調べます．

2種類のエミュレータ

- エミュレータは大別して「システム・エミュレータ」と「ユーザランド・エミュレータ」に分類できます
- システム・エミュレータ ... 装置全体をエミュレートします．エミュレータ上でLinuxなどのOSを 動作させることができます(「ゲストOS」と呼びます)
(例: VirtualBox, VMware など)
- ユーザランド・エミュレータ ... OS環境(アプリケーションの実行環境)をエミュレートします． OS上で動作するアプリケーション・プログラムを動作させることができます
(例: Wine, Windows Subsystem for Linux など)
- GDB付属のエミュレータ(の，多く)は，「ユーザランド・エミュレータ」です (両方の機能を持っているものもあります)

2種類のエミュレータの大きな違い

「システムコール命令が呼ばれたときの処理」が大きく異なります．それぞれ，以下のように動作します

- システム・エミュレータ
    - 「システムコール例外」という割込みが発生したと解釈し，割込みベクタに登録してある処理を実行します
(多くの場合，ゲストOSのカーネルの割込みハンドラが登録されているため，ゲストOSのカーネルに処理が移り，ゲストOSがシステムコールを処理します)
- ユーザランド・エミュレータ
    - 何かのOSカーネルと互換の動作をします
(例えばLinuxをエミュレートするなら，read()が呼ばれたらファイルのリードを行う，など)

「システムコール」とは何か

- 「システムコール」というのがよくわからない人は，要するに例えばLinuxなら open()とかread()とかwrite()とかのことだと思ってください

  ※ fopen()やfgetc()やfputc()は「システムコール」ではなく「ライブラリ関数」です

- システムコールは，OSカーネルがアプリケーション・プログラムに提供する サービスです

- アプリケーション・プログラムが(open()やread()などの)「システムコール」を 呼び出したら，OSカーネルがそれに応じた処理を行います

システムコール処理の流れ

- OSカーネルへのシステムコールは(多くの場合)以下のようにして行われます

  1. アプリケーション・プログラムがシステムコールの種類や引数を特定の レジスタ(もしくはスタック)に格納し, システムコール命令を実行します
  2. 割込みが発生し, OSカーネルに実行が移ります
  3. OSカーネルが特定のレジスタの値を参照して, 呼び出されたシステムコールを 判断します
  4. OSカーネルが当該の処理を行います

- 「特定のレジスタ」にどのレジスタをどのように使うのかは, OSカーネルの仕様として決められています (ABI:Application Binary Interface と呼びます)

ユーザランド・エミュレータには何が必要か

- ユーザランド・エミュレータは，OS環境(アプリケーションの実行環境)を エミュレートします
- 「OS環境をエミュレートする」ということは，「OSカーネルのシステムコール処理をエミュレートする」ということです

  ※ 本当はライブラリ環境やアプリケーション環境も必要ですが，最重要なのは，システムコールのエミュレーションです

ユーザランド・エミュレータには何があるのか

- GDB付属のエミュレータ(の，多く)は，「ユーザランド・エミュレータ」です
- そしてユーザランド・エミュレータはOSカーネルのシステムコール処理を エミュレートします
- つまりGDB付属のエミュレータのソースコード中には，「システムコール命令が呼ばれたら，何らかのOSカーネルと互換の動作を行う」という処理があるはずです
- ARMで言うと，例えばswi命令が呼ばれたらR0,R1,R2を引数として 当該のシステムコール処理を実行する，といった処理があることになります

エミュレータのシステムコール処理を探してみる

以下のようなことをやって，探してみよう

```
$ cd ~/gdb-7.3.1/sim/arm
$ grep SYSTEM_CALL *.c
$ grep system_call *.c
$ grep SYSTEMCALL *.c
$ grep SystemCall *.c
$ grep SYSCALL *.c
$ grep syscall *.c
```

システムコール処理を探す際のコツ

- ユーザランド・エミュレータのシステムコール処理の多くは，以下のような構造になっています
  - システムコールの種別に応じた処理をするための，巨大な switch～case
  - 上記 switch は，特定のレジスタ(システムコール番号)を見てcaseで 分岐している
  - case の内部にはシステムコールの種別に応じて，open()したり，read()したり している

  これを想定して，caseがひたすら連続しているような部分を探します．(grep case *.c してみるなど)

- 「難しそう．自分は読めるかなあ」と思っちゃったときは，「自分は天才」と思って読みます (重要)

ヒントがあります

```
$ cd ~/cross/exec
$ 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，
先頭に以下のよう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
(「システムコール・ラッパー」と言います)

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

システムコール処理(ARMの例)

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

unsigned
ARMul_OSHandleSWI (ARMul_State * state, ARMword number)
{
...
  switch (number)
    {
    case SWI_Read:
      if (swi_mask & SWI_MASK_DEMON)
        SWIread (state, state->Reg[0], state->Reg[1], state->Reg[2]);
      ...
      break;

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

演習3

1. ARMのエミュレータ用ソースコード内の，システムコール処理部分を探してください
2. どのようなシステムコールをサポートしているのか，見てみてください (Linuxのシステムコールだとは限りません)
3. 演習1で調べた「システムコール発行の仕様」と一致していることを確認して ください
4. lib-arm-elf.S のシステムコール呼び出し処理(システムコール・ラッパー) によって，エミュレータのシステムコール処理が実行される一連の流れを 理解してください
5. ARM以外のアーキテクチャでもやってみてください(V850など)

(やることその4)

エミュレータに独自の改造を加えます(独自命令の追加など). 追加した独自命令をサンプルプログラムから呼び出してみて, 動作を確認します.

改造の案

「カウンタ命令」を追加してみよう

- 命令を実行するたびに，1ずつ増えた値が指定されたレジスタに格納される命令
- 統計情報をカウントしたい場合などに使えないだろうか…？
- アトミックにカウントできるので，割込み処理などで利用できる可能性があるかも

修正を加えるファイル

- GDB
  - gdb-7.3.1/sim/arm/armemu.c
    カウンタ命令を追加
    (patch/counter/counter-gdb-armemu_c.diff)
- サンプルプログラム
  - cross/exec/lib-arm-elf.S
    カウンタ命令を呼び出す関数を追加
    (patch/counter/counter-cross-lib-arm-elf_S.diff)
  - cross/exec/sample.c
    カウンタ値の出力を追加
    (patch/counter/counter-cross-sample_c.diff)

ファイルの差分について

ファイルの差分は，diffコマンドで得ることができます

（使用例）

```
$ cd ~/gdb-7.3.1/sim/arm
$ mv armemu.c armemu.c.orig （オリジナルを保存）
$ cp armemu.c.orig armemu.c
$ vi armemu.c （ファイルに修正を加える）
```

（オリジナルに対しての差分を取得する）

```
$ diff -u armemu.c.orig armemu.c > ~/armemu.diff
```

~/patch にある *.diff というファイルは上記のようにして取得した，オリジナルのファイルに対する修正内容です

行頭に「-」がある行は，「削除された」という意味です
行頭に「+」がある行は，「追加された」という意味です

GDBの改造

counter-gdb-armemu_c.diffの内容

```
--- gdb-7.3.1/sim/arm/armemu.c.orig     2007-02-15 19:32:06.000000000 +0900
+++ gdb-7.3.1/sim/arm/armemu.c  2016-11-21 23:52:23.128436000 +0900
@@ -3835,25 +3835,32 @@
            case 0xf9:
            case 0xfa:
            case 0xfb:
            case 0xfc:
            case 0xfd:
            case 0xfe:
-           case 0xff:
              if (instr == ARMul_ABORTWORD && state->AbortAddr == pc)
                {
                  /* A prefetch abort.  */
                  XScale_set_fsr_far (state, ARMul_CP15_R5_MMU_EXCPT, pc);
                  ARMul_Abort (state, ARMul_PrefetchAbortV);
                  break;
                }

              if (!ARMul_OSHandleSWI (state, BITS (0, 23)))
                ARMul_Abort (state, ARMul_SWIV);

              break;
...(次ページへ続く)...
```

GDBの改造(続き)

counter-gdb-armemu_c.diffの内容(続き)

```
...(前ページからの続き)...
+
+               /* Counter instruction.  */
+             case 0xff:
+               {
+                 static int counter = 0;
+                 state->Reg[BITS (16, 19)] = counter++;
+               }
+               break;
+             }
+         }

 #ifdef MODET
     donext:
 #endif
```

サンプルプログラムの対応

counter-cross-lib-arm-elf_S.diffの内容

```
--- cross/exec/lib-arm-elf.S.orig       2012-10-18 21:37:01.000000000 +0900
+++ cross/exec/lib-arm-elf.S    2016-11-21 00:02:45.000000000 +0900
@@ -67,9 +67,15 @@
        .globl  __close
        .type   __close, %function
 __close:
        SWI(SWI_Close)
        mov     pc, lr

        .align 4
 _stack_addr:
        .long   _estack
+
+       .globl  counter
+       .type   counter, %function
+counter:
+       .long   0xeff00000
+       mov     pc, lr
```

サンプルプログラムの対応

counter-cross-sample_c.diffの内容

```
--- cross/exec/sample.c.orig    2012-10-18 21:37:01.000000000 +0900
+++ cross/exec/sample.c 2016-11-21 00:04:16.000000000 +0900
@@ -84,9 +84,12 @@
 int main()
 {
   int fd = 1;
   puts(fd, "Hello World! ");
   putxval(fd, data_value, 0);
   puts(fd, " This architecture is " ARCH "¥n");
+  putxval(fd, counter(), 0); puts(fd, "¥n");
+  putxval(fd, counter(), 0); puts(fd, "¥n");
+  putxval(fd, counter(), 0); puts(fd, "¥n");
   exit(0);
   return 0;
 }
```

パッチについて

ファイルの差分があれば，オリジナルのファイルに対して差分を加えることで，修正内容を反映した新しいファイルに更新できます

ファイルの差分のことを「パッチ」と言います．差分を反映することを「パッチを当てる」と言います

（パッチ当ての例）

```
$ cd ~/gdb-7.3.1/sim/arm
$ cp armemu.c.orig armemu.c     （オリジナルに戻す）
$ patch -p0 < ~/armemu.c.diff  （パッチを当てる）
```

本演習で配布するパッチ(~/patch以下のファイル)を当てると，オリジナルのファイルは*.orig というファイル名で保存されます．元に戻したいとき(別のパッチを当てるときなど)には，*.orig をコピーして元に戻してください

パッチを当てて，再ビルドして実行してみる

```
(GDBにパッチを当てて再ビルドする)
$ cd ~
$ patch -p0 < patch/counter/counter-gdb-armemu_c.diff
$ cd gdb-7.3.1
$ make  (エミュレータ(~/gdb-7.3.1/sim/arm/run)が再生成される)

(サンプルプログラムにパッチを当てて再ビルドする)
$ cd ~
$ patch -p0 < patch/counter/counter-cross-lib-arm-elf_S.diff
$ patch -p0 < patch/counter/counter-cross-sample_c.diff
$ cd cross/exec
$ make arm-elf.x

(実行例)
$ cd ~
$ ~/gdb-7.3.1/sim/arm/run ~/cross/exec/arm-elf.x
Hello World! abadface This architecture is arm-elf
0
1
2
$
```

終わったら，元に戻しておきましょう

```
$ cd ~
$ rm -fR cross
$ unzip cross-20130826.zip

$ cd ~/gdb-7.3.1/sim/arm
$ cp armemu.c.orig armemu.c
$ cd ~/gdb-7.3.1
$ make
```

演習4

1. 追加したい独自命令を考えて, 追加してみてください(CPUへの機能追加)

2. 追加したい独自システムコールを考えて, 追加してみてください(OSカーネルへの機能追加)

3. ARM以外のアーキテクチャでもやってみてください(V850など)

(やることその5)

同様のことを他アーキテクチャでやってみます．(V850など)

演習5

1. ここまでの内容で各自で試してみたいことを，ARM以外のアーキテクチャでもやってみてください

2. やってみた結果を, ARMのそれと比較してみてください
(比較することで, アーキ固有の部分はどこか，共通的な(本質的な)ものは何なのかを知ることができます)

(やることその6)

バッファオーバーラン脆弱性への攻撃を防ぐには，どのような機能を追加すれば可能か考え，実装し，検証してみます．

(やることその６－１)

各種アーキテクチャの脆弱性検証のサンプルの使いかた

サーバプログラムを展開

```
$ 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すると各アーキ向けにコンパイルされる)

sample.cにはバッファオーバーランの脆弱性があります

サーバプログラムのバッファオーバーラン脆弱性

```
$ cd ~/bof-server/cross
$ less sample.c
```

→ 以下の部分にサイズチェックもれによる
バッファオーバーラン脆弱性があります

```
int gets(int fd, char *str)
{
  unsigned char c;
  while (1) {
    c = getchar(fd);
...

int proc()
{
  char buf[16];

  /* バッファオーバーランの脆弱性がある */
  puts(1, "Input name:¥n"); /* フラッシュされるように改行を入れる */
  gets(0, buf);
...
```

サーバプログラムをエミュレータで実行

```
$ 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
$
```

脆弱性を検証コードで検証する

```
$ cd ~/exploit/cross
$ cat flag.txt
```

→ flag.txt の内容を確認

```
$ ./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！

※ 任意ファイルの内容の抜き出し
ネットワーク経由で動作していたら大問題！

(やることその6－2)

どのような脆弱性なのか？

検証コードを調べる

```
$ cd ~/exploit/cross

$ less arm-elf.S
```
→ 検証コードのソースコード(アセンブリ)を参照する

```
$ less arm-elf.d
```
→ 検証コードの逆アセンブル結果を参照する

以下のようなことが行われています

- サーバプログラムは，標準入力からread()していますが，バッファオーバーランの脆弱性があります
- 検証コードは，バッファオーバーランを発生させ，特定の実行コードをメモリ上に 書き込み，関数からの戻り先アドレスを実行コードの先頭に書き換えます
- 関数から返ると実行コードにジャンプし，実行コードが実行されます
- 実行コードは，カレントディレクトリの"flag.txt"をopen()し，read()し，標準出力にwrite()します

(やることその6－3)

エミュレータにどのような機能を追加すれば，その脆弱性への攻撃を防ぐことができるか？

攻撃を検知するための方法の案

- (案1)関数呼び出し命令が呼ばれたときに戻り先アドレスを内部で保持しておき，リターン時にアドレスをチェックする
  - CPUに対する機能追加を想定
  - 改造サンプル: ~/patch/retaddr/retaddr-armemu_c.diff
- (案2)read()で読み込まれる内容を監視し，システムコール呼び出しを行う 機械語コードがあり，さらにシステムコールが呼ばれたならば 不正な呼び出しと判断する
  - OSカーネルに対する機能追加を想定
  - 改造サンプル: ~/patch/readcheck/readcheck-armos_c.diff
- (案3)システムコール命令が連続して(間に関数コールなどが行われずに) 呼ばれた場合には，不正と判断する
  - CPUとOSカーネルに対する機能追加を想定
  - 改造サンプル: ~/patch/serialcall/serialcall-*_c.diff
    ※「やることその4」でのカウンタ命令の実装を流用

(案1) GDBの改造 (retaddr-armemu_c.diff)

```
--- gdb-7.3.1/sim/arm/armemu.c.orig     2007-02-15 19:32:06.000000000 +0900
+++ gdb-7.3.1/sim/arm/armemu.c  2016-11-21 22:08:35.683116000 +0900
@@ -474,6 +474,25 @@
   return 0;
 }

+static int retaddr_buf[16];
+static int retaddr_p = 0;
+
+static void push_retaddr(ARMul_State * state, int retaddr, int jmpaddr)
+{
+  fprintf(stderr, "Push retaddr: %08x %08x¥n", retaddr, jmpaddr);
+  retaddr_buf[retaddr_p++] = retaddr;
+}
+
+static void pop_retaddr(ARMul_State * state, int retaddr)
+{
+  fprintf(stderr, "Pop retaddr: %08x¥n", retaddr);
+  if (retaddr_buf[--retaddr_p] != retaddr) {
+    fprintf(stderr, "Detected invalid retaddr: %08x %08x¥n",
+            retaddr_buf[retaddr_p], retaddr);
+    exit(0);
+  }
+}
+
```

...(次ページへ続く)...

(案1) GDBの改造 (retaddr-armemu_c.diff(続き))

```
...(前ページからの続き)...
 /* EMULATION of ARM6.  */

 /* The PC pipeline value depends on whether ARM
@@ -3472,6 +3491,7 @@
              state->Reg[14] = (pc + 4) | ECC | ER15INT | EMODE;
 #endif
              state->Reg[15] = pc + 8 + POSBRANCH;
+             push_retaddr(state, state->Reg[14], state->Reg[15]);
              FLUSHPIPE;
              break;

@@ -3492,6 +3512,7 @@
              state->Reg[14] = (pc + 4) | ECC | ER15INT | EMODE;
 #endif
              state->Reg[15] = pc + 8 + NEGBRANCH;
+             push_retaddr(state, state->Reg[14], state->Reg[15]);
              FLUSHPIPE;
              break;

...(次ページへ続く)...
```

(案1) GDBの改造 (retaddr-armemu_c.diff(続き))

```
...(前ページからの続き)...
@@ -4126,6 +4147,7 @@
 static void
 WriteR15 (ARMul_State * state, ARMword src)
 {
+  pop_retaddr(state, src);
   /* The ARM documentation states that the two least significant bits
      are discarded when setting PC, except in the cases handled by
      WriteR15Branch() below.  It's probably an oversight: in THUMB
@@ -4192,6 +4214,7 @@
 WriteR15Branch (ARMul_State * state, ARMword src)
 {
 #ifdef MODET
+  pop_retaddr(state, src);
   if (src & 1)
     {
       /* Thumb bit.  */
```

(案2) GDBの改造 (readcheck-armos_c.diff)

```
--- gdb-7.3.1/sim/arm/armos.c.orig 2007-02-28 03:51:57.000000000 +0900
+++ gdb-7.3.1/sim/arm/armos.c 2016-11-21 22:31:20.334677000 +0900
@@ -341,6 +341,23 @@
     }
 }

+static int readcheck_open = 0;
+
+static void readcheck(ARMul_State * state, char c)
+{
+  static char readbuf[8];
+  static char swi[] = { 0x66, 0x00, 0x00, 0xef }; /* swi 0x66(open) */
+  static char ret[] = { 0x0e, 0xf0, 0xa0, 0xe1 }; /* mov pc, lr */
+
+  memmove(readbuf, readbuf + 1, sizeof(readbuf) - 1);
+  readbuf[sizeof(readbuf) - 1] = c;
+
+  if (!memcmp(&readbuf[0], swi, 4) && memcmp(&readbuf[4], ret, 4)) {
+    fprintf(stderr, "Detected syscall(open) data: %08x¥n", state->Reg[15]);
+    readcheck_open = 1;
+  }
+}
+
 static void
 SWIread (ARMul_State * state, ARMword f, ARMword ptr, ARMword len)
 {
...(次ページへ続く)...
```

(案2) GDBの改造 (readcheck-armos_c.diff(続き))

```
...(前ページからの続き)...
@@ -363,6 +380,10 @@
     for (i = 0; i < res; i++)
       ARMul_SafeWriteByte (state, ptr + i, local[i]);

+  if (res > 0)
+    for (i = 0; i < res; i++)
+      readcheck(state, local[i]);
+
   free (local);
   state->Reg[0] = res == -1 ? -1 : len - res;
   OSptr->ErrorNo = sim_callback->get_errno (sim_callback);
@@ -452,6 +473,12 @@
   struct OSblock * OSptr = (struct OSblock *) state->OSptr;
   int              unhandled = FALSE;

+  if (readcheck_open && (number == SWI_Open)) {
+    fprintf(stderr, "Detected reading and calling open(): %08x¥n",
+     state->Reg[15]);
+    exit(0);
+  }
+
   switch (number)
     {
     case SWI_Read:
```

(案3) GDBの改造 (serialcall-armemu_c.diff)

```
--- gdb-7.3.1/sim/arm/armemu.c.orig     2007-02-15 19:32:06.000000000 +0900
+++ gdb-7.3.1/sim/arm/armemu.c  2016-11-21 22:42:51.659653000 +0900
@@ -474,6 +474,12 @@
   return 0;
 }

+#ifdef MODE32
+int counter = 0;
+#else
+extern int counter;
+#endif
+
 /* EMULATION of ARM6.  */

 /* The PC pipeline value depends on whether ARM
```

...(次ページへ続く)...

(案3) GDBの改造 (serialcall-armemu_c.diff(続き))

```
...(前ページからの続き)...
@@ -3472,6 +3478,7 @@
              state->Reg[14] = (pc + 4) | ECC | ER15INT | EMODE;
 #endif
              state->Reg[15] = pc + 8 + POSBRANCH;
+             counter++;
              FLUSHPIPE;
              break;

@@ -3492,6 +3499,7 @@
              state->Reg[14] = (pc + 4) | ECC | ER15INT | EMODE;
 #endif
              state->Reg[15] = pc + 8 + NEGBRANCH;
+             counter++;
              FLUSHPIPE;
              break;

...(次ページへ続く)...
```

(案3) GDBの改造 (serialcall-armemu_c.diff(続き))

...(前ページからの続き)...

```
@@ -3838,7 +3846,6 @@
            case 0xfc:
            case 0xfd:
            case 0xfe:
-           case 0xff:
              if (instr == ARMul_ABORTWORD && state->AbortAddr == pc)
                {
                  /* A prefetch abort.  */
@@ -3851,6 +3858,11 @@
                ARMul_Abort (state, ARMul_SWIV);

              break;
+
+             /* Special instruction to get the counter.  */
+           case 0xff:
+             state->Reg[BITS (16, 19)] = counter;
+             break;
            }
        }
```

...(次ページへ続く)...

(案3) GDBの改造 (serialcall-armemu_c.diff(続き))

```
...(前ページからの続き)...
@@ -4126,6 +4138,7 @@
 static void
 WriteR15 (ARMul_State * state, ARMword src)
 {
+  counter++;
   /* The ARM documentation states that the two least significant bits
      are discarded when setting PC, except in the cases handled by
      WriteR15Branch() below.  It's probably an oversight: in THUMB
@@ -4192,6 +4205,7 @@
 WriteR15Branch (ARMul_State * state, ARMword src)
 {
 #ifdef MODET
+  counter++;
   if (src & 1)
     {
       /* Thumb bit.  */
```

(案3) GDBの改造 (serialcall-armos_c.diff)

```
--- gdb-7.3.1/sim/arm/armos.c.orig      2007-02-28 03:51:57.000000000 +0900
+++ gdb-7.3.1/sim/arm/armos.c   2016-11-21 22:41:54.865501000 +0900
@@ -452,6 +452,24 @@
   struct OSblock * OSptr = (struct OSblock *) state->OSptr;
   int              unhandled = FALSE;

+  {
+    static int old_counter = -1;
+    extern int counter;
+
+    /*
+     * Refer to the value of the counter directly here.
+     * But the counter belongs to CPU, and here is processing
+     * of an OS kernel. Therefore it is necessary to get
+     * the value of the counter by the special instruction.
+     */
+    if (counter == old_counter) {
+      fprintf(stderr, "Detected serial system call: %08x¥n",
+              state->Reg[15]);
+      exit(0);
+    }
+    old_counter = counter;
+  }
+
   switch (number)
     {
     case SWI_Read:
```

(やることその６－4)

検証してみよう！

攻撃を検知できることを検証する

（修正を加えていた場合には，元に戻してからパッチ当てする）
```
$ cd ~/gdb-7.3.1/sim/arm
$ cp armemu.c.orig armemu.c
```

（案１のパッチを当てて，GDBを再ビルドする）
```
$ cd ~
$ patch -p0 < ~/patch/retaddr/retaddr-armemu_c.diff
$ cd gdb-7.3.1
$ make
```

（サンプルプログラムが正常実行できることを確認）
```
$ ~/gdb-7.3.1/sim/arm/run ~/cross/exec/arm-elf.x
```

（サーバプログラムへの攻撃を検知して動作停止することを確認）
```
$ cd ~/exploit/cross
$ ./cat.pl ./enter.bin ./arm-elf.dat
  | ~/gdb-7.3.1/sim/arm/run ~/bof-server/cross/arm-elf.x
```

演習6

1. 案2と案3についても，パッチを当てて検証してみてください (パッチを当てる前に，ファイルを元に戻すことを忘れずに！)

2. 攻撃を防ぐ方法を考えて，実装してみてください

考察

案1～案3の方法を実環境で実現するためには，どのような対応が必要か考えてみよう

- 案1は，CPUの対応が必要
- 案2は，OSカーネルの対応が必要
- 案3は，CPUとOSカーネルの対応が必要
  - CPUには，アドレス範囲を設定する特殊命令の実装と，プログラムカウンタの値をチェックする機能の追加が必要
  - OSカーネルには，特殊命令によるアドレス範囲の設定処理の追加が必要
  - つまりCPUとOSカーネルの両方で対応し，協調動作する必要がある

考察

- 他にも場合によっては，例えばCPUに追加した特殊命令を使う実行コードを生成 するように，コンパイラで対応する必要性も考えられる
- 同様に，CPUに追加した特殊命令やOSカーネルに追加した特殊システムコールを 使うように，ライブラリで対応する必要性も考えられる
- CPUに特殊命令を追加しても，それを使うようにOSカーネルやコンパイラや ライブラリを対応させなければ意味は無い
- OSカーネルに特殊システムコールを追加しても，それを使うようにライブラリや アプリケーションを対応させなければ，やはり意味は無い
- 逆に，ライブラリやアプリケーションだけでできる対策を考えてみよう(効果が限定される・対応が面倒・本質的対応にならない場合も)

複数レイヤー間での連携と協調動作が必要

この講義で伝えたかったこと

- 低レイヤーを知らないと出てこないようなアイディアがある (CPUに機能追加する, OSカーネルに機能追加する, など).
またアイディアがあったとしても, 低レイヤーを知らないと検証できなかったり, 実現性や有効性を判断できなかったりする

- 効果的な脆弱性対策のためには, 特定のレイヤーだけに閉じた対策だけでなく, 複数のレイヤー(CPU, OSカーネル, コンパイラ, ライブラリ) をまたいで協調的に機能する方法を考えられることが重要

- システム全体を俯瞰して, 総合的に検討できることが重要
自分の専門以外のレイヤーも知り, 話を理解できることが重要
(特定のレイヤーのみ知っていればそれでいい, ということは無い)

- でも低レイヤーを学ぶ機会は少ないしみんなあまりやらないので, 勉強しておくとお得かも (他人と違うことや他人がやらないことをやっておくと, オンリーワンになりやすくて良いです)

- でも実際やると, 「今さらアセンブラなんてやって意味あるの」とか言われたりしますが, 気にしないのがコツです (笑)

おつかれさま
でした！