# エミュレータを改造して遊ぼう！
# ーSECCON2017決勝大会
# QEMUシステムコール問題の詳細ー

坂井弘亮

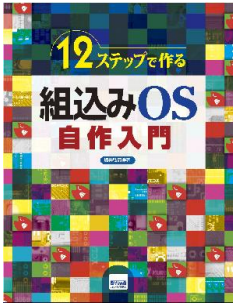
# 自己紹介

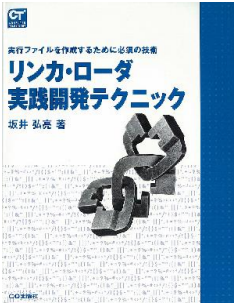
今までやってきたこと

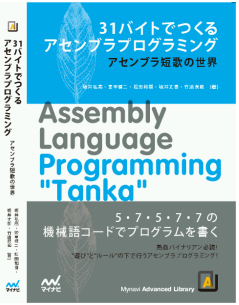
- 独自組込みOS開発(KOZOS)
- オープンソースカンファレンス出展
- セキュリティ・キャンプ講師
- SECCON実行委員・問題作成
- SecHack365委員
- 書籍執筆多数

他にもいろいろ. あとはググッてください

# 今まで書いた本


12ステップで作る
組込みOS
自作入門


実行ファイルを作成するために必須の技術
リンカ・ローダ
実践開発テクニック
坂井 弘亮 著


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


熱血!
アセンブラ入門
坂井弘亮[著]
やらなければ、
一生
わからん!!


ハロー
"Hello, World"
たった7行のCプログラムから解き明かす
OSと標準ライブラリのシゴトとしくみ
坂井弘亮[著]
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。
printf() 関数の先には何があるんだろう?
main() 関数が呼び出される前には何があるんだろう?
main() 関数からリターンする先はどこだ?


大熱血!
アセンブラ
入門
坂井 弘亮[著]
熱い心を持つエンジニアに贈る、
セキュリティの第一人者による渾身の書。
GCC4をひっさげて、
あの漢が帰ってきた!

セキュリティ研究者ではなく，組込み技術者です

# 本日のテーマ

- QEMU(エミュレータ)をいじって遊んでみよう! という話
- SECCON2017決勝大会(国際)の問題サーバ弐の詳細解説

# SECCONとは

SECCONとは

- 国内最大規模のセキュリティコンテスト
- 予選・国内決勝・国際決勝

ここ数年, 決勝大会(主に国際)のサーバ問題を担当

- 多種アーキテクチャExploit問題
- GDBデバッグプロトコル問題

主に低レイヤー分野を担当

本質的な防御のためには, 攻撃可能性を知る必要がある

# なぜ問題作成するのか(自分の場合)

考えていること

- 組込み分野はセキュリティ意識はまだまだ低い. 問題の提起
- 組込み分野とセキュリティ分野の橋渡し
- 目線の違い, 見えかたの違い(自分は組込み技術者)
- 自分はものづくりが好きなので, それを活かしたコミットがしたい (問題作成はものづくりとして純粋に楽しい)
- 低レイヤー学習の重要性を提起できる問題 (低レイヤーを知っていてこそ解ける問題, 解くことが面白い問題)

# こんな問題を作りたい

- 他の人が作らなさそうな，他の人と目線が異なる問題
- SECCON独自の問題
- 日本でやるなら日本らしい問題．他に無い問題
  - SECCONで多種アーキテクチャ問題といったら，10個とか20個とかだぜ！ みたいなもの
- 学習・復習になる問題．学習効果のある問題
  - 「セキュリティ学習」だけではなく「低レイヤー学習」とかのこと

# 考えていないこと

- 「今どきのCTFの問題」「CTFらしい問題」「世界のCTFに準じる問題」といった考えはありません
- そもそも自分はCTFプレイヤーではないので今どきのCTFとか世界のCTFとか よくわからない
- 自分なりの目的で問題を作っています
- 真似るなら, 自分がやることの意義もないだろう

# 今回作成した問題

- SECCON2017決勝大会(国際) サーバ弐
- 脆弱性のある, 多種アーキテクチャ向けのサーバプログラムが, QEMU上で動作している
- プログラムの実行ファイルはダウンロードできる
- 実行ファイルを解析し, 攻撃可能性を調べ, サーバ上のファイルを読み書きする, という内容

本質的な防御のためには, 攻撃可能性を知る必要がある

# サーバの関連ファイル

- サーバの関連ファイルは，以下で公開されています
- サーバのVMイメージ，構成ファイル，解答サンプルなど

http://kozos.jp/seccon/

# もう一度, 本日のテーマ

- QEMU(エミュレータ)をいじって遊んでみよう! という話
- SECCON2017決勝大会(国際)の問題サーバ弐の詳細解説

# まずはQEMUの話

# QEMUとは何か

- 広く利用されているエミュレータ
- 高速・多機能
- 多種アーキテクチャを対象としているため，組込み分野でも広く利用されている

# QEMUのソースコードで, システムコールの処理周りを見てみよう

# 実際に見てみる

# ここで, 前提知識

system(3)

- シェルを呼び出して任意のコマンドを実行するライブラリ関数
- 要するに任意のコマンドが実行できる, セキュリティ的にはやばい関数

# セキュリティ・キャンプの日々に，QEMUのソースコードを眺めていると，こんなものを見てしまった

(qemu/target/arm/arm-semi.c:arm_gdb_syscall())

```
case TARGET_SYS_SYSTEM:
    GET_ARG(0);
    GET_ARG(1);
    if (use_gdb_syscalls()) {
        return arm_gdb_syscall(cpu, arm_semi_cb, "system,%s",
                               arg0, (int)arg1+1);
    } else {
        s = lock_user_string(arg0);
        if (!s) {
            /* FIXME - should this error code be -TARGET_EFAULT ? */
            return (uint32_t)-1;
        }
        ret = set_swi_errno(ts, system(s));
        unlock_user(s, arg0, 0);
        return ret;
    }
```

# なにこれ超こわい！

- QEMUでなんらかの操作をすると, system() が呼ばれて任意のコマンド実行が できてしまう？
- semihostingという, システムコールに似た機能のようだ
- なにこれ超こわい！ どういうときにそうなるのか詳しく知って問題提起したい！ というのが今回のSECCONでの問題作成のきっかけ

# 次, SECCONの話

# 今回作成した問題

- SECCON2017決勝大会(国際) サーバ弐
- 脆弱性のある, 多種アーキテクチャ向けのサーバプログラムが, QEMU上で動作している
- プログラムの実行ファイルはダウンロードできる
- 実行ファイルを解析し, 攻撃可能性を調べ, サーバ上のファイルを読み書きする, という内容

本質的な防御のためには, 攻撃可能性を知る必要がある

# 今回作成した問題の詳細

- 実行ファイルを解析し，攻撃可能性を調べ，サーバ上のファイルを取得する，という内容
- semihostingを使うことで，ファイル中のキーワードを読むことができれば得点
- 競技者は，semihostingに気づく必要がある
- 問題作成の視点では，semihostingということを直接的には示唆せずに，うまく気づかせる，というバランスにする必要がある

# 問題にする上での課題

- サンプルがread/writeにsemihostingをそのまま使うだけでは，解析するとすぐに気づかれてしまい，それを流用するだけの 簡単な問題になってしまう
- semihostingに気づかせず，しかしエスパー問題にならないように ヒントは入れる必要がある (このあたりの難易度バランスが重要)
- read/writeには独自システムコールを使うことで semihostingに気づかせないが，exitにだけsemihostingを使うことでヒントにするという絶妙なバランス
- exitにだけsemihostingを使っているので，open/read/writeさせるためには qemuのソースコードを読んでsemihostingの呼び出し方法の詳細を知る必要がある (流用だけでは済まない)
- exitに失敗しているなら無限ループに入り無応答になるはずだが，接続が切断されるので，正常にexitできているはず…ということはわかる(はず)
- exitの処理を処理の流れ的にもアドレス的にもmain()の後に配置して，疑問に思うきっかけを作ってある

# 解いたチーム一覧

- 
- 

# 体験してみましょう

# まずは問題サーバに接続

HTTP接続, 問題説明を読む

- 3つのサーバプログラムが動いている
- カレントディレクトリにあるword.txtを読みフラグの取得
- チームフラッグを書き込むとflag.txtに反映されポイント

3つのアーキテクチャ

- ARM スマフォやラズパイ他, 多方面で利用されている組込み向け32ビットCPU
- MIPS (もともとは高性能ワークステーション向けだが) 現在は組込み向け32ビットCPU. RISCアーキテクチャの見本のCPU
- AArch64 ARMの64ビット・アーキテクチャ(今回初利用)

まずは体験

- ポートへの接続を試す
- arm-elf.xをダウンロードし簡単に見てみる

# 解いてみましょう
# (想定していた解法)

# クロスコンパイル環境

- 実行ファイルの解析のために, クロスコンパイル環境を構築する (説明のcross-gcc494参照. (参考)大熱血!アセンブラ入門)

- 説明ページにある情報
  - ARM/MIPS/AArch64の実行ファイル
  - QEMUにreadc/writecというスペシャルシステムコールのパッチを当てて 動作している

- まずはメジャーなARMでいきましょう
  - arm-elf.x をダウンロードして解析

# 逆アセンブルしてみる

- readc/writecの関数がある. svc命令でシステムコールを呼び出しているようだ
- しかしシステムコール番号らしきものが1とか2になっていて不自然
- Linuxのシステムコール番号一覧
- forkや_exitを潰してreadc/writecに置き換えている? (そんなばかな)

# システムモードで動作しているのでは

## QEMUの2種類の動作

- ユーザモード
  - CPUのユーザ命令とLinuxなどのOSカーネルをエミュレーションし，アプリケーション・プログラムを動作させる
- システムモード
  - CPUのすべての命令／動作をエミュレーションし，OSカーネルを動作させる

# OSレスのプログラムなのでは

つまりarm-elf.xはモニタから起動されてモニタの上で動作する，ベアメタルなプログラムを想定しているのではないだろうか

- OSカーネルそのものや，初期動作のテスト用プログラムなどが想定される
- システムコールを呼んでOSカーネルにサービス依頼する「ユーザ・アプリケーション」ではない
- readc/writecはデバッグのためにモニタが持つ，簡単な1文字入出力機能と 考えるのが妥当
- セキュリティ技術者的にはどうなのか知らないが，組込み技術者の目線では妥当 (と，思う)

# OSレスのプログラムなのでは

- 実機動作の場合は，モニタ側にデバッグ用の簡易システムコールとして シリアルへの1文字入出力のreadc/writecを実装しておく
- シミュレータ動作の場合は，シミュレータ側でシステムコール命令の実行時に readc/writecならば標準入出力を操作するような対応を入れておく
- そしてこういう独自システムコール追加は，組込み機器の開発ではよくあること
  - システムコール体系も独自で，そもそもLinuxやPOSIXでは全然違ったりする
  - 今回の場合はデバッグ用と割り切って，1文字入出力しか実装されていない
    (例：GDB内包の各種アーキテクチャ用シミュレータ)

# openやreadはあるのだろうか

ファイルの読み書きには, open/read/writeが必要だが

- そもそもopenが無いとファイルをオープンできないしreadが無いと任意のファイル の内容を読めない. それらはあるのか?
- パッチとしては「readc/writecのみ」と明記されているので, なさそう (システムコール番号を適当に変えてみたとしても見つからない)
- どうやってファイルの内容を読むのか?
- ここでqemuのソースコードを探してopenできるようなものが無いか探すという 道筋もあるのだが, ここではもうすこし実行ファイルを見てみよう

# exitはsemihosting_exit()になっている

arm-elf.xをよく見てみると

- exitは semihosting_exit() という関数によって実現されているようだが，readc/writecとは何か違う
- main()の終了後，semihosting_exit() という関数が呼ばれている．その後は無限ループになっているがプログラム自体は終了するので，無限ループには入らずにきちんと終了できているらしい
- しかしsemihosting_exit で行われている処理は，readc/writecとは何か違う (svc 0x123456 とは?)
- システムコール番号には semihosting_sys_exit (0x18)という値を渡している ようだ
- 独自追加のreadc/writecとはまた違う，別のシステムコール体系がある? しかしパッチとして機能追加しているのはreadc/writecのみのはず．QEMU既存の 処理で，そのようなシステムコール体系がある? そこにopenやreadがあるのでは?

# semihostingとは何か？

QEMUのソースを読む. semihostingやopenで検索してみよう

- semihostingという機能があるようだ
- 動作には条件があるようだが, exitができているので, これが動いている ことがわかる
- 見てみると...なんとsystem()がある!!! (驚愕)
- qemu起動時に -semihosting とかつけると有効になる機能のようだ. exitできているということは, これが有効になっているのでは？

# 解き筋をまとめると

- readc/writecというのがありシステムコール命令を読んでいるが，Linuxのシステムコール番号とぶつかっている
→ アプリケーション・プログラムではない？
- ベアメタル・プログラムで，readc/writecはモニタが持つデバッグ用の 簡易システムコールか？
- ということはreadc/writecと同等の方法でのopenやreadはなさそう．別の手段がある？
- exitはどうなっているのか．semihosting_exit()という関数があり，システムコール番号や呼び出し手順が全然違っている．しかしその後にある無限ループには入らず終了できているので，機能しているはず
- semihosting_exit()では何が起きているのか？ QEMUにはreadc/writecのパッチしか当てていないと明記されているので，QEMU既存の処理のはず．QEMUのソースコードを読んでみよう
- semihostingという機能があり，これが有効になっているようだ semihostingの中にopenやread/writeがあるので，これを使えばホストのファイルの 読み書きができるはず
- あとはsemihosting使ってファイルをオープンして読んで出力する

# 実際に解いてみる

- read/writeのサンプルを使ってフラグ取得・書き込みを行う
- readのサンプル
  - カレントディレクトリにあるword.txtを読みキーワードを取得すればポイント
- writeのサンプル
  - チームフラッグを書き込むとflag.txtに反映されポイント

# 解法のサンプルを見てみる

# system()を呼び出してみる

- system()を呼び出すには
  - chroot環境で動作しているので, 以下のコピーが必要
    - /bin/sh
    - 実行したいコマンド
    - 使われる共有ライブラリ

- system()を呼び出すサンプル
  - 実行してみる

# system()の呼び出し例を見てみる

# この問題の難しい点

# qemuにreadc/writec対応の独自パッチを当てているが

パッチの内容は非公開

- readc/writecという名前と, 実行ファイルの処理を読んで類推する必要がある (まあでも名前から明らかでもあるが)
- 実際の動作を見るために実行ファイルを手もとで動作させたいのだが, パッチが無いので, なんとかする必要がある
  - readc/writecの処理の内容を類推して自分で実装するなど
    - readc/writec部分のみならば, さほど難しくはない
    - ただそれでも, qemu内で標準入力の端末属性を変更している部分を 無効化しないと標準入力を期待通りに受けてくれないのでなんだか おかしい入力動作になる
  - 実行ファイル中でreadc/writecを呼び出している部分をNOPなどに 書き換えたり, qemuのデバッガ動作で動かしてスキップさせるなどして 工夫して実行するなど
- なのでいざとなったらあきらめて, 机上デバッグでがんばってアドレス計算 するなども必要
  - ただし単純なプログラムなので, それもさほど難しくはない

# semihostingという機能に気づけるかどうか

- semihostingというのは隠しておくが, そこに気づいてもらう, というような ゲームにしてある
- 気づいたところで, 使うには使いかたを調べる必要がある
  - semihosting_exit()では簡単な引数設定しかしていないので, 引数の設定方法 などが不明確 (そのようにするために, 引数が少ないexitをあえてsemihostingの ヒントに選んでいる)
  - qemuのソースコードを読んで調べる必要がある
  - アーキテクチャごとに調査が必要
  - ただしこれはパッチ無しのqemuでテストできるしソースコード読めばわかるはず なので, 極端に難しい話ではない (ひらめきが必要な話ではない)

# 改行コードが入ってはいけない

- チームフラッグの書き込みには, 改行コードの入力が必要
- しかし改行コードが攻略コードにそのまま入っていると, 攻略コードの注入時に そこで入力終了してしまう
  - 入力の終了を改行コードで判断しているため
  - これは実行ファイルを読めばわかる
- 改行コードを含まない攻略コードを書いて, 改行コードは内部で生成する必要がある

# この問題で提起したい問題

# この問題で提起したい問題

- semihostingというホスト資源にアクセスできる機能がある．qemu特有というわけでもないようだ
- -semihostingオプションをつけないと有効にならないため，ちゃんと使えば安全
- 知ってる人しか使わない機能なので，安全

…と言えるのだろうか？

# この問題で提起したい問題

例えばなんらかのプログラムの動作を見たくて, qemuで実行してみたいとします

- 「このプログラムはqemuで, -semihosting有効で実行してください」とか言われたら, どうしますか?
- インターネット上からダウンロードしたプログラムかもしれません(作った人がどういう人なのかわからないかも)
- 仕事の場合, 「こわいのでそもそも実行しない」「あきらめる」という選択肢は無いかもしれません
  - プロに「実行しなければいい」「あきらめればいい」という選択肢は 多くの場合, ありません
  - プロはそう簡単にはあきらめ(られ)ません

# この問題で提起したい問題

例えばなんらかのプログラムの動作を見たくて，qemuで実行してみたいとします

- 仕事の場合，qemuを動作させられる環境が限定されているかもしれません
  - 開発用環境が限定されており，VM作ってそこでやる，ということが できないかもしれません

- そのまま実行したら，system()呼んで何されるかわかりませんよ？
  - たとえ何をされたかわからない(たぶんされていない)としても，仕事でやっている場合は
    「何をされるかわからない瞬間があった」
    「何もされなかったという証明ができない」
    という時点で，システムを再インストールしなければならなくなるかも しれません

# semihostingの動作やqemuでの実装を知っていれば

semihostingの動作やqemuでの実装を知っていれば, 以下のような対策を 思いつけます

- qemuのsemihosting処理部分を改造して, まずはシステムコールが呼ばれたら その種類や引数を出力して強制終了させる
- 強制終了したときのシステムコールや引数を見て, 妥当ならば少しずつ システムコールを有効にしていく
  - 有効にしたとしても, ログは取るようにすれば, 問題無いことをログにより 他人に証明できる(仕事の場合はこういうのが大事)
- system()は危険すぎるので, 最後まで有効化しないか, 有効にするとしても できることを限定するように改造しておく

# semihostingの動作やqemuでの実装を知っていれば

- qemuの動作を知り，改造するという前提が持てれば，このような対策を思い付くことができます
- 自分で実装して実験することもできます
- 低レイヤーを知っていれば，こうした解決策を提案し，プロトタイプ実装や 実験をした上で，現実味をもって説明することができます
  - 人に説明するときは，この「現実味」というのがすごく重要

# 低レイヤーを知っていれば

- 「今どきアセンブラなんて知っている必要は無い」とかいう話をよく聞きます
- しかしそういうのはアセンブラを知らない人が，知らないのでそもそもこういう 対策を思い付くこともできないし，自分は思い付けないのでメリットも考えつか ないままに言っている場合が多いと思います
- 個人的には，アセンブラを知っていて役に立つことだらけです
- ただ低レイヤー(に限らないかもしれませんが)の知識は，このようにして自分自身で「役立てようとすること」が大事です
  (持っているだけで黙っていれば役に立つ，ということもありません)
- 逆に言えば，自分で役立てようとすれば，いろんなところですごく役に立ちます

# 問題には, 様々な意図があります

- 問題には, 様々な意図があります
- それらをすべて知ってほしいとは思いませんが, 無駄な知識というものは無く, 勉強にならない問題というものも無いです
- 本人がいかにそこから学ぶか, いかに役に立てるかです
(勉強になるかどうかは問題次第ではなく, 本人次第です)
- あと問題の意図がわかると面白いし学習モチベーションも上がるので, そうしたことを問題作成者に聞くとか, どこかで喋ってくれるように お願いするとかするといいでしょう
- 自分も問題作成のコミュニティに入ると, そういうのは本人に直接詳細を聞ける のでおすすめです. 自分でも問題を作ってみよう!

どうもありがとう
ございました