Step-Oriented Programmingによる任意コード実行の可能性

坂井弘亮

自己紹介

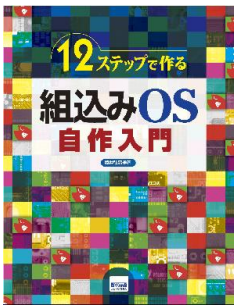
富士通株式会社 ネットワークサービス事業本部
富士通セキュリティマイスター(ハイマスター領域)
セキュリティ&プログラミングキャンプ(現セキュリティ・キャンプ)講師
SECCON実行委員
SecHack365実施協議会委員
クリティカルソフトウェアワークショップ プログラム委員
個人でのイベント出展・セミナー登壇多数
アセンブラ短歌 六歌仙の一人
バイナリかるた発案者
フリーソフトウェア作成
技術士(情報工学部門)

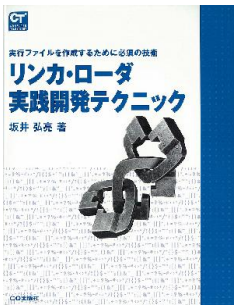
http://kozos.jp/

自己紹介

安価なマイコンボードで動作する独自組込みOS「KOZOS」をホビープログラミングとして 自作し，ブートローダー，シンプルなマルチタスク・カーネル，デバイスドライバ，簡易TCP/IPスタック，簡易Webサーバ，デバッガ対応，シミュレータ対応等を実装，フルスクラッチのソフトウェアによるWebサーバを動作させる．

さらにオープンソースソフトウェアとして公開し，各種イベントに出展・登壇 (オープンソースカンファレンスなど)．





自己紹介

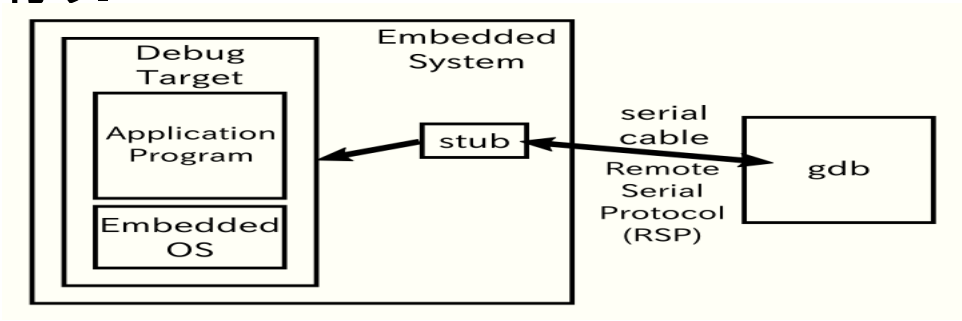
組込み技術者です

本日の話は，組込み技術者がセキュリティをやったらどうなるか，という話です

概要

Step-Oriented Programming による任意コード実行の可能性

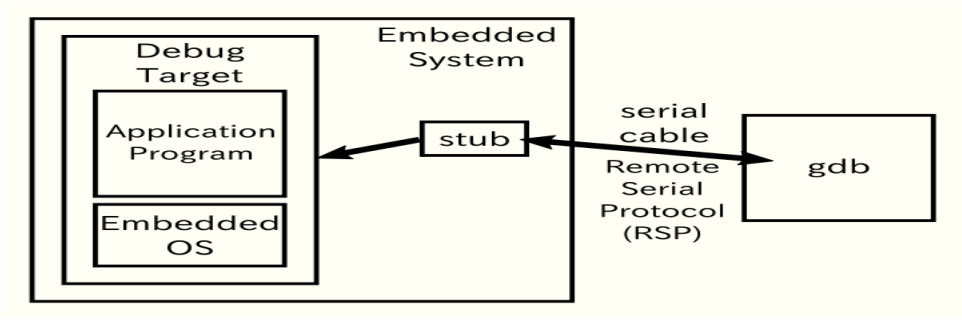
組込み機器ではホストPCと接続してリモートでプログラムのデバッグを行うために，スタブと呼ばれる独立した制御プログラムを内蔵し，メイン・プログラムを制御する ことでデバッガによるデバッグを可能にしている．スタブはレジスタ値やメモリの 読み書きなどの単純な制御のみを行うことで簡略化されており，解析などの複雑な 処理はホストPC上でデバッガが行う．



概要

Step-Oriented Programming による任意コード実行の可能性

ホストPC上のデバッガと組込み機器上のスタブとの通信は リモート・シリアル・プロトコル（RSP）と呼ばれるプロトコルにより，シリアル通信やTCP/IP通信により行われる．この通信が奪われた場合，スタブの任意の操作が可能となり，その場合の攻撃の可能性としては，プログラム・カウンタの値を変更しながらステップ実行を繰り返すことで 機械語コードの断片を組み合わせて任意コードを実行する Step-Oriented Programming（SOP）が考えられる．

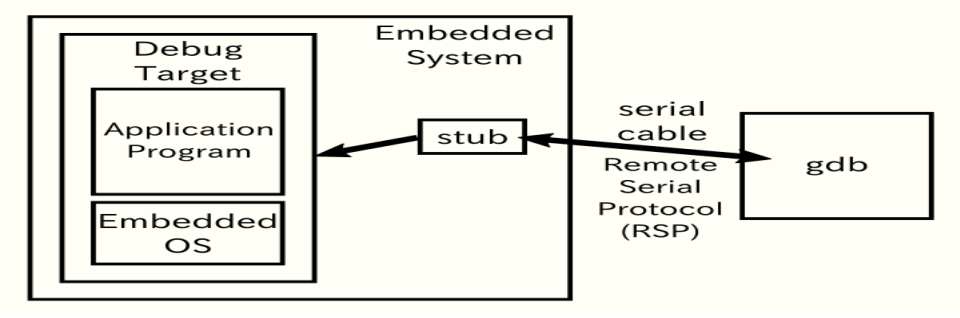


概要

Step-Oriented Programming による任意コード実行の可能性

これはDEPによるデータ領域の 実行防止やフラッシュROM上の機械語コードしか実行できないなどの理由で，注入された機械語コードの実行が不可能であったとしても，既存の機械語コードから 任意コードを組み立てての実行が可能である．

SOPによる攻撃の原理と実際に検証コードを組み立てて検証した結果について，デモを含めて説明する．



本日の流れ

- 組込み機器とは何か
- 組込み機器のリモートデバッグ対応
- リモートデバッグのための通信プロトコル
- Step-Oriented Programming (SOP) による攻撃可能性
- SECCON CTFでの実験

重要!

本研究の目的は通信が奪われた場合の攻撃の可能性を知ることで本質的な防御を 検討でき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ようなものでは ありません.

また本研究の目的は, 攻撃の危険を知ることで組込み機器のセキュリティを啓発する ことにあります.

他者のサーバや機器を許可無く攻撃・検証することは, 不法です. やってはいけません.

結論

- デバッグポートが開いていると, 何でもされ得る (当然)
- 機械語コードの注入ができないからといっても, 安全とは言えない
  (SOPにより, 任意コードの実行は可能)
- デバッグポートはきちんと閉じておこう!
  (可能ならばスタブのコード自体を削除する, 回路自体を削除するなど)

組込み機器とは何か

組込み機器とは何か

- 炊飯器, エアコン, 自動車, コンビニの中華まんスチーマ, ...
- マイコンでソフトウェア制御される機器

組込み機器とは何か

PCやサーバは「汎用機器」

- ユーザがアプリケーションをインストールして汎用的に使う (主役はアプリケーション)
- 資源は増設すればいい(ユーザがCPU増強やメモリ増設をできる)
- 資源は多いほうがいい(「大は小を兼ねる」の考え方)

組込み機器は「専用機器」

- 固定されたソフトウェアで固定の用途に使う (主役は機器そのもの)
- 資源が限定される(ユーザがCPU増強やメモリ増設をしない)
- 資源は必要なだけあればいい(「適材適所」の考え方)

組込み機器とは何か (セキュリティの視点で)

組込み向けマイコンでソフトウェア制御されている

- 結局はPCやサーバと同じ, コンピュータ・システム
- それらと同じ手法で攻撃されてしまう可能性がある

我々が意識しないところでも, そこらじゅうで大量に動いている (各種家電, 通信機器, スマートメーター, …)

PCやサーバに比べると...

- 扱っている情報資産は極少
- 個数は極大
- 近年その台数が増加しているが, 管理が不十分なことも多い

組込み機器のプログラムの開発

開発環境と実行環境が大きく異なる

- ホストとターゲットが別々になる
  - ホストは今どきのPCだったりサーバだったり
  - ターゲットはx86じゃなかったり8ビットマイコンだったり

- クロスコンパイル
  - ホストでコンパイルし，ターゲットで実行する

- リモートデバッグ
  - ホストでデバッガを起動し，ターゲットのデバッグポートに シリアルケーブルなどで接続してデバッグを行う

組込み機器のデバッグポート

多くの組込み機器は，デバッグ用のポートを持つ

- シリアルポートだったり，TCP/IPによる接続だったり

製品にも隠し機能で残っていたり，回路が残っていてコネクタを半田付けしたら使えてしまったり…

- 外にある組込み機器を，夜中にクラックされる懸念

デバッグポートを奪われたら，何でもできてしまう (当然)

- デバッガの上で動作しているプログラムを自由に制御できることと同じ
- メモリ書き換え，プログラムカウンタ変更など
- シェルコード注入して実行など何でも可能

デバッグポートを奪われたら
もちろん何でもできてしまうわけだが

しかし, 実際のところどのような攻撃がされ得るのだろうか. . . ?

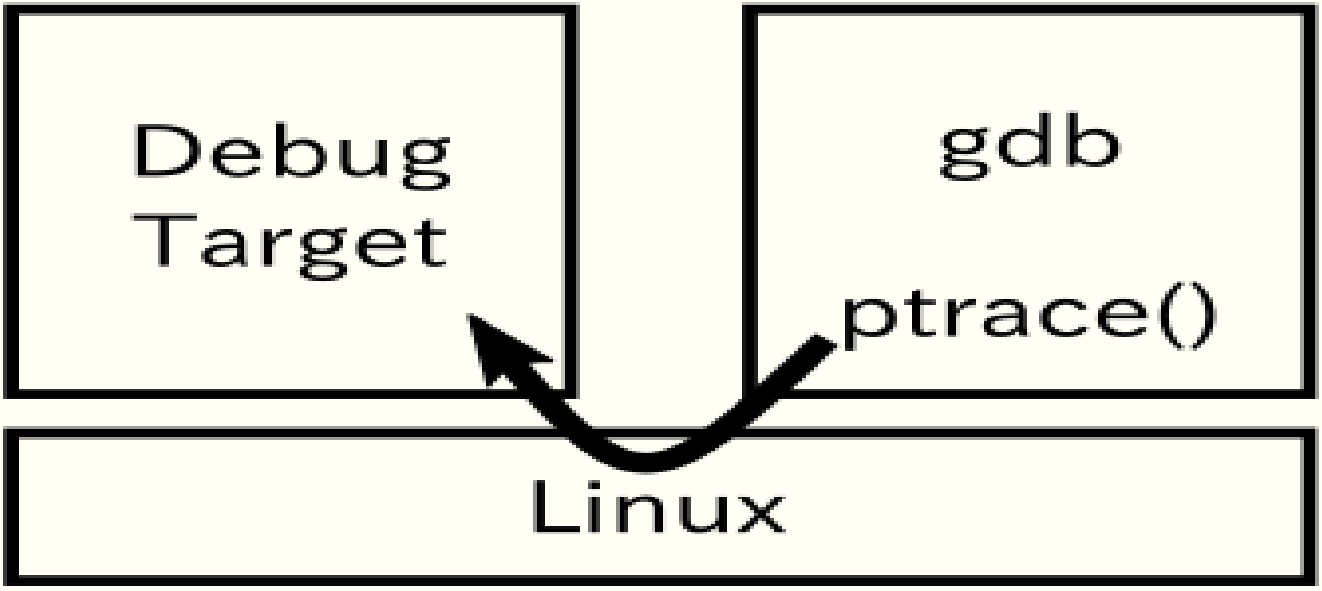
- たとえばマイコンでは, 実行コードを注入して実行するようなことが 原理的にできないアーキテクチャもある

そうしたことを考えてみようというのが今回のテーマです

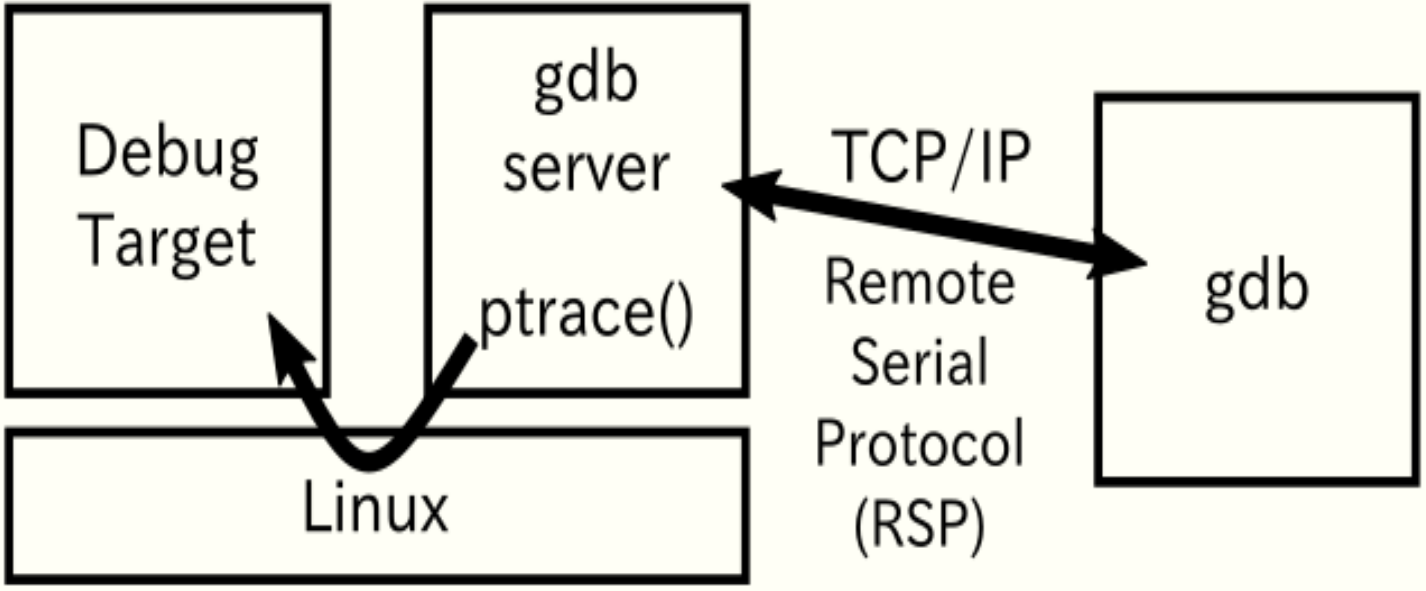
- デバッグポートがどのように奪われるかについては扱いません
- 今回のテーマは, デバッグポートが奪われたら何をされ得るか, です

組込み機器のリモートデバッグ対応

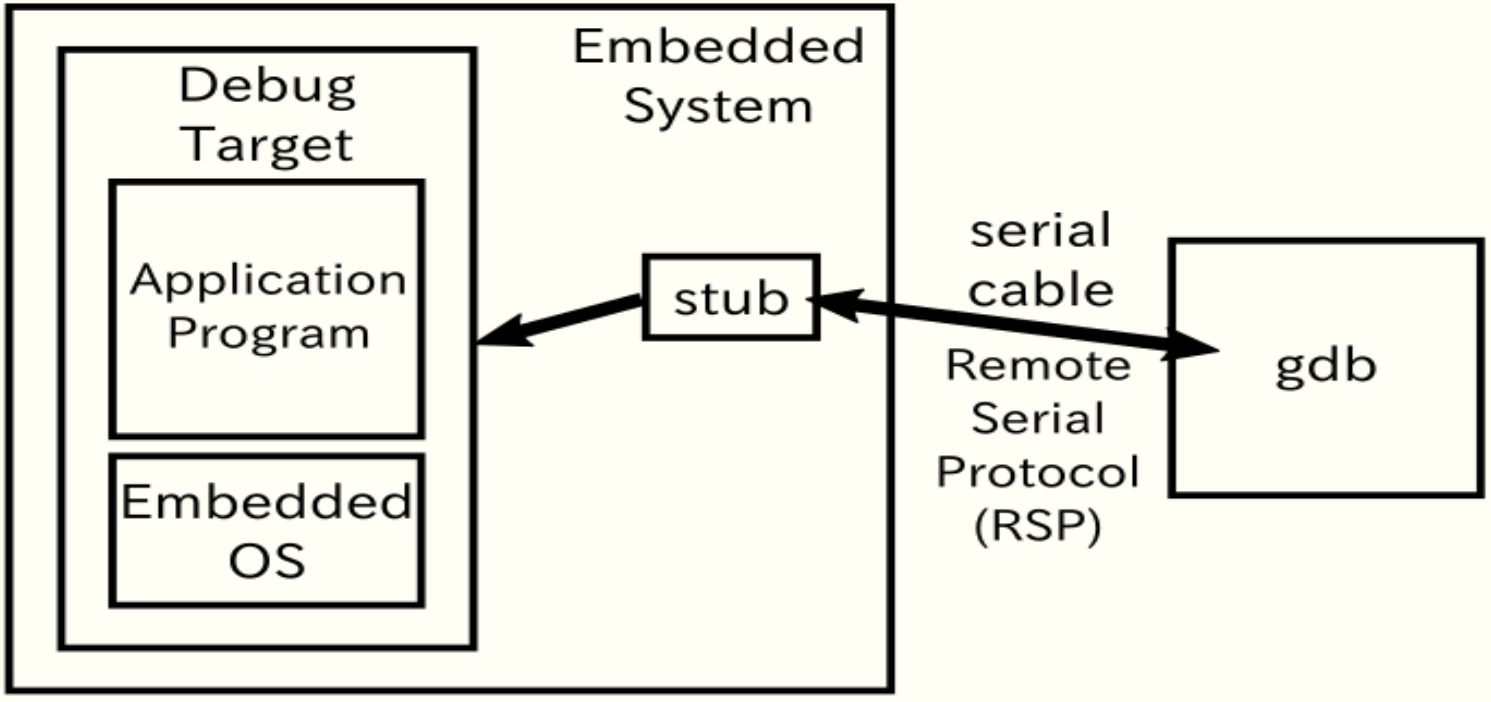
通常のデバッグ



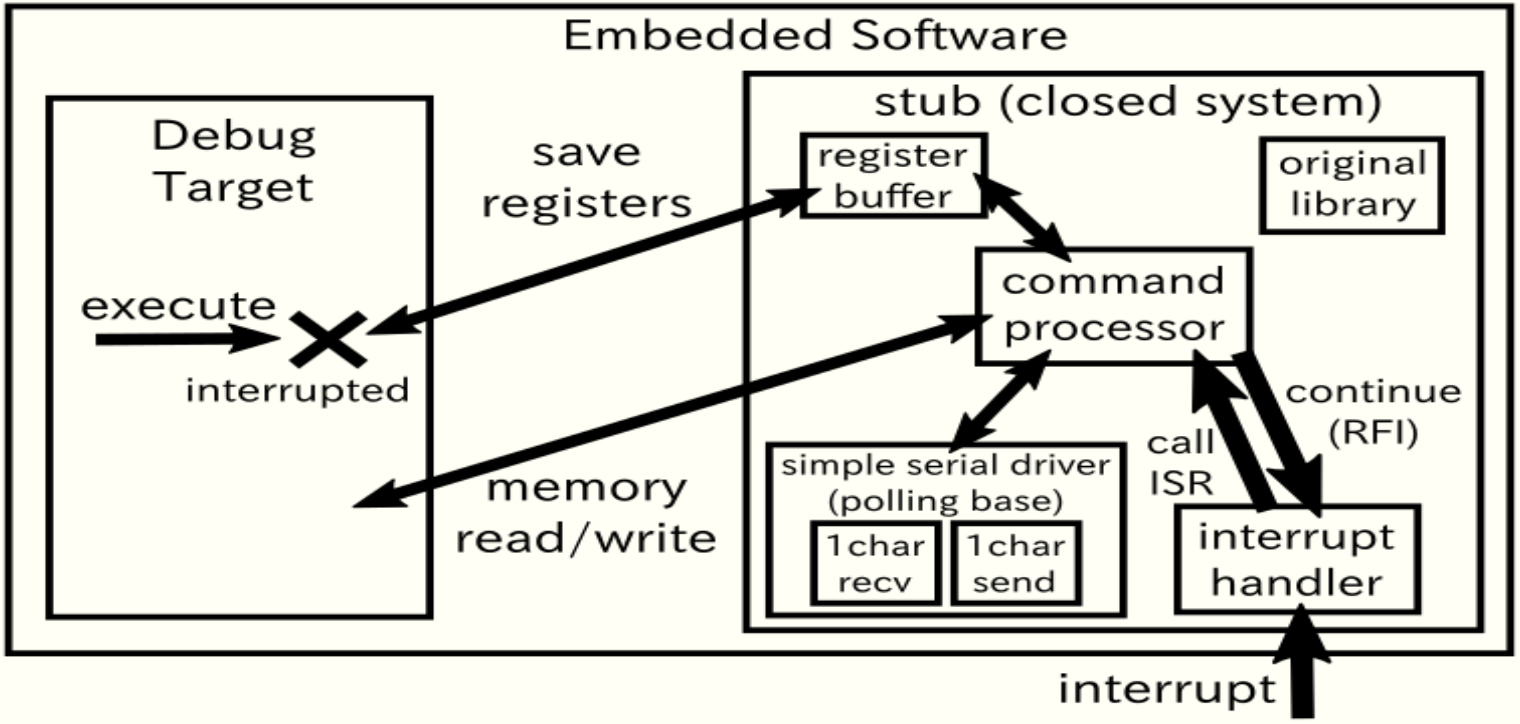
gdbserverを利用したデバッグ



組込み機器のデバッグ


Embedded System
Debug Target
Application Program
Embedded OS
stub
serial cable
Remote Serial Protocol (RSP)
gdb

スタブの構成


Embedded Software
Debug Target
execute
interrupted
save registers
memory read/write
stub (closed system)
register buffer
original library
command processor
simple serial driver (polling base)
1char recv
1char send
call ISR
continue (RFI)
interrupt handler
interrupt

スタブの実装

```
p = recvbuf;
switch (*(p++)) {
...
case 'g':
  read_memory(registers, sendbuf, REGISTERS_SIZE);
  break;

case 'G':
  write_memory(p, registers, REGISTERS_SIZE);
  stub_strcpy(sendbuf, "OK");
  break;
...
```

スタブの実装

```
case 'm':
  a2val(&p, &addr);
  if (*(p++) != ',') {
    stub_strcpy(sendbuf, "E01");
    break;
  }
  a2val(&p, &size);
  read_memory((void *)addr,
              sendbuf, size);
  break;
...
```

```
case 'M':
  a2val(&p, &addr);
  if (*(p++) != ',') {
    stub_strcpy(sendbuf, "E01");
    break;
  }
  a2val(&p, &size);
  if (*(p++) != ':') {
    stub_strcpy(sendbuf, "E02");
    break;
  }
  write_memory(p, (void *)addr, size);
  stub_strcpy(sendbuf, "OK");
  break;
...
```

リモートデバッグのための通信プロトコル

RSP (リモート・シリアル・プロトコル)

ホスト上のデバッガと, ターゲット上のスタブとの間で デバッグのための処理情報を伝達するためのプロトコル

簡単な操作コマンドのみが定義されている

- レジスタ読み書きやメモリ読み書きなど
- 複雑な操作はホスト側のデバッガが, コマンドを組み合わせて実現する

RSPでできること

- レジスタ値の取得
- レジスタ値の設定
- メモリのリード
- メモリへのライト
- 等々...

つまり, 何でもできる

メモリ上に実行コードを注入してプログラムカウンタを書き換えて実行できてしまう

RSPのプロトコル

フォーマット

```
$<command>{<parameter>}#<checksum>
```

コマンドとパラメータ

- 例えばステップ実行は「s」, 動作継続(continue)は「c」
- 後続してパラメータを持つものもある(メモリ読み書きなど)

チェックサム

- コマンド+パラメータの部分の加算値(1バイト)

RSPのプロトコル

応答

- 正常に受信できたらAckとして「+」を返す
  (チェックサムエラーなどはNakとして「-」を返し，再送をうながす)
- コマンド実行できたら「OK」を返す
- エラーは「Enn」を返す (nnはエラー種別を表す番号 例:E01)
- コマンドが未サポートの場合には，空コマンド(「$#00」)を返す

RSPのコマンド例

動作	コマンド列
ステップ実行	$s#73
レジスタ値の取得	$g#67
メモリのリード	$m2400,10#c0

プリミティブなコマンド

コマンド	意味
s	ステップ実行(step)
c	動作継続(continue)
g	レジスタ値の取得
G	レジスタ値の設定
m<address>,<size>	メモリのリード
M<address>,<size>	メモリへのライト
D	デタッチ

その他にもたくさんのコマンドが定義されているが，上のコマンドが使えればgdbでのだいたいのデバッグ操作は可能

もっと高級なコマンドもあるが，スタブが対応していなくて使えない場合は 別の簡単なコマンドを使うようにgdbが自動的に調整する

デモ
(RSPによるターゲットの制御)

注意

- gdbでのcontinueは，スタブにcコマンドを指示するだけではない
- つまり，gdbのコマンドがRSPのコマンドにそのまま対応しているわけではない

gdbはRSPの様々なコマンドを組み合わせて，continueなどの動作を実現している

動作継続(continue)の例

gdb上でブレークポイントから動作継続(continue)する際には，「cコマンド」をただ実行するわけではない

期待通りにcontinueするためには，以下のような操作が必要
(ソフトウェア・ブレークポイントの場合)

- sコマンドで1命令だけステップ実行で進める
- mコマンドで，ブレークポイントの位置の命令を読み込み保存する
- Mコマンドでブレークしていた箇所にトラップ命令を埋め込む (これをやらないと，次回に正常にブレークできないため)
- cコマンドで動作継続する

gdbがこれらのコマンドを自動的に発行することで，continueの動作が実現されている

ARMでのブレークポイント設定時の発行コマンド

```
(gdb) set debug remote 1
(gdb) target remote localhost:10000
...
(gdb) break main
Sending packet: $m20c8,4#ca...Ack
Packet received: 04e02de5
Breakpoint 1 at 0x20c8: file arm-elf.c, line 39.
(gdb) continue
Continuing.
Sending packet: $Z0,20c8,4#13...Ack
Packet received:
Packet Z0 (software-breakpoint) is NOT supported
...
```

Z0コマンドが使えるか試す
(パラメータ渡しができるソフトウェア・ブレークポイント設定)

→ 実装されていない

ARMでのブレークポイント設定時の発行コマンド

mコマンドで，ブレークポイントに配置されている
機械語コードが読み込まれる → 保存しておく

```
...
Sending packet: $m20c8,4#ca...Ack
Packet received: 04e02de5
Sending packet: $X20c8,0:#eb...Ack
Packet received:
binary downloading NOT supported by target
Sending packet: $M20c8,4:fedeffe7#e0...Ack
Packet received: OK
Sending packet: $vCont?#49...Ack
Packet received:
Packet vCont (verbose-resume) is NOT supported
Sending packet: $Hc0#db...Ack
Packet received:
Sending packet: $c#63...Ack
```

Xコマンドが使えるか試す
(バイナリ通信による高速なメモリ書き込み)

→ 実装されていない

Mコマンドで，ブレークポイントにトラップ命令を埋め込む

vContコマンドが使えるか調べる
(動作継続のマルチスレッド拡張)

→ 実装されていない

Hc0コマンドが使えるか試す
(スレッドごとの動作継続指定)
→ 実装されていない

cコマンドで動作継続する

徐々に簡単な命令に落とし込まれていく

ARMでのcontinue実行時の発行コマンド (ブレーク時)

gコマンドでレジスタ一覧を取得する
(プログラムカウンタが含まれるため，ブレーク位置がわかる)

```
Packet received: T05
Sending packet: $g#67...Ack
Packet received: 000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000001028000008200000c82000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000d3000000
Sending packet: $m20c8,4#ca...Ack
Packet received: fedeffe7
Sending packet: $qL1160000000000000000#55...Ack
Packet received:
Sending packet: $M20c8,4:04e02de5#0d...Ack
Packet received: OK

Breakpoint 1, main () at arm-elf.c:39
39      arm-elf.c: No such file or directory.
(gdb)
```

mコマンドで，ブレーク位置の機械語コードを確認する

Mコマンドで，ブレーク位置の機械語コードを元に戻す

ARMでのcontinue実行時の発行コマンド (continue実行時)

```
(gdb) continue
Continuing.
Sending packet: $Hc0#db...Ack
Packet received.
Sending packet: $s#73...Ack
Packet received: T05
Sending packet: $g#67...Ack
Packet received: 00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000c2800000820000 0cc2000
00000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000d3000000
Sending packet: $m20cc,4#f5...Ack
Packet received: 0100a0e3
Sending packet: $m20c8,4#ca...Ack
Packet received: 04e02de5
Sending packet: $M20c8,4:fedeffe7#e0...Ack
Packet received: OK
Sending packet: $Hc0#db...Ack
Packet received.
Sending packet: $c#63...Ack
```

sコマンドで，1命令だけステップ実行する
(ブレークポイントの次の命令で停止する)

gコマンドで，停止位置を確認する

mコマンドで，ブレークポイントの命令を取得し保存する
(次回のブレーク時の命令復旧のため)

Mコマンドで，ブレークポイントにトラップ命令を埋め込む
(トラップ命令は0xe7ffdefe)

cコマンドで，動作継続する

ステップ実行の実現

「sコマンド」によるステップ実行は，スタブ側で以下のように処理される

- ステップ実行例外を利用する方法
  - ステップ実行例外 ... 1命令を実行するごとに発生する例外
  - sコマンド受信時に，多くはCPUのステータスレジスタを操作して ステップ実行例外を有効にして動作継続する
  - 命令書き換えをせずに，1命令ごとにブレークできる
  - GDBのスタブのサンプルでは，x86，IA-64，68000がそのような実装になっている

ステップ実行の実現

「sコマンド」によるステップ実行は，スタブ側で以下のように処理される

- スタブ側でトラップ命令を埋め込む方法
  - sコマンド受信時に，スタブがプログラムカウンタの次の命令を 保存しトラップ命令に書き換えることでブレークさせる
  - 条件分岐命令がある場合には，分岐命令の直後と分岐先の両方にトラップ命令を 埋め込む必要がある
  - ブレーク時には保存していた命令に復旧する必要がある
  - GDBのスタブのサンプルでは，M32R，SHがそのような実装になっている

圧縮プロトコル

4文字以上の同じ文字の連続を圧縮できる

Gコマンドによるレジスタ設定などで効果がある
(設定不要なレジスタはゼロを指定することで，設定値を圧縮できる)

フォーマット

〈w〉*〈c〉

〈w〉	連続する文字
〈c〉	n = c - 29 として，<w>をさらにn個連続させる (c = '<' の場合は n = '<' - 29 = 60 - 29 = 31個)

圧縮プロトコル

ARMでの例

```
#   R0       R1      R2-R13    LR       PC
#+------++------++-------++------++------+
G00240000000000000*<0*<0*<000000024200000
```

0が32個連続 ×3回 → 0が96個 → レジスタ12個ぶんの設定

展開すると，以下のようになる

```
G00240000000000000000000000000000000000000000000000...
...000000024200000
```

Step-Oriented Programming (SOP) による攻撃可能性

デバッグポートを奪われたら, 本当に何でもできるのか？

デバッグポートを奪われたら, Mコマンドで機械語コードを注入して Gコマンドでプログラムカウンタを機械語コードを指すようにして cコマンドで実行してしまえば, シェルコード実行が簡単にできてしまう (当然)

デバッグポートを奪われたら，本当に何でもできるのか？

しかし，機械語コードを注入して実行できない場合もある

- 機械語コード領域が書き換えられない可能性
  - メモリ保護でガードされている
  - スタブがMコマンドによるメモリ書き込みの際に，アドレスチェックしている
  - 実行コードがフラッシュROM上にある

- データ領域が実行できない可能性
  - DEPが効いている
  - RAMの容量が少なく機械語コードを注入しづらい
  - ハーバード・アーキテクチャのマイコンで，RAM上の機械語コードが原理的に実行できない

デバッグポートを奪われたら, 本当に何でもできるのか？

ハーバード・アーキテクチャ

- RISC的には, 命令キャッシュとデータキャッシュが分離されていること
- マイコン的には, 機械語コードを置くフラッシュROMのアドレス空間とデータを置くRAMのアドレス空間が分離されていること (例: AVR)

(機械語コードを注入しての実行が, 原理的にできない)

しかし, だから安全だとは言えない

SOPの原理

機械語コードを注入して実行することができない場合でも，以下を繰り返せば，ROPと同様に既存の機械語コードを命令単位で組み合わせて 任意の機械語コードを実行されてしまう

- Gコマンドでプログラムカウンタを，実行したい機械語コードのアドレスに書き換える
- sコマンドでステップ実行する

RSPによるステップ実行をベースとした任意コード実行
(SOP: Step-Oriented Programming)

つまり，やはり危険である

SOPでできること

- 機械語コード領域を書き換えることができなかったり, RAM上の機械語コードを 実行できない場合でも, 既存の機械語コードを利用して組み立てることで任意コードが実行されてしまう

- プログラムカウンタの値を自由に設定できてステップ実行により 1命令単位で実行できるため, ROPなどよりも容易に任意コードの組み立てが可能

ステップ実行のために

命令書き換えができない場合にも，ステップ実行は可能

- ステップ実行で，ステップ実行例外が利用される場合には，たとえ機械語コードの 書き換えができないとしても，機械語コードを書き換えずにSOPが可能
- Mコマンドによるメモリ書き換えにはアドレスチェックが入っていても，ステップ実行によるトラップ命令埋め込みにはアドレスチェックが入っていない スタブの実装もある

ステップ実行のために

命令書き換えができない場合にも，ステップ実行は可能

- ステップ実行が不可能な場合にも，CPUがハードウェア・ブレークポイントの 機能を持っていてスタブがそれに対応(Z1コマンド)していれば，1つ先の命令にブレークポイントを仕掛けることでSOPは可能

- CPUがウォッチポイントの機能を持っていれば，レジスタ書き換えとロード命令や ストア命令と組み合わせることで，適当な箇所でブレークさせてSOPが可能

SECCON CTFでの実験

SECCON CTFで出題し実験

SECCON2016 オンライン予選でSOPによる任意コード実行の問題を実験的に出題

SECCON2016 決勝大会で最小SOPを競う競技形式にして出題

- サーバ上でARMなどの4種類のアーキテクチャの実行ファイルがシミュレータ上で 動作している
- デバッガの接続ポートが開いており，TCP経由でRSPで接続できる
- サーバのカレント・ディレクトリの「word.txt」というファイルからキーワードを読みスコアサーバにコミットすると得点
- サーバのカレント・ディレクトリの「flag.txt」というファイルに チームのキーワードを書き込むと定期的に得点
(最小サイズのSOPを作成したチームだけが書き込める)

注意

RSPで使えるコマンドは s, c, g, G, m の5種類

Mコマンドは無効化されている

- 競技として工夫の要素が無くなるため (メモリ書き換えができると, 機械語コードを注入して実行するだけの 簡単な問題になってしまう)
- 機械語コードを注入して実行することができない場合を想定

攻略にはSOPが要求される
果たして, SOPは使われるのか…?
(SOPは誰でも考えつく手法なのか…?)

重要!

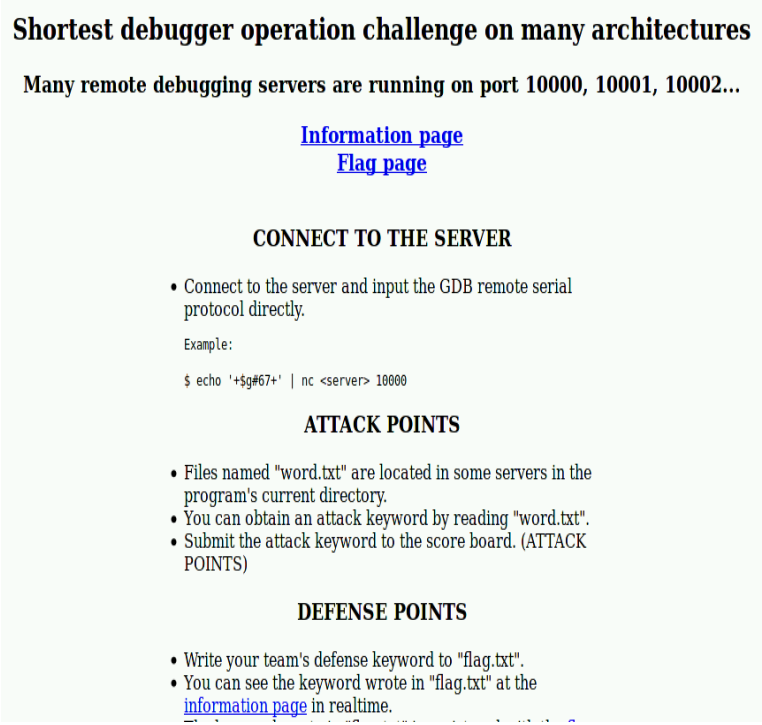
通信が奪われた場合の攻撃の可能性を知ることで本質的な防御を 検討できるようにするという目的で実施しています. 不正な攻撃を 助長するようなものではありません.

他者のサーバや機器を許可無く攻撃・検証することは, 不法です. やってはいけません.

出題した内容

競技者に与える情報は以下のみ

- TCPで接続するためのポート番号
- RSPで接続できる，ということ
- word.txtのキーワードを
  読むと得点，ということ
- flag.txtにキーワードを
  書き込むと得点，ということ
- 接続後に入力するコマンド例
  (「+$g#67+」)



Shortest debugger operation challenge on many architectures

Many remote debugging servers are running on port 10000, 10001, 10002...

Information page
Flag page

CONNECT TO THE SERVER

- Connect to the server and input the GDB remote serial protocol directly.

  Example:

  ```
  $ echo '+$g#67+' | nc <server> 10000
  ```

ATTACK POINTS

- Files named "word.txt" are located in some servers in the program's current directory.
- You can obtain an attack keyword by reading "word.txt".
- Submit the attack keyword to the score board. (ATTACK POINTS)

DEFENSE POINTS

- Write your team's defense keyword to "flag.txt".
- You can see the keyword wrote in "flag.txt" at the information page in realtime.

出題のポイント

- gdbで接続せず, RSPで直接操作するのが前提, ということに気づく必要がある
- 接続先のアーキテクチャが何なのか, 知る必要がある
- RSPで使えるコマンドを調べ, 使えるコマンドと使えないコマンドを知る 必要がある
- Mコマンドが使えないことに気づく必要がある
- どうすればファイルの読み書きができるか, 考える必要がある (SOPによる任意コード実行の手法を思いつくか...?)

しかし結果としては, 多数のチームがSOPにより攻略していた
(SOPは誰でも考えつく手法である, と言える)

アーキテクチャ一覧

アーキテクチャ名	種類	特徴
ARM	32ビットマイコン	4バイト固定長命令セット
H8/300	16ビットマイコン	可変長命令セット
SH	32ビットRISCマイコン	2バイト固定長命令セット
V850	32ビットRISCマイコン	4バイト固定長命令セット

対象サーバが分散すると競い合いにならないので，
分散しないように少数のアーキテクチャで実施

デモ
(RSPによる接続)

デモ
(SOPによるキーワード取得)

実施してみて

結果としては, 多数のチームがSOPにより攻略していた

- SOPはある程度の知識があれば, 誰でも考えつく手法であると言える (特別な手法ではないし, 新しい手法でもない)

GDB付属のARMシミュレータの浮動小数演算エミュレーションの組込みライブラリの 既存コードを利用

- 利用できる機械語コードが豊富になる
- そのようなライブラリの存在は知らなかった

RSPの圧縮プロトコルに気づいて活用できたチームが圧倒的に有利

結論

- デバッグポートが開いていると, 何でもされ得る (当然)
- 機械語コードの注入ができないからといっても, 安全とは言えない
  (SOPにより, 任意コードの実行は可能)
- デバッグポートはきちんと閉じておこう!
  (可能ならばスタブのコード自体を削除する, 回路自体を削除するなど)

どうもありがとう
ございました