脆弱性攻撃の理論と防御技術 I

at 福井大学 セキュリティゼミ 6/13

@elliptic_shiho

自己紹介

- 緑川 志穂(@elliptic_shiho)
- 何の変哲もない高校1年生
- 京都の田舎から来ました
- CTF(vuls, scryptos, Ellipse)
- 暗号が専門

はじめに

- 各章毎に一応休憩/質問タイムは取りますが、質問は別にいつしてもOKです
- 自由な感じでやっていきましょう

ねらい

- 脆弱性とは?といった基礎的な部分から、実際に現時点で使われている攻撃手法までの理解
- CTFを通じて実際に手を動かし、理解を深める
- 脆弱性に対する考え方、モラルについての理解

ねらい

- 脆弱性とは?といった基礎的な部分から、実際に現時点で使われている攻撃手法までの理解
  - 防御のためには攻撃を知らないとダメ
    - 火縄銃に対してRPG-7用意しても変な方向に打ってたら意味は無い
    - それにRPG-7高い
      - 効率よく!も重要

ねらい

- CTFを通じて実際に手を動かし、理解を深める
  - Capture The Flagという競技の説明
  - 毎回最後にテスト的な意味も含めて簡易的な物を行います
  - 実際に開催されたCTFの過去問からも出題します
    - 高ポイント狙って行きましょう

ねらい

- 脆弱性に対する考え方、モラルについての理解
  - 無闇矢鱈と攻撃するものじゃない
  - 脆弱性を作ることで起こる被害、二次被害、損害等に対する意識
    - 「動けばいい」では済まないのです...

いろいろと

- かたいことを言いましたが、楽しくやって欲しいです
  - セキュリティ分野は人材不足も叫ばれています
  - モバイルデバイスやIoTの普及に伴って更に攻撃対象も増えました。
  - 是非これらの知識を他に活かして欲しいです。

おしながき

- 基本的なアセンブラの読み方
- プログラムの解析演習
- 攻撃手法解説
- 攻撃演習(CTF)

おしながき

- 基本的なアセンブラの読み方
- プログラムの解析演習
- 攻撃手法解説
- 攻撃演習(CTF)

アセンブラ

- いわゆる機械語を人に読みやすくしたもの
  - 0xf4 <=> HLTとか普通脳内変換できない
- 原始的な操作しかできない

アセンブラ

- レジスタという概念がある
  - Cで言う変数のようなもの
- x86アーキテクチャならばeax, ecx, edx, ebx, esp, ebp, esi, edi

アセンブラ

- それぞれの名前は由来がある
  - eax = accumulator
  - ecx = counter
  - edx = data
  - ebx = base
  - esp = stack pointer
  - ebp = base pointer
  - esi = source index
  - edi = destination index

アセンブラ

- 何故このような名前なのか
  - eaxは算術演算の命令で短く記述できるようになっている(ex, add eax, ***命令)
  - ecxはカウンタとして扱うことを前提に命令体系が組まれている (ex, loop命令)
- 他にもあるので気になった方はお調べ下さい。

アセンブラ

- メモリ使わないの?

  →使います

- メモリはCPUと物理的に離れている(数十cm)
  - CPUにとっては非常にアクセスの遅いデバイス
- レジスタで基本的な処理をし、メモリアクセスは最小限にするのがベターとされている

アセンブラ

- 算術命令
  - add, sub, imul, idiv, or, and, not, xor, ...
- 転送命令
  - mov, lea, ...
- ジャンプ命令
  - jmp, call, ja, jne, jc, ...

アセンブラ

- 一時的にデータ退避したい時はどうするのか
  - メモリ上に専用の領域が用意されている
  - スタックといいます
- スタックの位置はespに代入されています

アセンブラ

- スタックはLIFO型のキューです
  - Last In, First Out
  - 情報やってるならわかると思います
- スタックに値を入れる時は"積む"、値を取り出す時は"取る"とか言います

アセンブラ

- スタックは実際はメモリ上にどう配置されてるのか?
  - 下に伸びていくメモリ領域

アセンブラ


上位アドレス
0xbffffc0c
0xbffffc08
0xbffffc04
0x12345678
0xdeadbeef
0xcafebabe
0xa1b2c3d4
下位アドレス0xbffffc00

アセンブラ

- こういう配置
- "下方向に伸びていく"という点によく引っかかるので注意

アセンブラ

- スタックに値を積んだり取り出したりする時はどうするのか
  - 単純に考えると
  - `mov [esp], imm32`
  - `sub esp, 4`
- 毎回書くのは面倒臭いよね?

アセンブラ

- スタック操作命令

  push/pop

- push eaxやpop dword [0xdeadbeef]のように書く

アセンブラ

- 大体説明するべきものはこれぐらい
- 後は実際にコンパイルされた結果を見たり、Intelの仕様書を読んでアセンブラを作ったりすると良さある

休憩/質問タイム

おしながき

- 基本的なアセンブラの読み方
- プログラムの解析演習
- 攻撃手法解説
- 攻撃演習(CTF)

プログラム解析

- 演習用のプログラムは以下よりダウンロードして下さい
- http://192.168.0.2:8080/

プログラム解析

- Q. IDA Pro Freeという便利なソフトがあってな
- A. 今回はアセンブラに慣れてもらう意味も含めて使いません
  - 使いたきゃ使ってもいいですが、演習では全てobjdumpを使います

プログラム解析

- 今回はobjdumpとreadelfというツールを使います
- 汎用的なディスアセンブルツール
- Linuxなら大抵の環境に入ってます

プログラム解析

- objdumpの簡単なオプション解説
- -d 逆アセンブルする指定
- -M (att|intel) アセンブラの形式をAT&T形式とIntel形式のどちらにするかを指定する
- -W 付けると横の文字数制限が無くなる

プログラム解析

- `objdump -d -M intel foo`
- 何やら出ましたね

- 読み方デモ

プログラム解析

- 目的は?
  - どんな動作をしているのか
- 見るべきポイント
  - どんな関数が使われているのか
  - jmp/call
  - 条件分岐っぽい場所も要点

プログラム解析

- なんとなく改行とかコメントとか入れるだけでもかなり様になる
- エディタはご自由に

プログラム解析

- readelf?
- てかそもそもELFとは?
  - Linuxの実行ファイルのフォーマット
- readelfはELFに含まれる情報をわかりやすく表示するツール
  - JPEG画像開くと撮影日時出てくるアレみたいな感じ

プログラム解析

- readelfの使い方

  -a 以下全てのオプション指定と同じ

  -h ヘッダ情報表示

  -l プログラムヘッダ/セグメントの表示

  -S セクションの表示

  -d 動的リンク情報表示

プログラム解析

- readelfの使い方

正直面倒臭いから-aだけでいい

プログラム解析

- `readelf -a`
- 沢山出た
- 読み方解説

プログラム解析

- 先ほどのobjdumpの結果と照らし合わせてアドレスを確認...
- ついでにC言語へ直してみましょう

プログラム解析

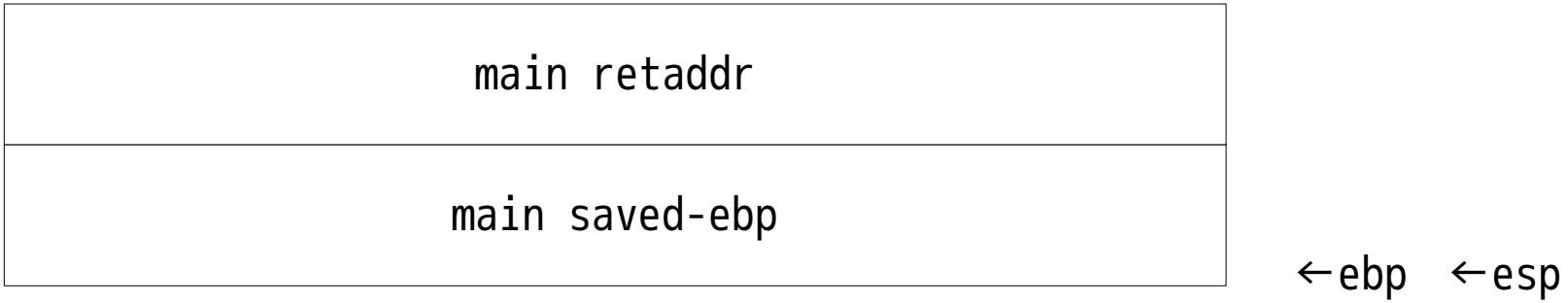
- `push ebp; mov ebp, esp`
  - 関数開始時の決まり文句
  - スタックの位置を新しくするためにある
  - このebpをベースポインタといいます

→ Cにする時には書かなくても良い

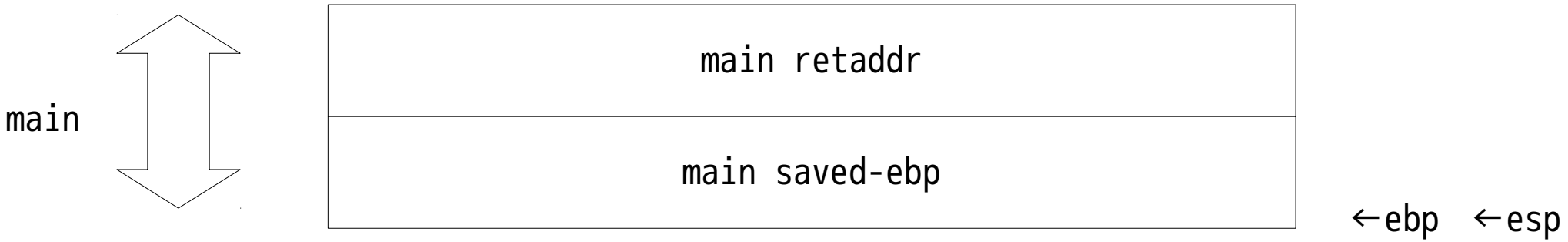
プログラム解析

- スタックポインタあんのにベースポインタってなんぞや
  - 関数ごとにスタックを切り替えるためにある
  - 図解

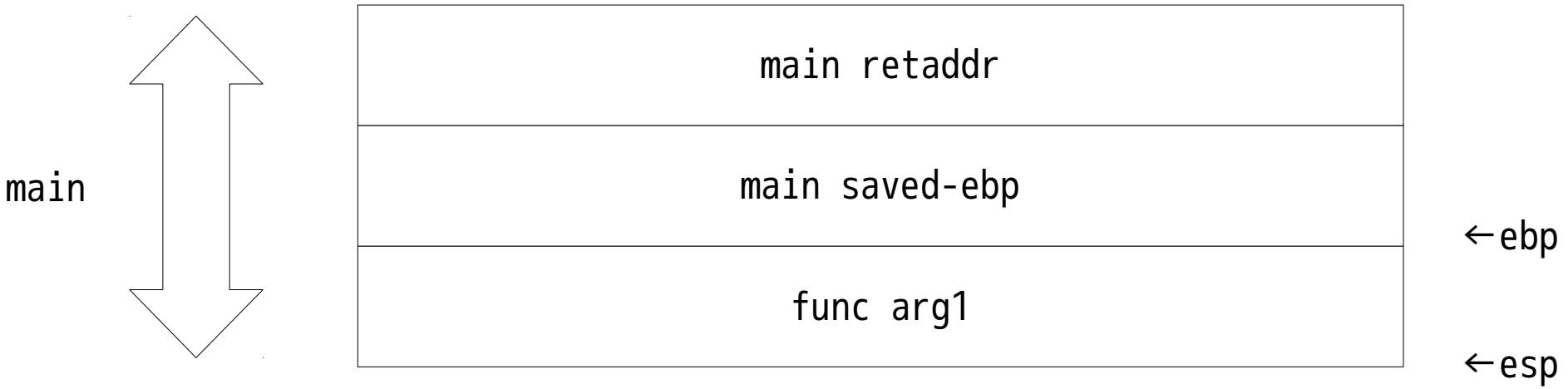
プログラム解析



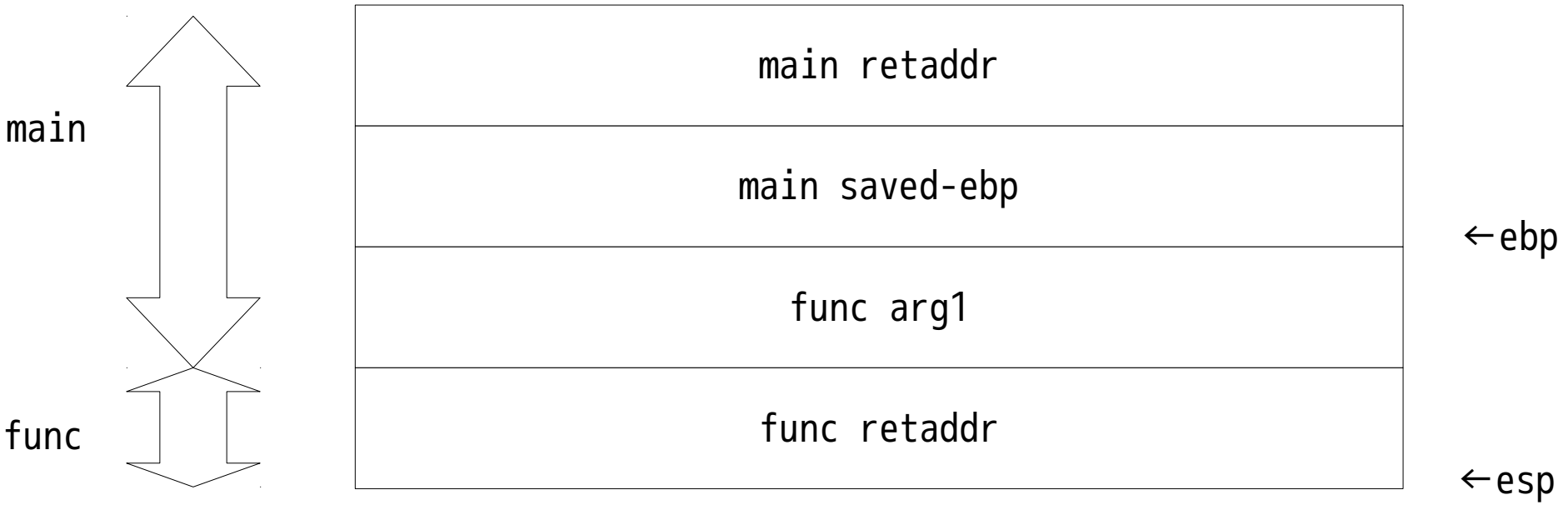
プログラム解析



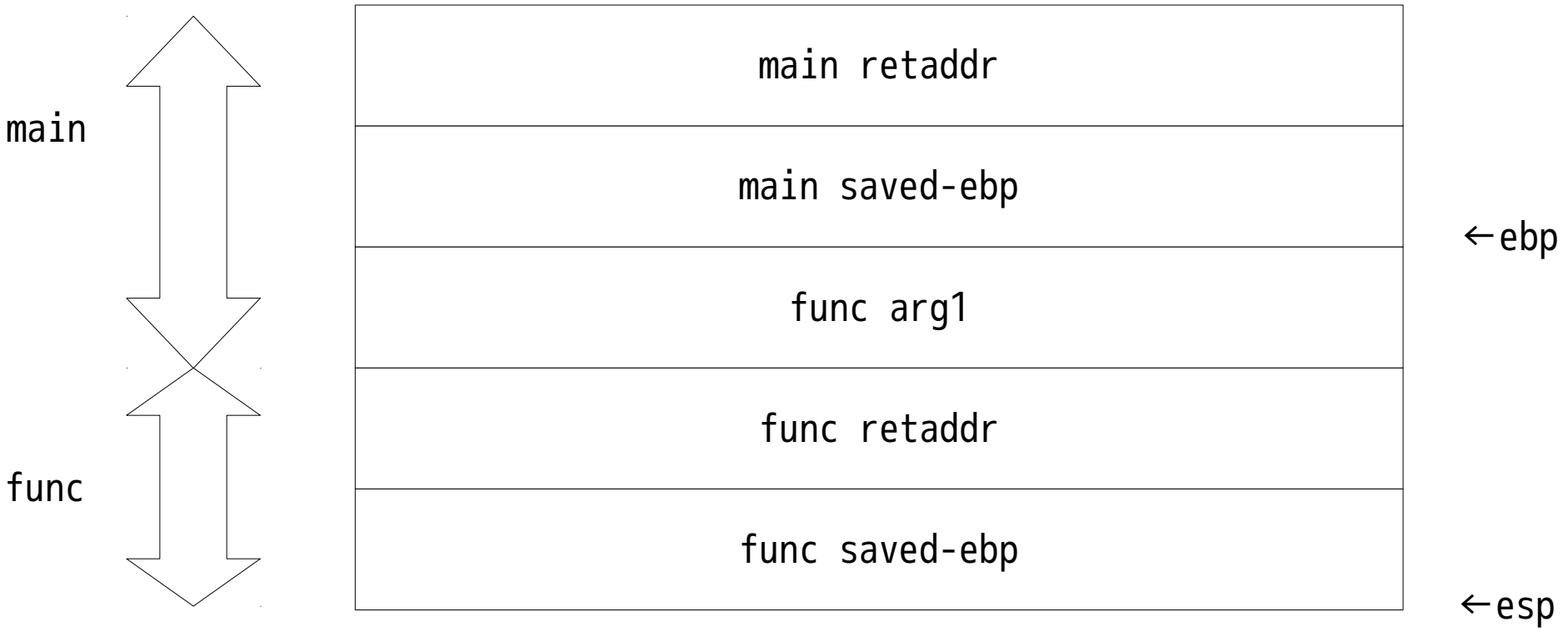
プログラム解析



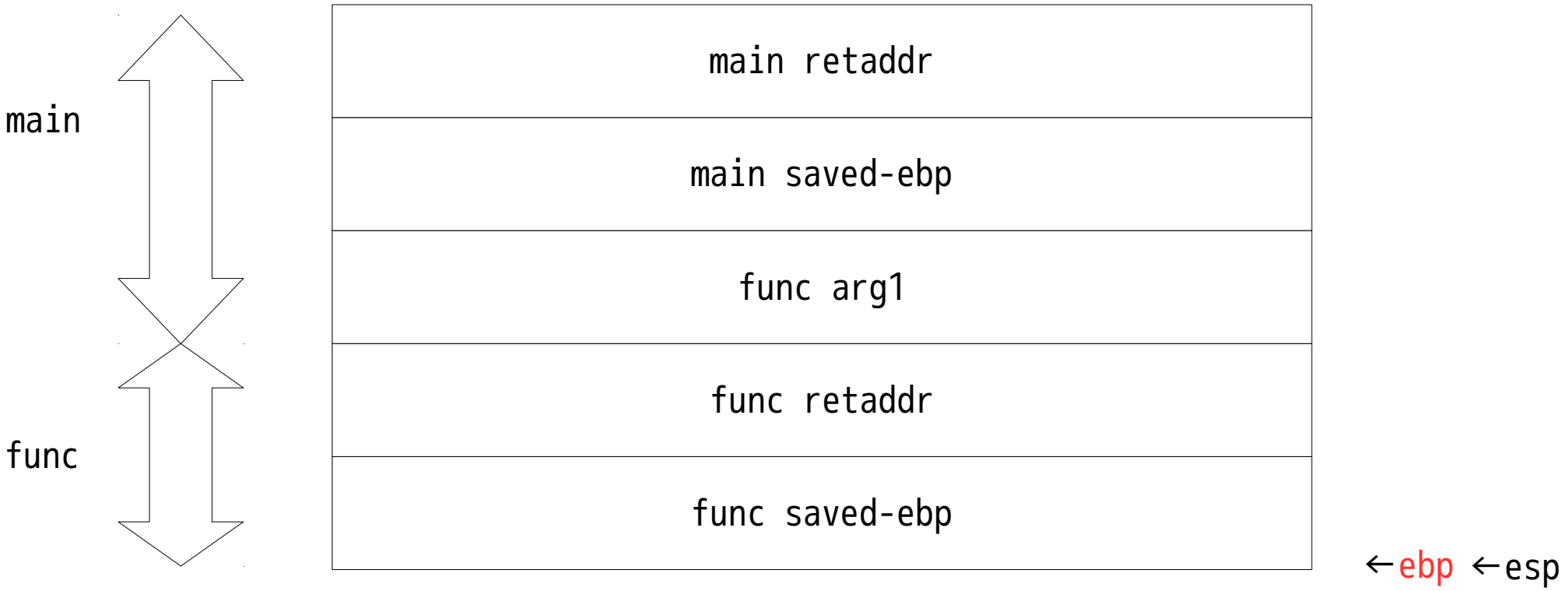
プログラム解析


main
func
main retaddr
main saved-ebp
func arg1
func retaddr
←ebp
←esp

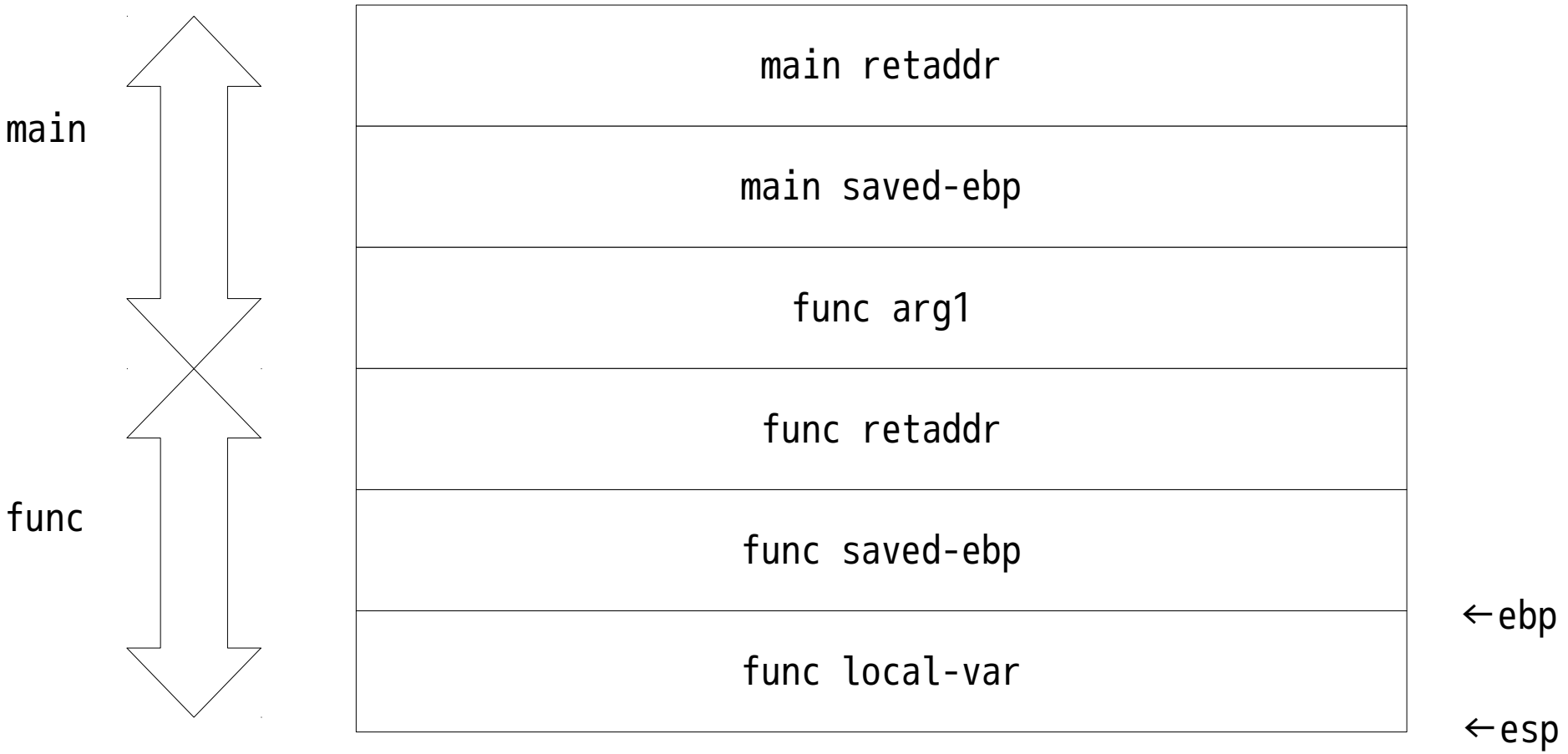
プログラム解析


main
func
main retaddr
main saved-ebp
func arg1
func retaddr
func saved-ebp
←ebp
←esp

プログラム解析


main
func
main retaddr
main saved-ebp
func arg1
func retaddr
func saved-ebp
←ebp ←esp

プログラム解析


main
func
main retaddr
main saved-ebp
func arg1
func retaddr
func saved-ebp
func local-var
←ebp
←esp

プログラム解析

- 何が嬉しいのか
  - ebpは相対的な基準
  - ebpを保存しておくことでespの位置を復元できる
  - 処理の簡略化

プログラム解析

- x86では関数の呼び出し時の引数はスタックに逆順に積む
- callの前に何か積んでたらまず間違いなく関数呼び出し
- write(1, 0xhogehoge, 13);

プログラム解析

- このアドレスなんぞ?

  - write関数は書き込み先のファイルディスクリプタ，書き込むバイト列のアドレス，書き込むバイト数の3つの引数を取る

- ファイルディスクリプタ?

プログラム解析

- ファイル操作の類は基本的にユーザープログラムは出来ない
  - I/O処理はカーネルだけの特権だから
- そのような処理をしたい時はカーネル側に代理で処理を依頼する必要がある
  - システムコールとかAPIとか言われる

プログラム解析

- 特にファイル操作の場合、まずはファイルに対して操作をするためのハンドルを取得する必要がある
  - openシステムコール
- この時取得するハンドル=識別番号がファイルディスクリプタ

プログラム解析

- なんも開いてないけど1番固定なのはなぜ?
  - デフォルトでは0, 1, 2が開いています
  - 0 = stdin
  - 1 = stdout
  - 2 = stderr

プログラム解析

- なんも開いてないけど1番固定なのはなぜ?
  - デフォルトでは0, 1, 2が開いています
  - 0 = stdin
  - 1 = stdout
  - 2 = stderr

プログラム解析

- つまり、write関数(実はシステムコールのラッパ関数)は、指定されたファイルディスクリプタへバイト列を指定された長さ書き込む

  → 今回は1に対して13バイト

  → 文字列を出力すると分かる

プログラム解析

- どのバイト列かを特定したい
- !!!実行時のアドレスなのでそのままファイルのオフセットにはならない!!!

プログラム解析

- アドレス特定には`readelf`が活躍する
- `readelf -S`した内容をもう一度読む
- 実行時との関係が書いてある!

プログラム解析

- アドレスを照らしあわせてバイナリエディタで読む...
- それっぽいのあるじゃん!

  → `write(1, "Hello World!\n", 13);`
- このプログラムは`Hello World`でした

プログラム解析

- 解析は大体こういうふうにやります
- 実際のCTFではライブラリがやってくれることも多いですが、自分でも出来る様になっておくことは大事

プログラム解析

- 余談: objdump, readelfはbinutilsというツール群の中の一つ
- binutilsには他にも便利なツールがたくさんある
- 少し紹介します

プログラム解析

- nm
  - シンボルを表示するプログラム
- シンボルとは:
  - 関数や変数の名前情報
  - 同時にアドレスも出せる

プログラム解析

- strings
  - ファイルの中に含まれる文字列を一覧表示
  - アドレスも出せるので便利
- これで見るだけでそのプログラムに付いてる防御機構も分かったりする

プログラム解析

- c++filt
  - c++の場合、クラスだのオーバーロードだの面倒な機構があるので単純にシンボルをその名前に出来ない
  - マングリングといいます
  - それを解除してくれるツール
- DEMO

プログラム解析

- binutils外のツールだと次のようなツールがあります
- xxd
  - 16進ダンプツール
  - 範囲指定等も可能なのでぱぱっと見たい時、パイプで繋いで出力結果確認等に便利

プログラム解析

- nasm, ndisasm
  - nasmというアセンブラと付属するディスアセンブラ
  - ぱぱっとコマンドでアセンブルしたりディスアセンブルする時に便利
  - シェルコード書くときなんかにも使う

プログラム解析

- asm_cmd, disasm_cmd
  - インタラクティブなアセンブラ/ディスアセンブラ
  - 自作ツール
  - ただの宣伝

プログラム解析

- ecalc
  - これも宣伝
  - コマンドラインから0x20+50とかを計算できる
  - 軽い上に任意精度

プログラム解析

- perl, python, ruby, …
  - どれか一つでいい(できれば複数)
  - Pwnに使うexploitぐらいは書けるようになっておきたい
    - exploitを書くのが本質ではないので、正直なところこれは大前提

プログラム解析

- perl, python, ruby, …
  - なんだかんだでCTFでは文字列処理がよく出てきます
    - この前のAttack&Defenseではcsvなテーブルをピボットしてちょっと弄る程度のコードを書きました
  - ワンライナーだったり、コードゴルフだったりはこういう時に役に立つ

プログラム解析

- ツールはどれも使い倒しましょう
- いくらでも応用できます
- 相互に活用出来るようにシェルの扱い方も同時に覚えるとなお良い

休憩/質問タイム

おしながき

- 基本的なアセンブラの読み方
- プログラムの解析演習
- 攻撃手法解説
- 攻撃演習(CTF)

攻撃技術解説

- お待ちかねですかね?
- ここからは実際に攻撃をする理論、防御手法をやります。
- 今日は3種類やります。

攻撃技術解説

- 攻撃手法 3種類
- Buffer Overflow
- Return-into-libc
- Format String Bug
  - GOT Overwrite

攻撃技術解説

- Buffer Overflow
- Return-into-libc
- Format String Bug
  - GOT Overwrite

Buffer Overflow

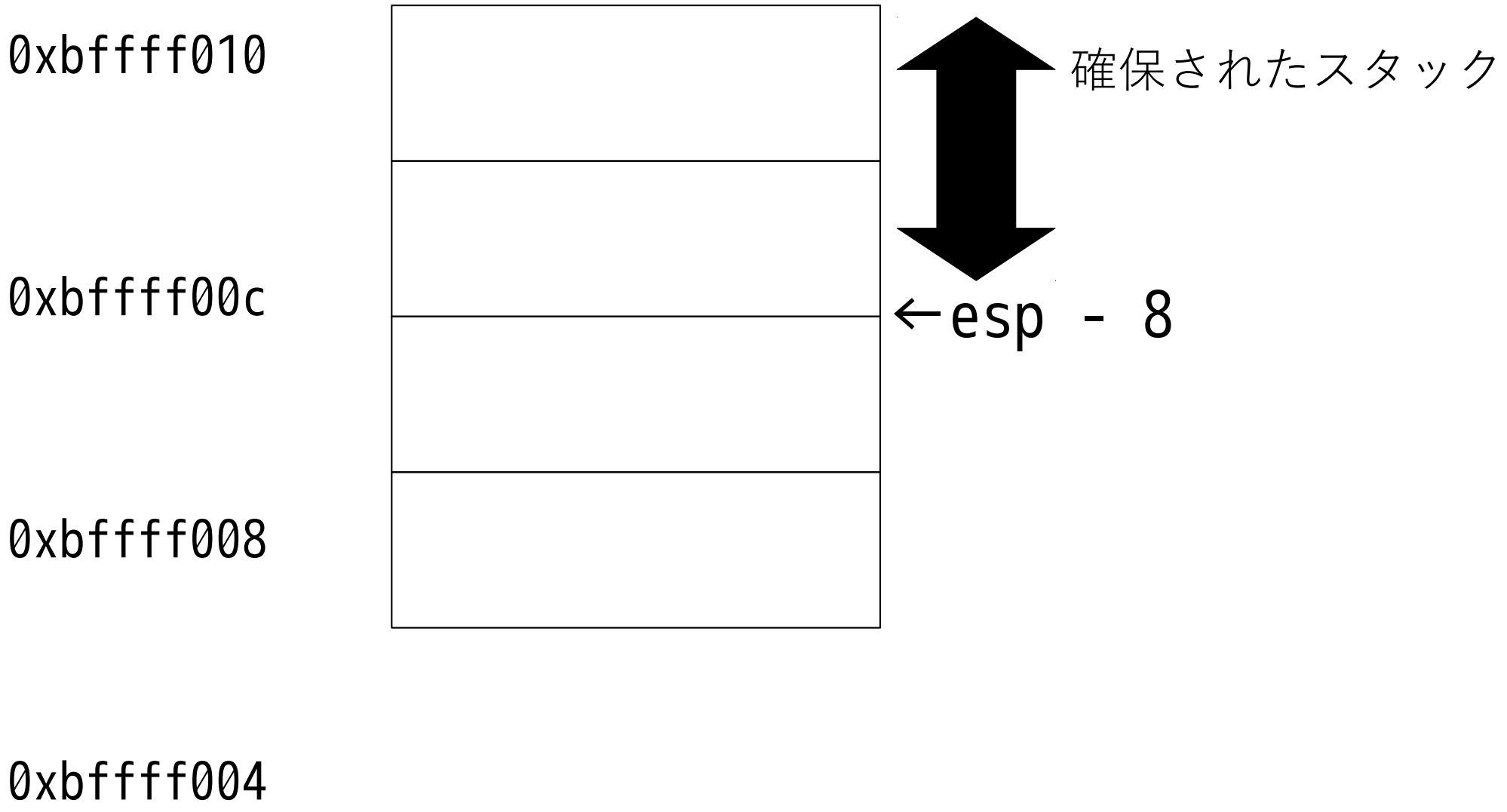
- 皆さん何かを入力するプログラムを書いたことはありますか?
  - あるよね?
- その時に何に文字列を入れましたか?
  - 殆どがchar型配列のようないわゆる"バッファ"だと思います

Buffer Overflow

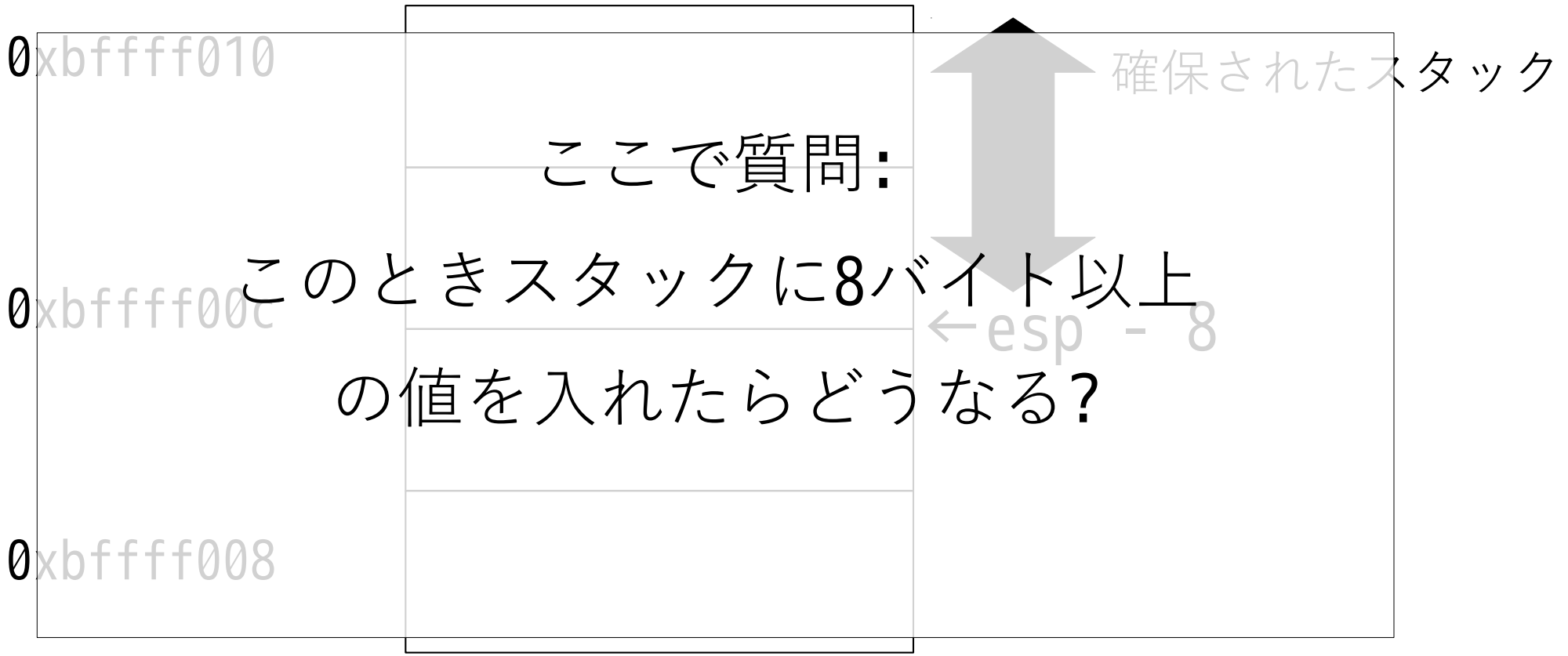
- C言語では関数内ローカル変数は主にスタック上に取られます
- デモプログラムを見てみましょう
- espが引き算されている = スタックの領域を確保している

Buffer Overflow

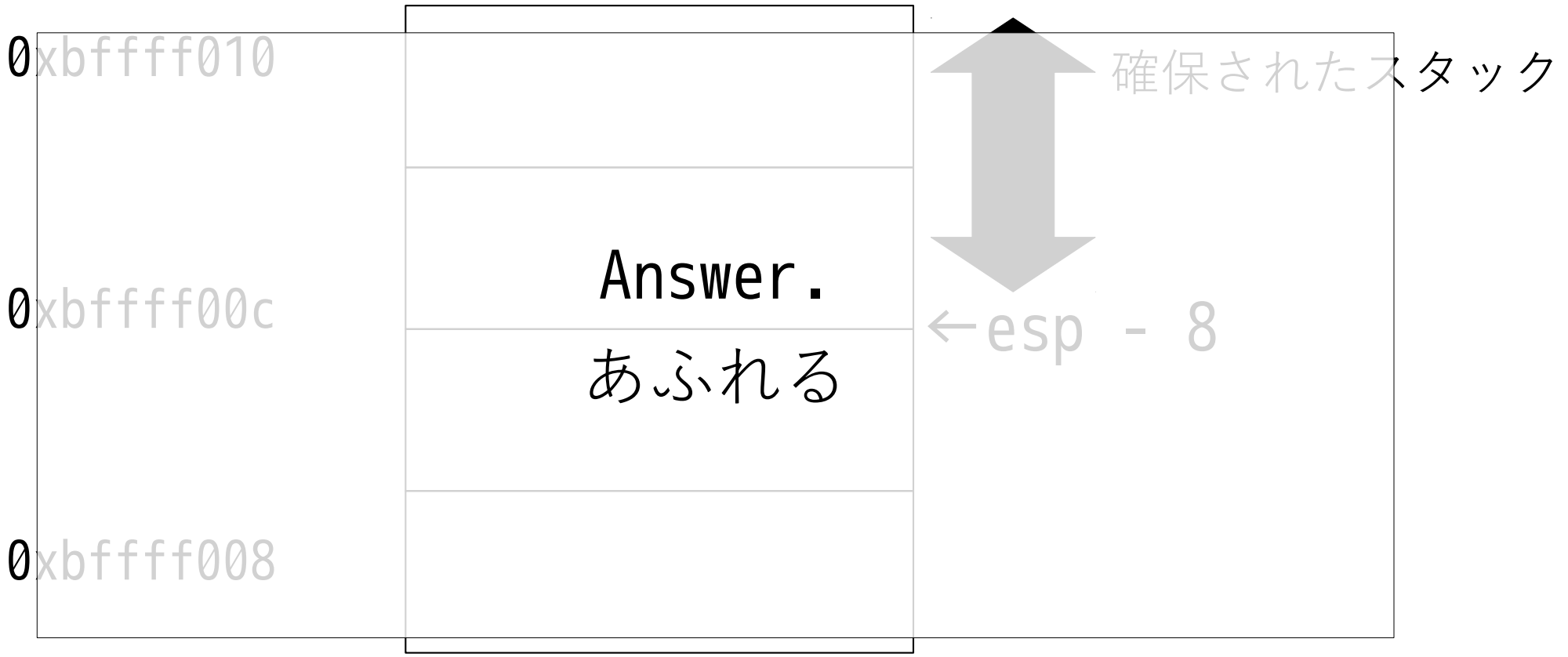
Buffer Overflow



Buffer Overflow



Buffer Overflow



Buffer Overflow

0xbffff004

Buffer Overflow

- 何が問題か
- callの仕組みを説明する必要がある

Buffer Overflow

- callは関数呼び出しの命令
  - jmpとの違いはなんなのか
- 決定的な違いは、「元に戻ってくるところ」にある

Buffer Overflow

- jmpだったら、絶対に関数の最後にはjmp [元の関数の次の命令のアドレス]があるはず
- あるか?無いなぁ。
- あるのはret命令

Buffer Overflow

- jmpだったら、絶対に関数の最後にはjmp [元の関数の次の命令のアドレス]があるはず
- あるか?無いなぁ。
- あるのはret命令

Buffer Overflow

- jmpだったら、絶対に関数の最後にはjmp [元の関数の次の命令のアドレス]があるはず
- あるか?無いなぁ。
- あるのはret命令 ←重要めっちゃ重要

Buffer Overflow

- ret命令は「スタックからEIPにアドレスをロードする命令」

- 復習: EIPは今実行してる命令のアドレス

Buffer Overflow

- call命令がjmpと違う点
- 「スタックに次の命令のアドレスを積んでからjmpする」

- もう分かりましたね?

Buffer Overflow

- 関数を呼び出す際には、スタックに戻る先のアドレスを積んで、それから`jmp`。
- 呼び出し先では処理をした後に`ret`で戻り先へ戻る

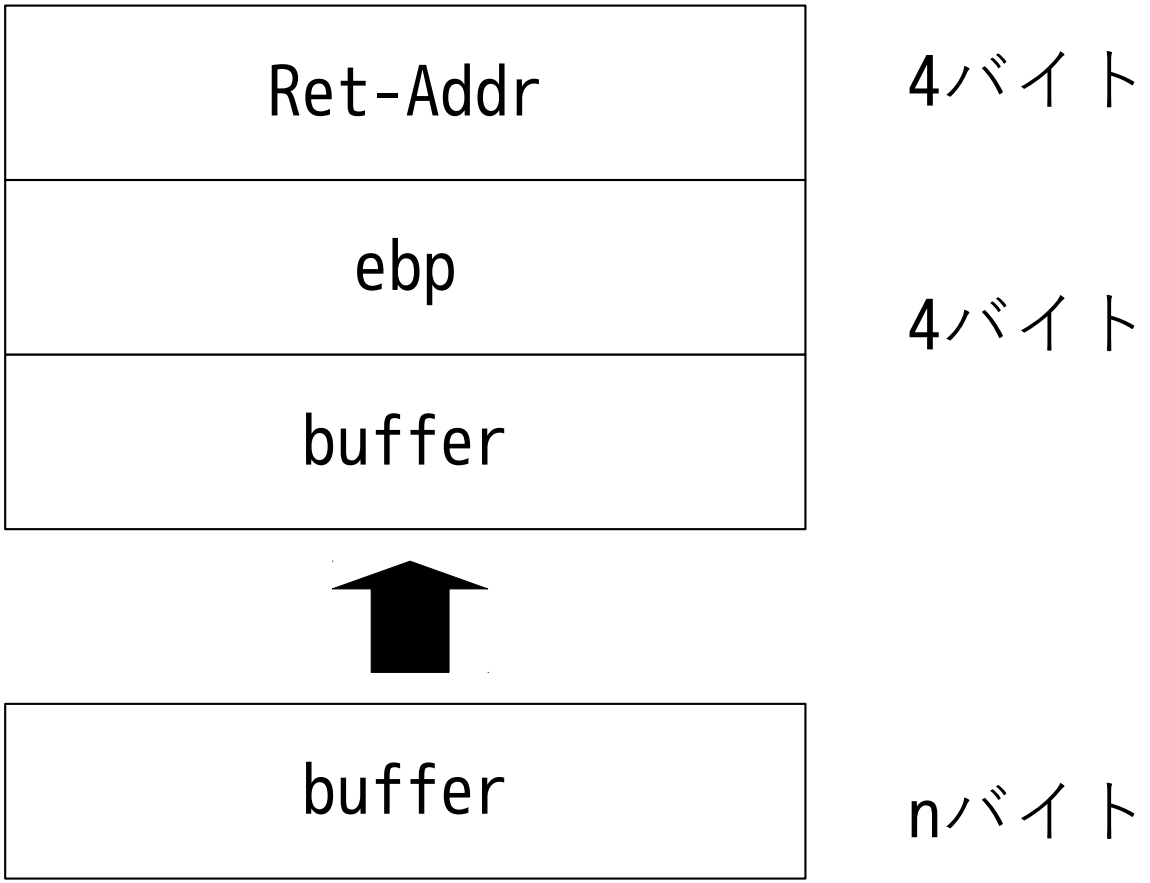
→戻る先のアドレスはスタックの上位アドレスにある!!

Buffer Overflow

- 整理: スタックには何が積まれているのか
  - 戻り先のアドレス
  - ベースポインター
  - バッファ

Buffer Overflow

- 図にするとこういう感じ



Buffer Overflow

- つまり
  - nバイト+4バイト書き込むと?

  → バッファが溢れる
- あふれた先には?
  - リターンアドレス

Buffer Overflow

- リターンアドレスを書き換えられると何が嬉しいのか
  - バッファのアドレスが分かっているならそれを書き込んでみる
  - すると、バッファの先頭にアドレスが移動する

Buffer Overflow

- 重要: バッファは自分でコントロール出来る
- つまり、バッファには任意の機械語を置ける

  → その機械語を実行できる!

Buffer Overflow

- これがBuffer Overflowという攻撃です
- バッファを溢れさせて、戻り先をバッファの先頭アドレスにする
- この時、バッファの内容を機械語にしておく事で好きな内容を実行できる

Buffer Overflow

- デモ
- 用語: この時に実行する機械語列のことをシェルコードと言います
  - 大抵/bin/sh=シェルを起動するためのコードだから

Buffer Overflow

- いつでも出来るわけではない
- ローカル変数にバッファが用意されていて、範囲を超えたユーザー入力が可能であれば可能。
  - バッファのアドレスが分かるor推測できることも
- 作り込みやすいことに変わりは無いので非常に危険かつ多い脆弱性

Buffer Overflow

- 対策技術も存在する
- ASLR
- NX
- SSP

Buffer Overflow

- ASLR
  - バッファのアドレス分かってるならアドレス自体をランダムにしてしまえばいんじゃね?
  - 有効
- ただし、32bitの場合、動く範囲が狭いので総当りされる可能性がある

Buffer Overflow

- ASLR
  - それに、普通は.text領域はランダムにならない
  - PIEという機能を使えば可能だけど、今回は触れません

Buffer Overflow

- NX
  - No eXecuteの略
  - スタックなんて実行する事まず攻撃以外で無いし、実行不可にしてしまえばいんじゃね?的防御
- 別の手法で攻撃できてしまう

Buffer Overflow

- SSP
  - Stack Smashing Protectorの略
  - ebp, eipの他に起動毎にランダムな値をはさみ、関数から抜ける前にチェックして違ったらその場でabort
- 非常に強力

Buffer Overflow

- however, SSPも万能ではない
  - 特殊な条件下であれば攻撃が出来てしまう
- もし興味があるなら調べてみてください。

Buffer Overflow

- ここからは自分の手で出来る対策です
- コードを書く際に参考にしてください。

Buffer Overflow

- まずバッファを超えたユーザー入力をするのが悪い

  - gets, fgetsのパラメータ設定ミス, sprintfで予想外の長さの文字列コピー, …

- getsはまず使わない ←そもそもC11で削除された
- sprintfは意外と気をつけてない人が多い

Buffer Overflow

- fgetsのパラメータはできれば定数で指定するのが良い
- sprintf, strcpy等の代わりにsnprintf, strncpy等の安全な関数を使う
- scanfはよく取りざたされるが、フォーマット文字列を適切に指定すれば非常に安全かつ便利

Buffer Overflow

- C11では規格レベルでそれなりに対策されています
  - getsの削除
  - フォーマット文字列への変更
  - strcpy_s関数等のセキュアな関数の追加
- いずれにしても新しいものはどんどん取り入れましょう

Buffer Overflow

- 基本原理
- 入力(ユーザー)を信用するな
- 普通の開発だけでなく、Webなどでも重要な原則
  - 何が来るか分からないならとりあえずでも最大限のチェックと対策を施すべき

Buffer Overflow

- 後で皆さんも実践しましょう!
- サーバーを用意してあるので、CTF形式で行いたいと思います

休憩/質問

攻撃技術解説

- Buffer Overflow
- Return-into-libc
- Format String Bug
  - GOT Overwrite

Return-into-libc

- Buffer Overflowの亜種
- 更に面倒かつlibcが分かっているor手元に存在する場合に使う

Return-into-libc

- libcとは?
  - Cで使う関数群(printf, fgets, system, ...)の本体があるライブラリファイル
  - バージョンごとにアドレスも変わるので決め打ち出来ない

Return-into-libc

- libcは必ずメモリ上に存在する
  - Cの関数を使っている限りは
- つまり、任意の関数のアドレスが分かる

Return-into-libc

- 復習: Buffer Overflowでは任意のアドレスを実行させられる
- 復習: x86の場合、引数はスタックに積む
- 復習: NX(No eXecutable)はスタックを実行不可能にする
- 事実: libcには任意の関数がある

  → Q.ここから導き出される攻撃は?

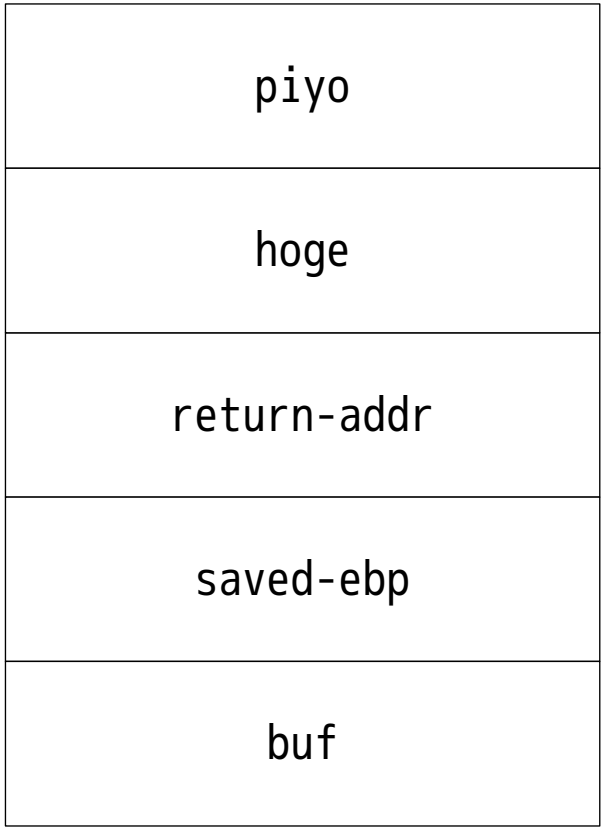
Return-into-libc

- A. libcの任意の関数に、引数を積んだようにスタックを作って飛ばす
  - system関数とか良いと思いませんか?
- しかも、libcには/bin/shという文字列が含まれている!

Return-into-libc

- スタックを作る?
  - スタック上のデータを弄る事が出来るから、スタック上に適切な配置でデータを置いてやることで関数呼び出しの時の引数配置を作る

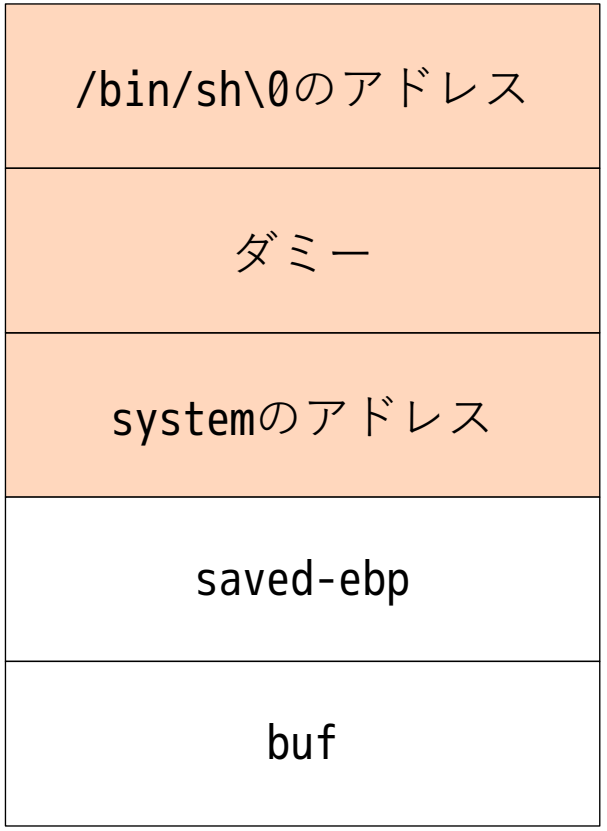
Return-into-libc



←return-addrの上にはなにかある

- 大抵は引数

Return-into-libc



←こういう感じに書き換える

= systemを呼び出すように仕向ける

= 好きな処理ができる!

Return-into-libc

- libc内の関数にReturnする=飛ぶため、Return-into-libcといいます
- libcが一緒に配布されている問題だったら大体はこれを使う

Return-into-libc

- デモ
  - もちろん後で実際にやります!
- ツールの使い方にも慣れましょう

Return-into-libc

- 攻撃時の課題
  - libcがないと使えない!
  - ASLRがある時はロードされるアドレスがバラバラ
  - Buffer Overflowと同等の脆弱性が必要

Return-into-libc

- それぞれにある程度の回避策はあります
  - 応用問題として演習で解説します。

休憩/質問

攻撃技術解説

- Buffer Overflow
- Return-into-libc
- Format String Bug
  - GOT Overwrite

Format String Bug

- 皆さん、printfは使ったことありますか?
  - 便利ですよね
- %xや%sを使うとフォーマット指定まで出来る

Format String Bug

- 皆さん、printfは使ったことありますか?
  - 便利ですよね
- %xや%sを使うとフォーマット指定まで出来る

Format String Bug

- 実は、printfファミリの関数で使えるフォーマットの中に%nというものが存在する
  - 何をするのか
  - 引数で指定されたアドレスにいままで出力した文字数を書き込む

Format String Bug

- 普通はなんの意味もない
- but こんなコードを書いてしまうと....?

```
char buf[256];
fgets(buf, 255, stdin);
printf(buf); // ;(
```

Format String Bug

- 普通はなんの意味もない
- but こんなコードを書いてしまうと....?

```
char buf[256];
fgets(buf, 255, stdin);
printf(buf); // ;(
```

Format String Bug

- 普通はどうしますか?
  - `printf("%s", buf);`が妥当でしょう
- 先ほどの例は何が危険か。
  - 入力にフォーマット文字列が与えられてしまう!

Format String Bug

- フォーマット文字列だけで引数指定してないならいんじゃね?
  - ダメ
  - C言語の闇

Format String Bug

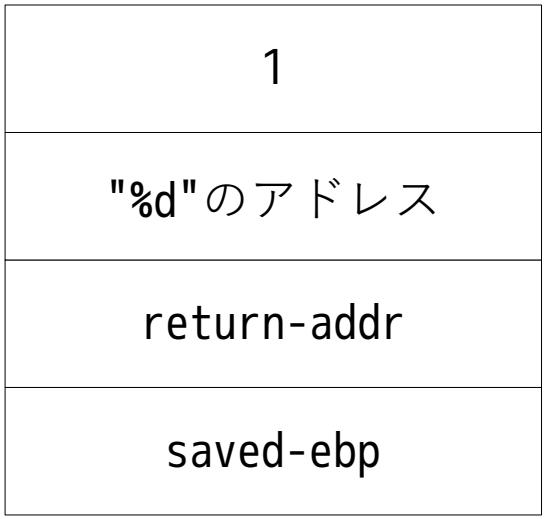
- C言語の可変長引数関数は少々特殊な扱い
- printfで例をあげて説明します

Format String Bug

- printf("%d", 1);
- printfはまずフォーマット文字列自体のアドレスを取得
- フォーマット文字列の中に%d = int型の整数があると分かった。

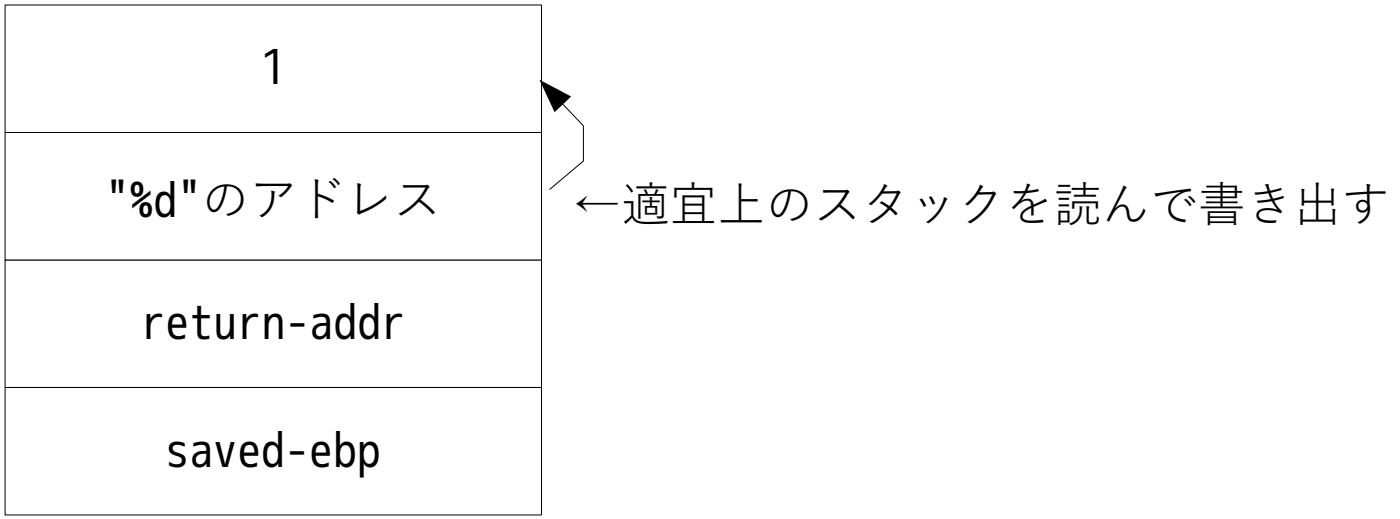
  →フォーマット文字列のアドレス + 4を読み出してintとして表示する!

Format String Bug



←適宜上のスタックを読んで書き出す

Format String Bug



Format String Bug

- %08x"だけ"を入れたらどうなるだろう?
  - スタックには適当なデータ
  - それでもprintfは読みだす
  -> 意図しない情報が盗まれる恐れ

Format String Bug

- バッファがスタック上にあったとしましょう
  - printfを呼び出す時,バッファはprintfよりスタックの上位にある
  - n番目の値を読み出せる識別子がありますね?

Format String Bug

- ここまでをデモ
  - 整理してみてください

Format String Bug

- 最初に戻って...
- %nの話でした

Format String Bug

- %nは引数で与えられたポインタへデータを書き込む

  →バッファがスタック上にあるならやり方次第で任意のアドレスを書き換えられるのでは?

Format String Bug

- これだけじゃない
- %sは何をするのか

  - 与えられたポインタのデータを'\0'が出るまで読んで書き出す

  → '\0'までの好きな位置のデータを読み出せる!

Format String Bug

- printf("%s", buf)とprintf(buf)の違いだけでこれだけ違いがあるんです
- Format String Bugは一般的には%nや%08xがブチ込めたらそう呼ばれます

Format String Bug

- 対策方法:
- printf/sprintf等のprintfファミリ関数では絶対に文字列表示にprintf("%s", buf)を使う
- C11以降を使う
  - 普通こんな機能使わないでしょ、ということでC11以降では%nは廃止されました

Format String Bug

- 攻撃をする上で私達が考える事は1つ

- いかにしてEIPをコントロールするか

Format String Bug

- .text領域は基本的に書き換え不可
- PIEじゃないASLR環境ならライブラリのアドレスやスタックも変動している
- じゃあどこを書き換えるのか。

攻撃技術解説

- Buffer Overflow
- Return-into-libc
- Format String Bug
  - GOT Overwrite

攻撃技術解説

- Buffer Overflow
- Return-into-libc
- Format String Bug
  - GOT Overwrite

GOT Overwrite

- ライブラリとは
  - ざっくり言えばよく使う関数群をまとめたもの
- 動的ライブラリと静的ライブラリがある
  - 動的ライブラリは実行時、静的ライブラリはコンパイル時にアドレスを調整する

GOT Overwrite

- 殆どは動的ライブラリを使う
  - libc.so.6もその一つ
- メリットとして、非常にバイナリ自体が小さくなることがある
  - 静的ライブラリを使うと、バイナリ自体に全ての関数を含めてしまうため。

GOT Overwrite

- 動的ライブラリは実行時にアドレスを調整する
- 誰が?いつ?どういうふうに?

GOT Overwrite

- 誰が?
  - ローダ
- ローダって?
  - 実行ファイルを設定されている通りに実際にメモリに配置し、スタック領域やライブラリのあれこれをまとめて請け負ってくれるソフトウェア

GOT Overwrite

- /lib/ld-linux.so.2が該当する
- まあ普段は気にしなくてもいい

GOT Overwrite

- じゃあどうやってアドレスをどうこうしてるのさ
  - 特定の場所に決まったメモリ領域を確保しておく
  - この場所は関数が必要になった時にローダが自動でその関数のアドレスを書き込んでくれる
  - 遅延リンクという仕組み
  - この時に関数のアドレスが書き込まれる場所をGlobal Offset Tableといいます
    - 略してGOT

GOT Overwrite

- じゃあどうやってアドレスをどうこうしてるのさ
  - 特定の場所に決まったメモリ領域を確保しておく
  - この場所は関数が必要になった時にローダが自動でその関数のアドレスを書き込んでくれる
  - 遅延リンクという仕組み
  - この時に関数のアドレスが書き込まれる場所をGlobal Offset Tableといいます
    - 略してGOT

GOT Overwrite

- 書き換えが可能?
- 特定のアドレス?
- しかも関数のアドレスが載ってるテーブル?

GOT Overwrite

- → このテーブルの値を一つでも書き換えてしまえばいいんじゃね?!
  - 正解
  - GOT Overwriteと言います

GOT Overwrite

- Format String Bugはよく使われる
  - 他にはUnlinkやFastbinsがよく出る
  - 発展的内容なので気になった方はお調べ下さい

GOT Overwrite

- 対策
  - そんな書き換えられるんだったら遅延させずに起動時に一括して書き換えて後読み書き不可能にしてしまおう
- RELRO(RELocation Read-Only)という防御機構

GOT Overwrite

- RELROへの反撃

  - Format String Bugがあるのならそのアドレスの内容を読んだり%08x連打してebpの場所探り当ててスタックのアドレスを計算してしまえばret-into-libcに持ち込める

GOT Overwrite

- 正直いたちごっこ
- まあこの辺りの流れは非常におもしろいのでぜひ調べてみてください。

GOT Overwrite

- デモ
  - GOT Overwriteはスタック書き換えをしない分、簡単に済むことが多い
  - 文字列を作るのが面倒なだけ

GOT Overwrite

- 余談です
- 実はテーブルの一部に関数を探してアドレスを書き込む関数のアドレスがあったりします

  -> それに飛ばすことで任意の関数のアドレスを取得&飛ぶことが可能

  - return-into-dl-resolveと言います
  - 超発展的なのでさらっと触れるだけで終わりです

休憩/質問

おしながき

- 基本的なアセンブラの読み方
- プログラムの解析演習
- 攻撃手法解説
- 攻撃演習

攻撃演習

- 楽しいですヨ
- CTF形式で行います
- 問題の中には過去に開催されたCTFの問題もあったりします

攻撃演習

- Q. いきなりやらせるの?
- A. そんなわけないです

攻撃演習

- 最初に各攻撃の具体的なアウトラインとシェルコードについての解説をします。
- 実際の時間は1時間ほどを予定しています。
  - その時間中も質問/ディスカッションして構いません

攻撃演習

- シェルコードの書き方
  - 正直な所、アセンブラでプログラミングをするだけです
  - というのも不親切なので、ちゃんと解説します

シェルコード

- シェルコードとはなにか?

  - 多くの場合、シェルを起動するための機械語列

  - たまに`cat flag`だけとか、何かしらの処理をするだけなんてのもある

シェルコード

- やりたいこと
  - /bin/shの起動
- 今回はシステムコールのexecveを使います

シェルコード

- Linuxではシステムコールの呼び出しをint 0x80という命令で行う
  - システムコールの番号はeaxにいれる
  - 引数はebx, ecx, edxにいれる

シェルコード

- execveの解説

  - Linuxのシステムコールで、番号は11番

  - execve(file, args, envp);のように使う

  - つまり、ebx = "/bin/sh\0", ecx = [NULL], edx = [NULL];

シェルコード

- 今回やりたいこと
- `execve("/bin/sh\0", NULL, NULL)`
- アセンブラにすると?

シェルコード

- eaxはexecveの番号なので11
- ebxは"/bin/sh\0"のアドレス
- ecx, edxはNULL = 0

シェルコード

- eaxはexecveの番号なので11
- ebxは"/bin/sh\0"のアドレス
- ecx, edxはNULL = 0

シェルコード

- どうやって作ろう
- 簡単なのはスタックを利用する方法

```
push "/sh\0"
push "/bin"
mov ebx, esp
```

シェルコード

- どうやって作ろう
- 簡単なのはスタックを利用する方法

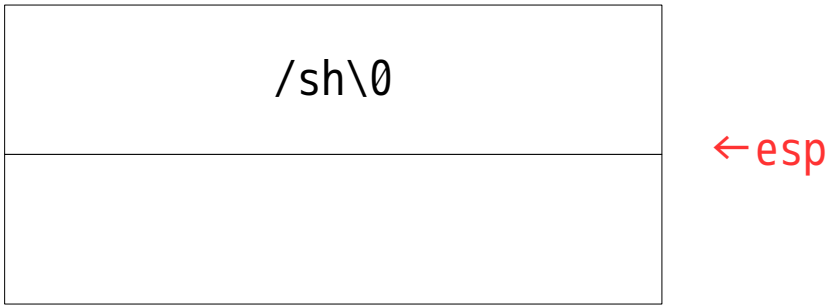
リトルエンディアン記述になるのに注意!

```
push 0x0068732f
push 0x6e69622f
mov ebx, esp
```

シェルコード

- どうやって作ろう
- 簡単なのはスタックを利用する方法

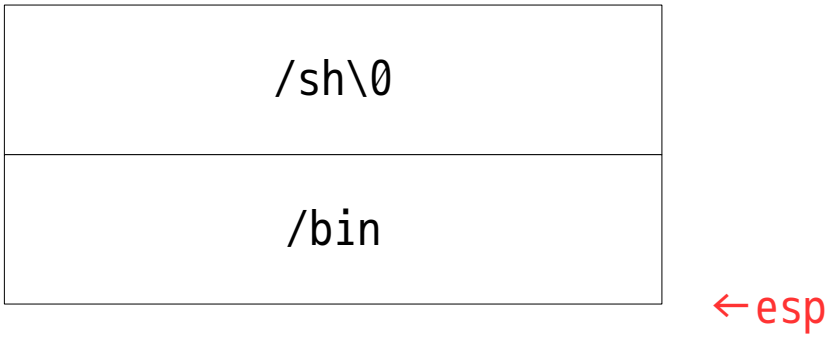
```
push 0x0068732f
push 0x6e69622f
mov ebx, esp
```



シェルコード

- どうやって作ろう
- 簡単なのはスタックを利用する方法

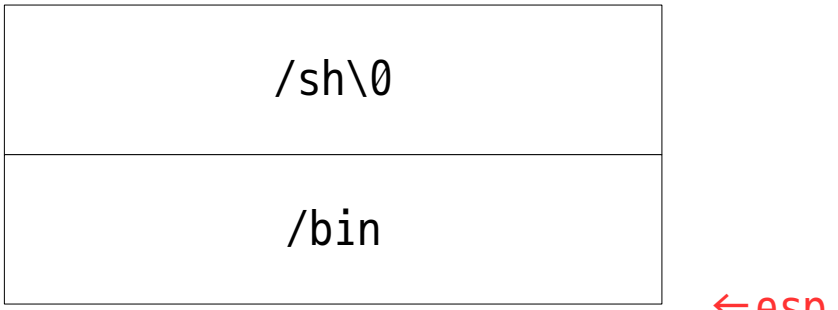
```
push 0x0068732f
push 0x6e69622f
mov ebx, esp
```



シェルコード

- どうやって作ろう
- 簡単なのはスタックを利用する方法

```
push 0x0068732f
push 0x6e69622f
mov ebx, esp
```



ebx = esp

シェルコード

- 今回はコレで行きます
- たまに'\0'が使えない、空白文字が使えない、制御文字が使えない、アルファベットしか使えない、etc...のような制限環境の問題も出る
- '\0'については今回少し触れます
- 他は個人課題にしておきます
  - 次回のCTFで出るかも?

シェルコード

- 実際にアセンブラコードに落としこんでみる

```
mov eax, 11
push 0x0068732f
push 0x6e69622f
mov ebx, esp
mov ecx, 0
mov edx, 0
int 0x80
```

シェルコード

- 実際にアセンブラコードに落としこんでみる

```
mov eax, 11
push 0x0068732f
push 0x6e69622f
mov ebx, esp
mov ecx, 0
mov edx, 0
int 0x80
```

← eax = 11

←

← さっきやりましたね?

←

← ecx = NULL つまり0

← edx = NULL つまり0

システムコール呼び出し

シェルコード

- 実際にアセンブラコードに落としこんでみる

```
00000000  B80B000000          mov eax,0xb
00000005  682F736800          push dword 0x0068732f
0000000A  682F62696E          push dword 0x6e69622f
0000000F  89E3                mov ebx,esp
00000011  B900000000          mov ecx,0x0
00000016  BA00000000          mov edx,0x0
0000001B  CD80                int 0x80
```

シェルコード

＞ ＞ ＞長い＜ ＜ ＜

シェルコード

＞ ＞ ＞ '\0'がある＜ ＜ ＜

シェルコード

- 実際にアセンブラコードに落としこんでみる

```
00000000  B80B000000          mov eax,0xb
00000005  682F736800          push dword 0x0068732f
0000000A  682F62696E          push dword 0x6e69622f
0000000F  89E3                mov ebx,esp
00000011  B900000000          mov ecx,0x0
00000016  BA00000000          mov edx,0x0
0000001B  CD80                int 0x80
```

シェルコード

- 実際にアセンブラコードに落としこんでみる

```
00000000  B80B000000          mov eax,0xb
00000005  682F736800          push dword 0x0068732f
0000000A  682F62696E          push dword 0x6e69622f
0000000F  89E3                mov ebx,esp
00000011  B900000000          mov ecx,0x0
00000016  BA00000000          mov edx,0x0
0000001B  CD80                int 0x80
```

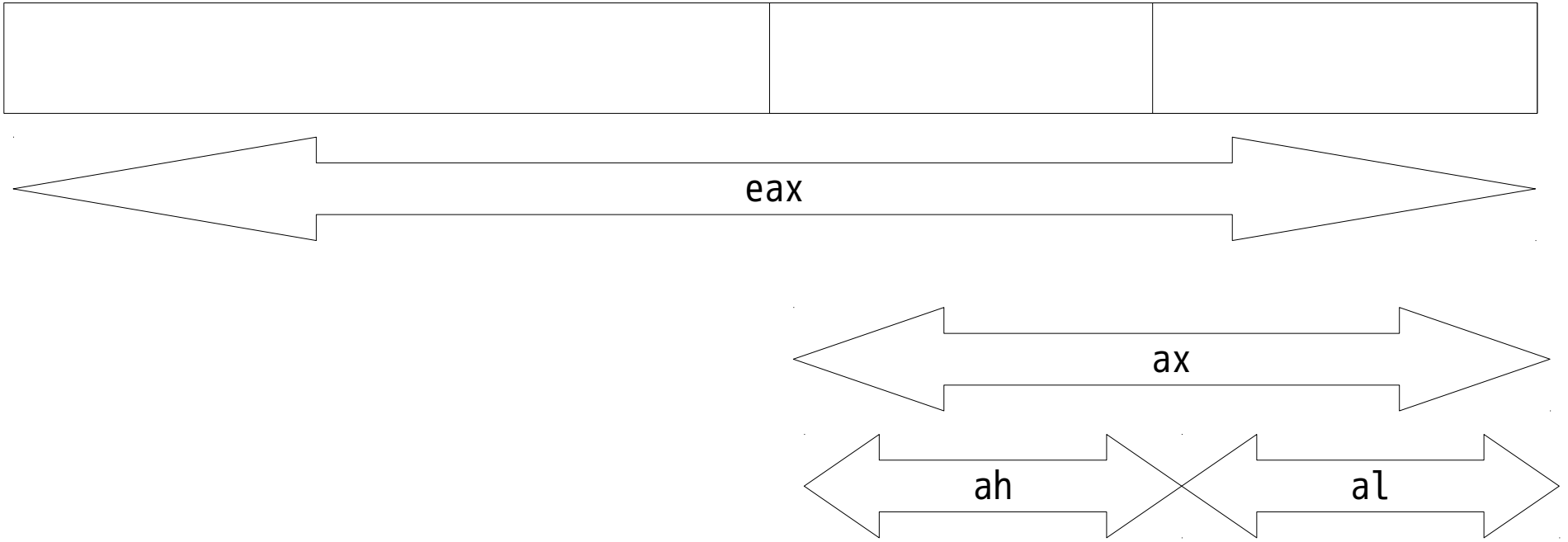
シェルコード

- この部分は32bit整数をレジスタにいれる時につきまとうヌルバイト
  - 対処を考える
  - 比較的簡単

シェルコード

- eax = 11に関しては、Linuxが最下位8bitしか見ていないことを利用する
- eaxは32bitレジスタ
  - 実は16bitだとaxになる
  - 更に、8bit単位だとal, ah(A Low, A High)

シェルコード



シェルコード

- eax, ebx, ecx, edxに関しては同じように16, 8bitレジスタが存在します
  - esp, ebp, esi, ediに関しては16bitレジスタまでしか存在しない
- ちなみに64bit環境になるとrax, rspのようにrが最初に付くようになります
  - レジスタもr15ぐらいまで増えます

シェルコード

- 今回はeaxの話をしたい
  - 最下位だけ見るならalでいいんじゃね?
    - That's Right!
  - それにmov r8, imm8は命令が短い!

シェルコード

- 実際にアセンブラコードに落としこんでみる

```
00000000  B00B                mov al,0xb
00000002  682F736800          push dword 0x0068732f
00000007  682F62696E          push dword 0x6e69622f
0000000C  89E3                mov ebx,esp
0000000E  B900000000          mov ecx,0x0
00000013  BA00000000          mov edx,0x0
00000018  CD80                int 0x80
```

シェルコード

- 他は?
  - 流石に全ビット見てるからr8法は無理
  - ゼロクリアするために幾つかのアセンブラ・イディオムが考えられています

シェルコード

- xor eax, eax
  - 皆さんXOR演算は知ってますよね?
  - 1 ^ 1 = 0, 0 ^ 0 = 0
  - これを同じレジスタ同士でやるとどうなるか
    - 0になる!
  - しかも2バイト, '\0'は出てこないため、今回のような場合に重宝する

シェルコード

- cdq
  - ?????
  - この命令はeaxに入っている値を符号拡張してeax, edxに代入する命令
    - 要するに符号を考えて代入してくれる命令

シェルコード

- eaxが0だったらどうなるか
  - 符号部分は0なので無視される
  - つまり、eax = edx = 0
  - 1バイト命令なので非常に良さ

シェルコード

- これらを組み合わせて書いてみる

```
00000000  31C0                xor eax,eax
00000002  99                  cdq
00000003  B00B                mov al,0xb
00000005  682F736800          push dword 0x68732f
0000000A  682F62696E          push dword 0x6e69622f
0000000F  89E3                mov ebx,esp
00000011  89D1                mov ecx,edx
00000013  CD80                int 0x80
```

シェルコード

- '\0'が残り一つ!
- しかも短い!
- このテクニックは覚えておきましょう
  - というかオペコード覚えておくと便利です

シェルコード

- 残り1つ

```
00000000  31C0                xor eax,eax
00000002  99                  cdq
00000003  B00B                mov al,0xb
00000005  682F736800          push dword 0x68732f
0000000A  682F62696E          push dword 0x6e69622f
0000000F  89E3                mov ebx,esp
00000011  89D1                mov ecx,edx
00000013  CD80                int 0x80
```

シェルコード

- 厄介
- 文字列中にあるのでいままでのようなテクニックは使えない
- どうするのか?
  - Linuxの仕様を逆手に取る

シェルコード

- Linuxは/bin/shと/bin//shは同一視する
  - つまりパスのディレクトリ指定が無いとスルーされる
- /bin = 4バイト, //sh = 4バイト
- つまり、残りは'\0'だけ
- push 0; push "//sh"; push "/bin"

シェルコード

- Linuxは/bin/shと/bin//shは同一視する
  - つまりパスのディレクトリ指定が無いとスルーされる
- /bin = 4バイト, //sh = 4バイト
- つまり、残りは'\0'だけ
- push 0; push "//sh"; push "/bin"

シェルコード

- push imm32はレジスタで代用できるじゃん
  - ご明察
- push eaxを使いましょう
  - よく見たらさっきeaxに'\0'入れたじゃん!
  - 利用しましょう

シェルコード

- 残り1つ

```
00000000  31C0                xor eax,eax
00000002  99                  cdq
00000003  89D1                mov ecx,edx
00000005  50                  push eax
00000006  682F2F7368          push dword 0x68732f2f
0000000B  682F62696E          push dword 0x6e69622f
00000010  89E3                mov ebx,esp
00000012  B00B                mov al,0xb
00000014  CD80                int 0x80
```

シェルコード

- '\0'を消せた!
- シェルコードはよく制限かかった問題が出ます
  - SECCON 2014 大阪大会で出た問題とか
  - あの問題おもしろいのでWriteupを探してみてください

休憩/質問

攻撃演習

- シェルコードの書き方は分かった。
- どのように問題を解けばいいの?

攻撃演習

- 問題文には大抵、xxx.xxx.xxx.xxx 12345のような形でIPアドレスとポート番号が書いてあります
- Webブラウザでアクセスしてもダメで、直接ソケット通信をする必要があります

攻撃演習

- 簡単なやり方
- telnetやnetcatを使う
- nc 192.168.0.2 9000を実行してみてください
- telnetでもいいです

攻撃演習

- なんか出ましたね?
- 直接ソケットを作って通信することが多いです
- 今回はPythonで書きます

攻撃演習

- Pythonの書き方については触れません
- 実際に書きます

攻撃演習

- socketというライブラリを使います
- structは文字列の変換に。
- その他好きなライブラリがあれば使って下さい

攻撃演習

- 取り敢えず読み込みだけするコードを試してみましょう
- バッファのアドレスが手に入りますね
- ここだけ切り出して数値にしてしまいましょう
- そして変数へ代入。

攻撃演習

- 攻撃文字列にはまずシェルコードを。
- 次にバッファのサイズ分まで適当な文字列を。
- 最後にバッファのアドレスを`struct.pack`を使って代入。
  - `struct.pack` = 数値を文字列に。
  - `struct.unpack` = 文字列を数値に。

攻撃演習

- 実行してみましょう
- シェルが取れました。
- ASLRもSSPもなしなので簡単ですね!

休憩/質問

CTF

- さて、CTFです
- CTFとはなにか?
  - Capture the Flagの略
  - 問題を解いて、入手した旗を立てるイメージ
  - 幾つか種類がある

CTF

- 種類?
  - Jeopardyスタイル
  - アメリカのクイズ番組から来てる
  - 画像を見たとおり、問題ごとにポイントが割り振られてて、問題を解くとそのポイントが手に入る
  - 多くのオンラインCTFがこの形

CTF

- Attack and Defenseスタイル
  - 各チームにサーバーが一台渡されて、そのサーバー上で動くプログラムに対して修正して脆弱性を防いだり、相手のサーバーのプログラムを攻撃したりすることでポイントを得る
  - いわゆる"攻防戦"
  - 余談:博多でつい最近(6/7)行われました
    - 優勝してきました

CTF

- King of Hillスタイル
  - 問題サーバーというサーバーがいくつかあって、そのサーバーで動いてるプログラムであったり何かしらのサービスに対しての攻撃をしたりそのまま占拠して他の攻撃を弾くことで独占することでポイントを得る
  - SECCON本戦がこの形式

攻撃演習

- 今回はJeopardyスタイルです
- 問題は5つほどあります
- ちょっと頭をひねる問題がありますが、論理的に考えれば解けます

攻撃演習

- CTFのアドレスはhttp://192.168.0.2:4000/です
- 時間は頃合いを見て。

攻撃演習

CTF

質問/ディスカッション自由

攻撃演習

- CTF終了
- 問題の答え合わせをします

攻撃演習

- 答え合わせ適当

攻撃演習

- 以上です。
- 難しいけど、楽しいでしょ?
- 次回もやるのでお楽しみに...

攻撃演習

- 幾つか他にもCTFがあるので紹介をします

攻撃演習

- ksnctf http://ksnctf.sweetduet.info/
  - 基本的な問題から応用問題まで幅広く
  - 良問揃い
  - Villager BはPwnをする上では登竜門的存在

攻撃演習

- akictf http://ctf.katsudon.org/
  - ksnctfの派生CTF
  - Web/Cryptoの問題が多い
    - Pwnの演習が終わった後はCryptoを考えています
  - 難易度も全体的に高く、継続して取り組める

攻撃演習

- EDCTF http://ctf.npca.jp/
  - 強豪チームEpsilon DeltaのCTF
  - Pwnの難しい問題が...
  - もし興味を持ったなら是非やりましょう

攻撃演習

- Pwnable.kr http://pwnable.kr/
  - Pwnをやるならここの問題はやっておけ!的な場所
  - バッファオーバーフローすらしないような問題からカーネルの脆弱性を突くような問題まで...

攻撃演習

- その他、本についても。
  - HACKING:美しき策謀
  - リバースエンジニアリングバイブル
  - アナライジング・マルウェア
  - Linkers & Loaders
- などなど。

まとめ

- まとめ。
- アセンブラの読み書きを実践した。
- プログラムの解析手法について理解し、いくつかのツールの使用方法を理解した。

まとめ

- 実際に使われる攻撃手法とそのための防御策を3つ理解した。
  - Buffer Overflow
  - ret-into-libc
  - GOT Overwrite
- CTFを通じた実践演習で各手法に対する攻撃方法について深く理解できた。

まとめ

- 再三言っていますが、実際に攻撃に転用するのはやめましょう。
  - モラル面を守って、攻撃が許可された範囲内で。
  - 今の日本だと不正アクセス禁止法なんてものもありますしね...

まとめ

- 今回、攻撃手法に主眼を置いたのはそれに対する防御であったり、攻撃されないようなプログラムを書くことの重要性を知ってもらうためです。
- セキュリティ業界はまだまだ生まれたばかりばかりです
  - 先んじてやっておけばこの先ワンチャンあるかもしれませんよ?

次回予定

- 復習
- Heap Overflow
- Use After Free
- Return-Oriented-Programming (ROP)
- 特殊なシェルコード
- 簡易CTF

次回予定

- ※あくまで予定です
- 変更が入る可能性もあります

御静聴ありがとうございました

お疲れ様でした!