

SBOF概論

Binary Exploitについて

- 実行バイナリの脆弱性を付いてメモリ空間を引っ掻き回し、想定外の動作をさせる攻撃
- メモリ管理はロジックだけを書いていると中々意識しない部分だが、任意コード実行に繋がることが多いため重要
- だいたいの場合、プログラムカウンタ(x86ではripというレジスタに入っている)を自由に設定して任意コード実行を目指す
 - returnアドレスの書き換え、フックや関数テーブルの書き換え等

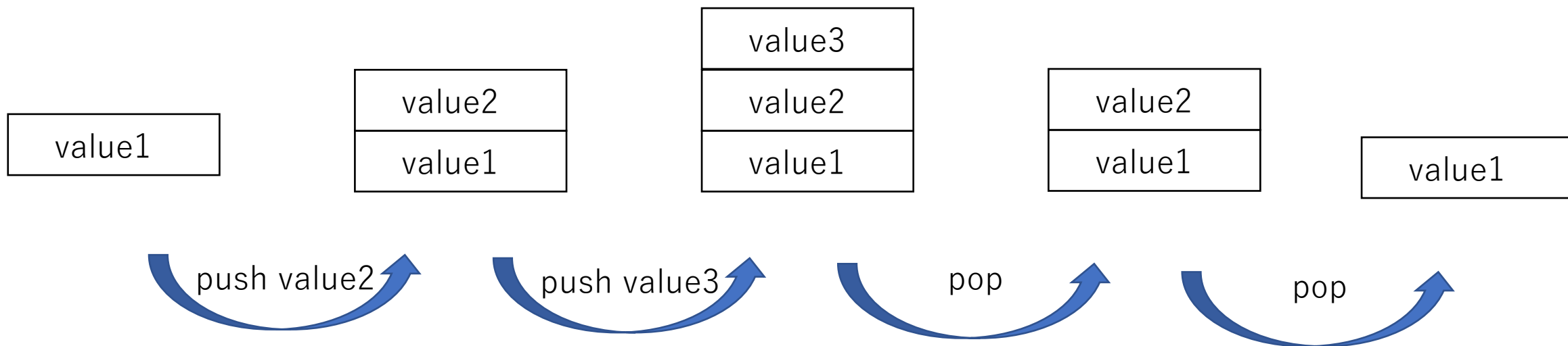
バッファオーバーフロー

- 一般的には「データ用に用意されたサイズ」を超えて書き込みを行ってしまうこと
- メモリ上において、データの下に配置された別のデータを誤って上書きしてしまう
- 今回は「スタック」上のバッファオーバーフローについて扱うが「ヒープ領域」等、別の領域でも同様のことが発生しうる

スタック

- 先入れ後出しのデータ構造で入れる(先頭に「積む」)ことをpush、先頭から取り出すことをpopという
- 実行バイナリではメモリ上にスタック用の領域を確保し、スタックの先頭アドレスを`rsp`というレジスタで管理している
 - スタックの底のアドレスも`rbp`というレジスタで管理している
- 以下の説明ではメモリアドレスは図の下の方が大きい値とする
 - つまりpushするとスタックは上に伸びるのでrspは小さくなる

スタック



スタックとローカル変数

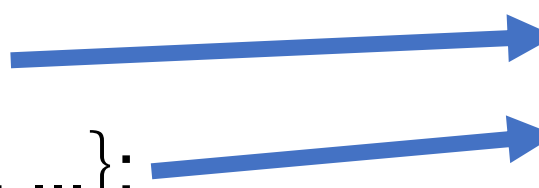
```
int main() {
```

```
    int x = 0xdeadbeef;
```

```
    int arr[10] = {1, 2, 3, ...};
```

```
    ...
```

```
}
```



0xdeadbeef
0x1
0x2
0x3
⋮

アセンブリ言語で変数にアクセスする時は大抵`rbp`からのアドレス差分を用いる

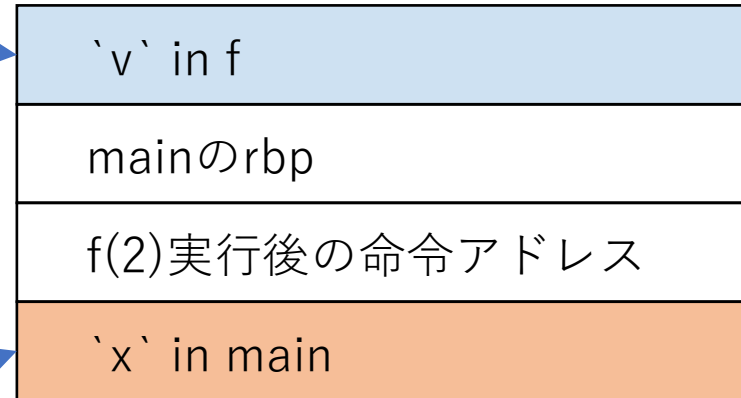
関数のcallとreturn

- 関数をcallする時、呼び出し先のローカル変数のための領域が欲しいので元のスタック領域の上に確保する
 - よって元のスタック領域のrspが新たなrbpになる
- 一方、returnして呼び出し元に戻る時に、どこに戻るかという情報が必要なので事前にスタックに積んで退避しておく
 - 元のrbpも必要なので事前にスタックに積んで退避しておく
- return時は逆向きの操作を行ってripの設定とrbpの復元を行う

関数呼び出しとスタック

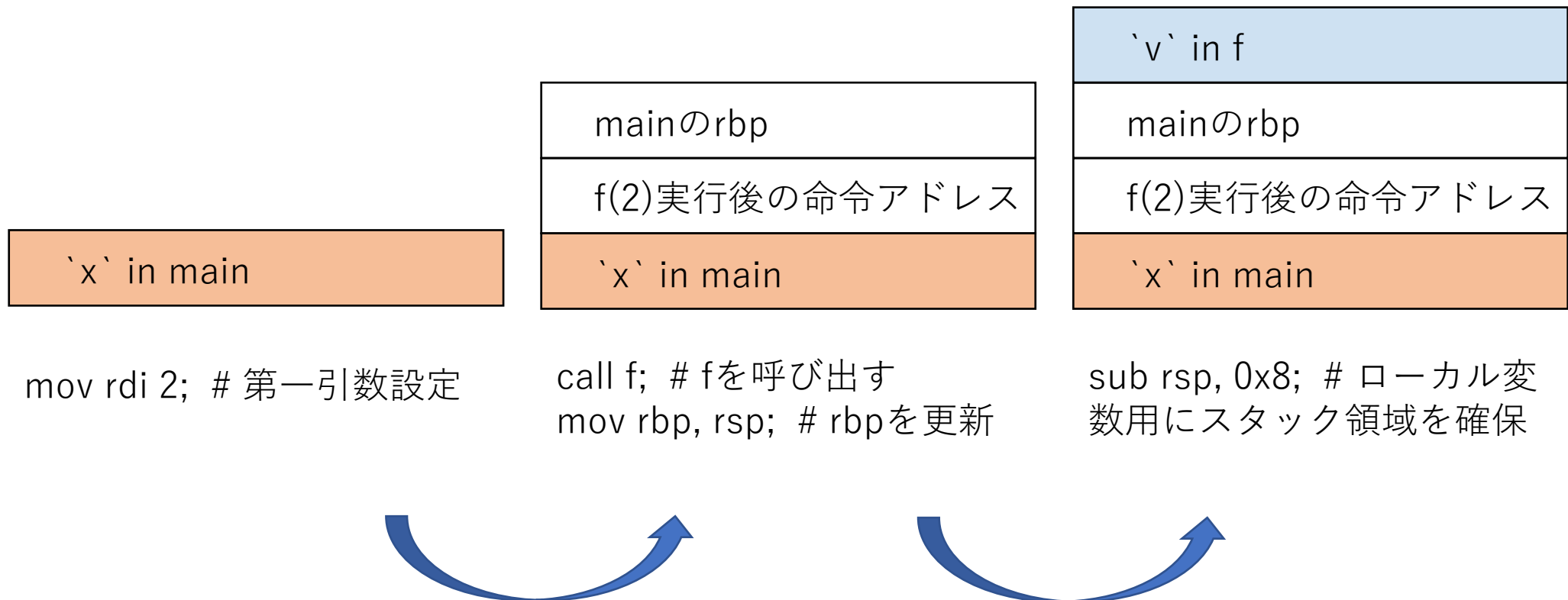
```
int f(int v) {  
    return v+2;  
}
```

```
int main() {  
    int x;  
    x = f(2);  
    ...  
}
```



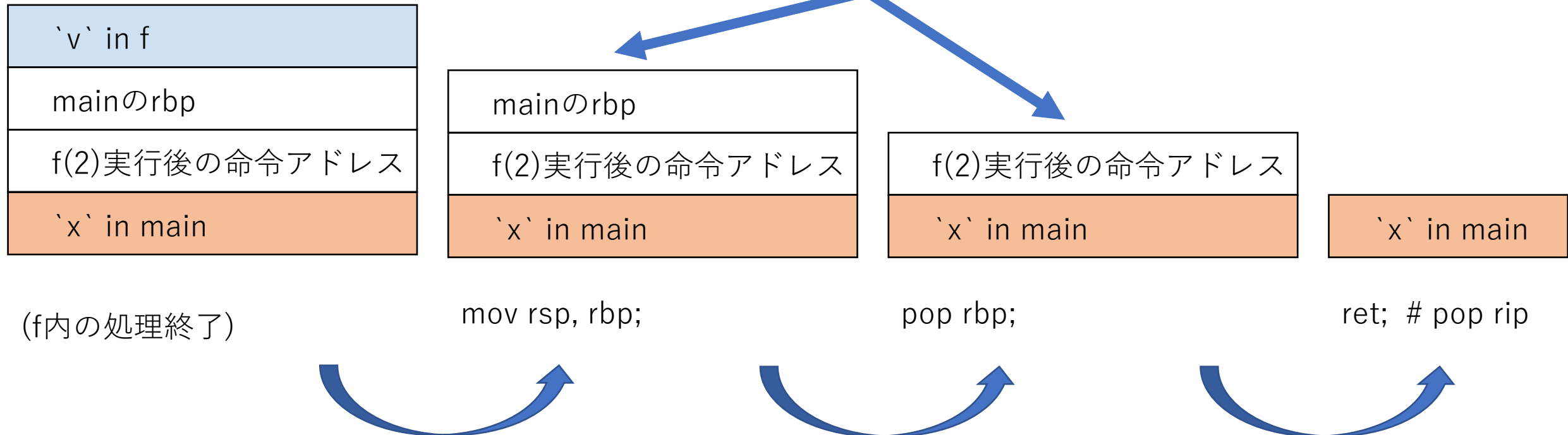
f(2)がcallされる時にスタックに積まれる値

関数呼び出しとスタック



関数を抜ける時のスタック

この2つは`leave`という1つの命令に
まとめられている



SBOFによって出来ること

- 自身の下の変数を書き換える
 - 認証用の変数を0(false)から1(true)に書き換える
 - 重要な値(サイズや添字)を書き換えて他のバグに繋げる
- 関数のリターンアドレスはローカル変数の下に存在するのでここを書き換えてリターン先を別のアドレスにする

SBOFが起こる原因

- 入力関数の不適切な使用
 - ``gets(buf)``, ``scanf("%s", buf)``: ``buf`` に対して「任意長」の書き込み
 - ``fgets(buf, size, stream)``: ``buf`` のサイズを超えた ``size`` を設定
- 範囲指定ミス (OOB: Out of Bounds)
 - 配列サイズを超えた添字や負の添字を指定するといった添字指定ミス
- ロジックのバグ
 - サイズや添字指定変数を大きい方向へ書き換えることが出来る等

攻擊手法

概観: メモリ空間

- 実行バイナリ自体が置かれる領域
 - 実行コードや、グローバル変数はここに置かれる
- スタック領域
 - ローカル変数や関数呼び出し時の情報が置かれる
- ヒープ領域
 - 主に可変長の領域が欲しい時にここから切り出す
- その他
 - ライブラリやリンクのコードが置かれる領域等

概観: バイナリとメモリ空間

- バイナリはメモリに配置される時、役割やパーミッションごとにセクションという単位に分かれる
 - パーミッションはRead, Write, Executeの3つがある
- 主なセクション
 - .textセクション: 実行コードが配置 (RX)
 - .rodataセクション: “read only”なデータが配置 (R)
 - .dataセクション: 読み書きが可能なデータが配置 (RW)

攻撃: Shellcode

- 書き込み可能な領域に機械語のコードを書いてそこを実行する
- だいたいの攻撃がシェルを起動するコードを書くことからこう呼ばれる
- スタック上を実行不可能にするNXという防衛機構が追加
 - NX有効下でも、mmapシステムコールによって(R)WX領域を作ってそこへ飛ばす事は可能

攻撃: ROP

- retがスタックの値をripに設定する事を利用して制御を奪う
- NX有効化では真に自由な命令は実行出来ないが、「バイナリ中に最初から存在する命令」は実行出来る
- `pop rdi; ret` という命令があれば、スタック上の値を第一引数に設定してその下の値をripに入れることが出来る
 - 下の値を飛ばしたい関数のアドレスにすれば任意関数を任意の引数を設定して実行できる

攻撃: ROP

rip: pop rdiのアドレス

pop rdi; ret のアドレス
0x1
関数fのアドレス

rip: retのアドレス

関数fのアドレス

rip: 関数fのアドレス

関数fのアドレス



pop rdiによって0x1がrdiに代入される



retによってripが関数fのアドレスに設定される



f(1)が実行される

SBOFに対する防衛機構

防衛機構: NX bit (No eXecute bit)

- メモリ空間を実行のための「コード領域」と読み書きのための「データ領域」に分けて後者を実行不可能にする機構
- スタック上に置いたコードは実行出来なくなるため、シェルコード攻撃を防ぐことが出来る
- mmapシステムコールでRWX領域は作ることが出来るので任意のアドレスに対してこれが適用されているわけではない

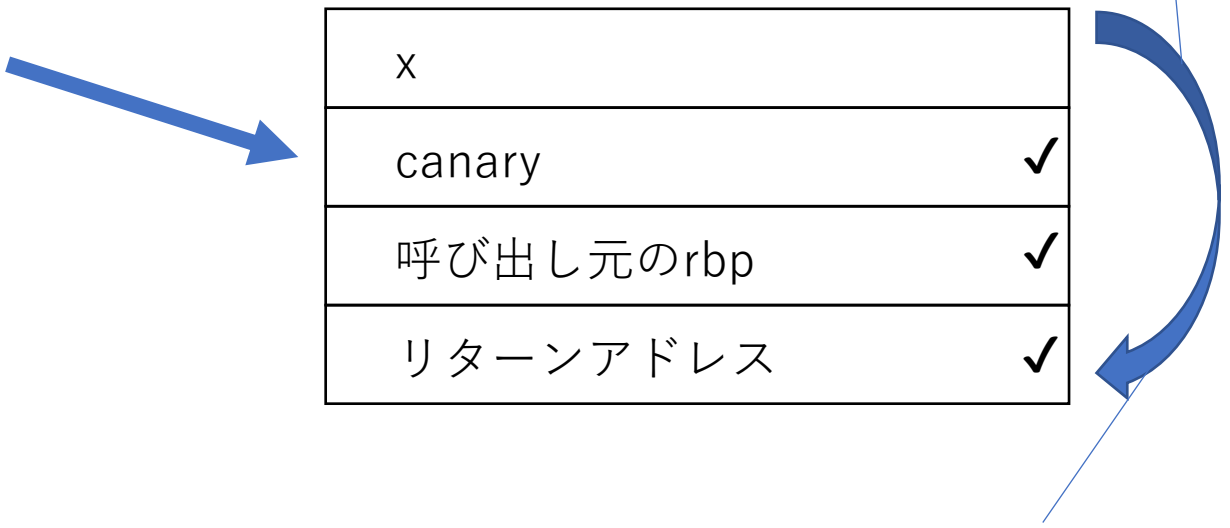
防衛機構: Canary

- SBOFを検知するための機構で、スタックの最下部に番兵用の値を置いてこれが書き換わっていないかを確認める
 - 基本的に不変なレジスタ(`gs:0x14`)の値を用いる
- 他のバグと併せてこの値を読むことが出来れば回避可能
 - BOFは(範囲を超えて)「読み出す」場合もある
 - ``int arr[16]``に対して``arr[i]``を表示する時に、for文の範囲指定を間違えて16以上の``i``を指定出来るとスタックの下の方を覗ける

防衛機構: Canary

```
Int main() {  
    int canary = gs:0x14;  
    int x;  
    ...  
    if (canary != (gs:0x14)) {  
        stack_chk_fail();  
    }  
}
```

ローカル変数部からリターンアドレスを書き換えようとするとき
canaryも書き換わる



x	
canary	✓
呼び出し元のrbp	✓
リターンアドレス	✓

canaryの書き換えを検知すると
stack_chk_fail() という関数でエ
ラー終了する

防衛機構: PIE

- Portable Independent Executableの略でコード領域の配置アドレスをランダムにする
 - 逆にPIEが無効下なら、機械語がどのアドレスに存在しているかは既知
- ROPをするにはコード領域のアドレスを知る必要があったのでここがランダムの場合は別のバグ等でリークする必要がある
 - Overread等で、スタックの下の方を覗いてリターンアドレスを特定すると配置アドレスが判明する

防衛機構: Intel CET

- call時にリターンアドレスをスタックに積むが、“shadow stack”と呼ばれる領域にも積んで、ret時に照合する
- よって、自由なアドレスへretすることが出来なくなる
- リターンアドレスの書き換えに大幅な制限が加わることでROP終了

防衛機構(?): メモリ安全な言語を使う

- だいたい言語の使い方を間違えている事が主な原因
 - C言語: 論外
 - C++: `std::vector`等でも領域外参照は相変わらず可能
- Rustはメモリ安全を徹底して作られた言語なので不安全な参照等をしようとする「前に」コンパイラに捕捉される
 - 所有権: そのメモリをどの変数が有しているか
 - ライフタイム: そのメモリはどのスコープまで有効か
- Rustの言語仕様自体によるバグ以外のバグ、つまり書き手によるバグの混入が殆ど起こらない

その他の面白いBinary Exploit

- Format String Attack (書式文字列攻撃)
 - ``printf(buf)``のように自由に書式文字列を設定出来てしまう際にスタック上の値を自由に読み「書き」出来てしまう攻撃
- Heap Based Exploit
 - Heap領域と呼ばれる可変長領域のメタ情報を書き換える攻撃
 - `malloc`や`free`の使い方を誤る事で発生
 - だいたいの場合、任意アドレス書き込みに持ち込んで関数ポインタを書き換える