

\$?

@MysticDoll

Twitter @MysticDoll

Name Mitsuru Takigahira

いま しゃかいじん 1 年目
ヤのつく組織の正社員をやっている

今日は\$?の話をします

具体的には以下の C コード

```
void main () { }
```

をコンパイルしてできた a.out を実行後の\$?の話です

\$?を追うことになった経緯

- ① 貸与 PC 配られる
- ② gcc で適当にコード書いて遊ぶ
- ③ void main() { } での exit code ってどう決まるんだろう？
- ④ 技術研修中ところどころやることねえな
- ⑤ とりあえず色々追ってみるか～

\$?を追うことになった経緯

- ① 貸与 PC 配られる
- ② gcc で適当にコード書いて遊ぶ
- ③ `void main() { }` での exit code ってどう決まるんだろう？
- ⑤ とりあえず色々追ってみるか～

調査した環境

- Arch Linux (glibc 2.27)
- CentOS 6.9 (glibc 2.12)
- 共に x86_64

何故この環境か

- Arch Linux は普段使い環境なので自明
- いろいろあった
 - 外部ベンダーから渡される PC、VM、何もおきないはずがなく...
- これらの環境でやったおかげで変なことが起こって少し良かった

exit code

- プログラムの exit code は main 関数の返り値
 - 何も考えず gcc でコンパイルした場合はそうなる
- `void main() { }` で記述された関数の返り値は未定義

⇒ ではどうやって `void main() { }` で書かれた関数の返り値が決まるのか

ABI

- どのレジスタをどの値に使うかは ABI 仕様により定まる
 - x86_64 System V の場合は `%rax` の値が関数の返り値として使われる¹
- つまり、`void main() { }` で何もしないプログラムでは、`main` 関数実行時の `%rax` が返り値として採用される

- カーネルの `exit syscall` の実装を見ると

```
SYSCALL_DEFINE1(exit, int, error_code)
{
    do_exit((error_code&0xff)<<8);
}
```

となっていて、下位 8 ビットのみをエラーコードとして利用している
(右 8 ビットシフトしている部分が見つからなかったなので誰か教えて)

glibc

- main 関数は glibc の `__libc_start_main` から呼ばれる
 - 実際には `_start` 内でレジスタの初期化をして `__libc_start_main` で `main` を呼び出す処理を gcc が作ってくれてる
- glibc のソースの `csu/libc-start.c` を見ると良い²

¹https://wiki.osdev.org/System_V_ABI#x86-64

²[git://sourceware.org/git/glibc](https://sourceware.org/git/glibc)

調べる

やってみる

- 無のプログラム内で%rax に入れた値が exit code になるか確かめる
- 無のプログラムの実行とその exit code の確認

やってみる

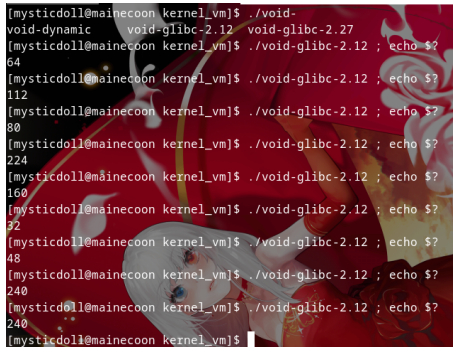
exit code とレジスタ

- ① `$ echo 'void main() { __asm__("mov $123, %rax"); }' \`
`| gcc -xc - -O0 -o exit.out;`
- ② `$/exit.out ; echo $? ;`

⇒ 123 が返ってくる。

無のプログラムの exit code

変なことが起きた

A terminal window with a red and white floral background. The prompt is [mysticdoll@mainecoon kernel_vm]\$. The first command is ./void-dynamic void-glibc-2.12 void-glibc-2.27. Subsequent commands are ./void-glibc-2.12 ; echo \$? which outputs 64, 112, 80, 224, 160, 32, 48, 240, and 240 respectively. The final command is ./void-glibc-2.12 ; echo \$? which outputs 240.

```
[mysticdoll@mainecoon kernel_vm]$ ./void-  
void-dynamic void-glibc-2.12 void-glibc-2.27  
[mysticdoll@mainecoon kernel_vm]$ ./void-glibc-2.12 ; echo $?  
64  
[mysticdoll@mainecoon kernel_vm]$ ./void-glibc-2.12 ; echo $?  
112  
[mysticdoll@mainecoon kernel_vm]$ ./void-glibc-2.12 ; echo $?  
80  
[mysticdoll@mainecoon kernel_vm]$ ./void-glibc-2.12 ; echo $?  
224  
[mysticdoll@mainecoon kernel_vm]$ ./void-glibc-2.12 ; echo $?  
160  
[mysticdoll@mainecoon kernel_vm]$ ./void-glibc-2.12 ; echo $?  
32  
[mysticdoll@mainecoon kernel_vm]$ ./void-glibc-2.12 ; echo $?  
48  
[mysticdoll@mainecoon kernel_vm]$ ./void-glibc-2.12 ; echo $?  
240  
[mysticdoll@mainecoon kernel_vm]$ ./void-glibc-2.12 ; echo $?  
240  
[mysticdoll@mainecoon kernel_vm]$
```

Figure: glibc 2.12 を static link したプログラムの様子 ³⁴

³CentOS6.9 では glibc-2.12 を使っているため これは Arch でこのバージョンをリンクしたやつ

⁴これ以降のバージョン及び shared な glibc を使う場合は値は毎回同じ

更に調べる

- 何もしないプログラムの実行時の exit code の決定タイミング
- glibc 2.12 を静的リンクした場合、毎回 exit code が変わる原因の特定
 - CentOS 6.9 では検証時に exit code が毎回変わる現象が起きたため

確かめる

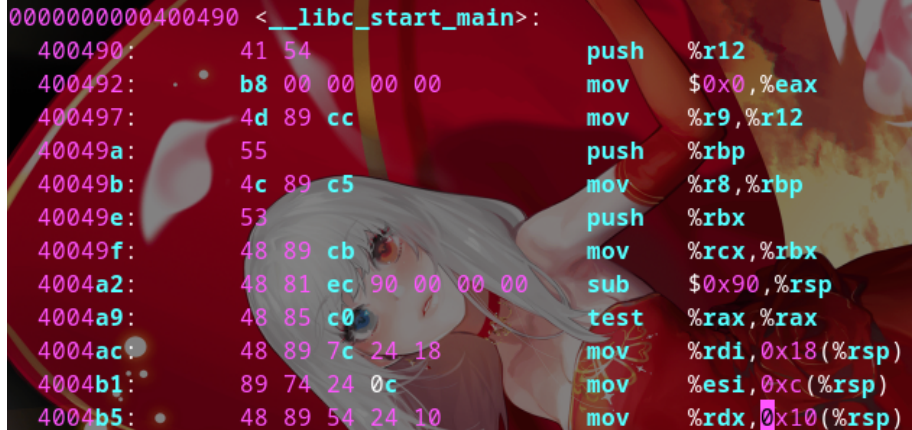
逆アセンブル

- ① `$ echo 'void main() { }' | gcc -xc -O0 -static -o void;`
- ② `$ objdump -D void | less;`

読む

- 当然 main 関数のセクションを見ても何もしていないため、
`__libc_start_main` のセクションを読む

static link した glibc 2.12 での処理を追う



```
0000000000400490 <__libc_start_main>:
 400490:      41 54                push    %r12
 400492:      b8 00 00 00 00      mov     $0x0,%eax
 400497:      4d 89 cc             mov     %r9,%r12
 40049a:      55                  push    %rbp
 40049b:      4c 89 c5             mov     %r8,%rbp
 40049e:      53                  push    %rbx
 40049f:      48 89 cb             mov     %rcx,%rbx
 4004a2:      48 81 ec 90 00 00 00 sub     $0x90,%rsp
 4004a9:      48 85 c0             test    %rax,%rax
 4004ac:      48 89 7c 24 18       mov     %rdi,0x18(%rsp)
 4004b1:      89 74 24 0c          mov     %esi,0xc(%rsp)
 4004b5:      48 89 54 24 10       mov     %rdx,0x10(%rsp)
```

Figure: glibc 2.12 libc_start_main

static link した glibc 2.12 での処理を追う

__libc_start_main での処理

- 第一引数でメイン関数のポインタを受けとり、そしてそれを実行する
- 実際にアセンブリを見ると `mov %rdi, 0x18(%rsp)` しており、
`%rsp + 0x18` の位置にメイン関数のポインタが入ることがわかる

static link した glibc 2.12 での処理を追う

```
400609: 48 8d 44 24 20      lea    0x20(%rsp),%rax
40060e: 64 48 89 04 25 00 03  mov    %rax,%fs:0x300
400615: 00 00
400617: 48 8b 15 4a 72 2a 00  mov    0x2a724a(%rip),%rdx      # 6a7868 <__environ>
40061e: 48 8b 74 24 10      mov    0x10(%rsp),%rsi
400623: 8b 7c 24 0c      mov    0xc(%rsp),%edi
400627: ff 54 24 18      callq  *0x18(%rsp)
40062b: 89 c7      mov    %eax,%edi
40062d: e8 0e 0a 00 00      callq  401040 <exit>
```

Figure: glibc2.12 での main 関数呼び出し

glibc 2.12 での処理を追う

glibc 2.12 での main 呼び出し部分

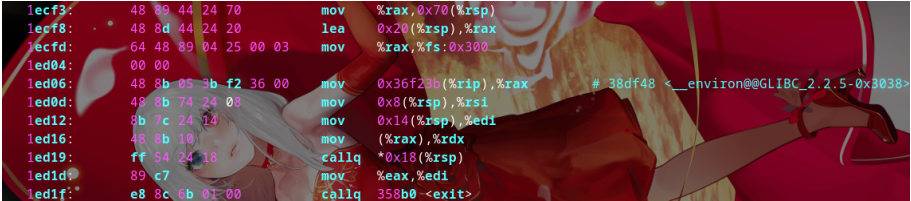
- `callq *0x18(%rsp)` している部分が `main` 関数の実行
- 実際に返り値 (つまり `%eax` の値) を `exit` 関数の第一引数に渡している
- よく見るとこの画像最初の部分で `lea 0x20(%rsp), %rax` している
 - `%eax` は `%rax` の下位 32 ビットなので、ここで返り値が決まる
 - つまり `%rsp + 0x20` が返り値として定まる

返り値がランダムになる理由

ASLR (Address Space Layout Randomization)

- Linux2.6.12 以降ではデフォルトで有効になっている
- 共有ライブラリ及びスタック領域の位置がランダムに定まる
- つまりスタックポインタ (%rsp) + 0x20 の値もランダムに決まる
- 実際 sysctl コマンドなどで ASLR を無効化すると定数が返ってくる

Shared な glibc 2.12 の場合



```
1ecf3: 48 89 44 24 70      mov    %rax,0x70(%rsp)
1ecf8: 48 8d 44 24 20      lea    0x20(%rsp),%rax
1ecfd: 64 48 89 04 25 00 03 mov    %rax,%fs:0x300
1ed04: 00 00
1ed06: 48 8b 05 3b f2 36 00 mov    0x36f23b(%rip),%rax # 38df48 <__environ@@GLIBC_2.2.5-0x3038>
1ed0d: 48 8b 74 24 08      mov    0x8(%rsp),%rsi
1ed12: 8b 7c 24 14      mov    0x14(%rsp),%edi
1ed16: 48 8b 10      mov    (%rax),%rdx
1ed19: ff 54 24 18      callq  *0x18(%rsp)
1ed1d: 89 c7      mov    %eax,%edi
1ed1f: e8 8c 6b 01 00      callq  358b0 <exit>
```

Figure: Shared な glibc-2.12 の場合

Shared な glibc 2.12 の場合

- static link の際と同じように `%rax` を変更している
- その後に `%rip` (プログラムカウンタ) + `0x36f23b` を `%rax` に入れている
- よってこの値が毎回返ってくる

glibc 2.27 の場合

```
400ff1: 48 8b 15 70 2d 2a 00    mov     0x2a2d70(%rip),%rdx
400ff8: 48 8b 74 24 10          mov     0x10(%rsp),%rsi
400ffd: 8b 7c 24 0c            mov     0xc(%rsp),%edi
401001: 48 8b 44 24 18          mov     0x18(%rsp),%rax
401006: ff d0                 callq   *%rax
401008: 89 c7                 mov     %eax,%edi
40100a: e8 a1 5f 00 00        callq   406fb0 <exit>
```

Figure: glibc 2.27 の main 関数実行

glibc 2.27 の場合

- main 関数のアドレスを `%rax` に入れてから `callq *%rax` している
- そのため `%rax` は常に main 関数のアドレスとなり、定数が返ってくる
- この呼び出し方は最新の ABI を読むと
Position-Independent Indirect Function Call という項に書いてある
 - 古い ABI は見つからず diff は取れていないが多分そういうことですね

わかったこと

- 意識せずにコンパイルしてるが、コンパイラはいろいろしている
- 事前知識がないと glibc が悪いと気づかずカーネルとか読みがち
- 同期の蛙と餃子がちょこちょこ付き合ってくれて楽ができた (ありがとう)
- みなさんも楽しく気軽に `objdump -D` しましょう

おわり