Shellcoder's Challenge の手引き 2

■システムコール

OS（オペレーティングシステム）とはざっくりと説明すると「キーボード入力、ディスプレイ出力、メモリ管理、ディスクアクセスといった全般的なハードウェアの管理を引き受けてくれるコードの集まり」です。まぁハードウェア管理以外にもいろいろとやることはあるのですが、それはここでは置いておきます。

アプリケーション（ユーザーランドで動作するコード）が何かの文字をディスプレイに出力したい場合、ハードウェア（ディスプレイ）を管理している OS にその旨を伝えなければなりません。

OS はアプリケーションのためにシステムコールを提供します。

/usr/include/i386-linux-gnu/asm/unistd_32.h

```
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H
/*
 * This file contains the system call numbers.
 */
#define __NR_restart_syscall      0
#define __NR_exit                 1
#define __NR_fork                 2
#define __NR_read                 3
#define __NR_write                4
#define __NR_open                 5
#define __NR_close                6
#define __NR_waitpid              7
#define __NR_creat                8
#define __NR_link                 9
#define __NR_unlink              10
#define __NR_execve              11
#define __NR_chdir               12
```

システムコールとは、OS がアプリケーションのために用意したライブラリ（のようなもの）です。これらを組み合わせてアプリケーションを作成します。

文字列をディスプレイに表示するには、通常は printf や puts を使うわけですが、これらも内部では（最終的には）システムコールを呼んでいます。

syscall.s

```
.global main
main:
        mov     $13,%edx # length of string
        lea     ss, %ecx # addr of string
        mov     $1, %ebx # stdout
        mov     $4, %eax # write sys-num is 4
        int     $0x80
        xor     %eax, %eax
        ret
ss:
        .string "Hello World!¥n"
```

Ubuntu/x86 で上記のコードを実行すると Hello World!と表示されます。

```
$ gcc syscall.s -o syscall
$ ./syscall
Hello World!
```

C 言語にするとこんな感じになるでしょう。

```
write(1, "Hello World!¥n", 13);
```

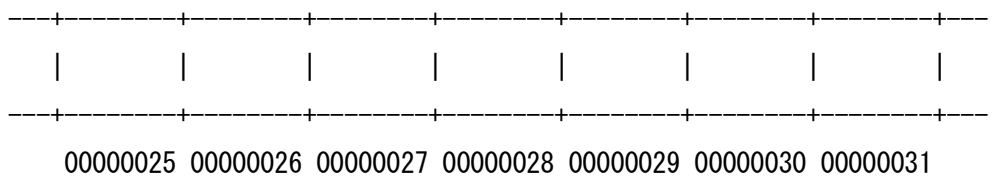
eax がシステムコール番号で unistd_32.h をみると write は 4 と定義されているので、eax には 4 を入れます。これは環境（OS）によって違うので各環境によって調べなければなりません。ebx、ecx、edx はそれぞれ引数です。順番に 1、”Hello World!”、そして 13 が入れられます。この状態で int $0x80（もしくは sysenter）を実行することで write システムコールを呼び出せます。

ちなみに Windows の場合は ntdll.dll 内の関数から sysenter が呼ばれています。とはいっても Windows においては直接 sysenter を呼ぶ機会はあまりありませんが…。

■スタック

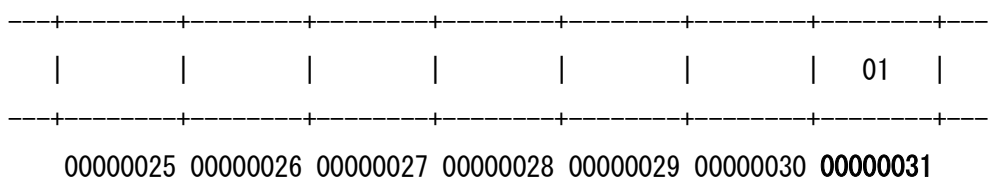
続いてスタックについて解説します。

スタックとはメモリ管理の概念です。よく「筒」や「積み上げる皿」を例に解説されますが、スタックと呼ばれるメモリが実際に存在するわけではなく、あくまでも「筒や積み上げられる皿のような使われ方をされるメモリ」という感じです。

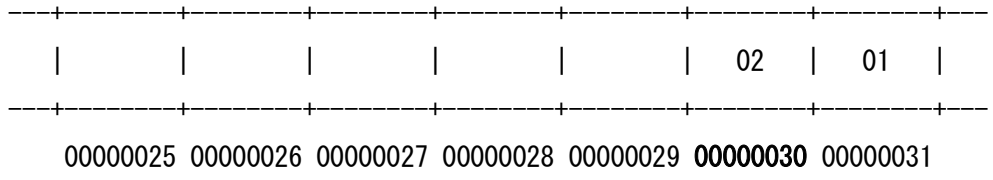


実行されたプログラムは、最初にスタックの基点となる場所を決めます。ここでは 00000031 を基点（ebp）としましょう。

基点（ebp）が決められた後、例えば push 命令が実行されると、スタックにデータが格納されます。格納されるデータは当然 push されたものです。仮に「push 0x01」が実行されたとしましょう。

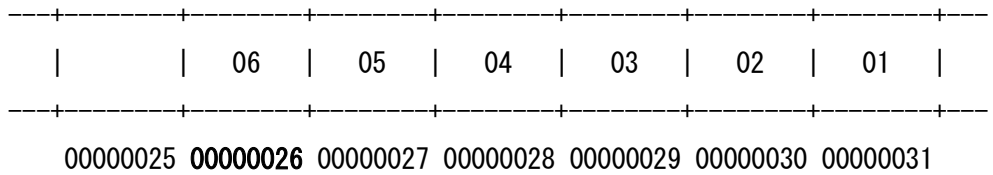


00000031 に 01 が格納されました。さらに push 命令を、次は「push 0x02」を実行しましょう。



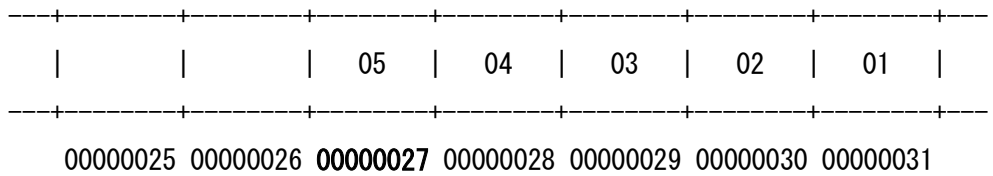
最初に格納されたのは 01 であり、2 番目に格納されたのが 02 です。

このようにスタックはメモリアドレスの低位（減算方向）に向かって伸びます（成長します）。push 命令を実行し続ければ、その分だけメモリの低位に向かって値が格納されていきます。

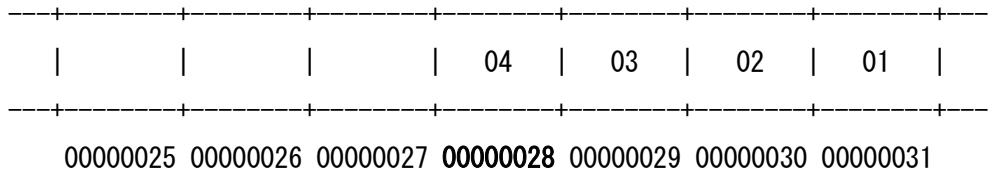


上記は 01～06 までの値が順番に push された状態です。

同じようにして、今度はスタックからデータを取り出すことを考えます。スタックからデータを取り出す場合は pop 命令です。pop 命令が実行されると、スタックのもっとも低位のアドレスからデータが取り出されます。このスタックのもっとも低位のアドレスをスタックのトップ（esp）と呼びます。これまで**太字**にしていたアドレスです。



最後に push されたデータ 06 が最初に pop されました。よって、次に pop 命令が実行されたら、05 がデータとして取り出されます。この時、スタックのトップ（esp）は 00000027 から 00000028 へと変わります。つまり、基点（ebp）とは異なりスタックのトップ（esp）は常に（push/pop 毎に）変動するわけです。



ここではわかりやすさのために 000000XX といった値を使いましたが、これだとすぐにスタックを使いきってしまう（数回の push で 00000000 になってしまう）ので、一般的には 0xbffffXXX といったような「高いアドレス」を基点としてスタートします。

■ レベル 9

マシン語を逆アセンブルし、それを解読して答えを見つける問題です。

Level 9 ★

Ubuntu/x86環境で以下のマシン語を実行した。
表示される文字列を答えよ。

```
b8 61 41 61 41 53 50 ba 04 00 00 00 bb 01 00 00
00 b8 04 00 00 00 89 e1 cd 80 58 31 c0 5b c3 90
```

逆アセンブルせずとも読めたらそれが一番ですが、逆アセンブルするとこのようになります。ちなみにこれは Intel 記法ですね。

```
b861416141: mov eax, 0x41614161
        53: push ebx
        50: push eax
ba04000000: mov edx, 0x4
bb01000000: mov ebx, 0x1
b804000000: mov eax, 0x4 # write system-call
      89e1: mov ecx, esp
      cd80: int 0x80
        58: pop eax
      31c0: xor eax, eax
        5b: pop ebx
        c3: ret
        90: nop
```

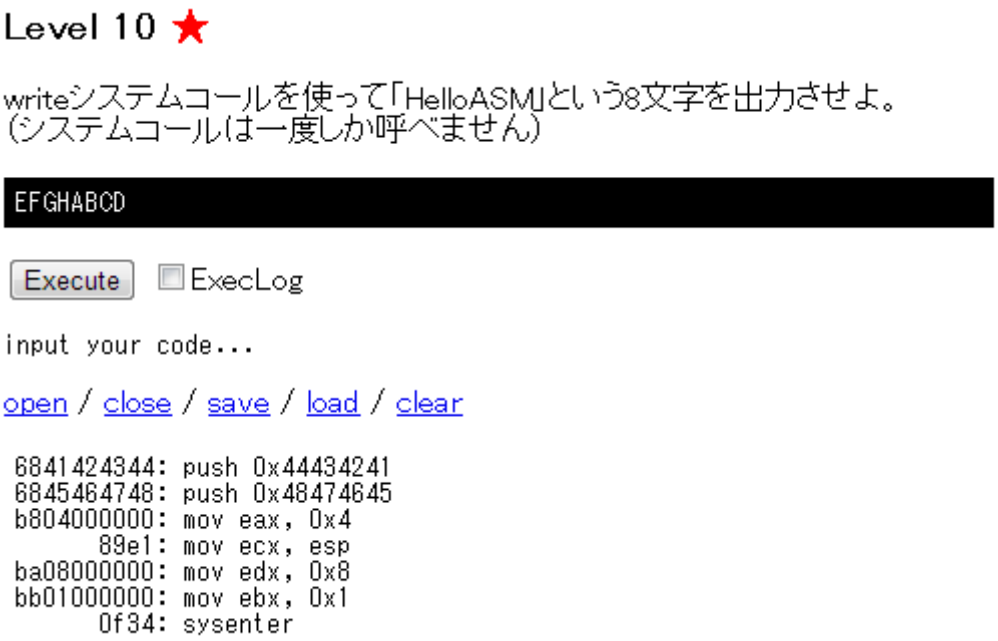
まず eax が 4 なので write システムコールですね。そして ebx が 1 なので stdout、edx は 3 番目の引数（サイズ）なので、size=4 となります。つまり 4 文字の String が出力されるわけです。

そして肝心の出力される文字 ecx には esp が入れられています。esp はスタックのトップを指しますので最後に push された値を確認すると eax です。そして eax には 0x41614161 が入れられています（1 行目）。

というわけで、ASCII コード表により 0x41='A'、0x61='a'から AaAa という 4 文字が正解かなとおもいきや違います。答えは aAaA です。

■レベル１０

ここからマシン語を書く問題になります。



まずは使い方から。

open で編集開始、close で編集終了、save で現在のコードを保存、load で保存されたコードをロード、clear で保存されたコードを削除となります。

Execute は現在のコードを実行し、その結果を表示します。デフォルトのコードは write システムコールで EFGHABCD を出力します。ExecLog にチェックを入れると命令毎のレジスタの状態が表示されます。デバッグに使いましょう。

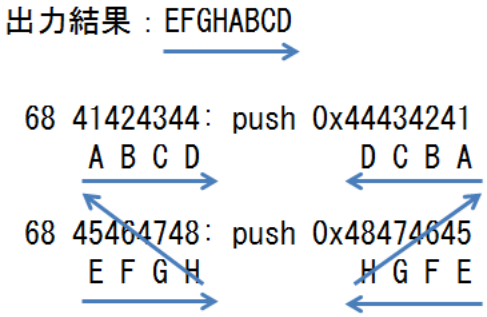
最後に、「正直、マシン語の直書きがツライ」という方は、アセンブラコードを書いたあと最後に'='を追加してみてください。少しは楽になるかもしれません。

```
mov eax, ecx=
```

では問題の解説に移りましょう。

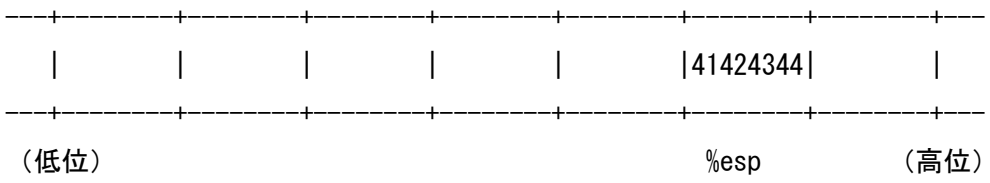
とりあえず Execute をクリックすると EFGHABCD と表示されます。HelloASM という文字列を表示させるという問題で文字列長は同じ 8 なので、E に対応するところに H、F に対応するところに e といった具合に置き換えていけばうまくやれそうです。

push に渡したデータとマシン語コード、そして実際に出力されている値を見比べて、どのように配置が変わるかを確認してください。この辺、最初はややこしいですが、慣れると簡単だ（というか、ああーなるほど）と思います。

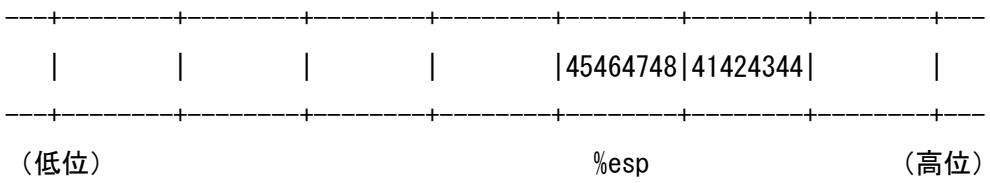


マシン語、アセンブラコード、そして実際の出力はこのような関係になっています。

最初の push で 0x44434241 がスタックへ送られます。push はこれを逆にしてスタックへ格納します。

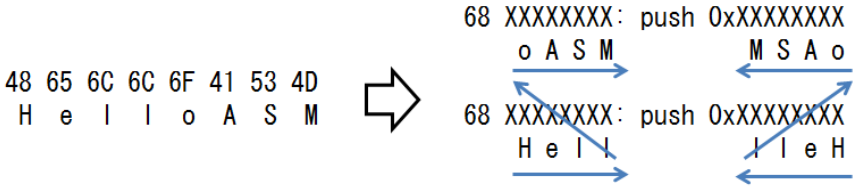


続いて、2 度目の push で 0x48474645 が送られます。



そして esp の値が ecx へコピーされて write システムコールが呼ばれますので、結果として 4546474841424344（EFGHABCD）という順番で出力されます。

あとはこれらを HelloASM に変えるだけです。



Enjoy Hacking ☺