系统程序设计作业Unit1

2014302580341 卓越二班 余璞轩

1. 秘密信息

> *From: CTE*
> *To: You*
> *Excellent!You got everything!*

2.密码

> *3 777 4 44*

(On MacOS 10.12 Sierra)

3.process_keys12函数

原文件中的`process_keys12()`和与它相关的函数。

```
void process_keys12 (int * key1, int * key2) {
    *((int *) (key1 + *key1)) = *key2;
}

char * extract_message1(int start, int stride) {
    int i, j, k;
    int done = 0;

    for (i = 0, j = start + 1; ! done; j++) {
        for (k = 1; k < stride; k++, j++, i++) {
            if (*(((char *) data) + j) == '\0') {
                done = 1;
                break;
            }
            message[i] = *(((char *) data) + j);
        }
    }
```

```
    message[i] = '\0';
    return message;
}
```

第一条信息由`stride`和`start`决定，而这两者又由`dummy`决定，所以可以猜测到`process_keys12()`改变了`dummy`值。

`(int*)(key1 + *key1)`是`dummy`的地址值。这里`key1`是地址，`*key1`是`key1`的数值，所以说`key1=&dummy-&key1`，这两个变量在main函数中声明的时候隔了三个int型变量，所以`key1=3`。

4.前两位密码在破解程序中的作用

```
char * extract_message1(int start, int stride) {
    int i, j, k;
    int done = 0;

    for (i = 0, j = start + 1; ! done; j++) {
        for (k = 1; k < stride; k++, j++, i++) {
            if (*(((char *) data) + j) == '\0') {
                done = 1;
                break;
            }
            message[i] = *(((char *) data) + j);
        }
    }
    message[i] = '\0';
    return message;
}
```

尝试理解`start`和`stride`的作用，`start`是从第几个字符开始读，而`stride`是每读`stride-1`字符然后隔一个字符再读。

把data数组转换为字符输出出来，结果为：

> *ccccccccFFrromo: mFr:ie ndC*
> *TTo:E Y*
> *ouT*
> *Gooo:d! NYowo tury*
> *cEhoxoscineg lkelyse3,n4 tto! fYoroceu a cgalol tto eextvraectr2 yantd*
> *havioind gth!e*

结合原文开头为"From:"，这里`start=9, stride=3`可以满足题意。

然后因为key2就等于dummy值，由

```
start = (int)(*(((char *) &dummy)));
stride = (int)(*(((char *) &dummy) + 1));
```

得知，`dummy`的后四位为0309即可，所以`key2`的值可以为`int('0309',16)=777`

5.process_keys34()函数

```
void process_keys34 (int * key3, int * key4) {
    *(((int *)&key3) + *key3) += *key4;
}
```

一开始光看这个函数还不是太懂在做什么，但是分析一下main函数就知道这个函数应该是**修改了它call stack附近的某个值**，导致main函数的走向跟我们想的不太一样。

6.第一次call process_keys34()后

```
int main (int argc, char *argv[])
{
    int dummy = 1;
    int start, stride;
    int key1, key2, key3, key4;
    char * msg1, * msg2;

    key3 = key4 = 0;
    if (argc < 3) {
        usage_and_exit(argv[0]);
    }
    key1 = strtol(argv[1], NULL, 0);
    key2 = strtol(argv[2], NULL, 0);
    if (argc > 3) key3 = strtol(argv[3], NULL, 0);
    if (argc > 4) key4 = strtol(argv[4], NULL, 0);

    process_keys12(&key1, &key2);

    start = (int)(*(((char *) &dummy)));
    stride = (int)(*(((char *) &dummy) + 1));

    if (key3 != 0 && key4 != 0) {
```

```
        process_keys34(&key3, &key4);
    }

    msg1 = extract_message1(start, stride);

    if (*msg1 == '\0') {
        process_keys34(&key3, &key4);
        msg2 = extract_message2(start, stride);
        printf("%s\n", msg2);
    }
    else {
        printf("%s\n", msg1);
    }

    return 0;
}
```

第一次在if中执行了`process_keys34()`，应该打印出msg2，所以肯定是跳转到最后一个if的第一个条件里去了。

但是肯定不是跳进`process_keys34(&key3, &key4);`这句，否则会陷入某种死循环；所以可以肯定跳进了`msg2 = extract_message2(start, stride);`这句。

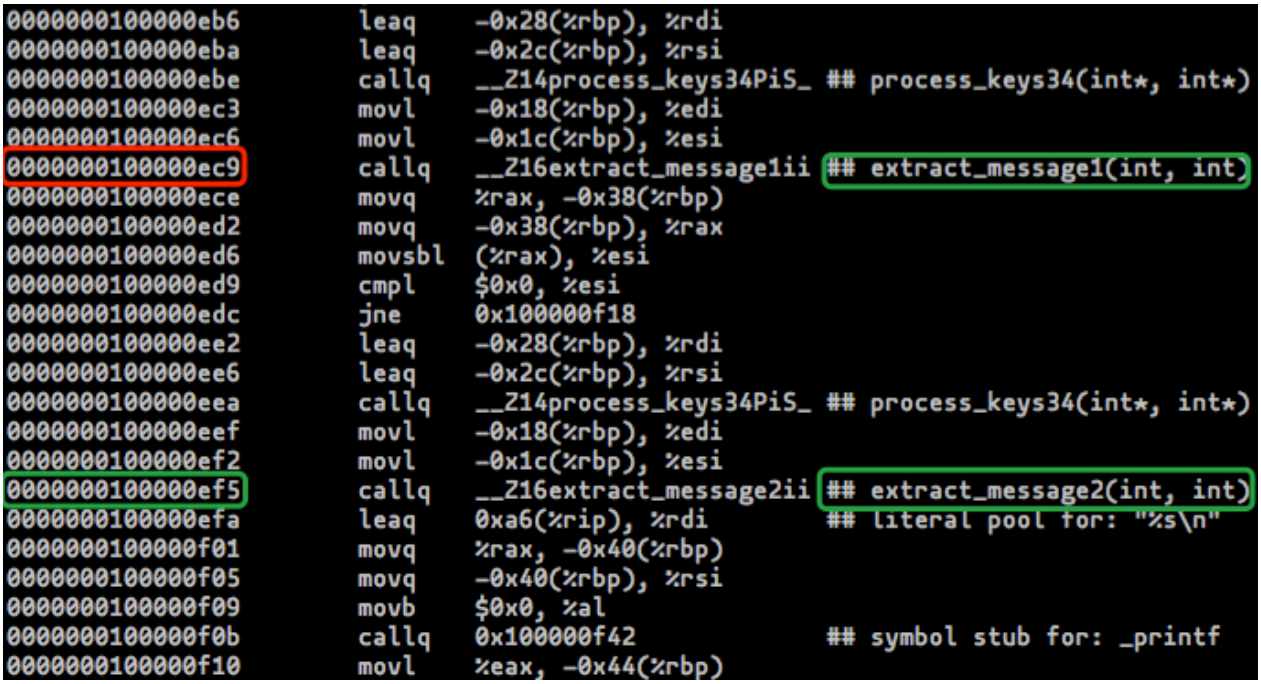
7.后两位密码在破解程序中的作用

很明显，main函数中的控制流进入`process_keys34()`后程序运行的方向似乎失去了常理。仔细看该函数。

```
void process_keys34 (int * key3, int * key4) {
    *(((int *)&key3) + *key3) += *key4;
}
```

可以推断这个函数改变了自己在堆栈上的返回地址。

我们用otool(mac版objdump)查看一下反汇编信息。



上下分别是理应进入的`extract_message1()`和`extract_message2()`的相应地址，两者相减得到44就是`key4`的值。

```
>>> int('ef5',16)-int('ec9',16)
44
```

最后一步要求`key3`。

在操作数大小为64位的架构中，参数按照顺序通过寄存器传递。在x86-64的架构里，栈指针是%rsp。看一下main函数里call函数的参数是如何传进去的。

```
0000000100000ed9 cmpl $0x0, %esi
0000000100000edc jne 0x100000f18
0000000100000ee2 leaq -0x28(%rbp), %rdi
0000000100000ee6 leaq -0x2c(%rbp), %rsi
0000000100000eea callq __Z14process_keys34PiS_ ## process_keys34(int*, int*)
0000000100000eef movl -0x18(%rbp), %edi
0000000100000ef2 movl -0x1c(%rbp), %esi
0000000100000ef5 callq __Z16extract_message2ii ## extract_message2(int, int)
0000000100000efa leaq 0xa6(%rip), %rdi ## literal pool for: "%s\n"
0000000100000f01 movq %rax, -0x40(%rbp)
0000000100000f05 movq -0x40(%rbp), %rsi
0000000100000f09 movb $0x0, %al
0000000100000f0b callq 0x100000f42 ## symbol stub for: _printf
```

两个leaq之后就开始movl，并没有压栈PUSH。同时把`0000000100000ef5`这个地址压入栈。再看看`process_keys34()`函数。

```
__Z14process_keys34PiS_:
0000000100000c50        pushq   %rbp
0000000100000c51        movq    %rsp, %rbp
0000000100000c54        movq    %rdi, -0x8(%rbp)
0000000100000c58        movq    %rsi, -0x10(%rbp)
0000000100000c5c        movq    -0x10(%rbp), %rsi
0000000100000c60        movl    (%rsi), %eax
0000000100000c62        movq    -0x8(%rbp), %rsi
0000000100000c66        movslq  (%rsi), %rsi
0000000100000c69        addl    -0x8(%rbp,%rsi,4), %eax
0000000100000c6d        movl    %eax, -0x8(%rbp,%rsi,4)
0000000100000c71        popq    %rbp
0000000100000c72        retq
0000000100000c73        nopw    %cs:(%rax,%rax)
```

先把%rbp入栈，此时栈上没有参数，然后movq用%rsp的值取代了%rbp原有的值，所以现在%rsp和%rbp都指向栈顶。因为该函数对`key3`进行了取址操作，所以要把它保存进内存。现在再把两个参数`key3` `key4`放进堆栈，所以现在栈顶长这个样子：

High
ef5（返回地址）
%rbp
key3
key4
Low

原函数中的`*(((int *)&key3) + *key3) += * *key4`等价于`*(&key3 + *key3) += *key4`。`key3`的地址离应该跳转到的地址有16bit，所以`key3=16/4=4`。

```
┌─Lucius@Macintosh  ~/Documents/SystemLevelProgramming/Unit1  <master*>
└─$ ./a.out 3 777 4 44
From: CTE
To: You
Excellent!You got everything!
┌─Lucius@Macintosh  ~/Documents/SystemLevelProgramming/Unit1  <master*>
└─$
```

At 10/23/2016