Software Exploitation
Code Reuse Attacks

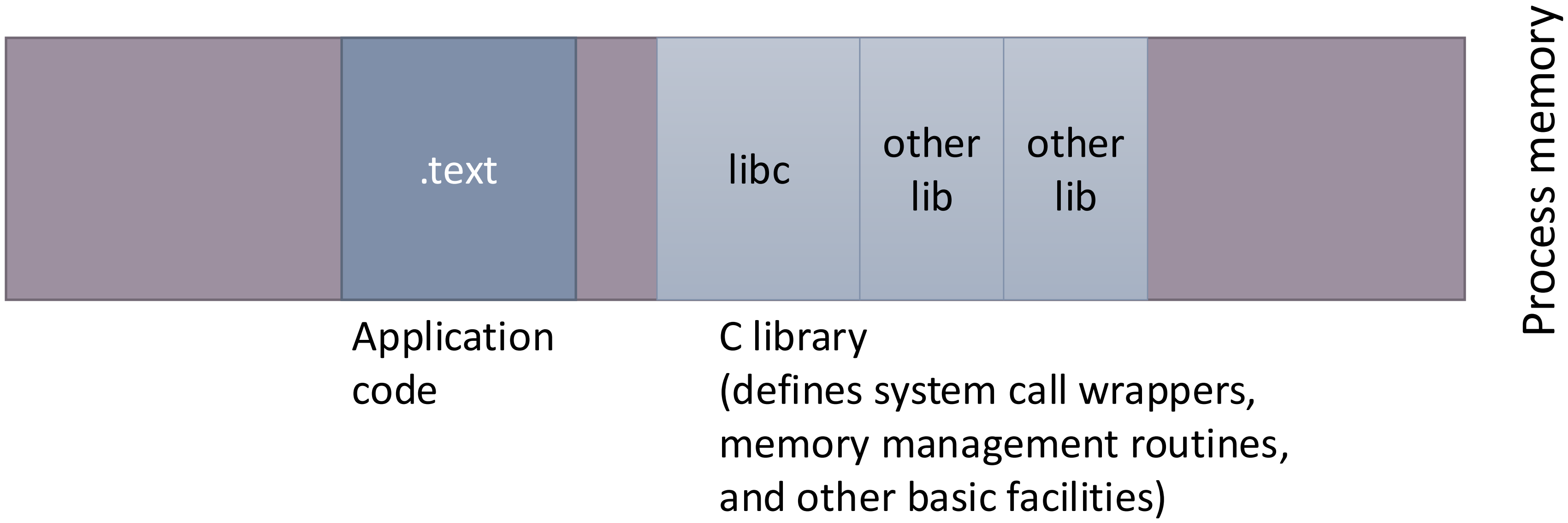
Georgios (George) Portokalidis

Ret2libc Attacks

- **What can I do if I control the return address when I cannot inject code**?

Ret2libc Attacks

- **What can I do if I control the return address when I cannot inject code**?
- Return to an existing function (e.g., a libc function)



Discovering Linked Libraries

```
$ ldd /bin/ls
	linux-vdso.so.1 (0x00007ffc83b62000)
	libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f9edfdf1000)
	libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1 (0x00007f9edfbe8000)
	libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9edf83d000)
	libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f9edf5cf000)
	libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f9edf3cb000)
	/lib64/ld-linux-x86-64.so.2 (0x00007f9ee0016000)
	libattr.so.1 => /lib/x86_64-linux-gnu/libattr.so.1 (0x00007f9edf1c6000)
	libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f9edefa9000)
```

ret2libc

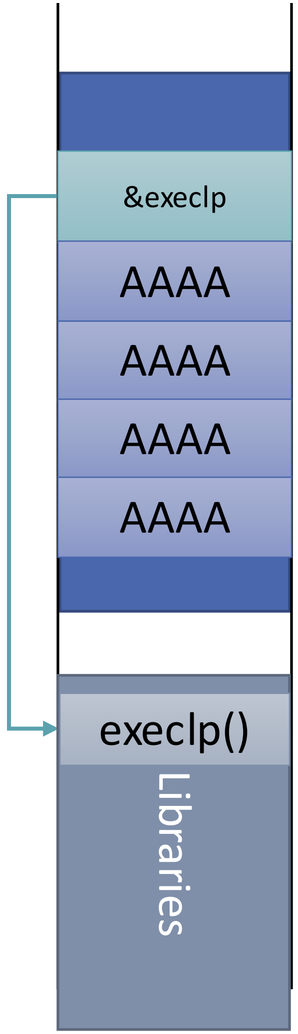
- Replace return address with the address of an **existing** function

- Example:

```
int execlp(const char *file, const char *arg, ...
               /*, (char *) NULL */);
```

- Executes file with command-line arguments
  - stdin and stdout connected to current process

- How to prepare arguments?
  - Arguments are passed using registers
    - First 6 integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, and R9
  - RBP, RBX, and R12–R15 are callee saved
  - RAX used for function return



Preparing Arguments (ret2libc on 64-bits)

```
const char *file = "/bin/sh";

execlp(file, file, NULL);
```

```
call execlp
```

```
execlp:
    ...
```

RDI → file
RSI → file
RDX → NULL

Enter Gadgets

- Gadgets are small sequences of instructions

```
mov qword ptr [rax], rdx        pop rax
xor eax, eax                    ret
ret
```

- Gadgets end in an indirect control-flow instruction (`ret` , jmp ptr, call ptr)
- Attackers can chain gadgets to execute code
- Different gadgets can achieve the same functionality
  - Example: `add $8, $esp` vs `pop %edi; pop %esi`

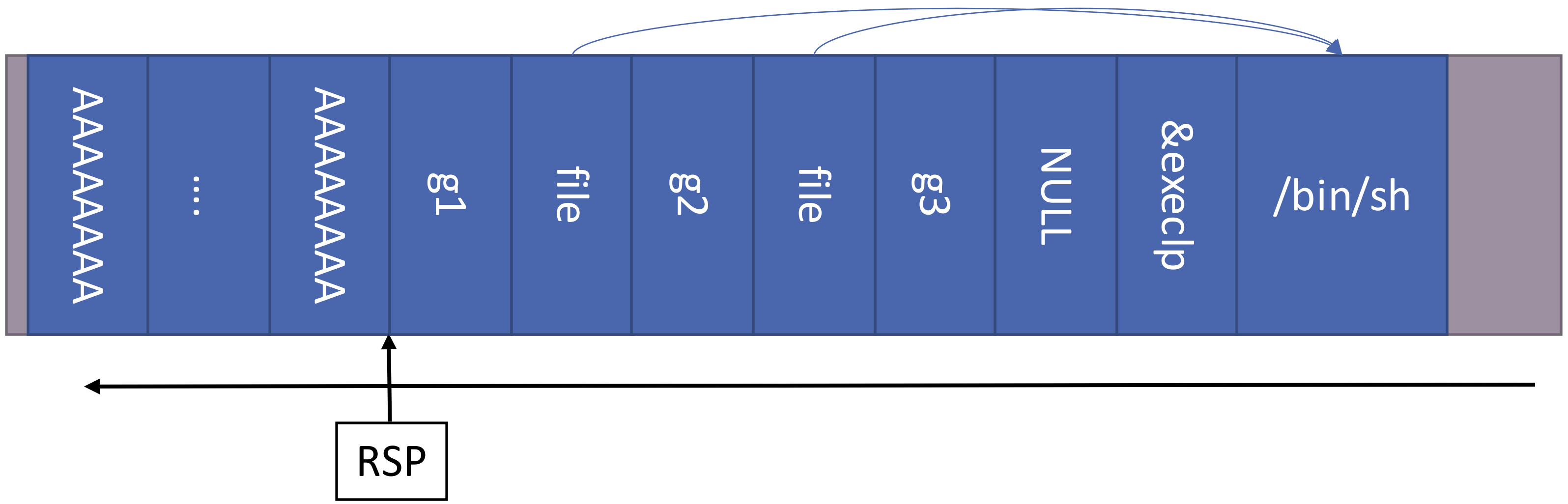
Preparing Arguments in the Stack (ret2libc on 64-bits)

```
const char *file = "/bin/sh";

execlp(file, file, NULL);
```

```
call execlp
```

```
execlp:
    ...

g1   : pop rdi
g1+1 : ret

g2   : pop rsi
g2+1 : ret

g3   : pop rdx
g2+1 : ret
```

RDI → file
RSI → file
RDX → NULL



Finding Gadgets in Code

```
mov (%rcx),%rbx
test %rbx,%rbx
je 41c523 <main+0x803>
mov %rbx,%rdi
callq 42ab00
mov %rax,0x2cda9d(%rip)
cmpb $0x2d,(%rbx)
je 41c4ac <main+0x78c>
mov 0x2cda8d(%rip),%rax
ret
test %rbx,%rbx
mov $0x4ab054,%eax
cmove %rax,%rbx
mov %rbx,0x2cda6a(%rip)
test %rdi,%rdi
je 41c0c2 <main+0x3a2>
mov $0x63b,%edx
mov $0x4ab01d,%esi
callq 46cab0 <sh_xfree>
ret
```

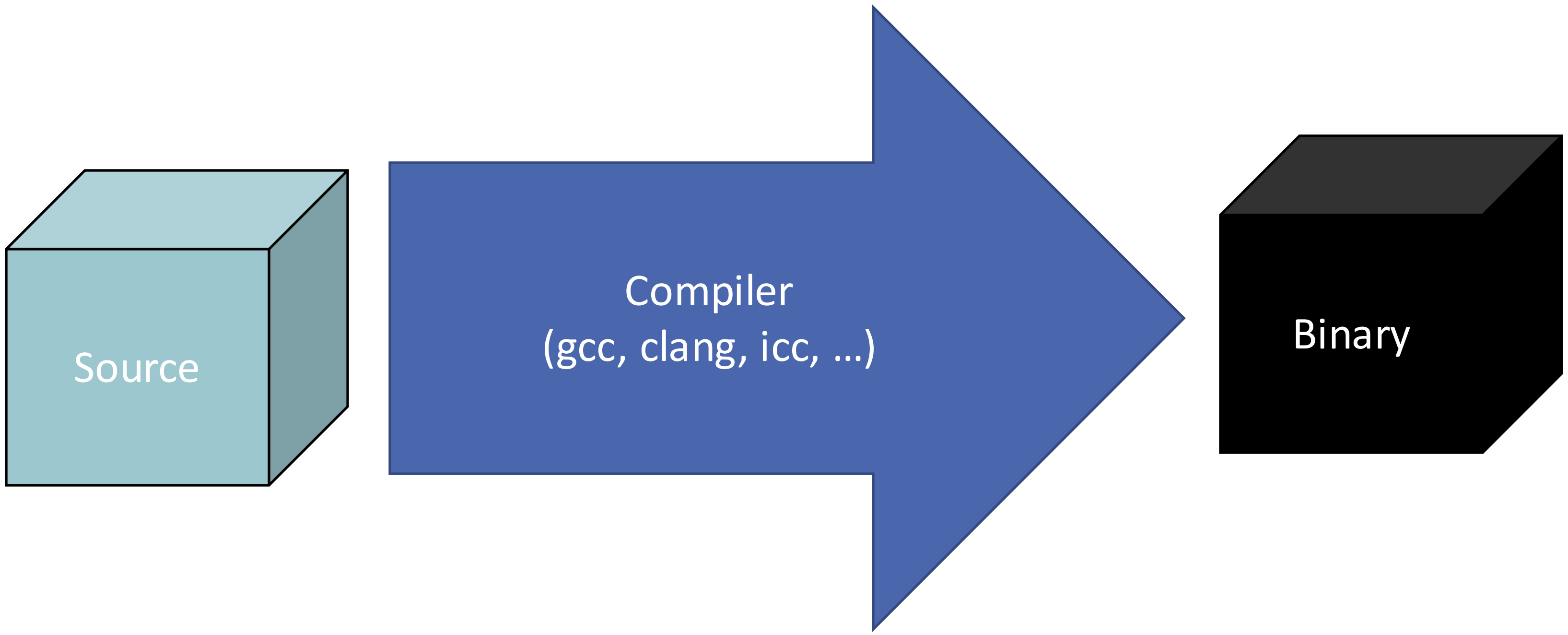
```
mov %rax,0x2d2945(%rip)
mov 0x2cda16(%rip),%rax
test %rax,%rax
je 41c112 <
movzbl (%ra
callq 41b64
mov 0xb8(%r
cmp 0xc(%rsp),%r15
mov %rax,0x2d2670(%rip)
je 41c214 <main+0x4f4>
xchg %ax,%ax
mov (%rsp),%rdx
movslq %r15d,%rax
mov (%rdx,%rax,8),%r14
ret
je 41c214 <main+0x4f4>
cmpb $0x2d,(%r14)
jne 41c214 <main+0x4f4>
movzbl 0x1(%r14),%r12d
movl $0x0,0x18(%rsp)
cmp $0x2d,%r12b
```

Gadgets

```
je 41c440 <main+0x720>
xor %ebp,%ebp
mov $0x4c223a,%ebx
add $0x1,%r14
jmp 41c1a3 <main+0x483>
cmp (%rbx),%r12b
mov %ebp,%r13d
jne 41c188 <main+0x468>
mov %rbx,%rsi
test %eax,%eax
xchg %ax,%ax
jne 41c188 <main+0x468>
movslq %ebp,%rax
ret
cmpl $0x1,0x4ab3c8(%rax)
je 41c461 <main+0x741>
mov (%rsp),%rcx
add $0x1,%r15d
movslq %r15d,%rdx
mov (%rcx,%rdx,8),%rdx
test %rdx,%rdx
je 41cefd <main+0x11dd>
```

Beyond Intended Instructions

- We call Instructions emitted by the compiler *intended instructions*



Beyond Intended Instructions

- Example of intended instructions

Compiler
(gcc, clang, icc, …)

```
;-- main:
 0x080489a3      8d4c2404       lea ecx, [esp + 4]
 0x080489a7      83e4f0         and esp, 0xfffffff0
 0x080489aa      ff71fc         push dword [ecx - 4]
 0x080489ad      55             push ebp
 0x080489ae      89e5           mov ebp, esp
 0x080489b0      56             push esi
 0x080489b1      53             push ebx
 0x080489b2      51             push ecx
 0x080489b3      81ec0c020000   sub esp, 0x20c
```

Beyond Intended Instructions

- Example of intended instructions
- Instructions have different sizes
  - Can start at any byte (no alignment requirements by CPU)

Compiler
(gcc, clang, icc, …)

```
;-- main:
 0x080489a3    8d4c2404      lea ecx, [esp + 4]
 0x080489a7    83e4f0        and esp, 0xfffffff0
 0x080489aa    ff71fc        push dword [ecx - 4]
 0x080489ad    55            push ebp
 0x080489ae    89e5          mov ebp, esp
 0x080489b0    56            push esi
 0x080489b1    53            push ebx
 0x080489b2    51            push ecx
 0x080489b3    81ec0c020000  sub esp, 0x20c
```

Unintended Instructions

- *Unintended instructions* are formed by bytes within or between intended instruction

```
;-- main:
0x080489a3      8d4c2404        lea ecx, [esp + 4]
        0x080489a4      4c              dec esp
        0x080489a5      2404            and al, 4
0x080489a7      83e4f0          and esp, 0xfffffff0
0x080489aa      ff71fc          push dword [ecx - 4]
0x080489ad      55              push ebp
0x080489ae      89e5            mov ebp, esp
0x080489b0      56              push esi
0x080489b1      53              push ebx
0x080489b2      51              push ecx
0x080489b3      81ec0c020000    sub esp, 0x20c
```

Unintended Instructions

- *Unintended instructions* are formed by bytes within or between intended instruction

```
;-- main:
0x080489a3      8d4c2404        lea ecx, [esp + 4]
        0x080489a4      4c              dec esp
        0x080489a5      2404            and al, 4
0x080489a7      83e4f0          and esp, 0xfffffff0
0x080489aa      ff71fc          push dword [ecx - 4]
        0x080489ab      71fc            jno 0x80489a9
0x080489ad      55              push ebp
0x080489ae      89e5            mov ebp, esp
0x080489b0      56              push esi
0x080489b1      53              push ebx
0x080489b2      51              push ecx
0x080489b3      81ec0c020000    sub esp, 0x20c
```

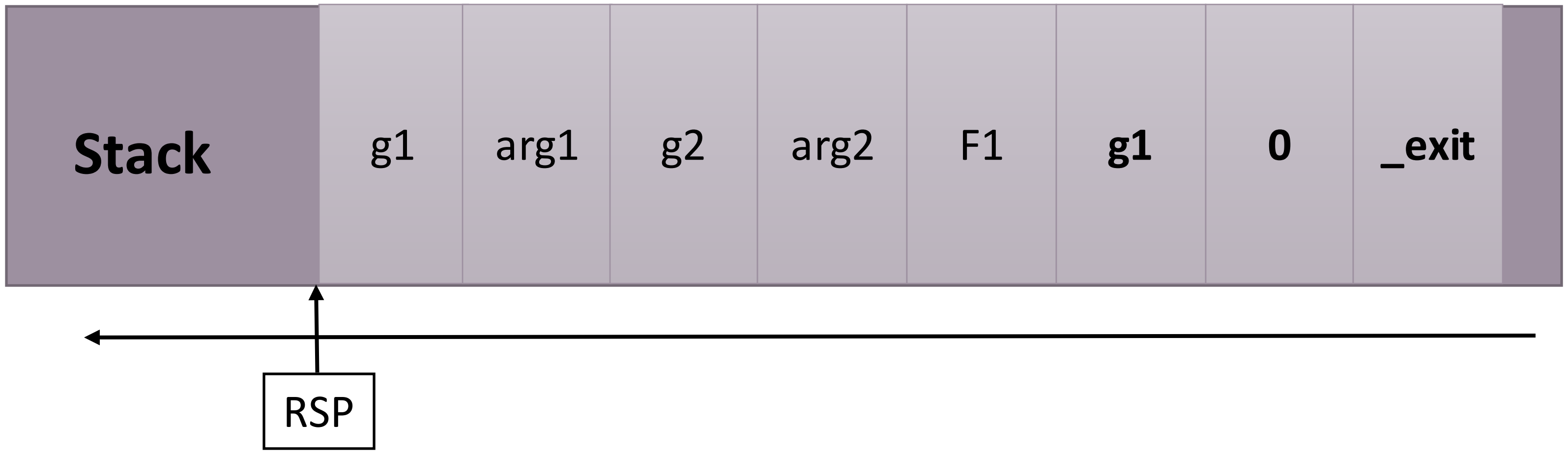
Unintended Instructions

- *Unintended instructions* are formed by bytes within or between intended instruction

```
;-- main:
 0x080489a3      8d4c2404        lea ecx, [esp + 4]
                     0x080489a4      4c              dec esp
                     0x080489a5      2404            and al, 4
 0x080489a7      83e4f0          and esp, 0xfffffff0
 0x080489aa      ff71fc          push dword [ecx - 4]
                     0x080489ab      71fc            jno 0x80489a9
 0x080489ad      55              push ebp
 0x080489ae      89e5            mov ebp, esp
                     0x080489af      e556            in eax, 0x56
 0x080489b0      56              push esi
 0x080489b1      53              push ebx
 0x080489b2      51              push ecx
 0x080489b3      81ec0c020000    sub esp, 0x20c
```

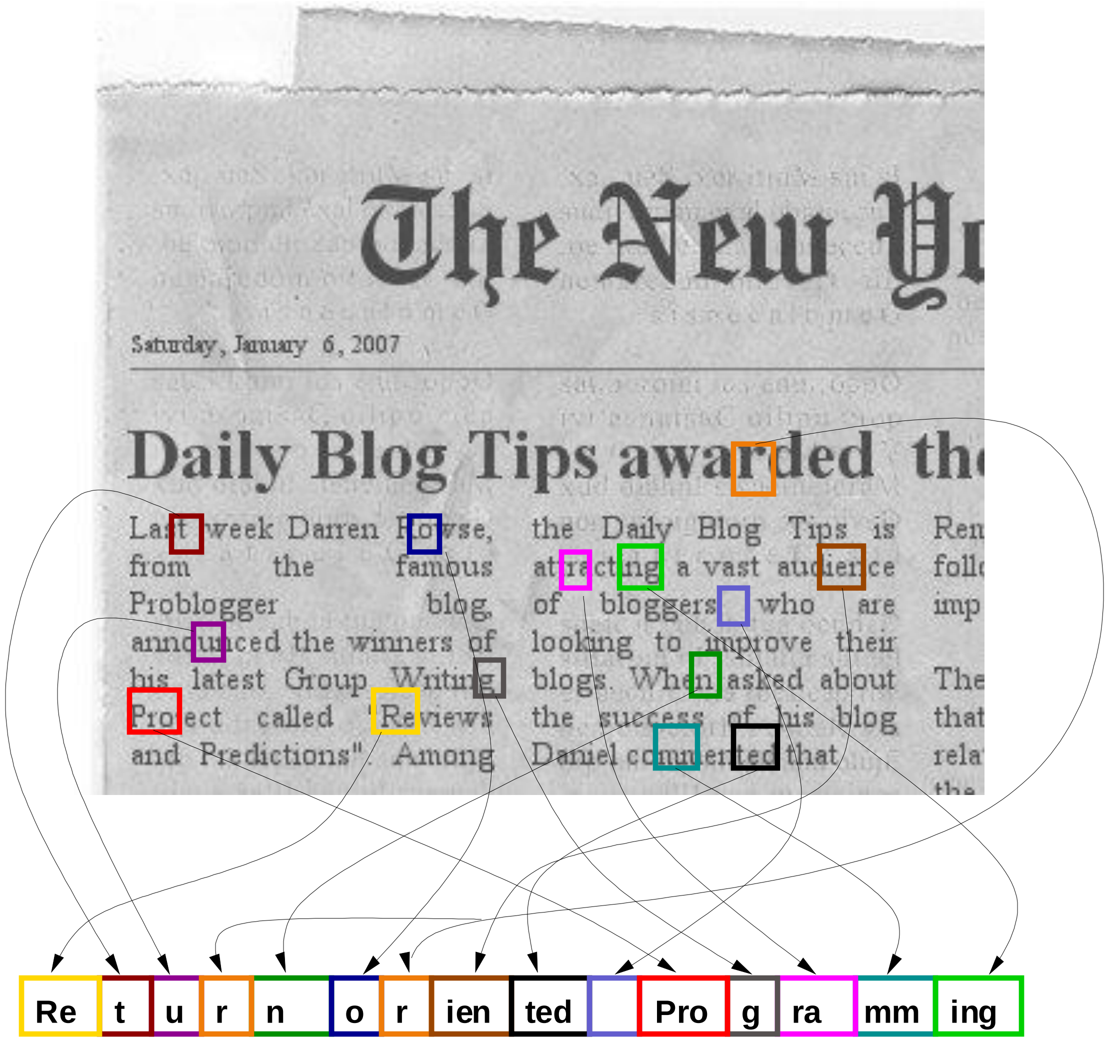
Function Chaining on 64-bit

```
F1(arg1, arg2):
    ...
g1   : pop rdi
g1+1 : ret

g2   : pop rsi
g2+1 : ret


exit(0):
    ...
```



Return-Oriented Programming

- Creating a whole new program by chaining gadgets ending in a ret
  - Weird machines
- Use the stack like a tape providing the data for the computation and the instruction pointer
- Provides a Turing complete machine
  - Read/write memory
  - Conditional branching
  - ALU operations
  - Perform system calls



https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini

An Example: Add 2 Number and Store the Result

Payload

0x00	...
esp	0xb8800000
	0x00000001
	0xb8800010
	0x00000002
	0xb8800020
	0xb8800010
	0x00400000
	0xb8800030
	...
0xff	

Code

```
0xb8800000:
  pop eax
  ret
  ...
0xb8800010:
  pop ebx
  ret
  ...
0xb8800020:
  add ebx, eax
  ret
  ...
0xb8800030:
  mov eax, [ebx]
  ret
```

Actions

```
 eax = 1
 ebx = 2
 eax += ebx
 ebx = 0x400000
*ebx = eax
```

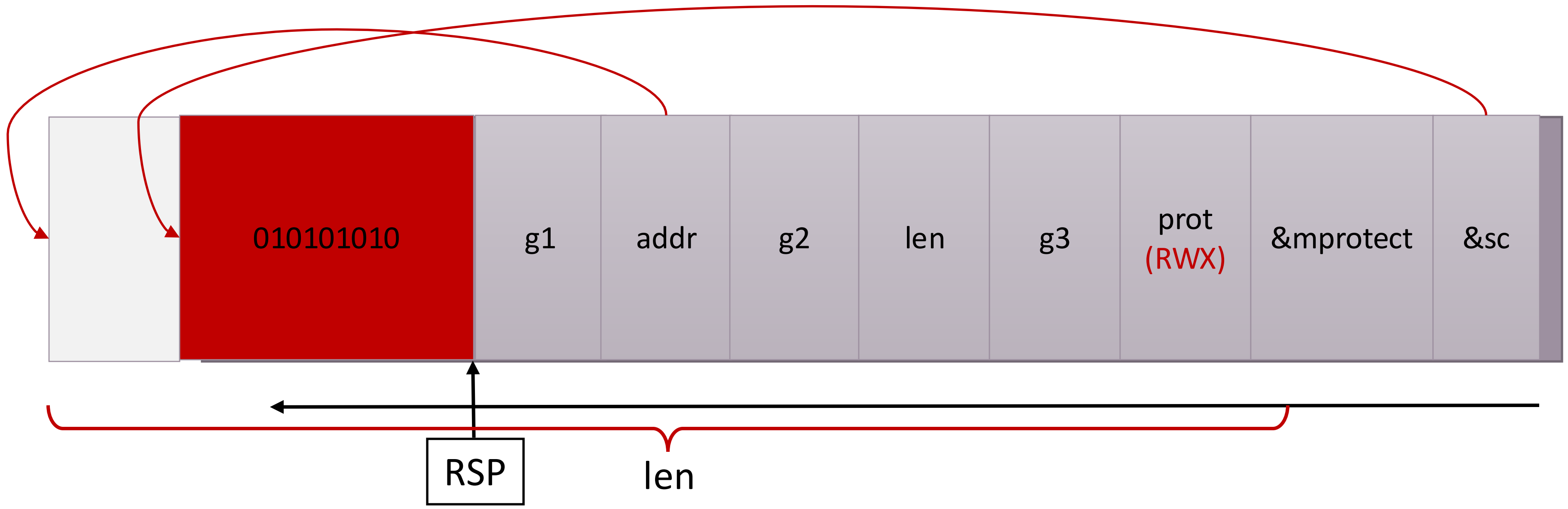
Multi-stage Exploits

- ROP is complicated
- It is easier to create a first-stage ROP payload for bypassing NX
  - Allocate W+X memory, copy shellcode (2nd stage payload) within, and execute
  - Make memory area containing shellcode (2nd stage payload) executable and execute
- However, there are also pure-ROP exploits
  - In-the-wild exploit against Adobe Reader XI
  - CVE-2013-0640

Multi-stage Exploits

- Call `mprotect(void *addr, size_t len, int prot)` on shellcode to turn it eXecutable
  - Or allocate executable memory and copy shellcode
- Execute shellcode

```
g1    : pop rdi
g1+1  : ret

g2    : pop rsi
g2+1  : ret

g3    : pop rdx
g2+1  : ret
```



Appendix: Code-Reuse Attacks on 32-bit Programs

Preparing Arguments in the Stack (ret2libc on 32-bits)

```
const char *file = "/bin/sh";

execlp(file, file, NULL);
```

```
call execlp
```

```
execlp:
    ...
```

Preparing Arguments in the Stack (ret2libc on 32-bits)

```
const char *file = "/bin/sh";

execlp(file, file, NULL);
```

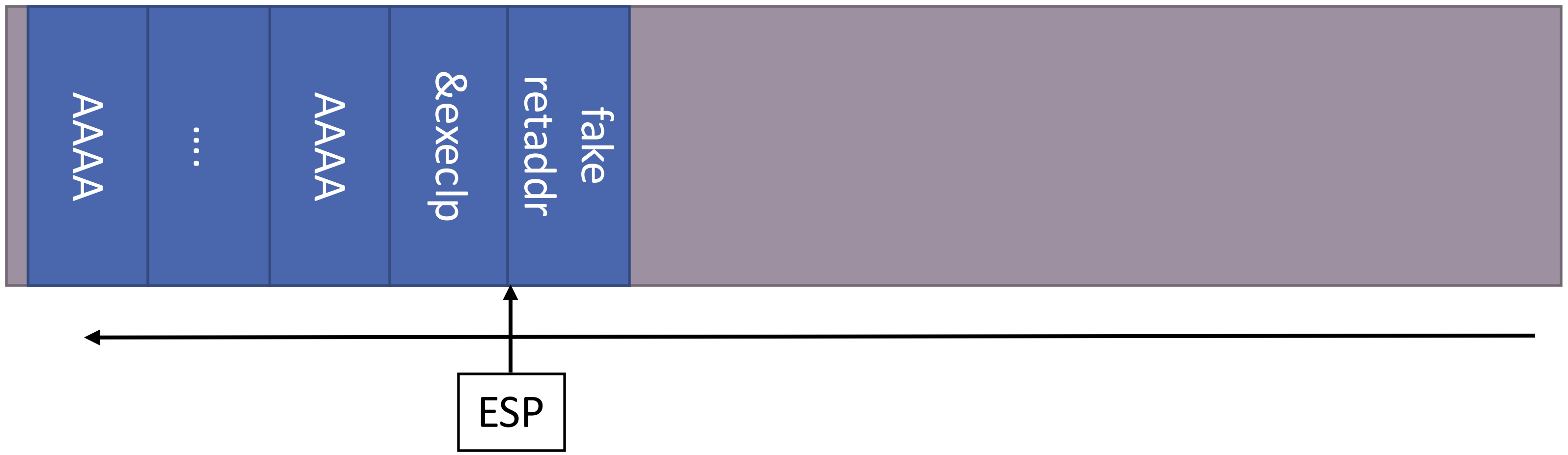
```
call execlp          execlp:
                         ...
```

Preparing Arguments in the Stack (ret2libc on 32-bits)

```
const char *file = "/bin/sh";

execlp(file, file, NULL);
```

```
call execlp
```

```
execlp:
    ...
```

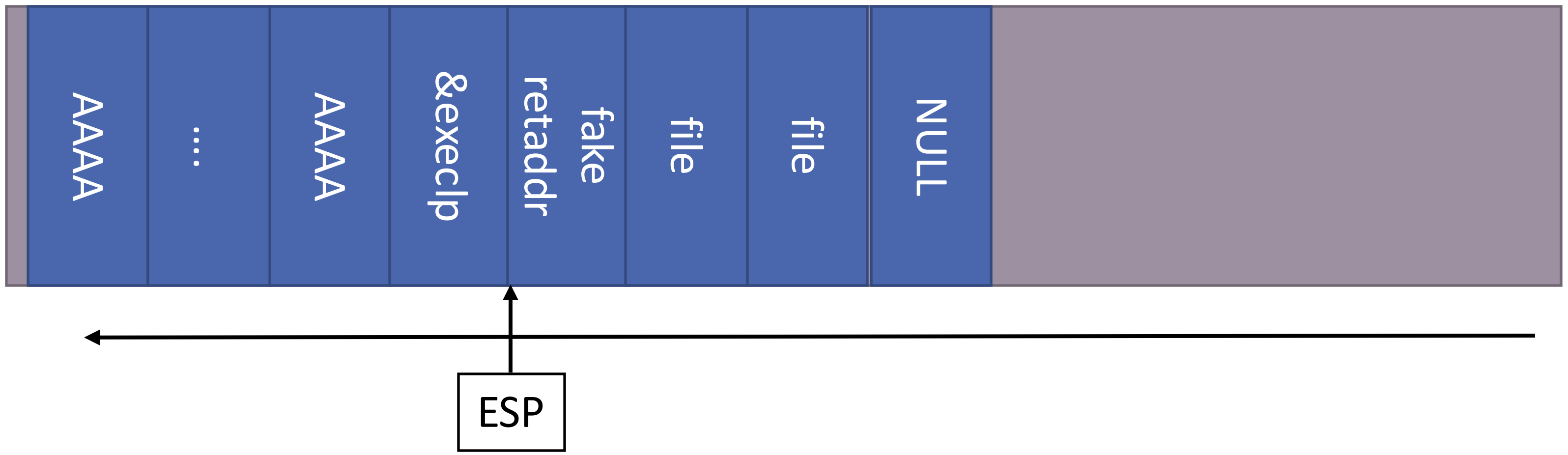


Preparing Arguments in the Stack (ret2libc on 32-bits)

```
const char *file = "/bin/sh";

execlp(file, file, NULL);
```

```
call execlp
```

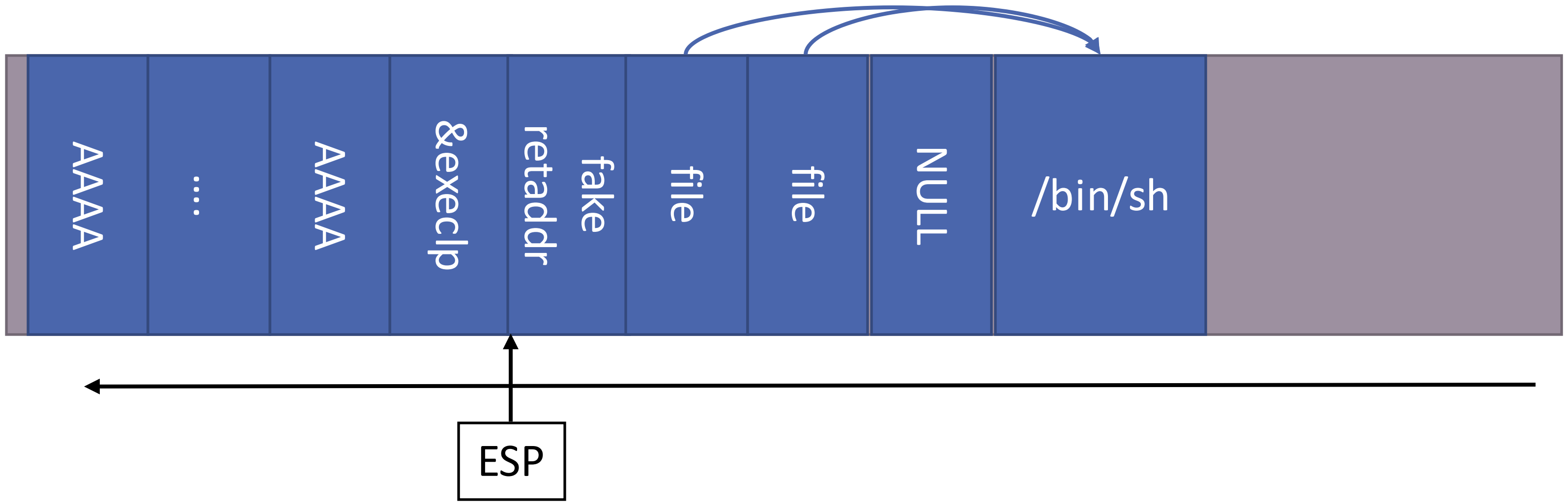
```
execlp:
    ...
```



Preparing Arguments in the Stack (ret2libc on 32-bits)

```
const char *file = "/bin/sh";

execlp(file, file, NULL);
```

```
call execlp        ➡  execlp:
                          ...
```



Chaining Functions

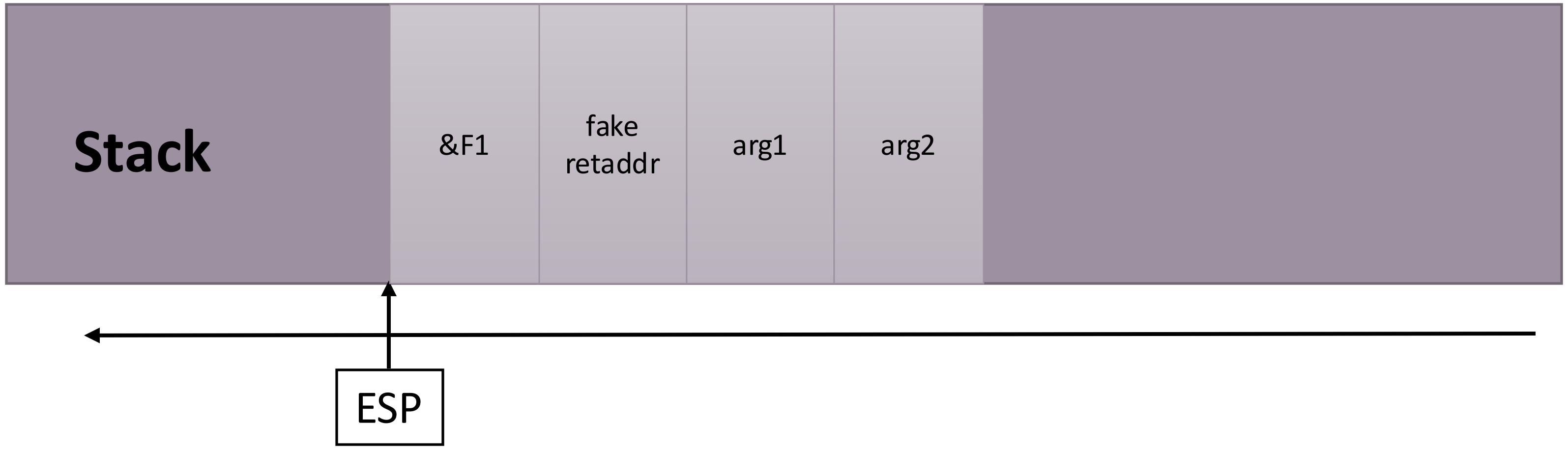
F2(..)

F1(..)

Chaining Functions in 32-bit

- How to call a 2nd function after the initial ret2libc?

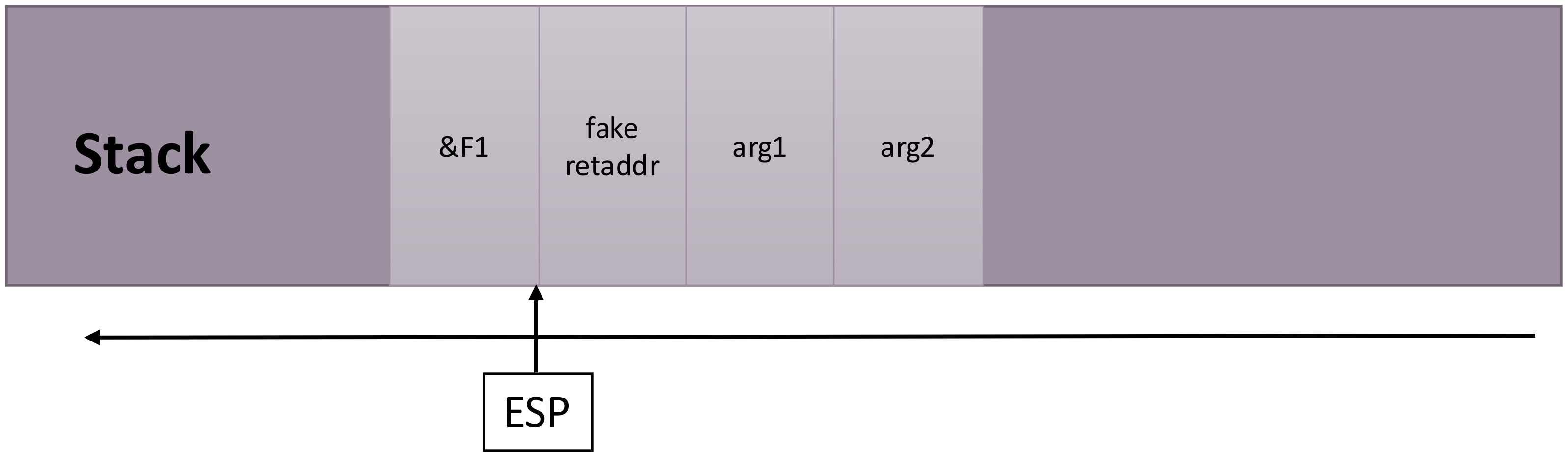
```
F1(arg1, arg2):
    ...


exit(arg3):
    ...
```



Chaining Functions in 32-bit

- How to call a 2nd function after the initial ret2libc?

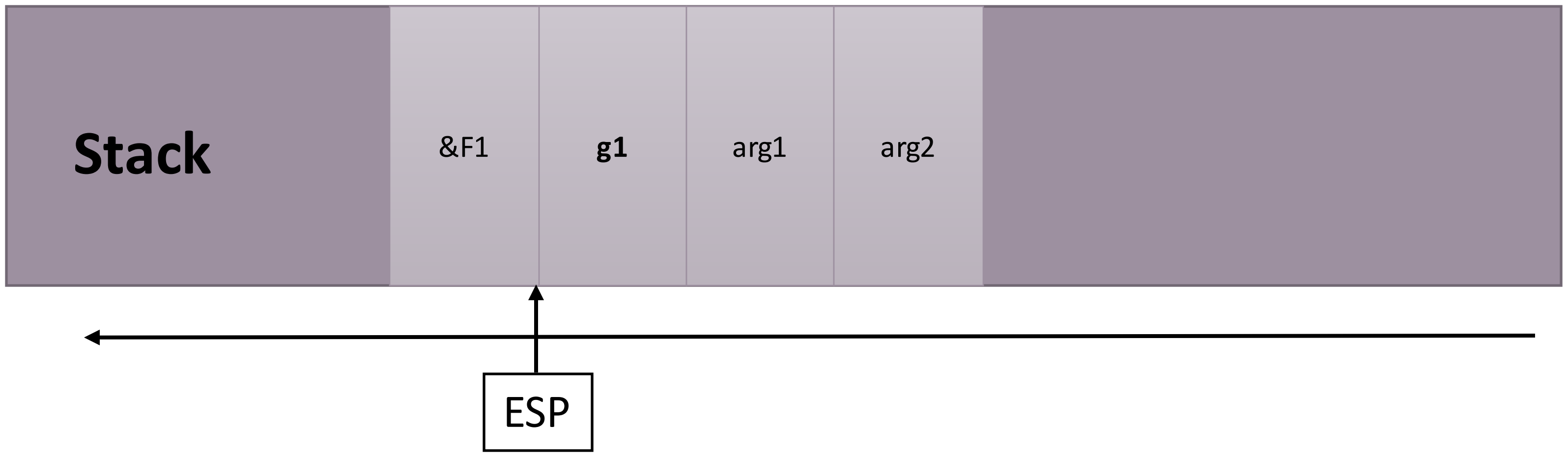
```
F1(arg1, arg2):
    ...


exit(arg3):
    ...
```



Chaining Functions in 32-bit

- How to call a 2nd function after the initial ret2libc?
- Use gadgets to move SP

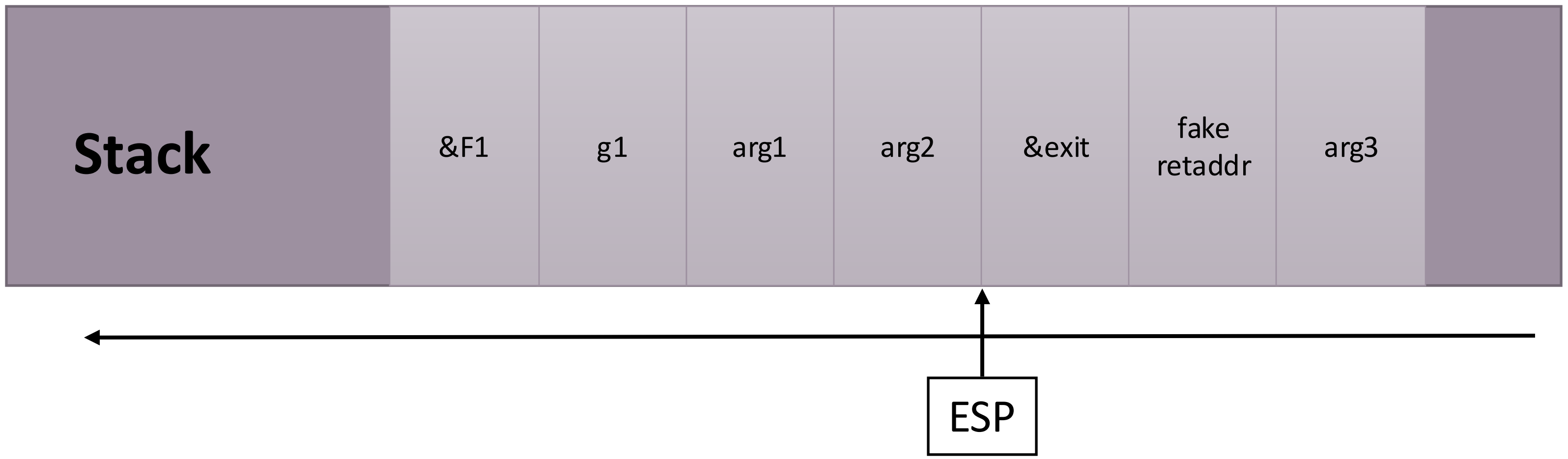
```
F1(arg1, arg2):
     ...

g1:  pop edi ; pop ebp ; ret

exit(arg3):
     ...
```



Chaining Functions in 32-bit

- How to call a 2nd function after the initial ret2libc?
- Use gadgets to move SP

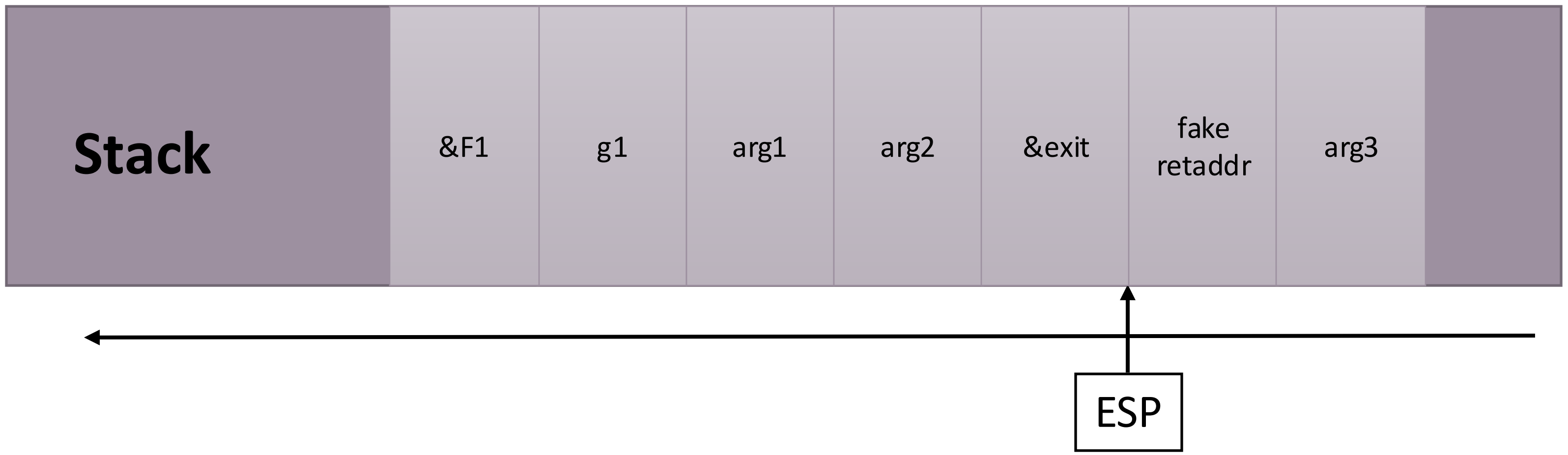
```
F1(arg1, arg2):
    ...

g1:  pop edi ; pop ebp ; ret

exit(arg3):
    ...
```



Chaining Functions in 32-bit

- How to call a 2nd function after the initial ret2libc?
- First, use gadgets to move SP
- Add another series of fake retaddr and arguments in the stack

```
F1(arg1, arg2):
    ...

g1:  pop edi ; pop ebp ; ret

exit(arg3):
    ...
```

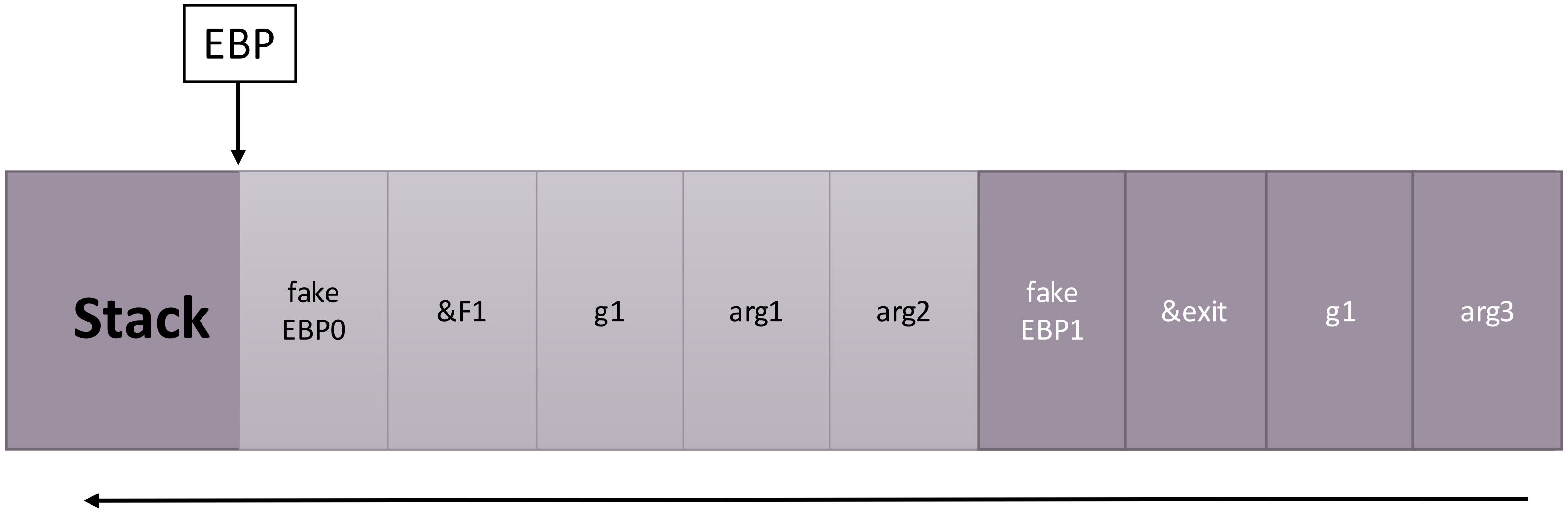


Chaining Functions in 32-bit with Frame Pointers

- Functions have a leave gadget before ret

```
leave //mov ebp, esp; pop ebp;
ret   //return
```

```
F1(arg1, arg2):
    ...

g1:   leave ; ret

exit(arg3):
    ...
```

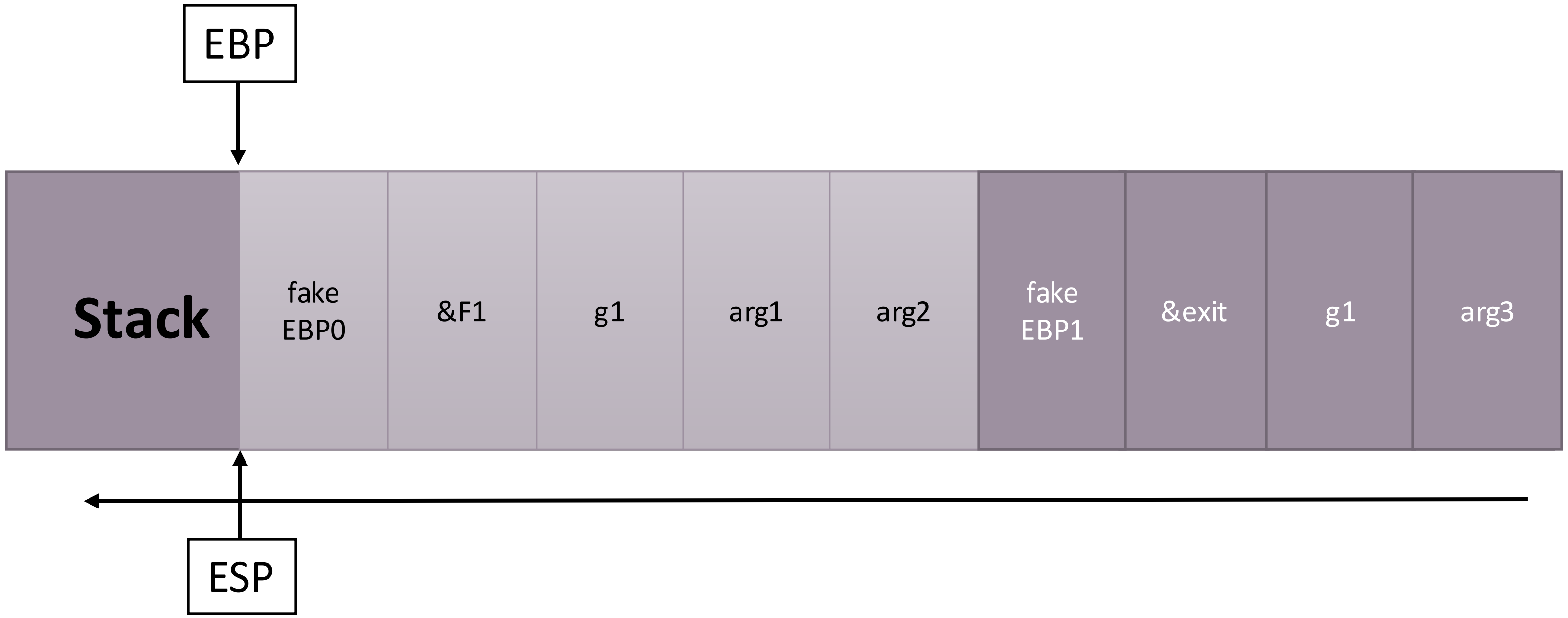


Chaining Functions in 32-bit with Frame Pointers

- Functions have a leave gadget before ret

```
leave //mov ebp, esp; pop ebp;
ret   //return
```

```
F1(arg1, arg2):
    ...

g1:  leave ; ret

exit(arg3):
    ...
```

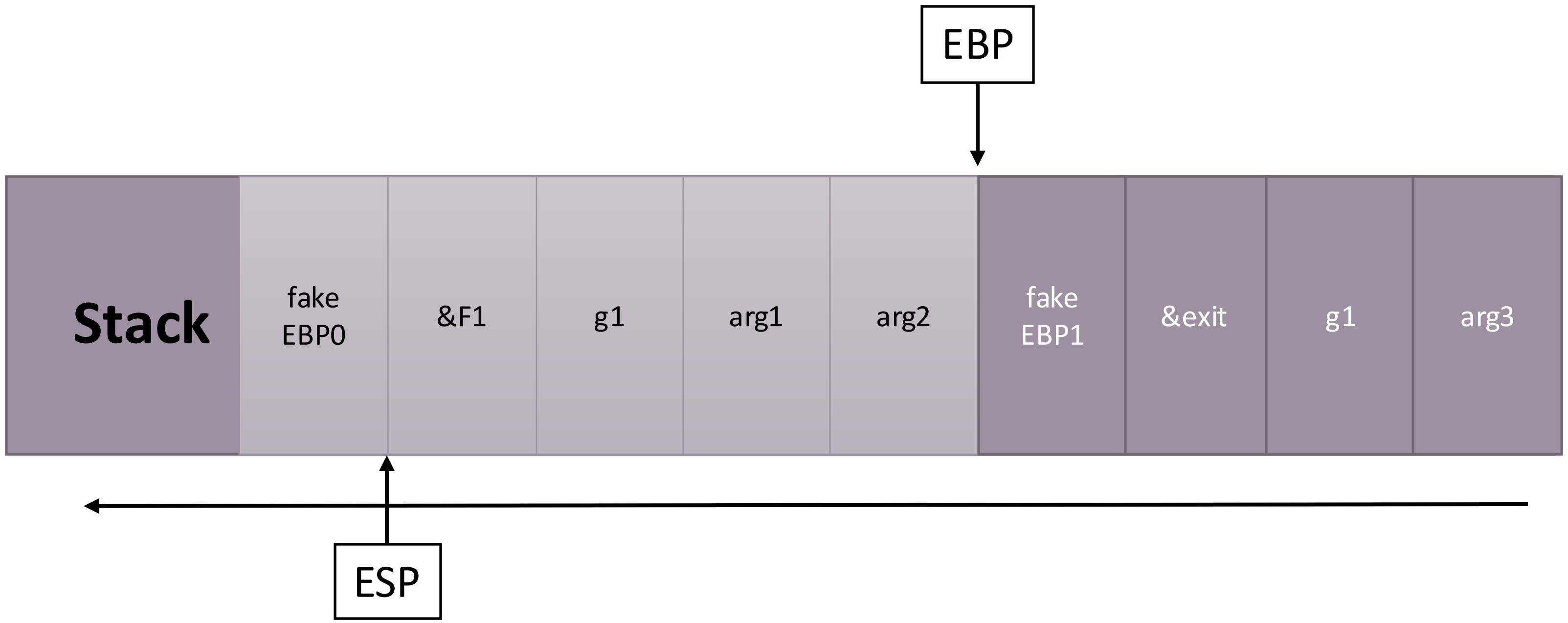


Chaining Functions in 32-bit with Frame Pointers

- Functions have a leave gadget before ret

```
leave //mov ebp, esp; pop ebp;
ret   //return
```

```
F1(arg1, arg2):
    ...

g1:  leave ; ret

exit(arg3):
    ...
```

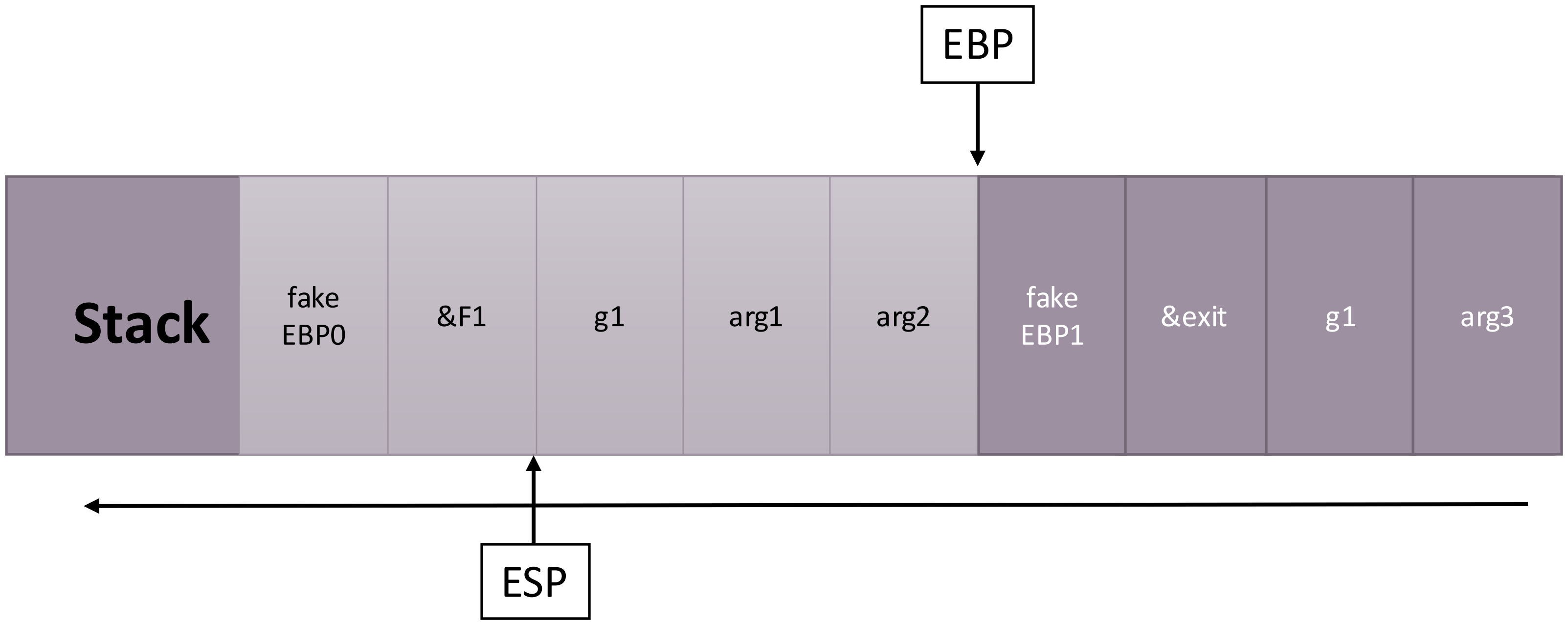


Chaining Functions in 32-bit with Frame Pointers

- Functions have a leave gadget before ret

```
leave //mov ebp, esp; pop ebp;
ret   //return
```

```
F1(arg1, arg2):
    ...

g1:  leave ; ret

exit(arg3):
    ...
```

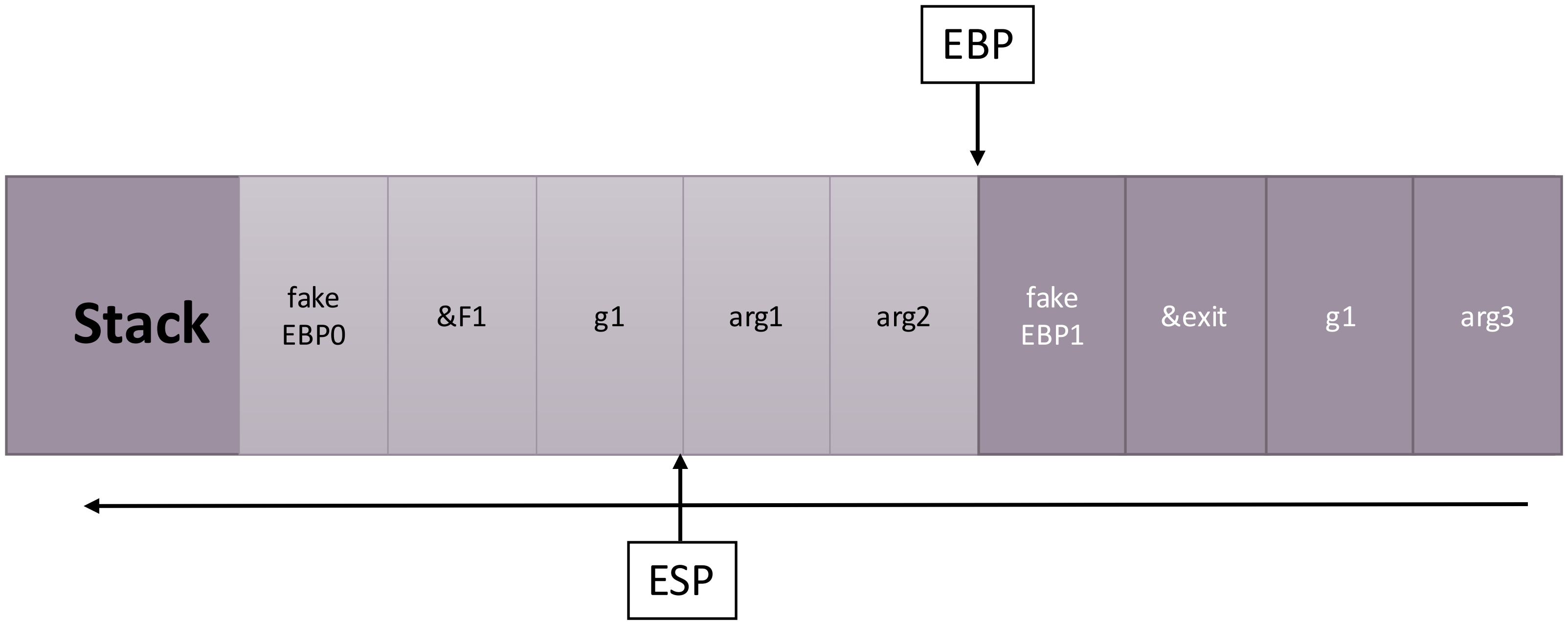


Chaining Functions in 32-bit with Frame Pointers

- Functions have a leave gadget before ret

```
leave //mov ebp, esp; pop ebp;
ret   //return
```

```
F1(arg1, arg2):
    ...

g1:  leave ; ret

exit(arg3):
    ...
```

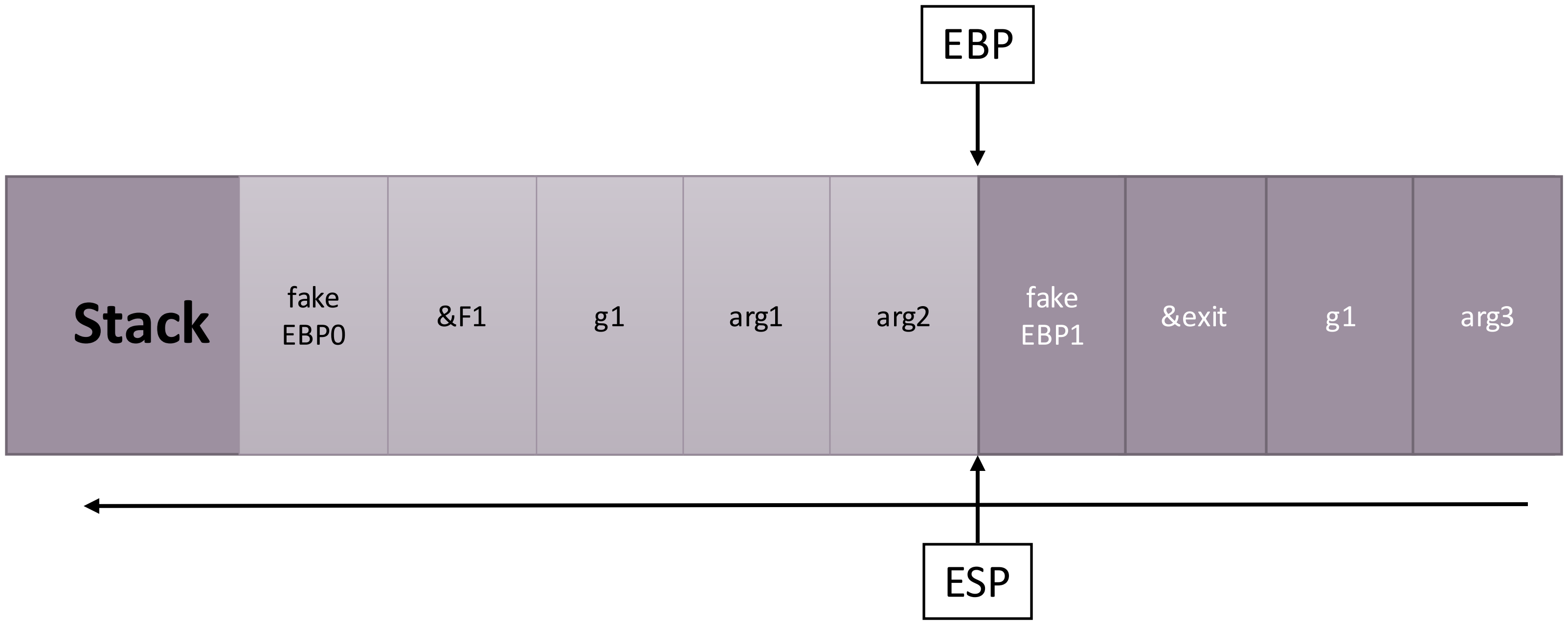


Chaining Functions in 32-bit with Frame Pointers

- Functions have a leave gadget before ret

```
leave //mov ebp, esp; pop ebp;
ret   //return
```

```
F1(arg1, arg2):
    ...

g1:  leave ; ret

exit(arg3):
    ...
```

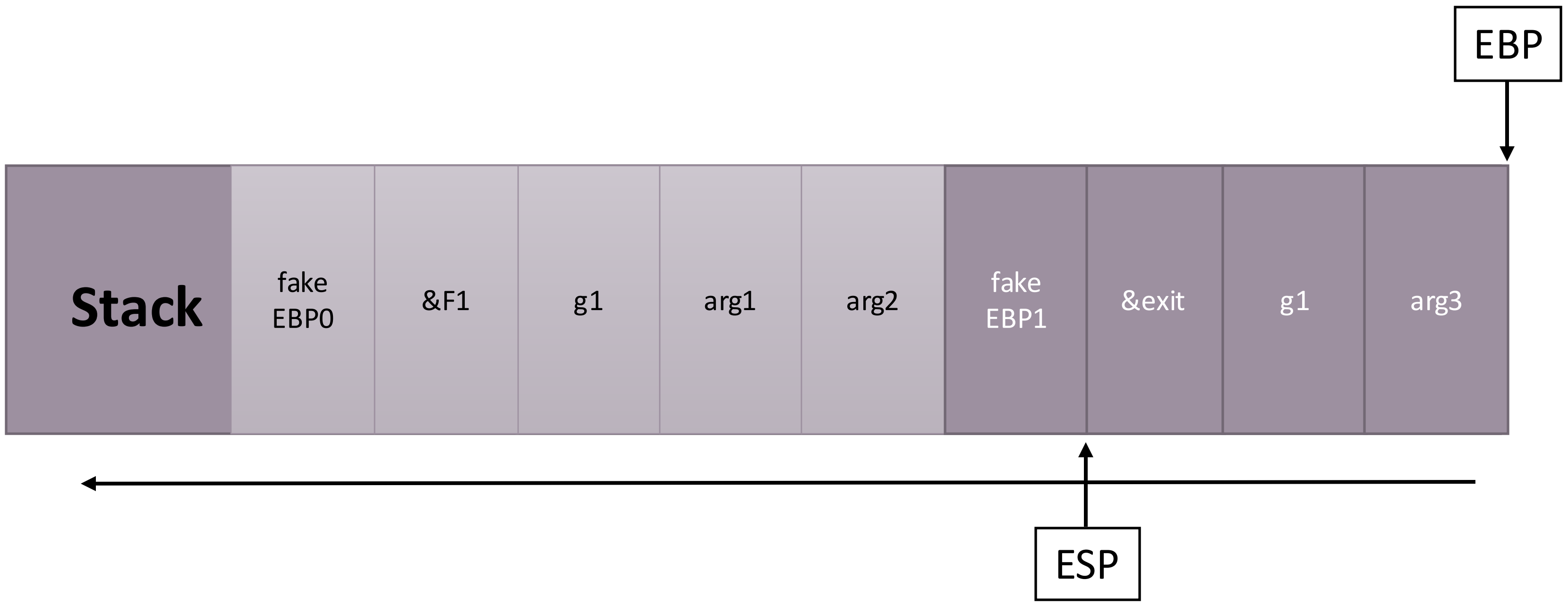


Chaining Functions in 32-bit with Frame Pointers

- Functions have a leave gadget before ret

```
leave //mov ebp, esp; pop ebp;
ret   //return
```

```
F1(arg1, arg2):
    ...

g1:   leave ; ret

exit(arg3):
    ...
```

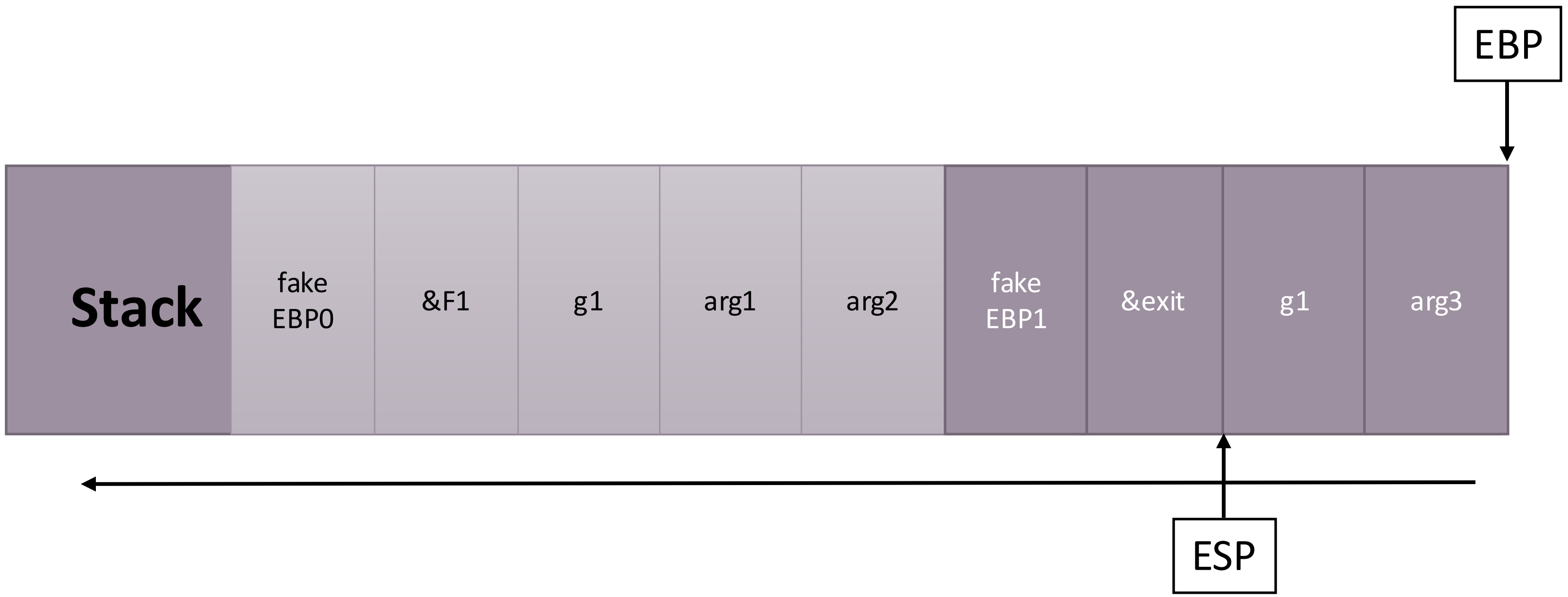


Chaining Functions in 32-bit with Frame Pointers

- Functions have a leave gadget before ret

```
leave //mov ebp, esp; pop ebp;
ret   //return
```

```
F1(arg1, arg2):
    ...

g1:   leave ; ret

exit(arg3):
    ...
```

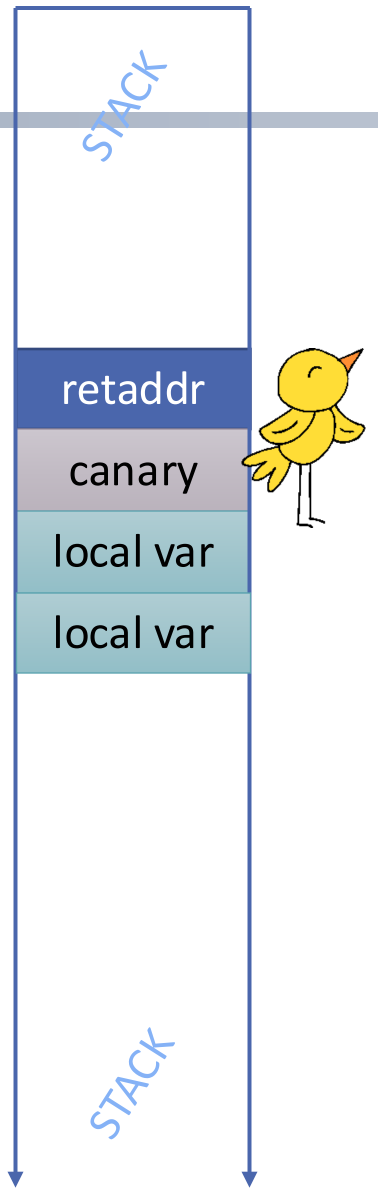


Software Exploitation
Stack Defenses

Georgios (George) Portokalidis

StackGuard

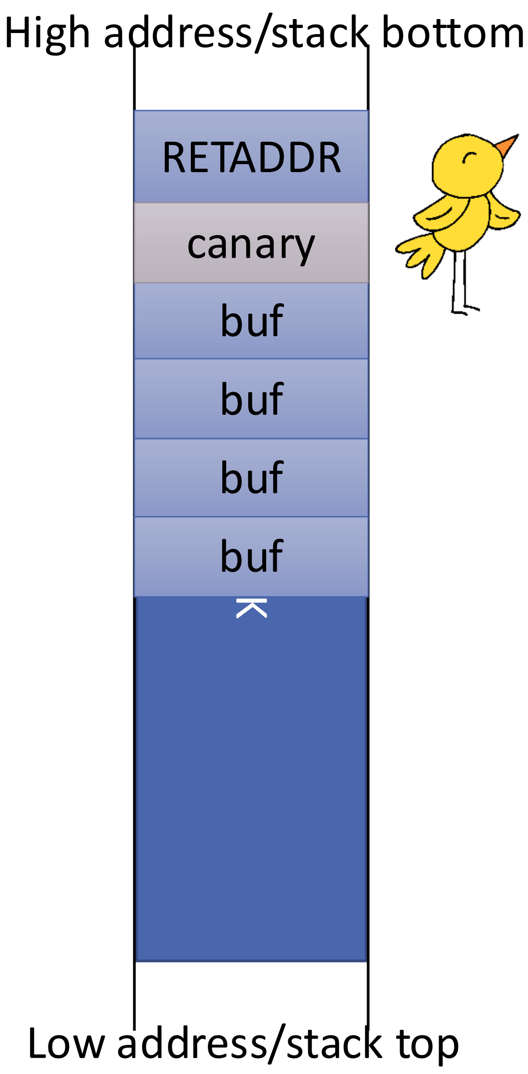
- Insert special values, called canaries, between local variables and function return address
- Canary values are inserted on function entry
- Canaries are verified before a function returns
  - Program stops if the canary has changed



Stack Overflow With Canary

```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return 0;
}
```

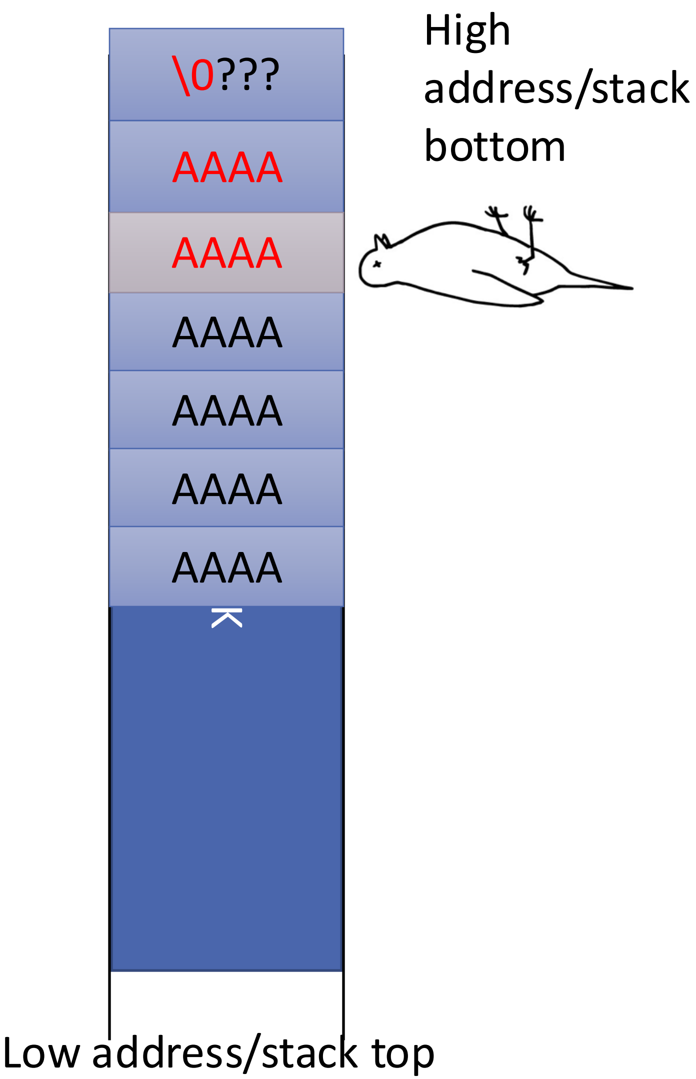
./mytest AAAAA



Stack Overflow with Canary

```c
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return 0;
}
```

```
./mytest AAAAAAAAAAAAAAAAAAAAAAAA
```



Canary Types

- Random canary: (used in Visual Studio, gcc, etc.)
  - Choose random bytes at program startup
  - Insert canary bytes into every stack frame
  - Verify canary before returning from function

- Terminator canary:

  Canary = 0 (null), newline, linefeed, EOF
  - String functions will not copy beyond terminator
  - Hence, attacker cannot use string functions to corrupt stack

Example: C code

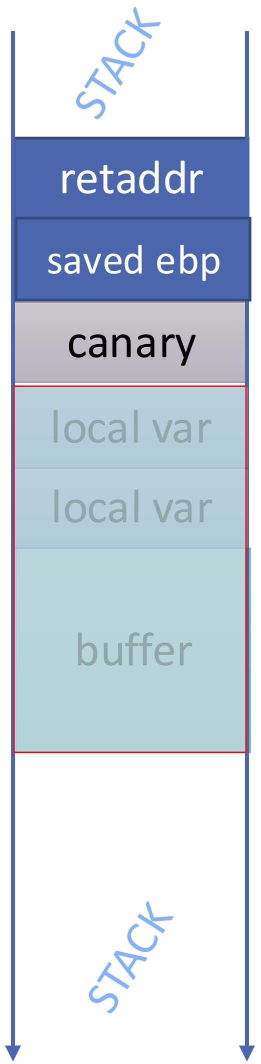
```
void do_echo(const char *str)
{
        char echo[128] = "echo: ";
        int i;

        for (i = 6; *str != '\n'; i++) {
                echo[i] = *str++;
        }
        echo[i] = '\0';

        puts(echo);
}
```

Example: Compiled Code

```
00000000004006aa <do_echo>:
  4006aa:       48 81 ec 98 00 00 00    sub    $0x98,%rsp
  4006b1:       48 89 fe                mov    %rdi,%rsi
  4006b4:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  4006bb:       00 00
  4006bd:       48 89 84 24 88 00 00    mov    %rax,0x88(%rsp)     Store canary
  ...
  400725:       48 8b 84 24 88 00 00    mov    0x88(%rsp),%rax
  40072c:       00
  40072d:       64 48 33 04 25 28 00    xor    %fs:0x28,%rax       Verify canary
  400734:       00 00
  400736:       75 0f                   jne    400747 <do_echo+0x9d>
  400738:       48 81 c4 98 00 00 00    add    $0x98,%rsp
  40073f:       c3                      retq
  ...
  400747:       e8 c4 fd ff ff          callq  400510 <__stack_chk_fail@plt>
```

Alignment of Stack Buffers and Canaries

- The order of local variables may be important
- Buffer overflows could allow important local variables to be controlled



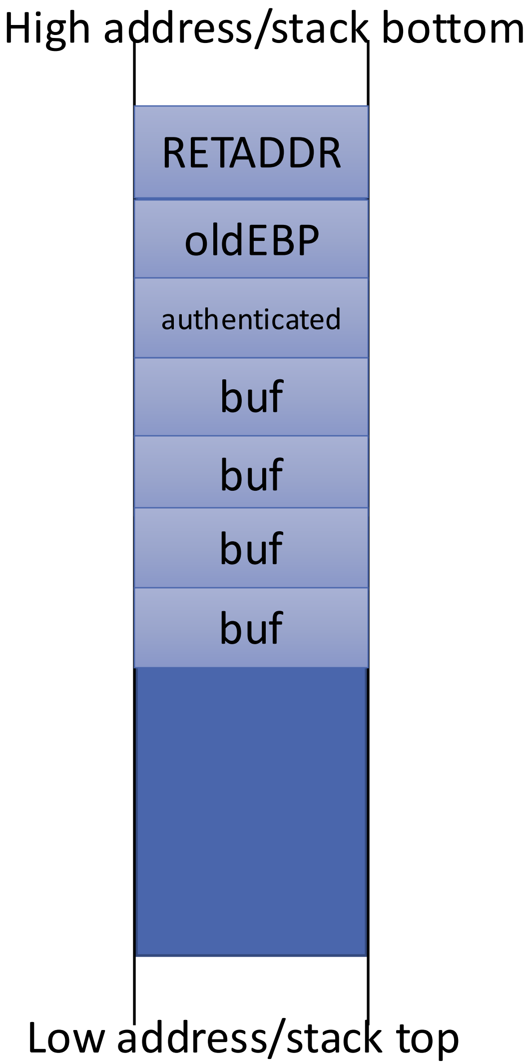
Non-Control Data Attacks

- Attacks overwriting data not directly used in control flow

- Essentially corrupting program state that affects its security
  - For example: Disabling/Bypassing a security mechanism

Example: Non-Control Data

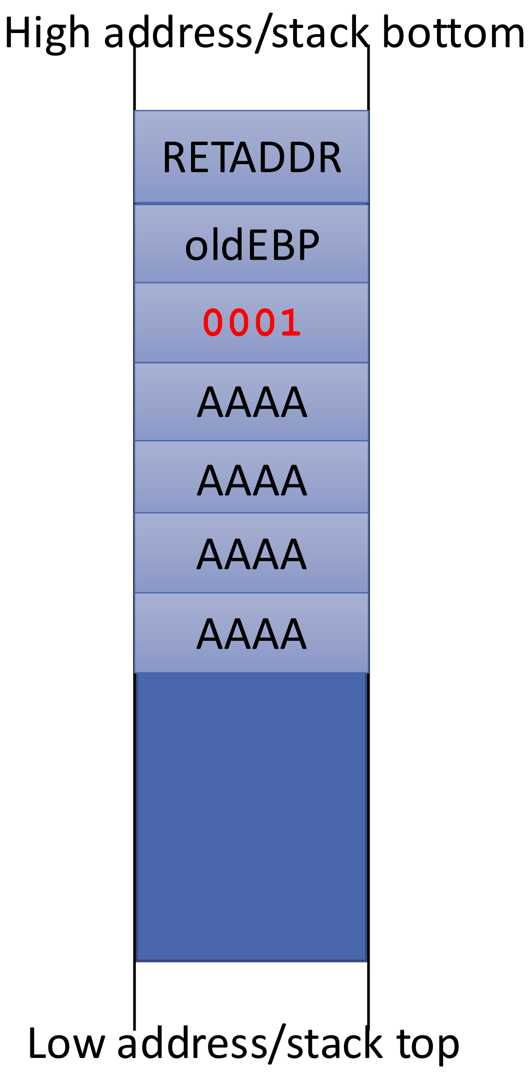
```
static int mytest(char *str)
{
      int authenticated = 0;
      char buf[16];

      read(STDIN_FILENO, buf, 32);
      if (check_pass(buf))
            authenticated = 1;

      do_something(authenticated);
}
```



Example: Non-Control Data

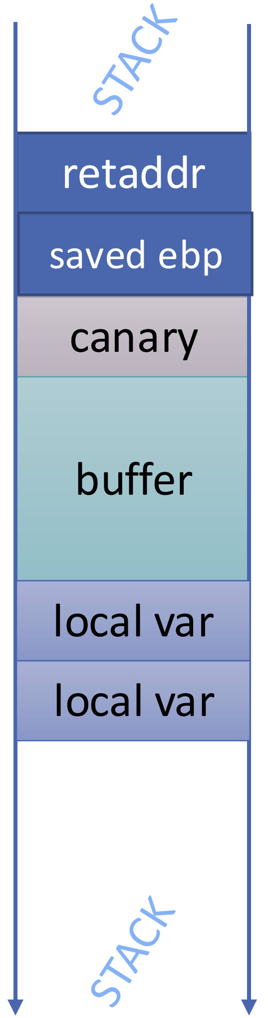
```
static int mytest(char *str)
{
      int authenticated = 0;
      char buf[16];

      read(STDIN_FILENO, buf, 32);
      if (check_pass(buf))
            authenticated = 1;

      do_something(authenticated);
}
```

./mytest AAAAAAAAAAAAAAAA\x01\x00\x00\x00



Buffers Allocated Adjacent to Canary

- Overflow will always corrupt the canary
- When multiple stack buffers exist, then one could still overflow into another
- Underflow also possible
  - Overwriting bytes before the beginning of the buffer



Problems

- Canaries can be omitted in small functions or non-string buffers
- Canaries/keys can be leaked
- Bugs may leave canaries untouched
  - Non-linear overflows or arbitrary write bugs

Fortified Source

- Defining _FORTIFY_SOURCE during compilation introduces buffer overflow checks for the following functions:
  - memcpy, mempcpy, memmove, memset, strcpy, stpcpy, strncpy, strcat, strncat, sprintf, vsprintf, snprintf, vsnprintf, gets
  - **If** the size of the destination buffer can be statically determined
- Requires optimization level >=1 (-O1)
- _FORTIFY_SOURCE == 1
  - checks that shouldn't change the behavior of conforming programs are performed → overflows will still abort execution
- _FORTIFY_SOURCE == 2
  - Some more checking is added, but some conforming programs might fail

Example: FORTIFY_SOURCE

```
char buf[128];
int i;

strcpy(buf, s1);
for (i = 1; i < n; i++) {
        strcat(buf, s1);
}
```

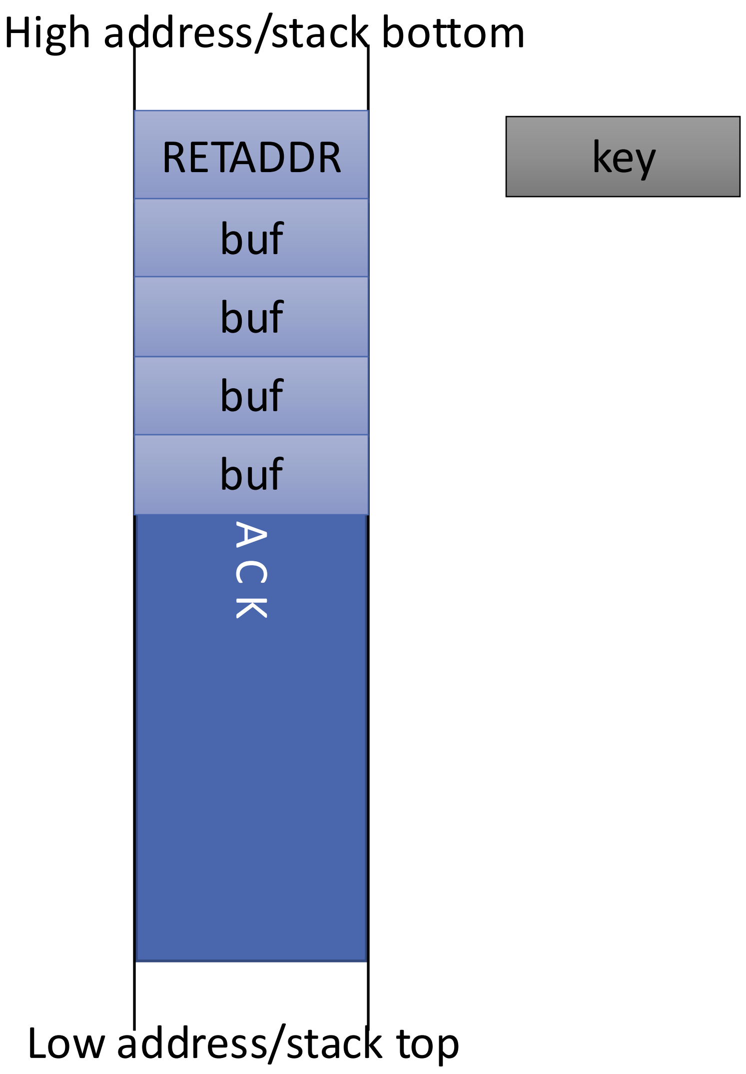
```
char buf[128];
int i;

__strcpy_chk(buf, s1, 128);
for (i = 1; i < n; i++) {
        __strcat_chk(buf, s1, 128);
}
```

```
mov     edx,0x80
mov     rsi,r12
mov     rdi,r13
call    44bd40 <__strcat_chk>
```

Appendix: Other Stack Defenses

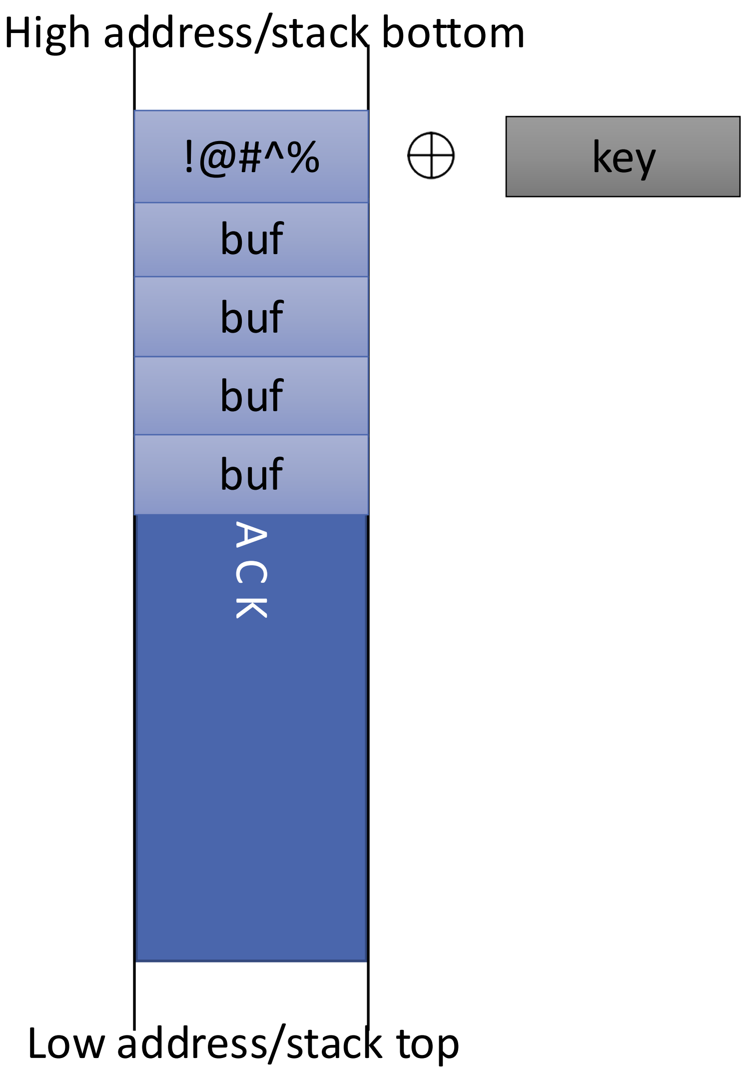
StackShield

- **Address obfuscation instead of canary**
- Encrypt return address on stack by XORing with random string
- Decrypt just before returning from function
- Attacker needs decryption key to set return address to desired value



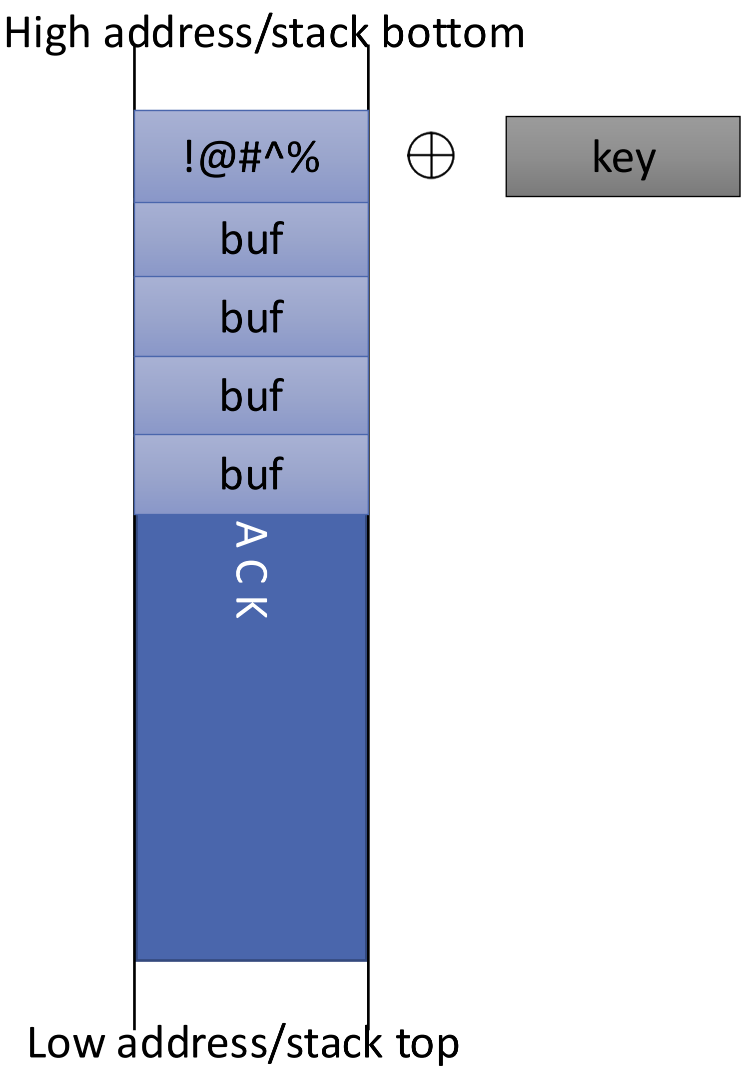
Example: StackShield

```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```

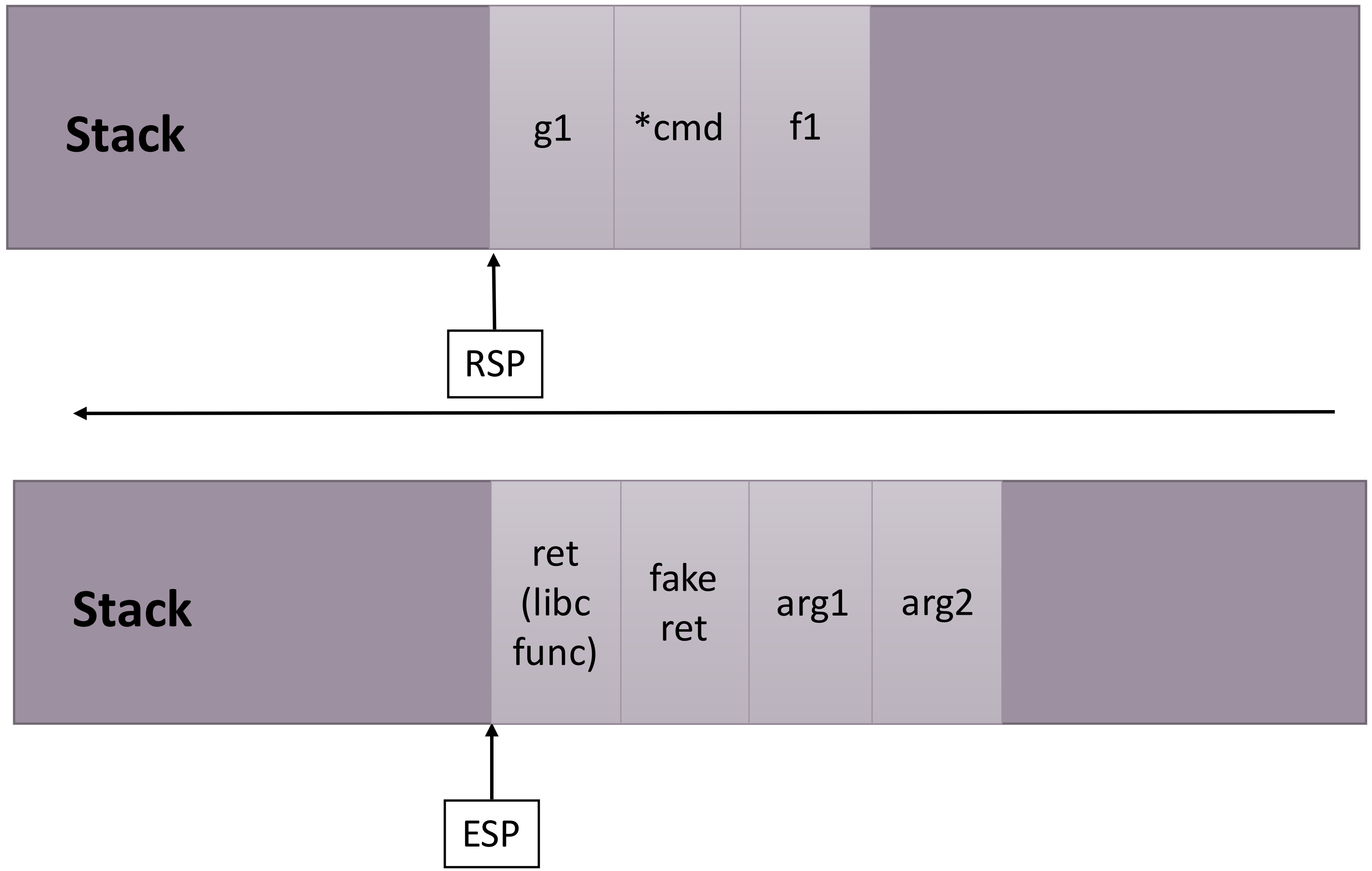


Example: StackShield

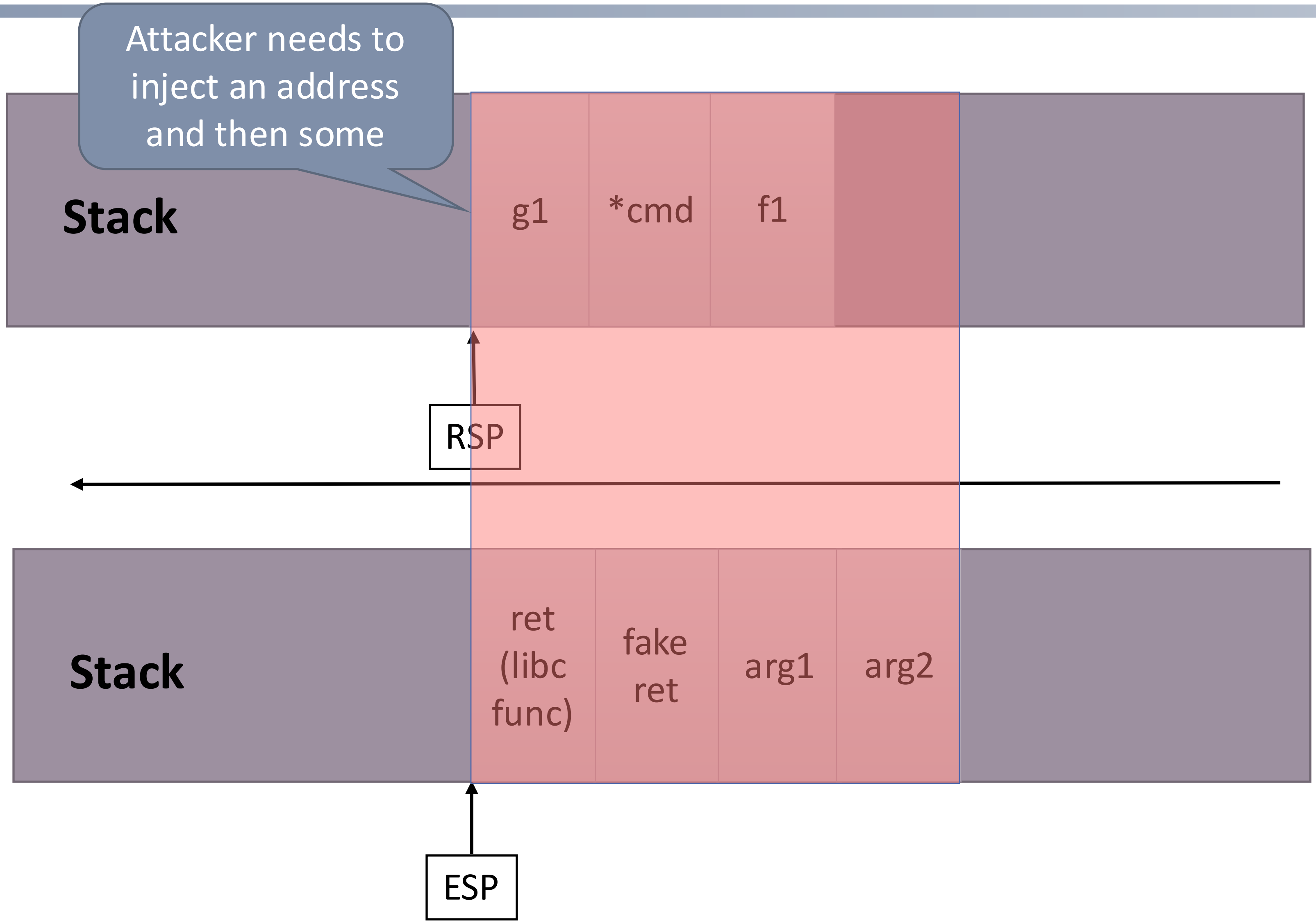
```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```



ASCII Armored Address Space

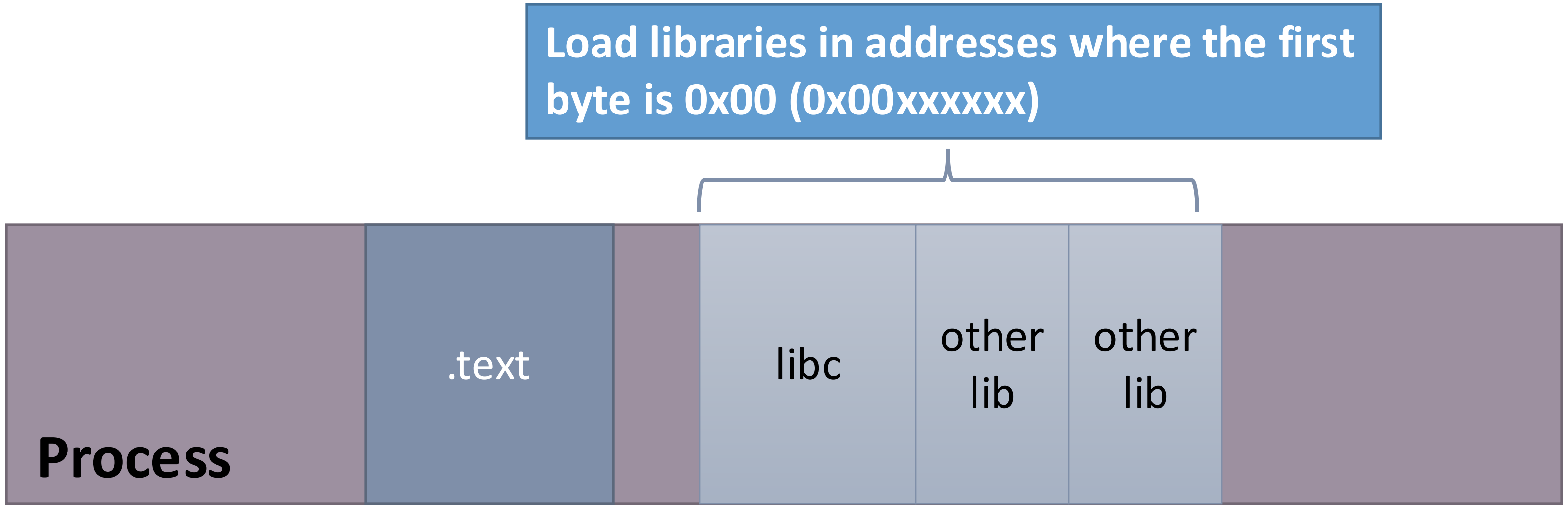


ASCII Armored Address Space

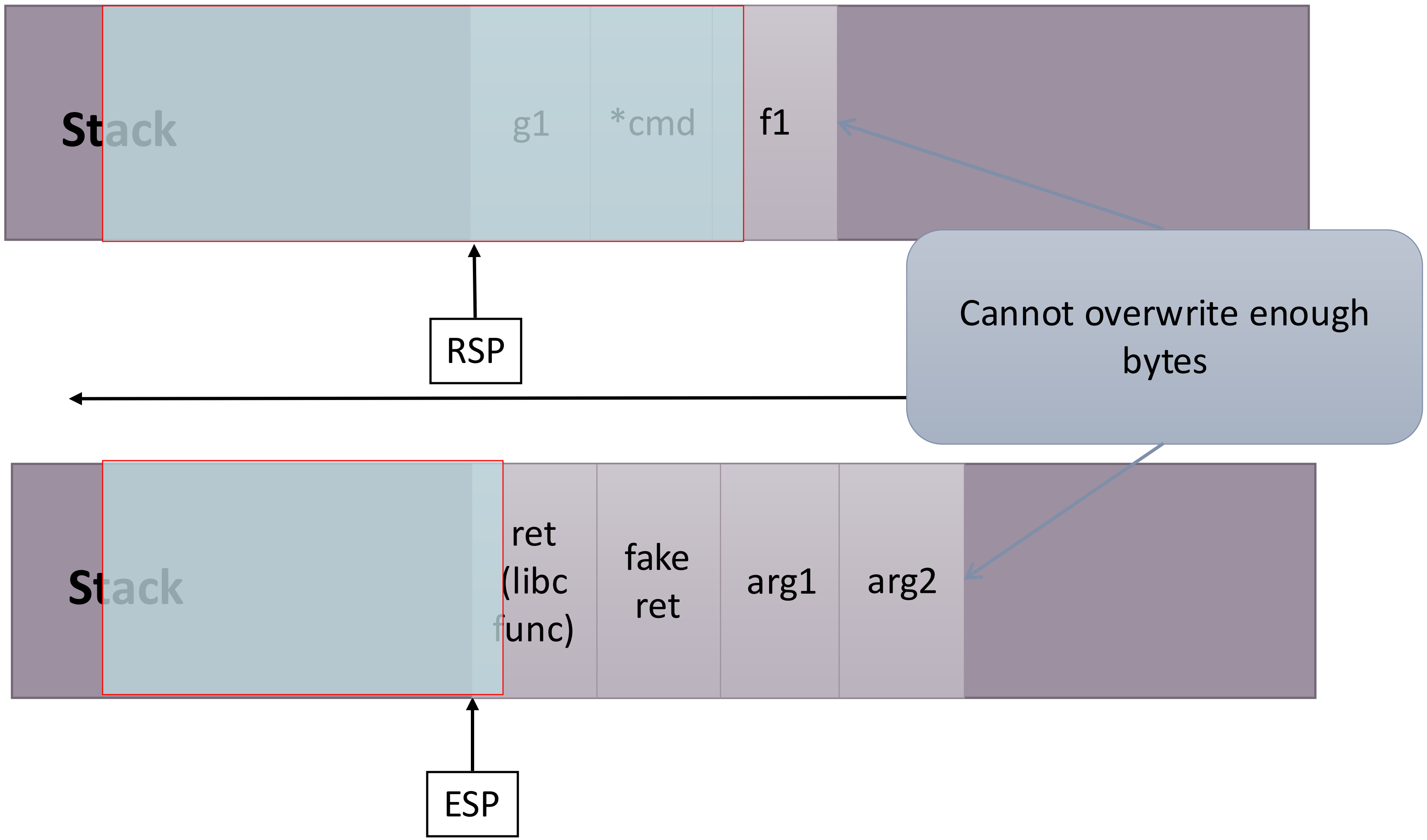


ASCII Armored Address Space

- **Observation:** strcpy() stops copying on the first null byte!



ASCII Armored Address Space



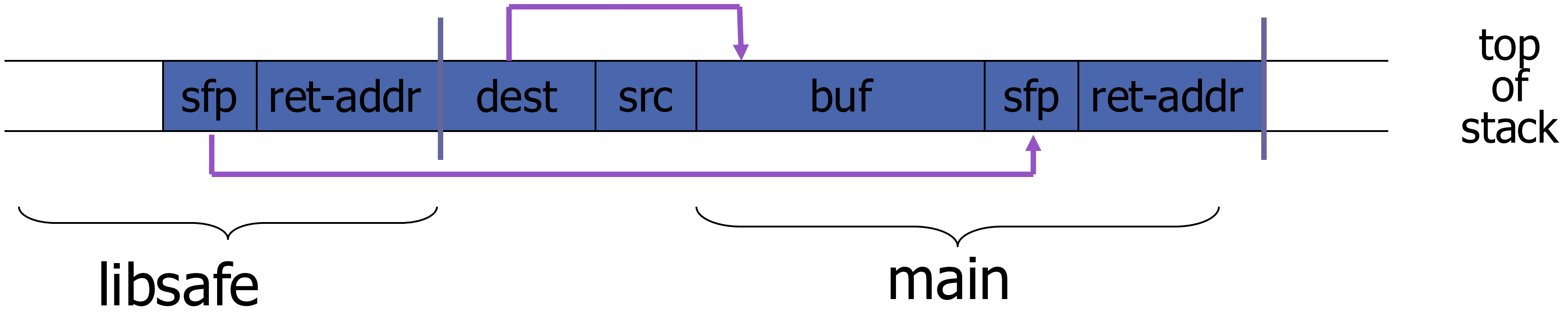
Problems

- Other methods of copying data may not have the same limitation: gets(), read(), fread(), custom copy routines, etc.
- Not all buffers are protected

Run time checking: Libsafe

- Old dynamically loaded library
- Intercepts calls to strcpy (dest, src), etc. through library interposition at load time
- Validates sufficient space in current stack frame:

  |frame-pointer – dest| > strlen(src)
  - If so, does strcpy()
  - Otherwise, terminates application

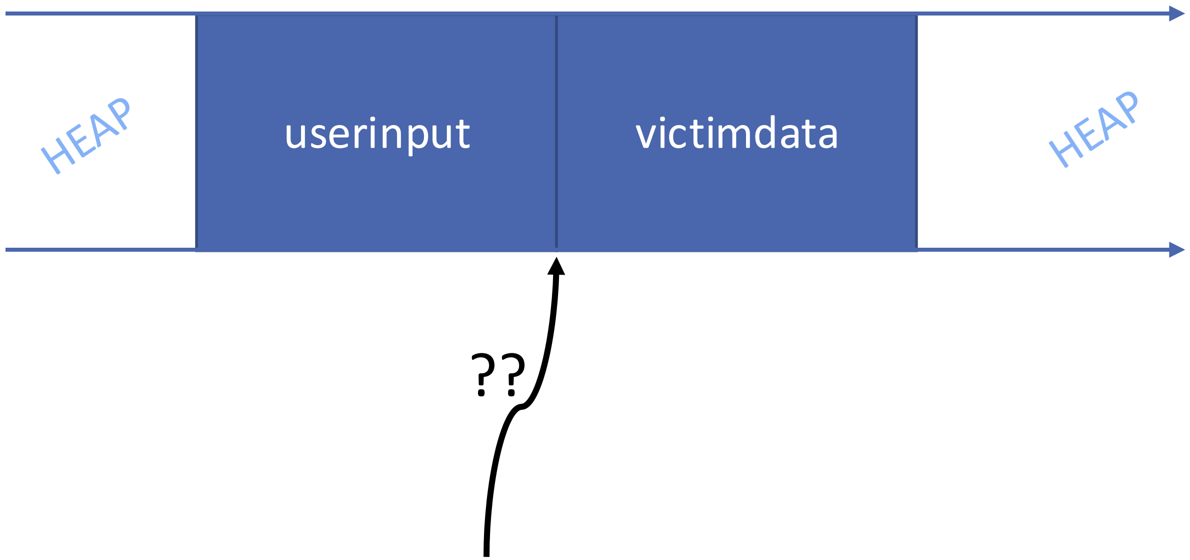


Software Exploitation
Heap Overflows

Georgios (George) Portokalidis

Understanding the Heap

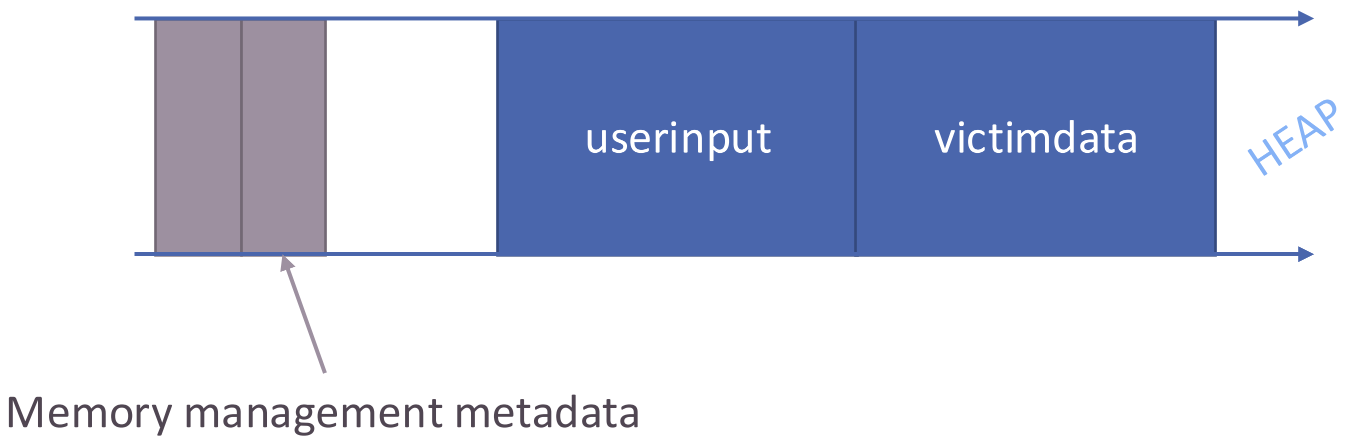
- The layout of buffers in memory depends on the implementation off the allocator (i.e., malloc)




```
char *userinput = malloc(20);
char *victimdata = malloc(20);
```

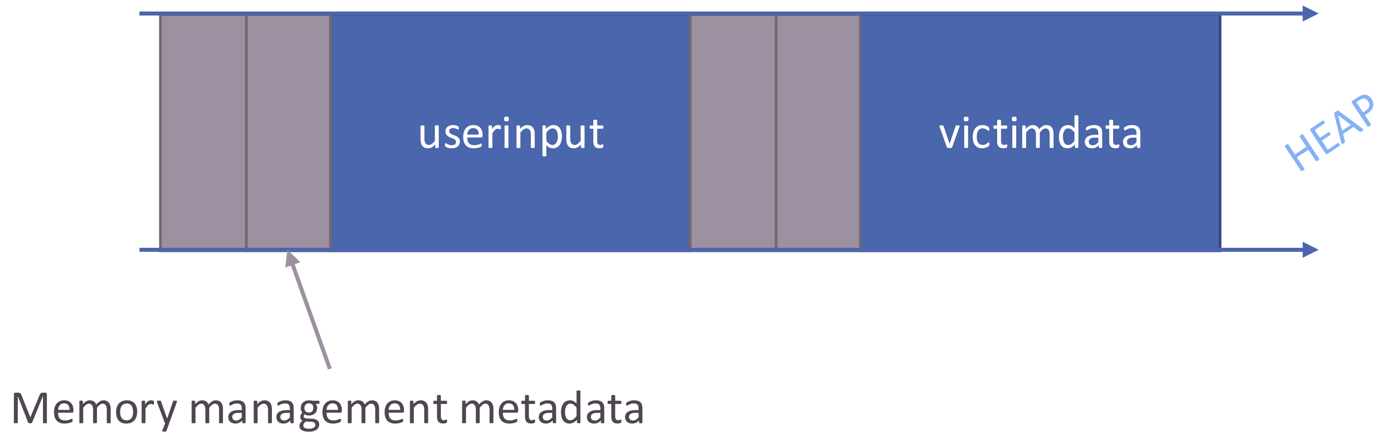
Metadata

- The heap also stores metadata about allocated areas
  - **Stored in separate area**



Metadata

- The heap also stores metadata about allocated areas
  - Stored in separate area
  - **Stored inline (interleaved) with program data**



malloc() Implementations

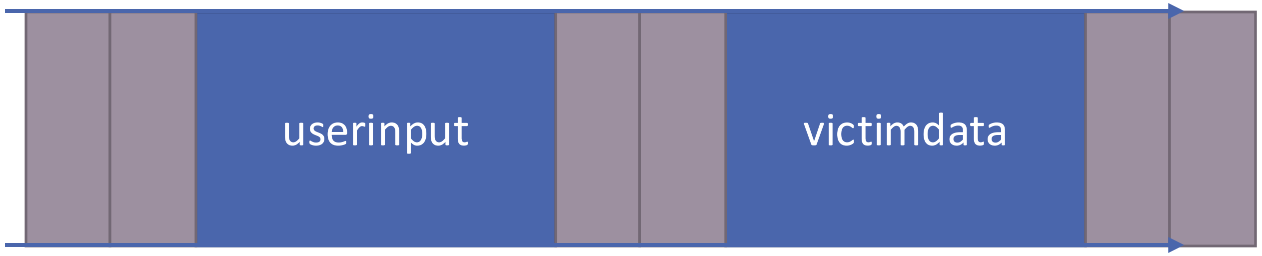
- dlmalloc – General purpose allocator
- ptmalloc2 – glibc
- jemalloc – FreeBSD and Firefox
- tcmalloc – Google
- libumem – Solaris
- …

glibc malloc()

- https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/
- Heap memory is obtained from the kernel using the brk() or mmap() system calls
  - Provides plenty of “raw” space
- The allocator splits memory into **arenas**
  - Each thread gets its own arena
  - Each arena has its own metadata
- Memory within the arena is split into **chunks** and given to program through various allocation functions (e.g., malloc())
  - Chunks are organized in bins, usually through double linked-lists

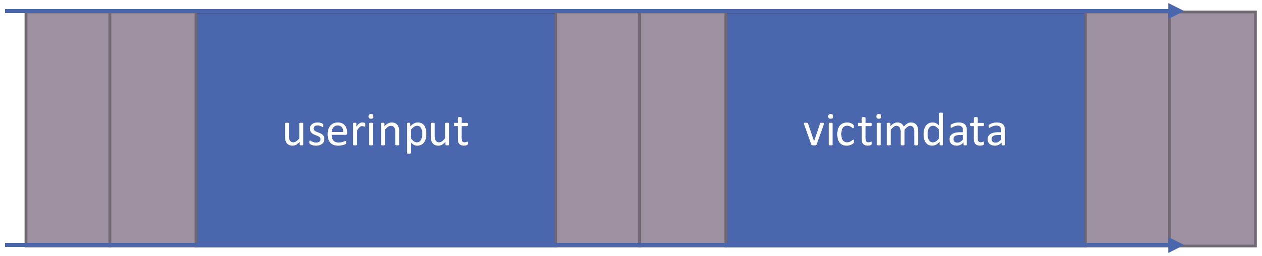
Controlled Buffer Contains Data

- Victim data is used as data
- Example: victimdata stores the filename to read from or write to

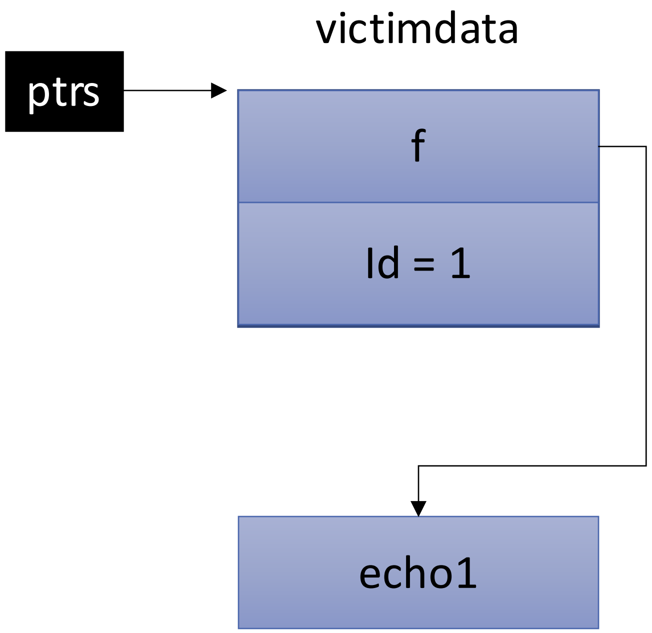


Controlled Buffer Contains Pointers

- A pointer that will be later used by the program is stored in victim data:
  - Function pointer:
    - That will be used in a call → control-flow hijacking
  - Data pointer:
    - That will be read → Arbitrary read bug
    - That will be written with user-controlled data → Arbitrary write bug
      - Can overwrite a code pointer (such as a return address)

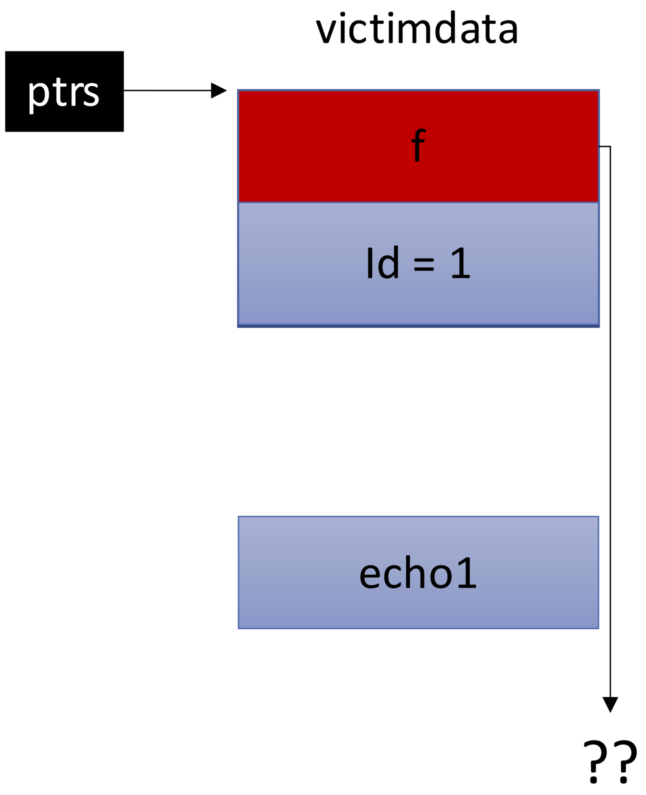


Example: Control Function Pointer



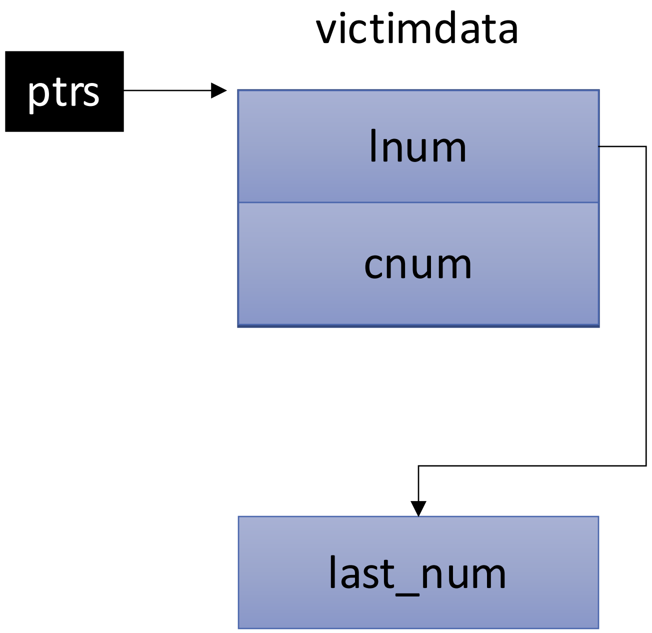

```
struct heap_ptrs {
        void (*f)(const char *);
        int id;
};

…

struct heap_ptrs *ptrs = malloc(…);

ptrs[0].id = 1;
ptrs[0].f = echo1;
…

ptrs[i].f(echo_buf);
```

Example: Control Function Pointer



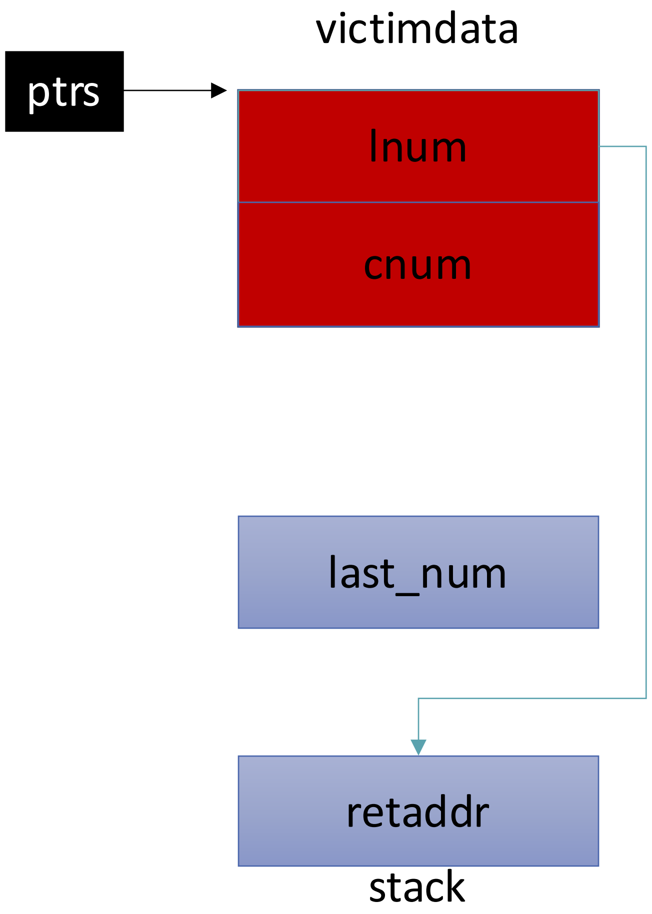

```
struct heap_ptrs {
        void (*f)(const char *);
        int id;
};

…

struct heap_ptrs *ptrs = malloc(…);

ptrs[0].id = 1;
ptrs[0].f = echo1;
…

ptrs[i].f(echo_buf);
```

Example: Control Data Pointer



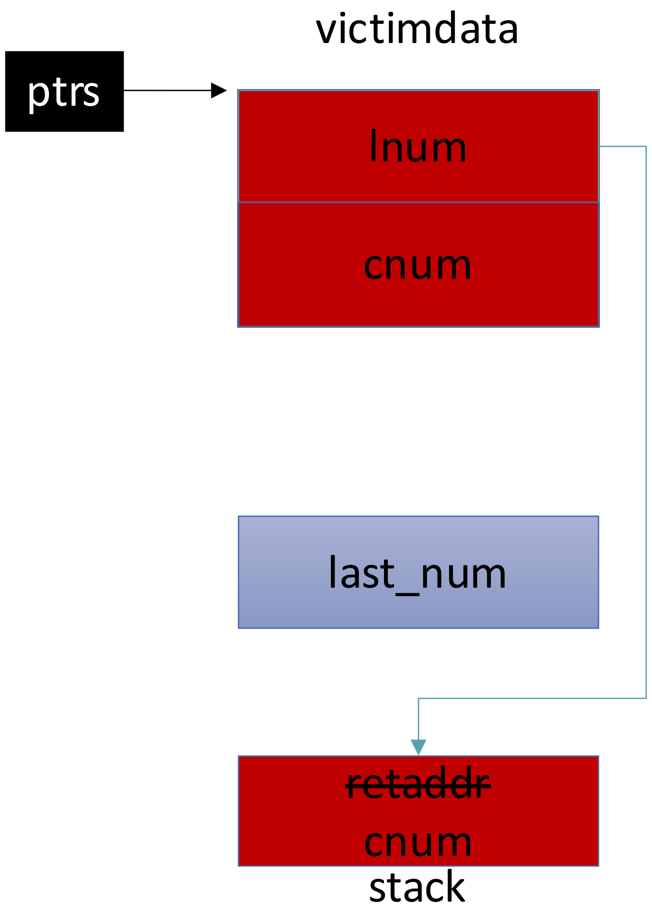

```
struct heap_ptrs {
        long *lnum;
        long cnum;
};

long last_num;

…

struct heap_ptrs *ptrs = malloc(…);

ptrs->lnum = &last_num;

…

*ptrs->lnum = ptrs->cnum;
```

Example: Arbitrary Write




```
struct heap_ptrs {
    long *lnum;
    long cnum;
};

long last_num;

...

struct heap_ptrs *ptrs = malloc(...);

ptrs->lnum = &last_num;

...

*ptrs->lnum = ptrs->cnum;
```

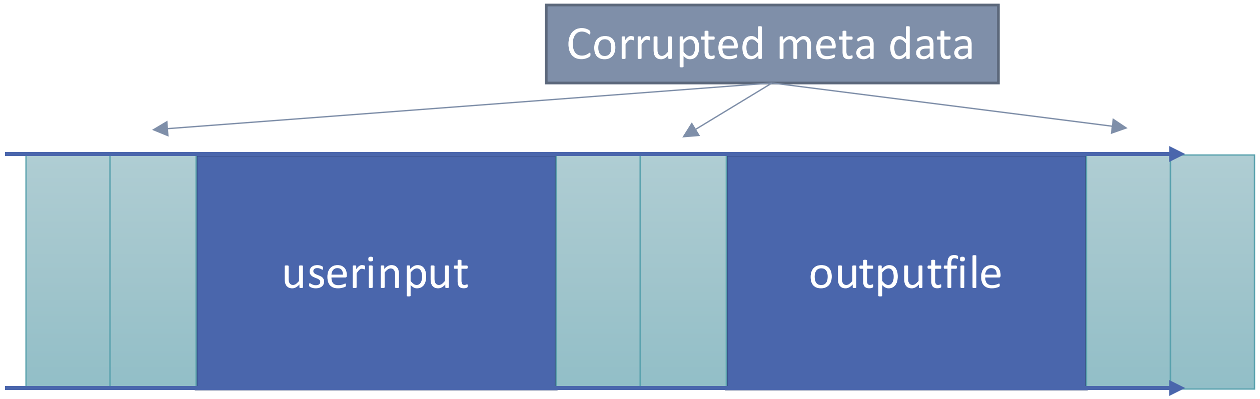
Example: Arbitrary Write




```
struct heap_ptrs {
        long *lnum;
        long cnum;
};

long last_num;

…

struct heap_ptrs *ptrs = malloc(…);

ptrs->lnum = &last_num;

…

*ptrs->lnum = ptrs->cnum;
```

Heap-Specific Pattern: Corrupted Metadata

- Use of the corrupted meta data and may lead to an arbitrary write, corrupting a code pointer or security critical data
- More in the appendix



Heap Overflows In Practice

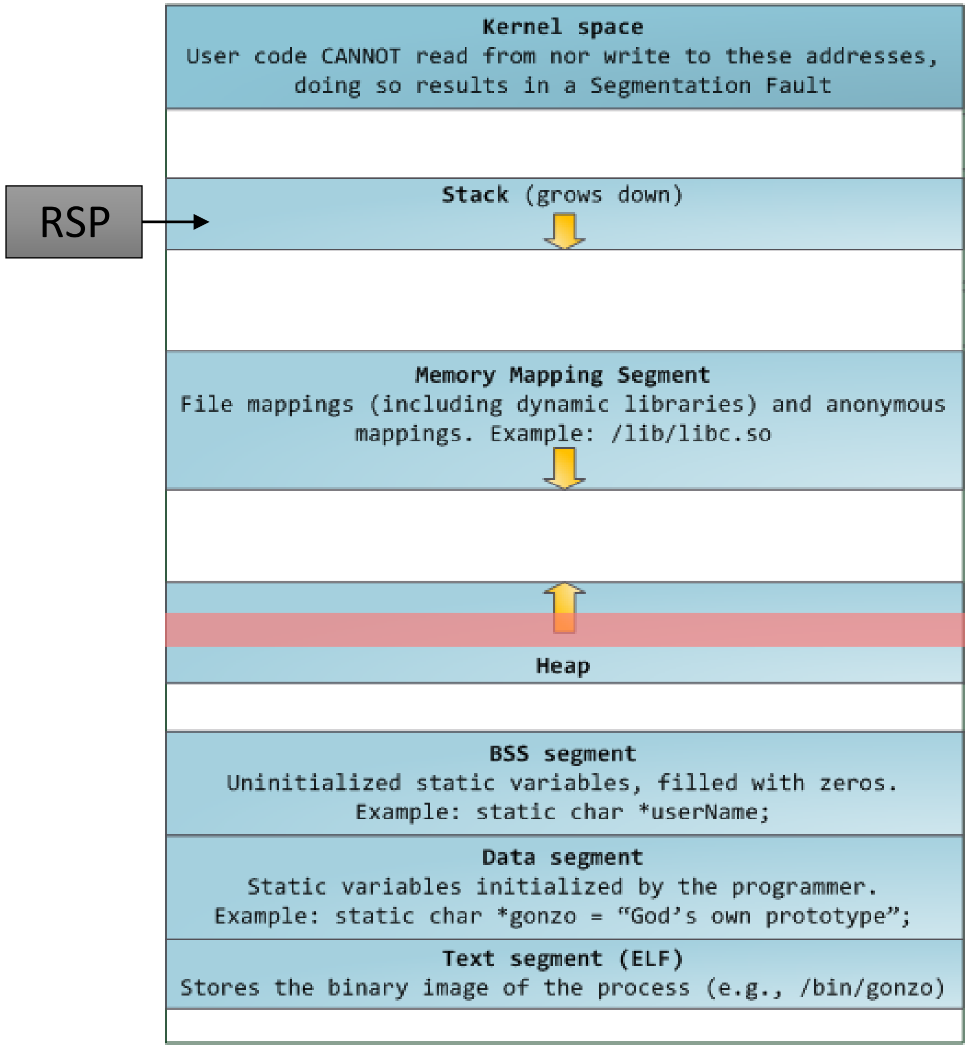
- Exploiting the allocator depends on
  - The allocator’s implementation
  - The sequence of allocator calls in the program
- The attacker may need to “guide” the program to perform a long sequence of allocations and deallocations to **align** the objects in the heap
  - Referred to as memory massaging

ROP and Heap Overflows

Heap to Stack

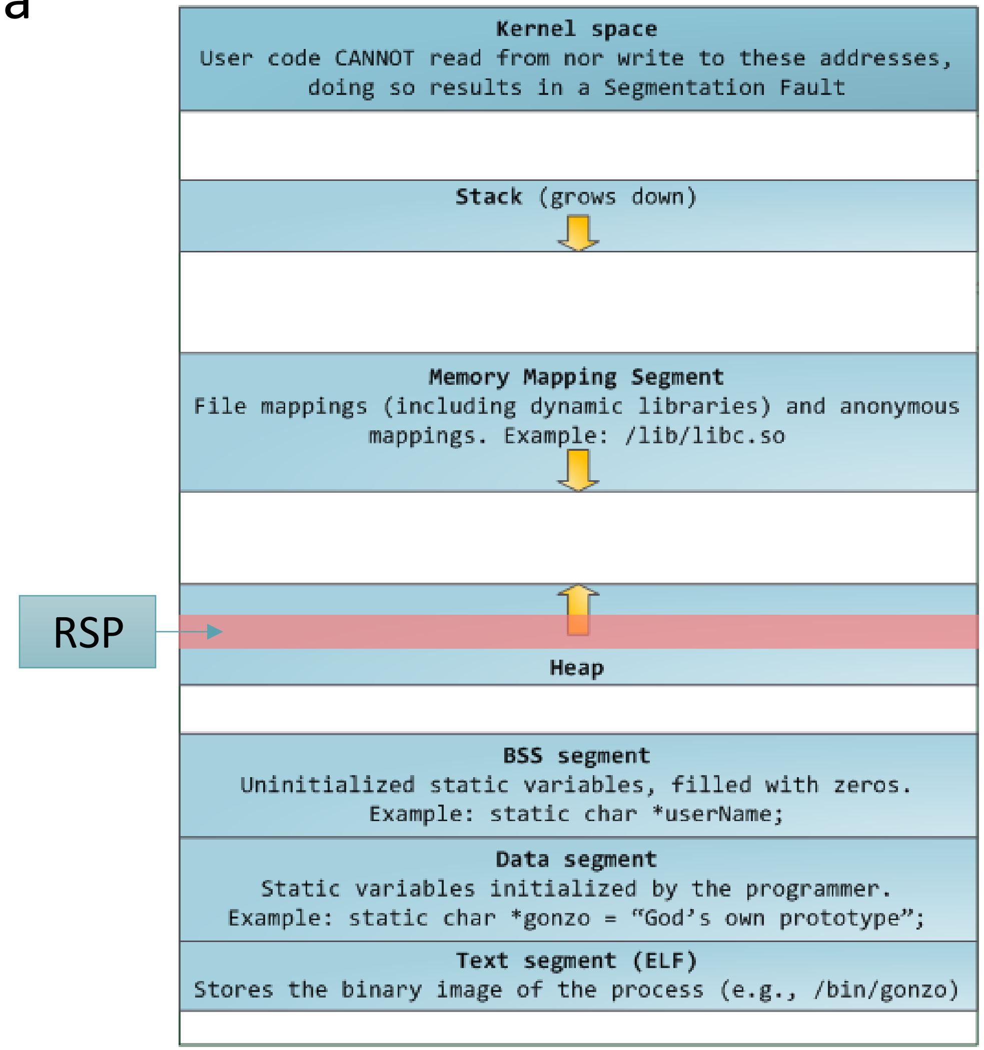
- Attacker controls:
  - the outcome of a call * or jmp *
    - E.g., by overwriting a function pointer in the heap
  - An area in the heap
- **ROP requires controlling the data under RSP**

??



Enter Stack Pivoting

- Make the stack pointer point to user data



Enter Stack Pivoting

- **Solution 1**

- Requirements:
  - A register points to the controlled buffer on the heap
  - An exchange gadget with RSP and that register exists

- How:
  - Execute the gadget

```
xchg r**, rsp
...
ret
```

Enter Stack Pivoting

- **Solution 2**

- Requirements:
  - A gadget that adds/subs a large value from the stack pointer
  - The result of the above points the SP to user-controlled data

- How:
  - Execute the gadget

```
add 0x***, rsp
...
ret
```

```
sub 0x***, rsp
...
ret
```

Enter Stack Pivoting

- **Solution 3**

- Requirements:
  - You control RBP
  - A leave gadget exists

- How:
  - Execute the gadget

```
movl    %ebp, %esp
pop     %ebp
```

```
leave
ret
```

More Stack Pivoting

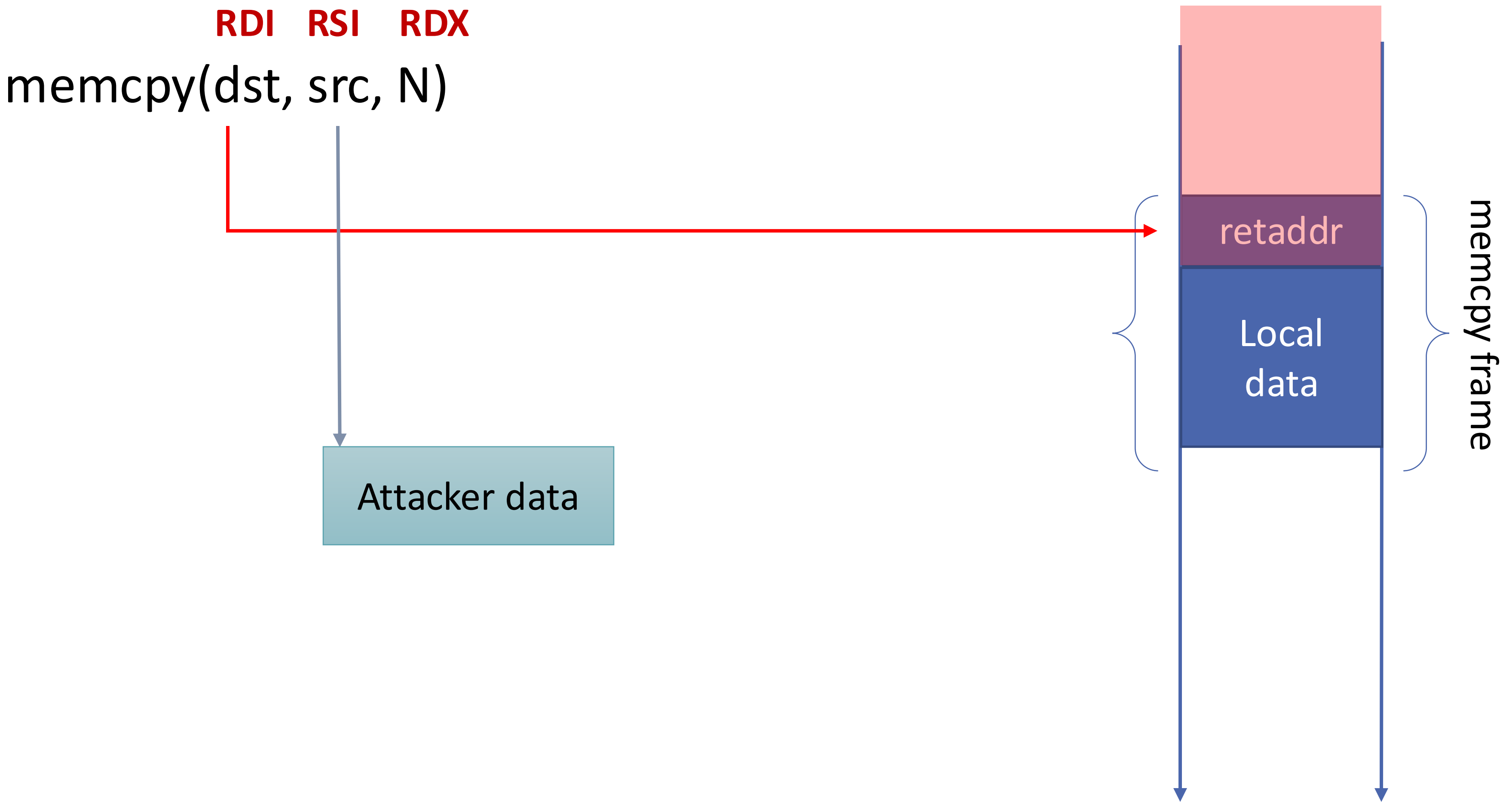
- Combining multiple pivots is possible
  - For example, executing a sub rsp, 0x**** gadget in a loop
- Any instruction sequence that updates the RSP with user-controlled data will do
- Example:

```
push rax
pop rsp
...
ret
```

Defenses and Bypasses

- Check that RSP is pointing into the stack area
  - Potentially expensive (how often should I check the RSP?)
- Can be subverted by …
  - …corrupting the saved stack boundaries
  - …using a gadget that copies your buffer into the stack
    - For example, find a gadget that calls memcpy()

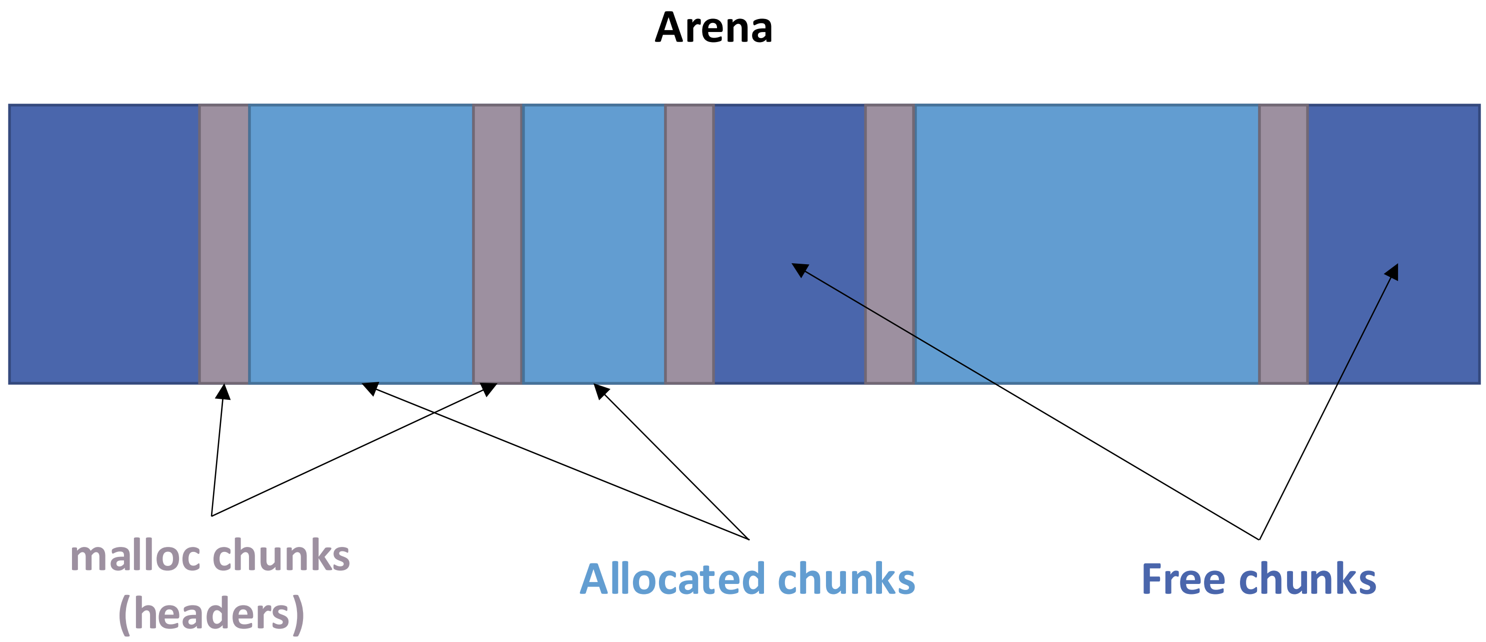
Memcpy()



RDI RSI RDX
memcpy(dst, src, N)
Attacker data
retaddr
Local data
memcpy frame

Appendix: Attacks Through Allocator-metadata Corruption

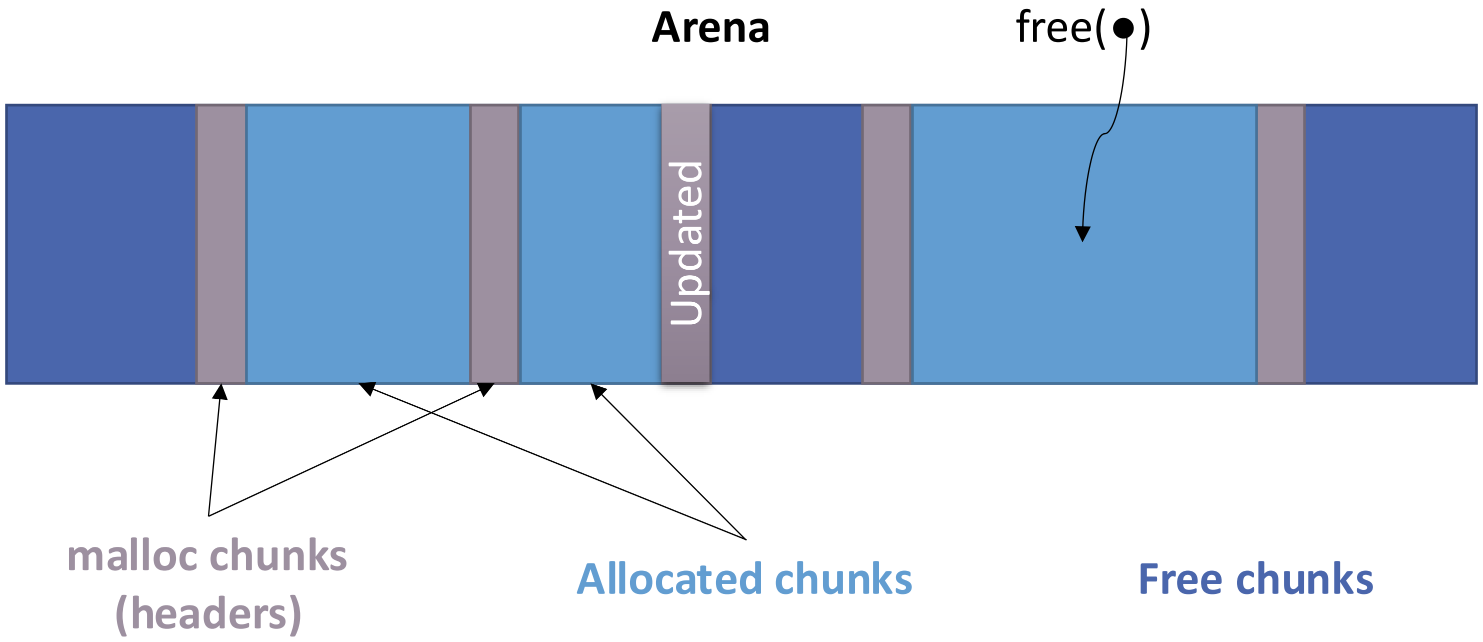
Heap Arena Structure



No two free chunks can be adjacent.

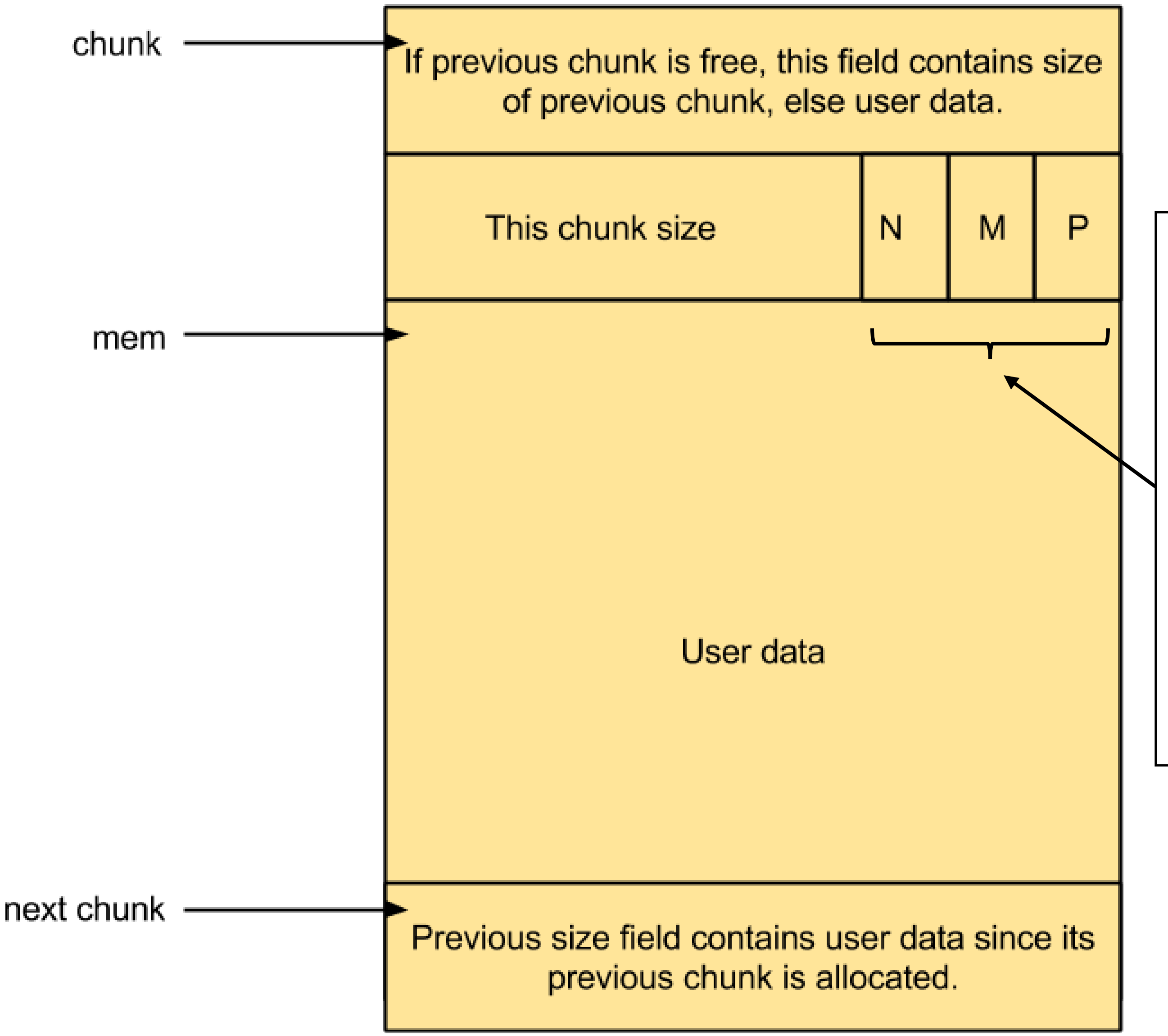
Appendix: Attacks by Corrupting Heap Metadata

Heap Arena Structure



No two free chunks can be adjacent.

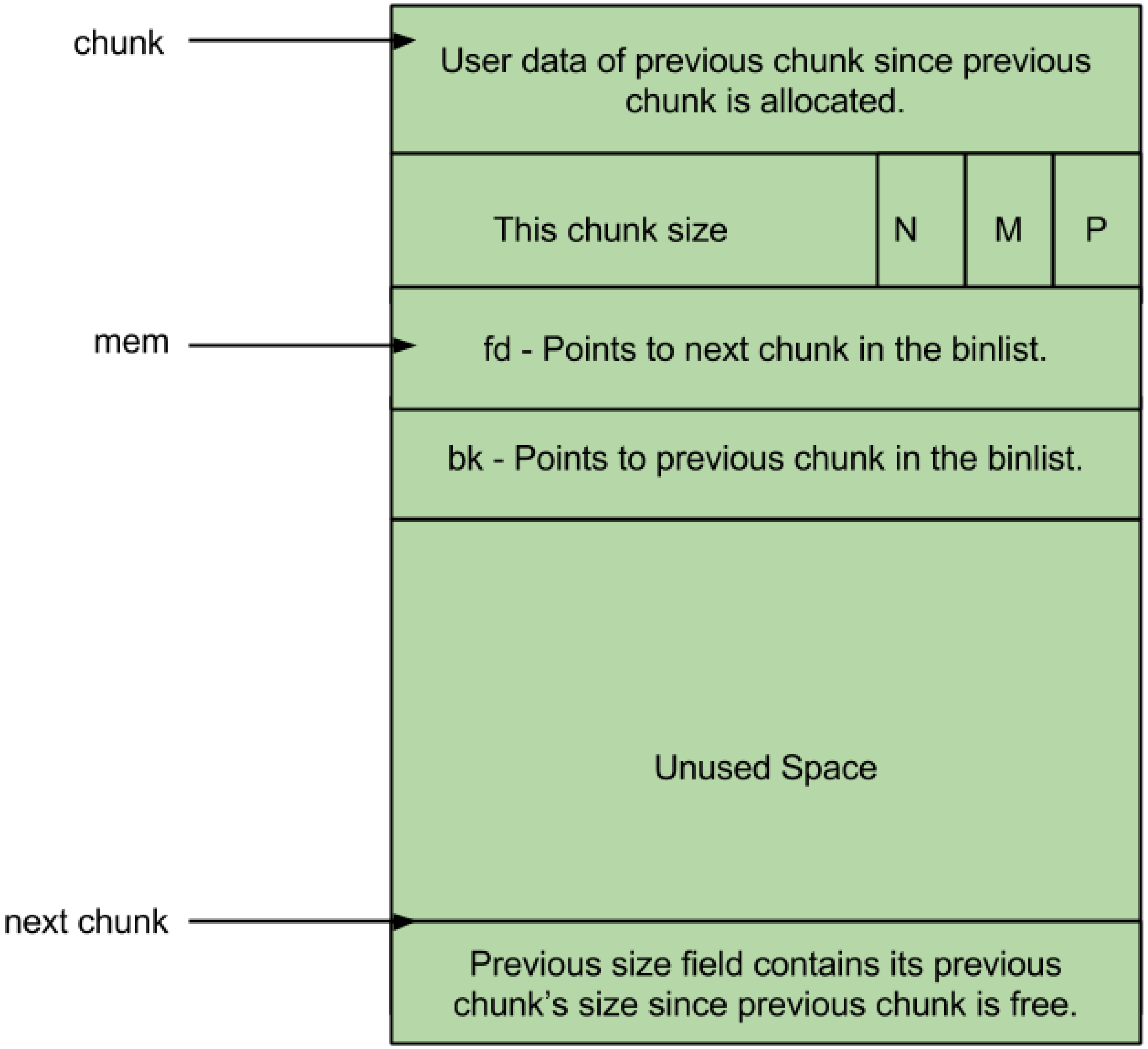
Adjacent free chunks are merged together



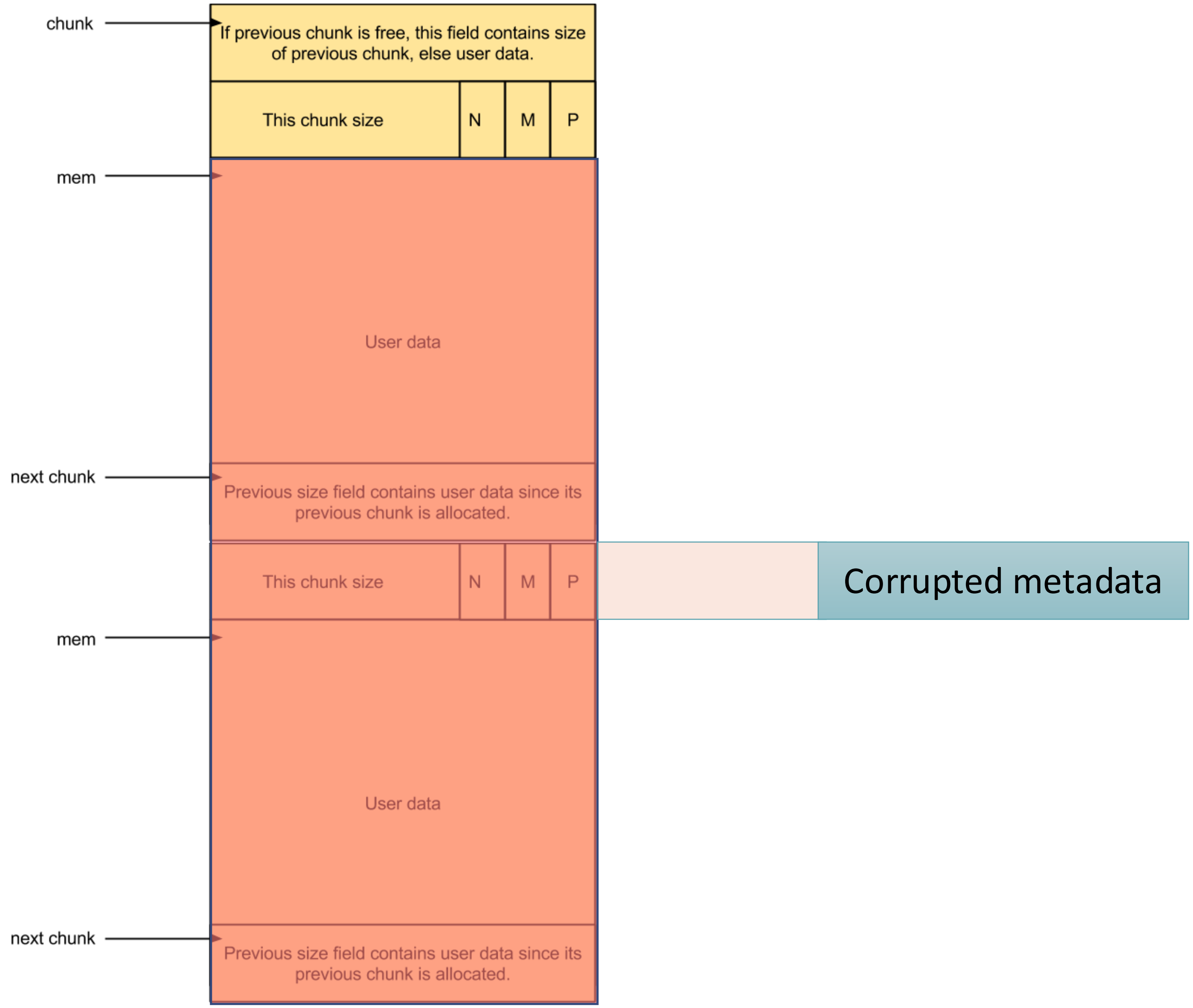
Bitmap

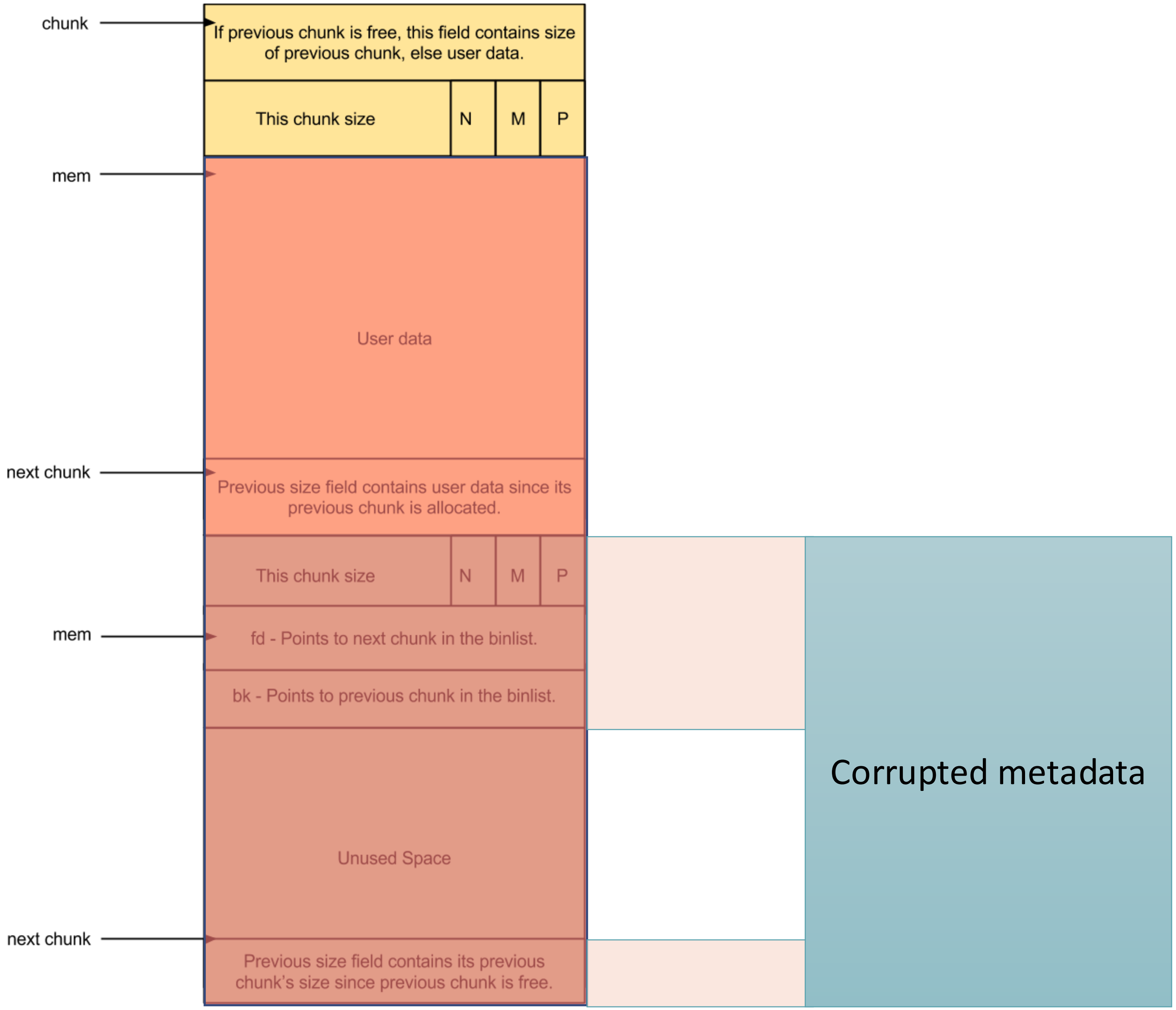
- P - This bit is set when previous chunk is allocated
- M - This bit is set when chunk is mmap’d
- N - This bit is set when this chunk belongs to a thread arena.

Allocated Chunk


chunk
User data of previous chunk since previous chunk is allocated.
This chunk size
N
M
P
mem
fd - Points to next chunk in the binlist.
bk - Points to previous chunk in the binlist.
Unused Space
next chunk
Previous size field contains its previous chunk’s size since previous chunk is free.

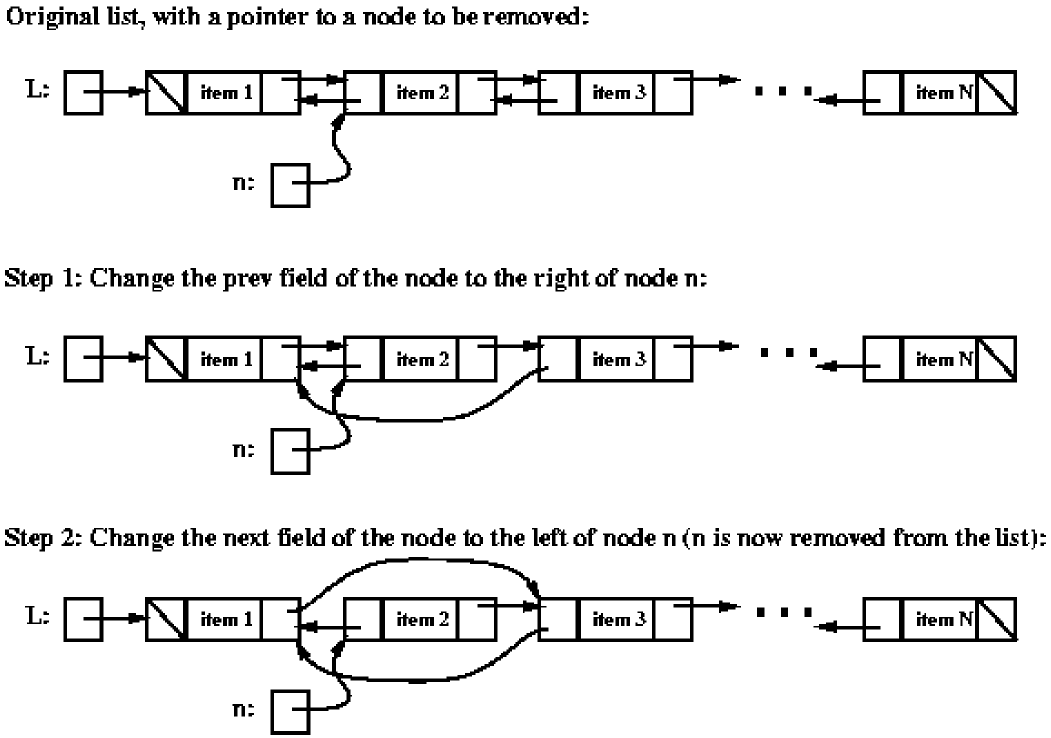
Free Chunk


chunk
If previous chunk is free, this field contains size of previous chunk, else user data.
This chunk size
N
M
P
mem
User data
next chunk
Previous size field contains user data since its previous chunk is allocated.
This chunk size
N
M
P
Corrupted metadata
mem
User data
next chunk
Previous size field contains user data since its previous chunk is allocated.


chunk
If previous chunk is free, this field contains size of previous chunk, else user data.
This chunk size
N
M
P
mem
User data
next chunk
Previous size field contains user data since its previous chunk is allocated.
This chunk size
N
M
P
mem
fd - Points to next chunk in the binlist.
bk - Points to previous chunk in the binlist.
Unused Space
next chunk
Previous size field contains its previous chunk's size since previous chunk is free.
Corrupted metadata

Linked-list Manipulation to Arbitrary Write

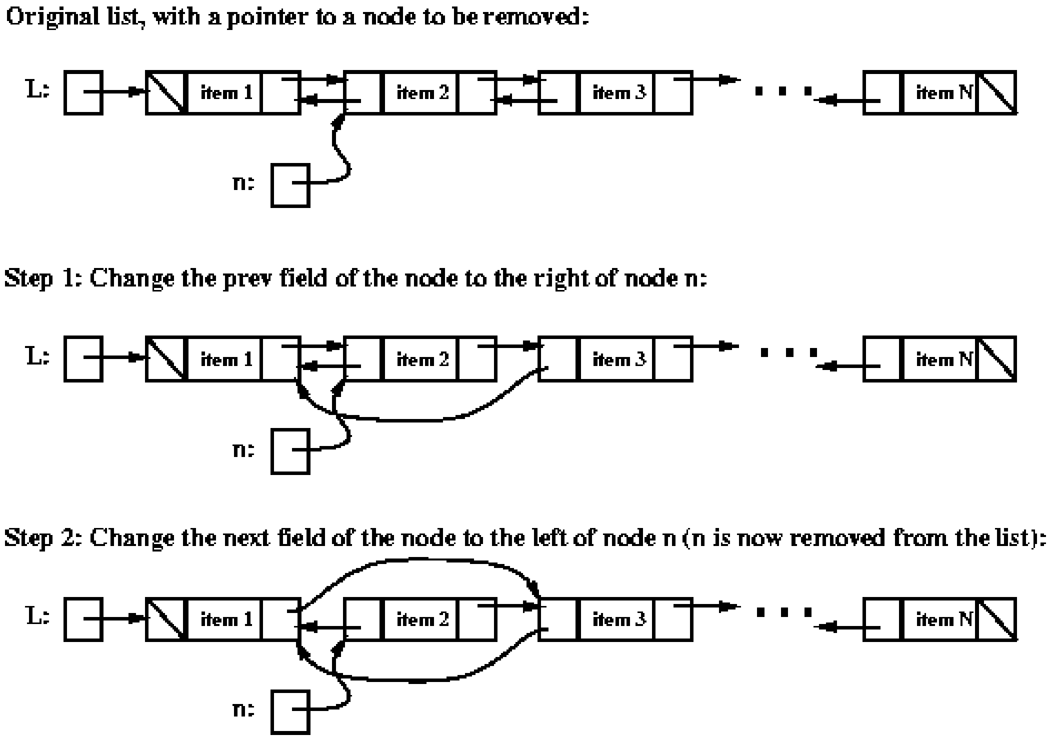
Corrupted pointers attacker controlled next and prev pointers due to the overwritten n



Remove *n*

```
n->next->prev = n->prev;

n->prev->next = n->next;
```

Linked-list Manipulation to Arbitrary Write



Remove *n*

*(n->next + prev_offset) = n->prev

n->next->prev = n->prev;

*(n->prev + next_offset) = n->next

n->prev->next = n->next;

Heap Integrity Checks

- Used to be enabled when environment variable `MALLOC_CHECK_` > 0
- On by default on most recent systems
- Values of metadata pointers are checked for viability
  - Example: Ensures pointers point within the current arena

Remove *n*

*(n->next + prev_offset) = n->prev

n->next->prev = n->prev;

*(n->prev + next_offset) = n->next

n->prev->next = n->next;

Software Exploitation Address Space Layout Randomization (ASLR)

Georgios (George) Portokalidis

Fixed Process Layout

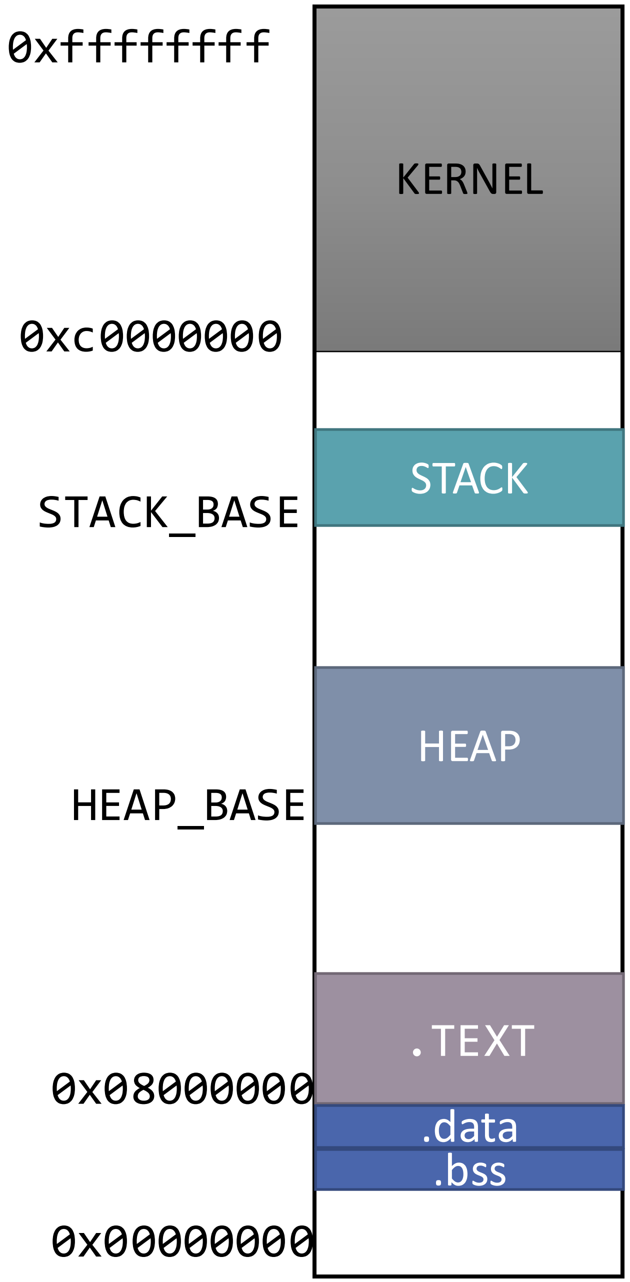
- The programs we have exploited (this far) have a fixed memory layout
- Data segments start at the same address
- ELF binary is loaded at the same address
- Shared libraries are loaded at the same address

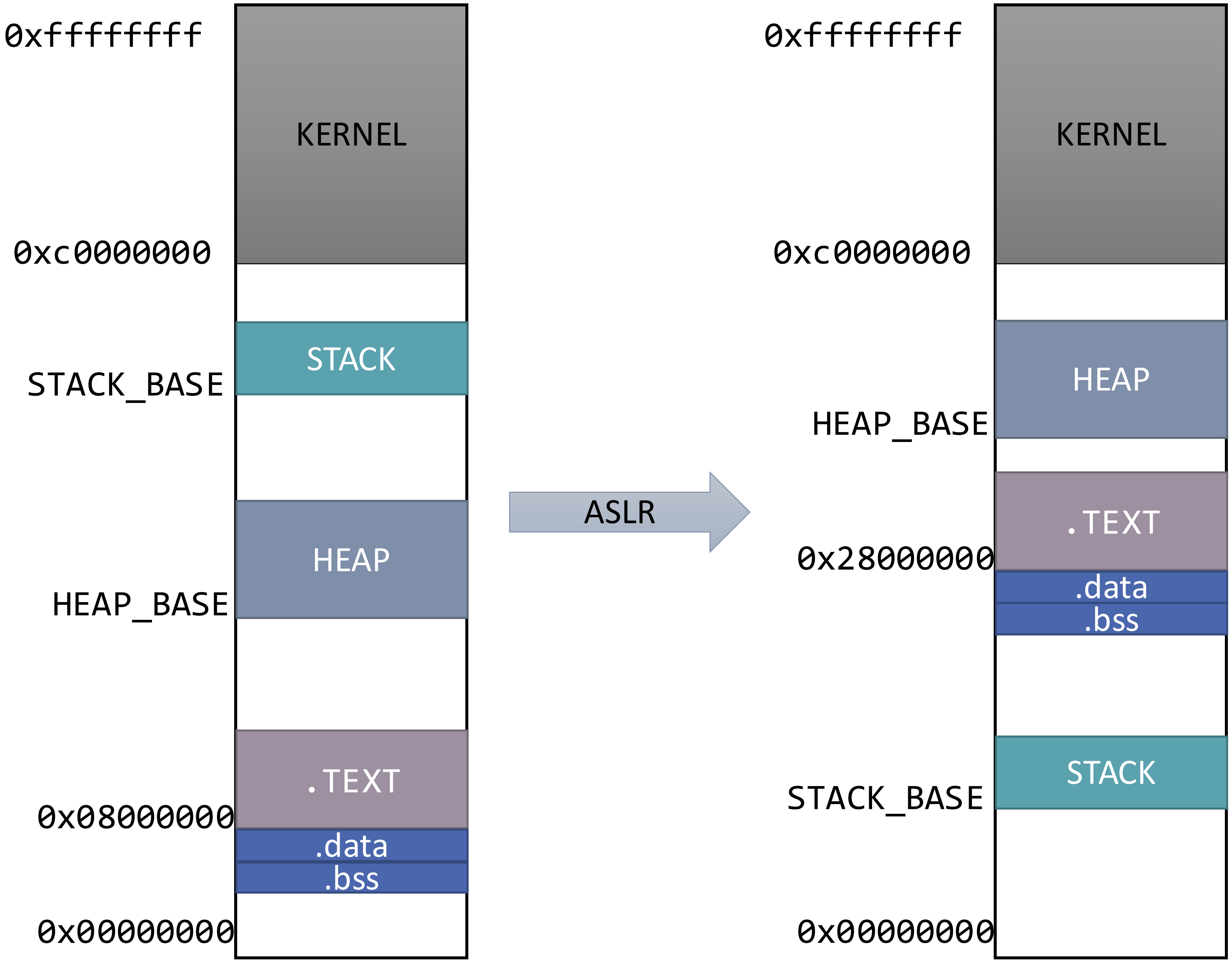
One Attack Fits All

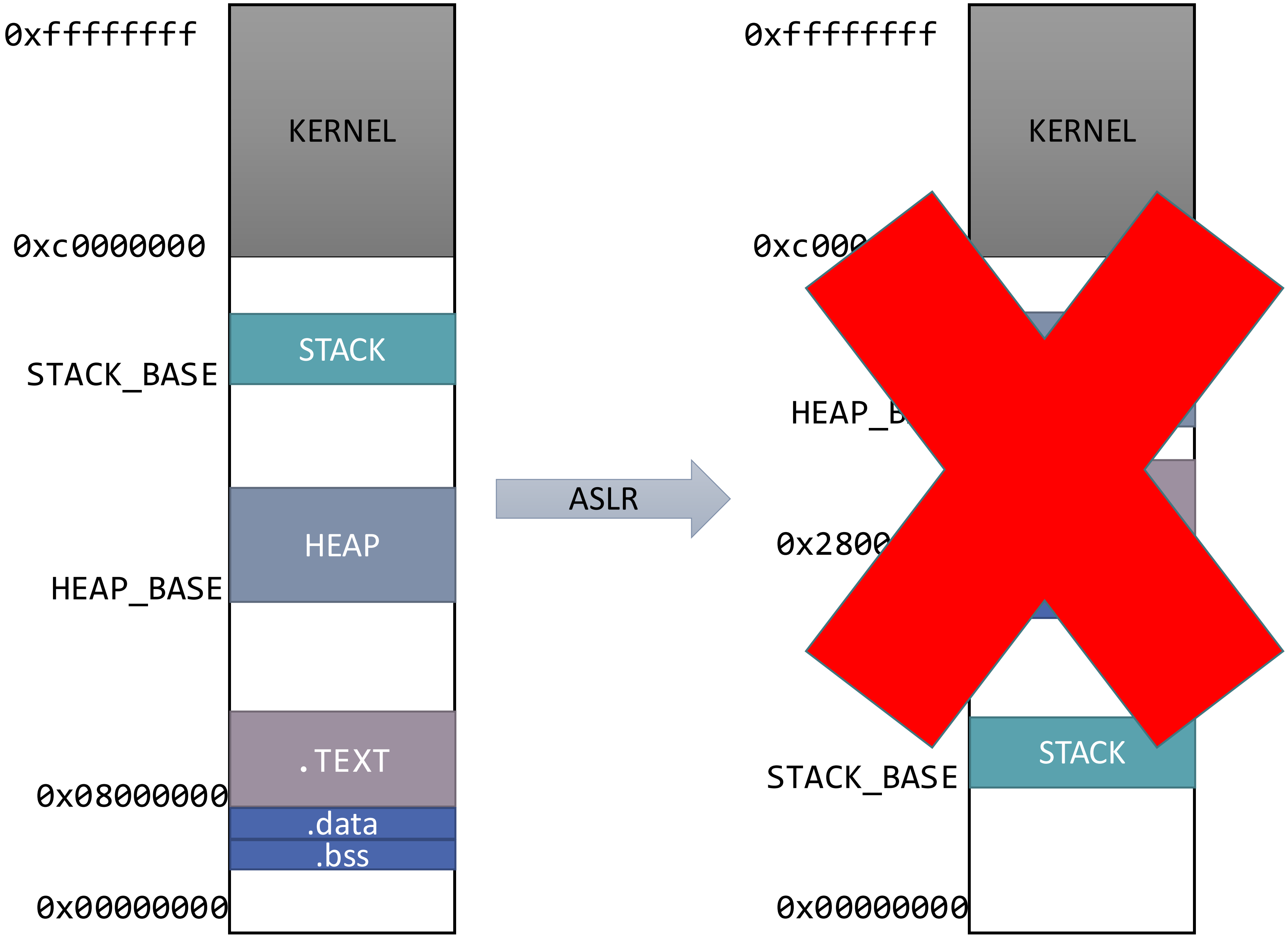
- Fixed process layout → facilitates exploit development
- Attacker can statically discover …
  - … the location of their data
  - … the location of code (gadgets, functions, etc.)
- An exploit developed on one system will work on all other systems running the same software

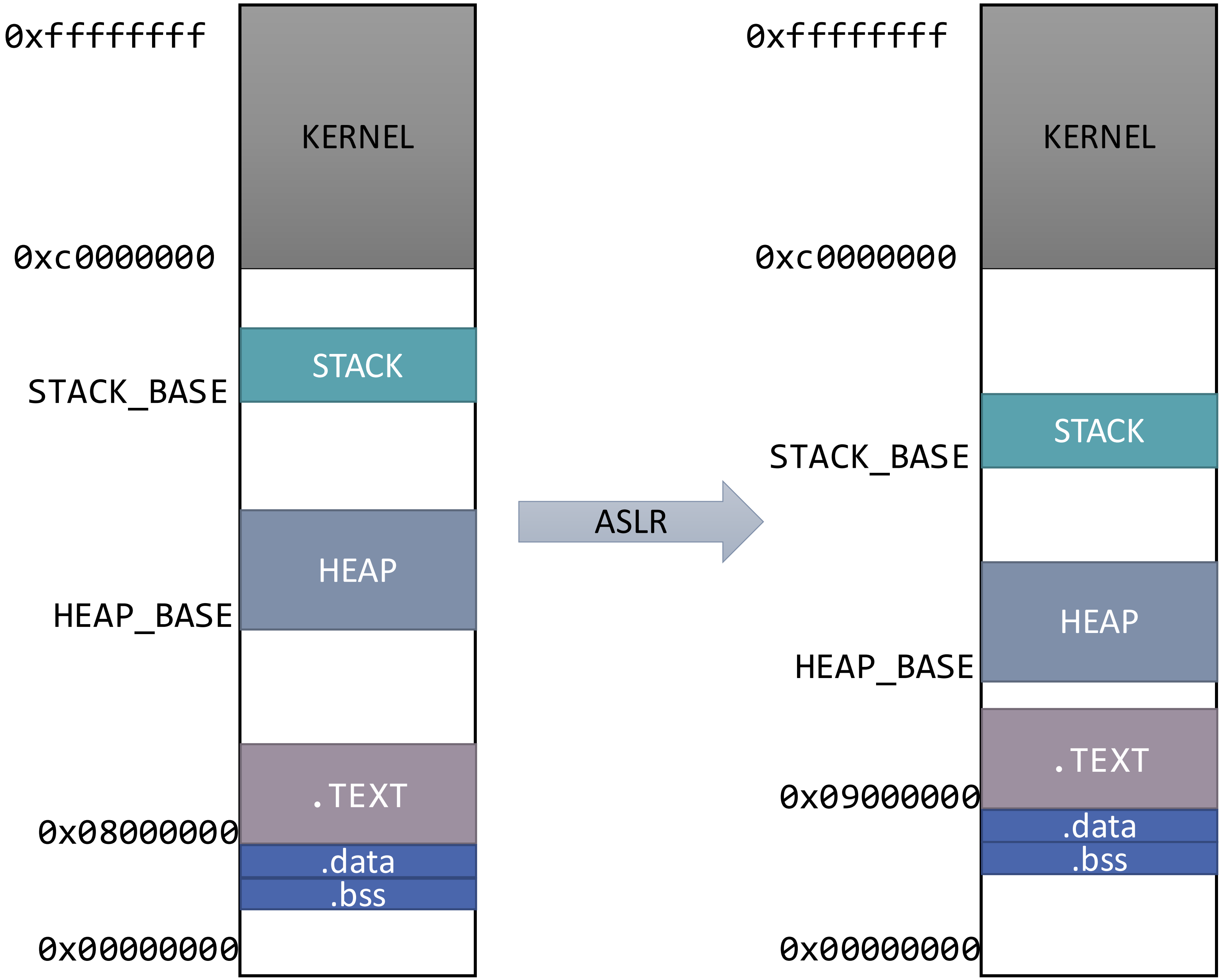
Address Space Layout Randomization (ASLR)

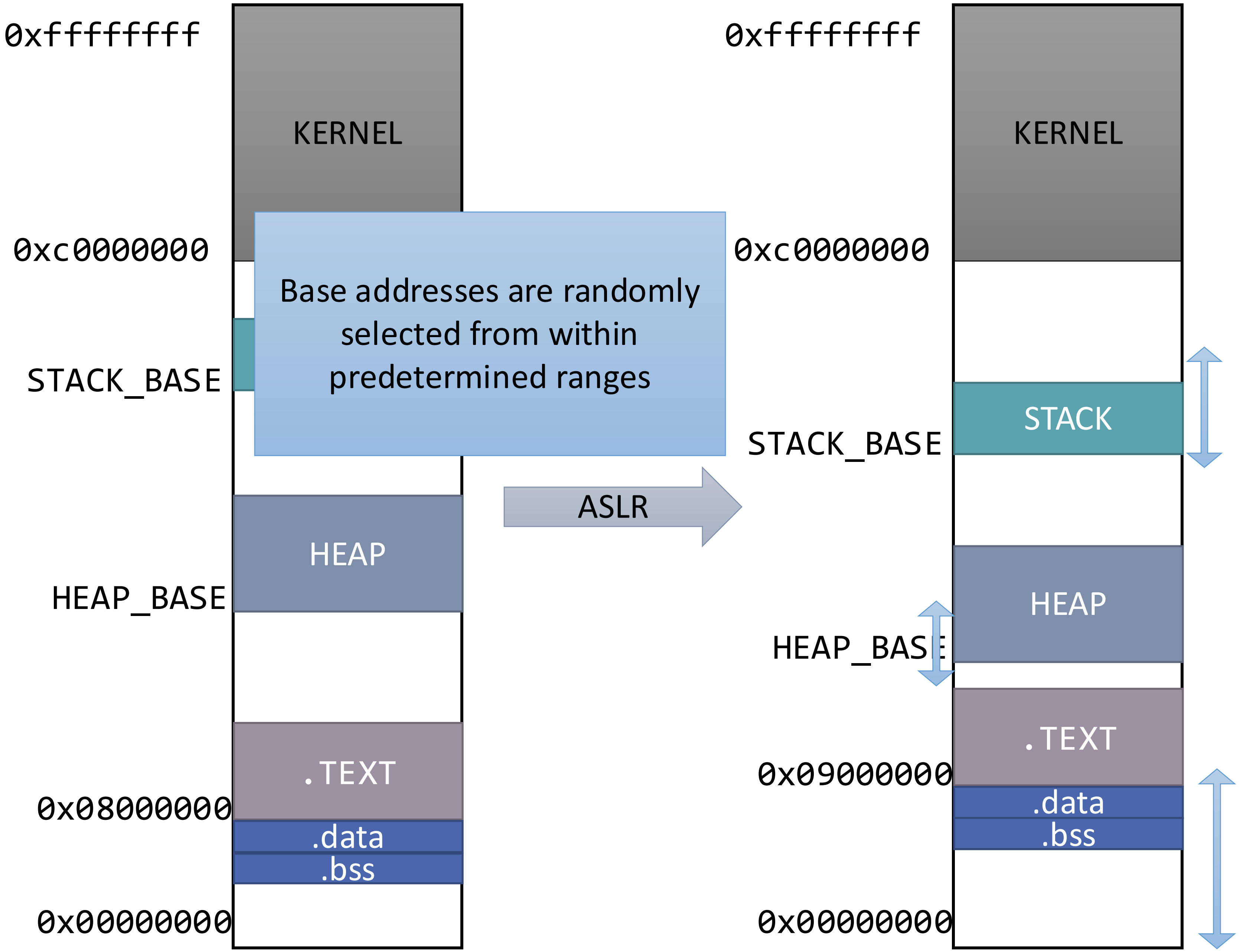
- Ideal version → when starting up a process, randomly pick the base address where each data and code segment will be loaded
- Introduce uncertainty for the attacker → need to guess the location of code and their data

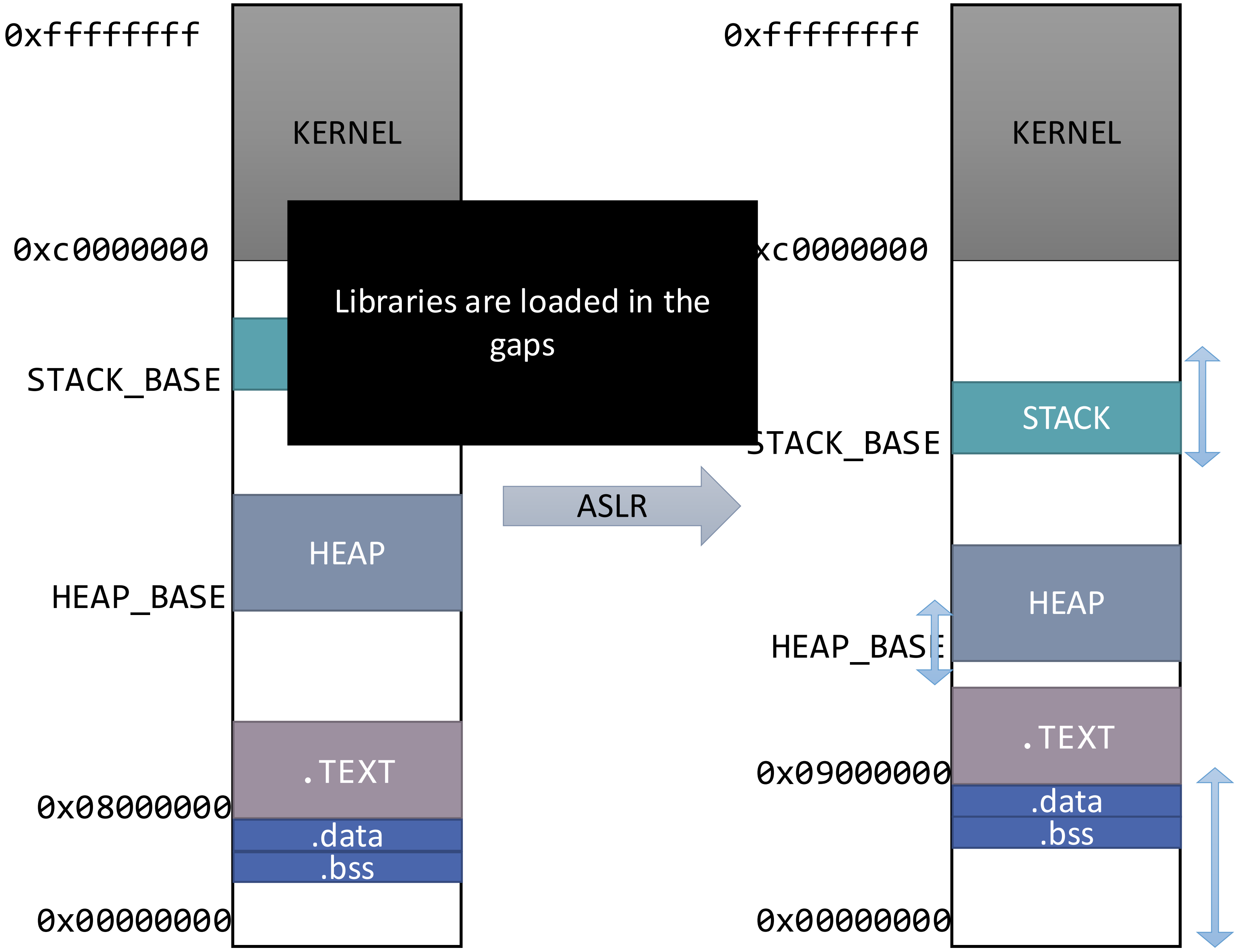

0xffffffff
KERNEL
0xc0000000
STACK
STACK_BASE
HEAP
HEAP_BASE
.TEXT
0x08000000
.data
.bss
0x00000000


0xffffffff
KERNEL
0xc0000000
STACK
STACK_BASE
HEAP
HEAP_BASE
.TEXT
0x08000000
.data
.bss
0x00000000
ASLR
0xffffffff
KERNEL
0xc0000000
HEAP
HEAP_BASE
.TEXT
0x28000000
.data
.bss
STACK
STACK_BASE
0x00000000


0xffffffff
KERNEL
0xc0000000
STACK
STACK_BASE
HEAP
HEAP_BASE
.TEXT
0x08000000
.data
.bss
0x00000000
ASLR
0xffffffff
KERNEL
STACK
STACK_BASE
0x00000000


0xffffffff
KERNEL
0xc0000000
STACK
STACK_BASE
HEAP
HEAP_BASE
.TEXT
0x08000000
.data
.bss
0x00000000
ASLR
0xffffffff
KERNEL
0xc0000000
STACK
STACK_BASE
HEAP
HEAP_BASE
.TEXT
0x09000000
.data
.bss
0x00000000


0xffffffff
KERNEL
0xc0000000
STACK_BASE
HEAP
HEAP_BASE
.TEXT
0x08000000
.data
.bss
0x00000000
Base addresses are randomly selected from within predetermined ranges
ASLR
0xffffffff
KERNEL
0xc0000000
STACK
STACK_BASE
HEAP
HEAP_BASE
.TEXT
0x09000000
.data
.bss
0x00000000


0xffffffff
KERNEL
0xc0000000
STACK_BASE
HEAP
HEAP_BASE
.TEXT
0x08000000
.data
.bss
0x00000000
Libraries are loaded in the gaps
ASLR
0xffffffff
KERNEL
STACK
STACK_BASE
HEAP
HEAP_BASE
.TEXT
0x09000000
.data
.bss
0x00000000

Example

```
unsigned long getEBP (void) {
    __asm ( ”movl %ebp ,%eax ” );
}

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

No ASLR

```
> ./getEBP
EBP:bffff3b8

> ./getEBP
EBP:bffff3b8
```

With ASLR

```
> ./getEBP
EBP:bfaa2e58

> ./getEBP
EBP:bf9114c8
```

ASLR in Practice

- Amount of randomness can differ between OSes

ASLR in Linux

- Rs: number of bits randomized in the stack area
- Rm: number of bits randomized in the mmap() area
- Rx: number of bits randomized in the main executable area
- Ls: least significant randomized bit position in the stack area
- Lm: least significant randomized bit position in the mmap() area
- Lx: least significant randomized bit position in the main executable area

32-bit Linux

- Rs = 24, Rm = 16, Rx = 16, Ls = 4, Lm = 12, Lx = 12

64-bit Linux

- Much larger entropy

ASLR in Windows

- Vista and Server 2008
- Stack randomization
  - Find Nth hole of suitable size (N is a 5-bit random value), then random word-aligned offset (9 bits of randomness)
- Heap randomization: 5 bits
  - Linear search for base + random 64K-aligned offset
- EXE randomization: 8 bits
  - Preferred base + random 64K-aligned offset
- DLL randomization: 8 bits
  - Random offset in DLL area; random loading order

ASLR in Practice

- Amount of randomness can differ between OSes
- Adoption was gradual
  - Not all code can be moved (relocated)