CSCI 4907/6545 Software Security
Fall 2025

Instructor: Jie Zhou

Department of Computer Science
George Washington University

Slides materials are partially credited to Gang Tan of PSU.

Announcements

- Assignment 1 due today
- Assignment 2 released
- (Mini-) Assignment 1.5: Prepare at least one question about anything covered so far for the next lecture

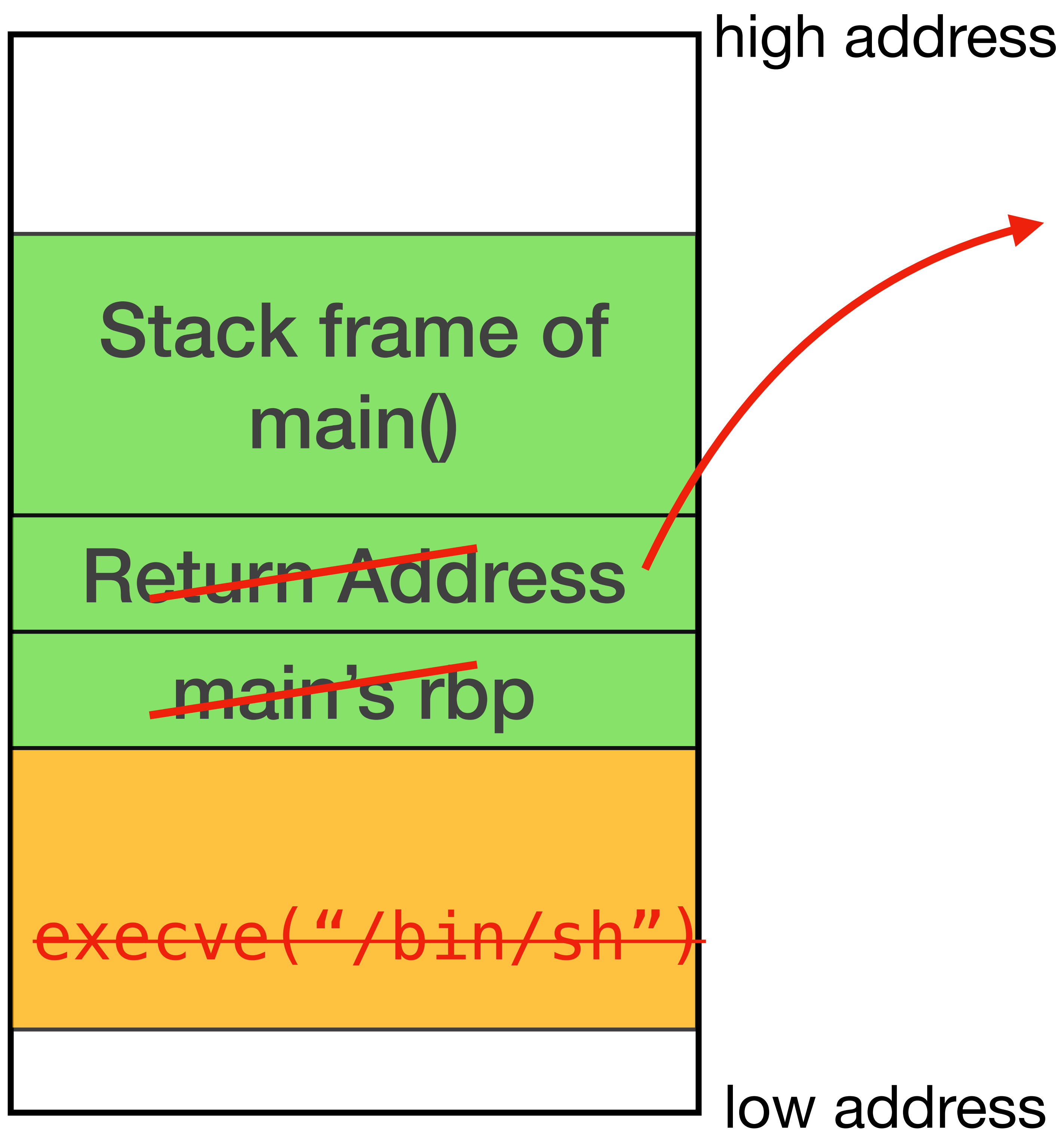
Outline

- Review: ROP, Integer Overflows, and Heap Overflows
- Temporal Memory Safety
- Format String Vulnerabilities

Limitations of `ret2libc` Attacks

- On AMD64 (and many other arch, e.g., AArch64), function arguments are first passed via registers instead of stack.
- Limited exploitable functions
  - `system()` and other “profitable” library functions could be removed.
- Can only execute straight-line code
  - Desired malicious computation may be invalidated by functions themselves.

Exploiting Existing and Executable Code



"return" to `system()`

```
(gdb) disassemble system
Dump of assembler code for function system:
   0x00007ffff7c50d70 <+0>:     endbr64
   0x00007ffff7c50d74 <+4>:     test   %rdi,%rdi
   0x00007ffff7c50d77 <+7>:     je     0x7ffff7c50d80 <system+16>
   0x00007ffff7c50d79 <+9>:     jmp    0x7ffff7c50900
   0x00007ffff7c50d7e <+14>:    xchg   %ax,%ax
   0x00007ffff7c50d80 <+16>:    sub    $0x8,%rsp
   0x00007ffff7c50d84 <+20>:    lea    0x1878f5(%rip),%rdi
   0x00007ffff7c50d8b <+27>:    call   0x7ffff7c50900
   0x00007ffff7c50d90 <+32>:    test   %eax,%eax
   0x00007ffff7c50d92 <+34>:    sete   %al
   0x00007ffff7c50d95 <+37>:    add    $0x8,%rsp
   0x00007ffff7c50d99 <+41>:    movzbl %al,%eax
   0x00007ffff7c50d9c <+44>:    ret
```

How about setting the argument and executing the same instructions from other places?

What really matters are the instructions and how they are arranged.

Return-oriented Programming (ROP)

An exploit technique that allows arbitrary code execution without calling any functions.

- Exploiting memory corruption bugs
  - Often starting with a corrupted return address
- Chaining code sequences, called *gadgets*, that end with a `ret`
  - Generally, gadgets ending with control flow transfer instructions, e.g. `jmp`
- Turing-complete
  - Memory operations
  - Arithmetic and logic
  - Control flow

Return-oriented Programming (ROP)

- Use `ret` to jump to the “profitable” instructions to the attacker’s interest

```
ret
```

⟺

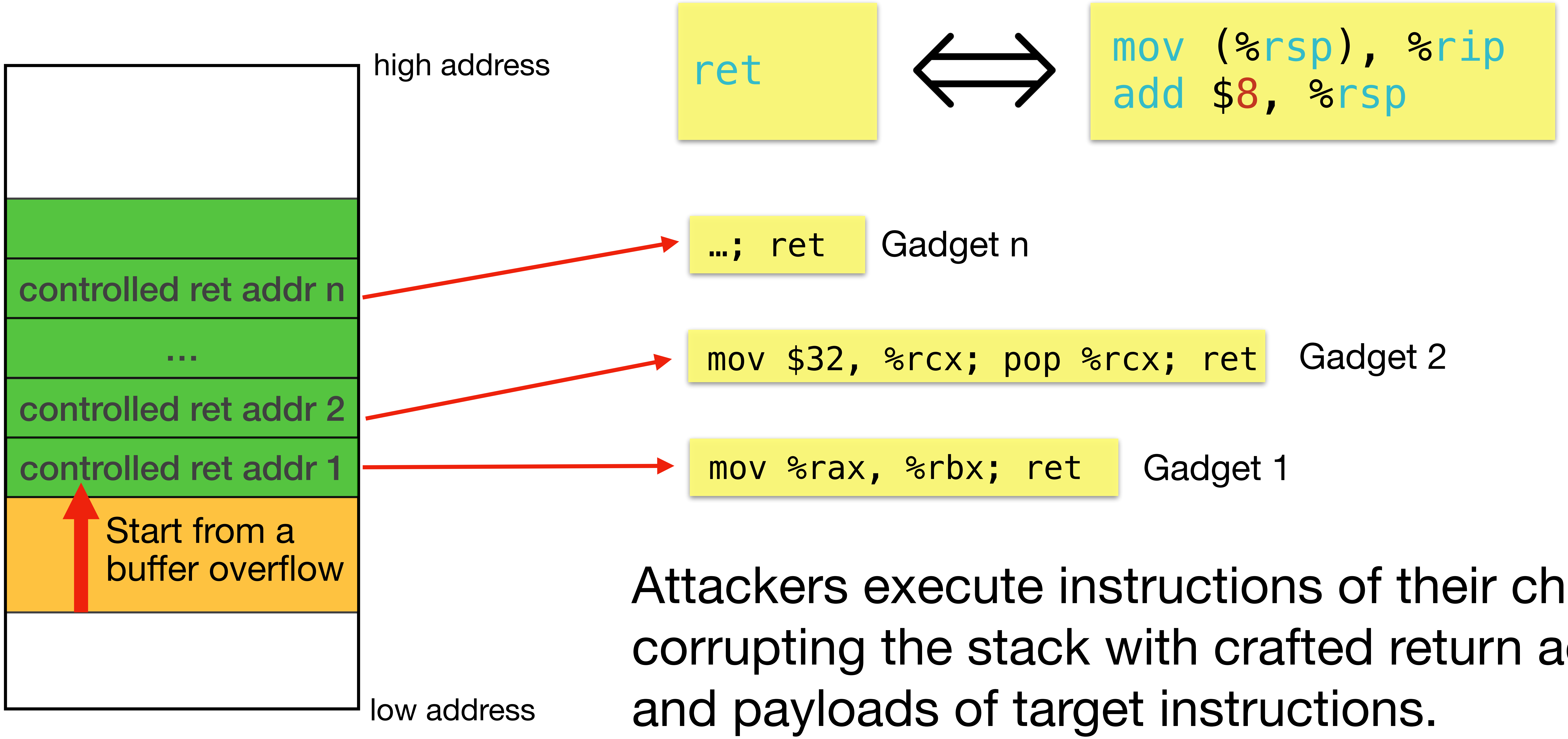
```
mov (%rsp), %rip
add $8, %rsp
```

Note this instruction is for demonstration. `rip` cannot be explicitly updated.

Controlling what `rsp` enables controlling the next to-be-executed instruction.

- Use `rsp` as a confused deputy for `rip`
  - Attackers use `rsp` to control the flow of the victim program.

Chaining Multiple ret

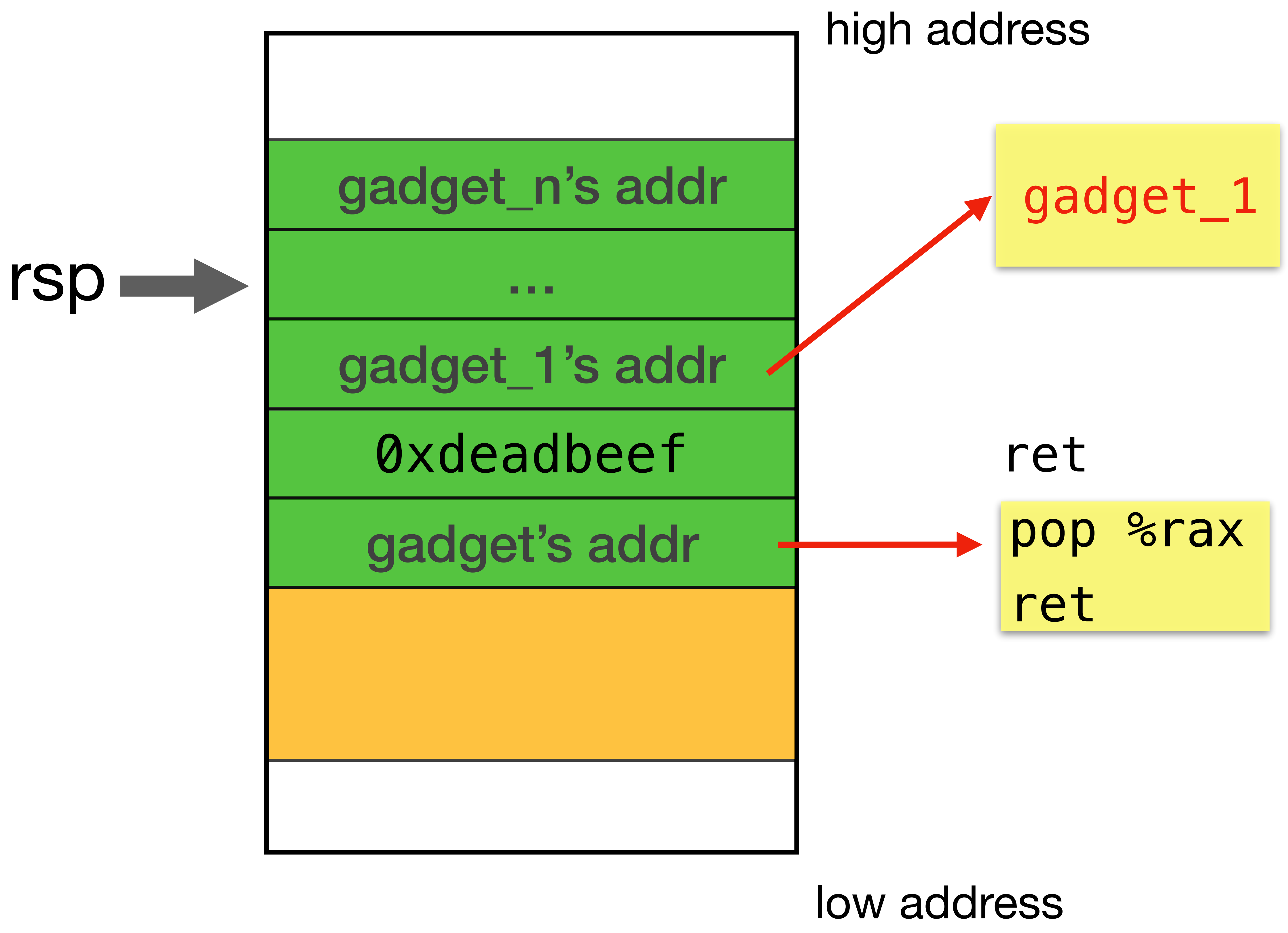


Attackers execute instructions of their choosing by corrupting the stack with crafted return addresses and payloads of target instructions.

Loading a Constant

How to load an arbitrary constant (e.g. `0xdeadbeef`) into a register?

- Option 1: Pop the constant to the target register

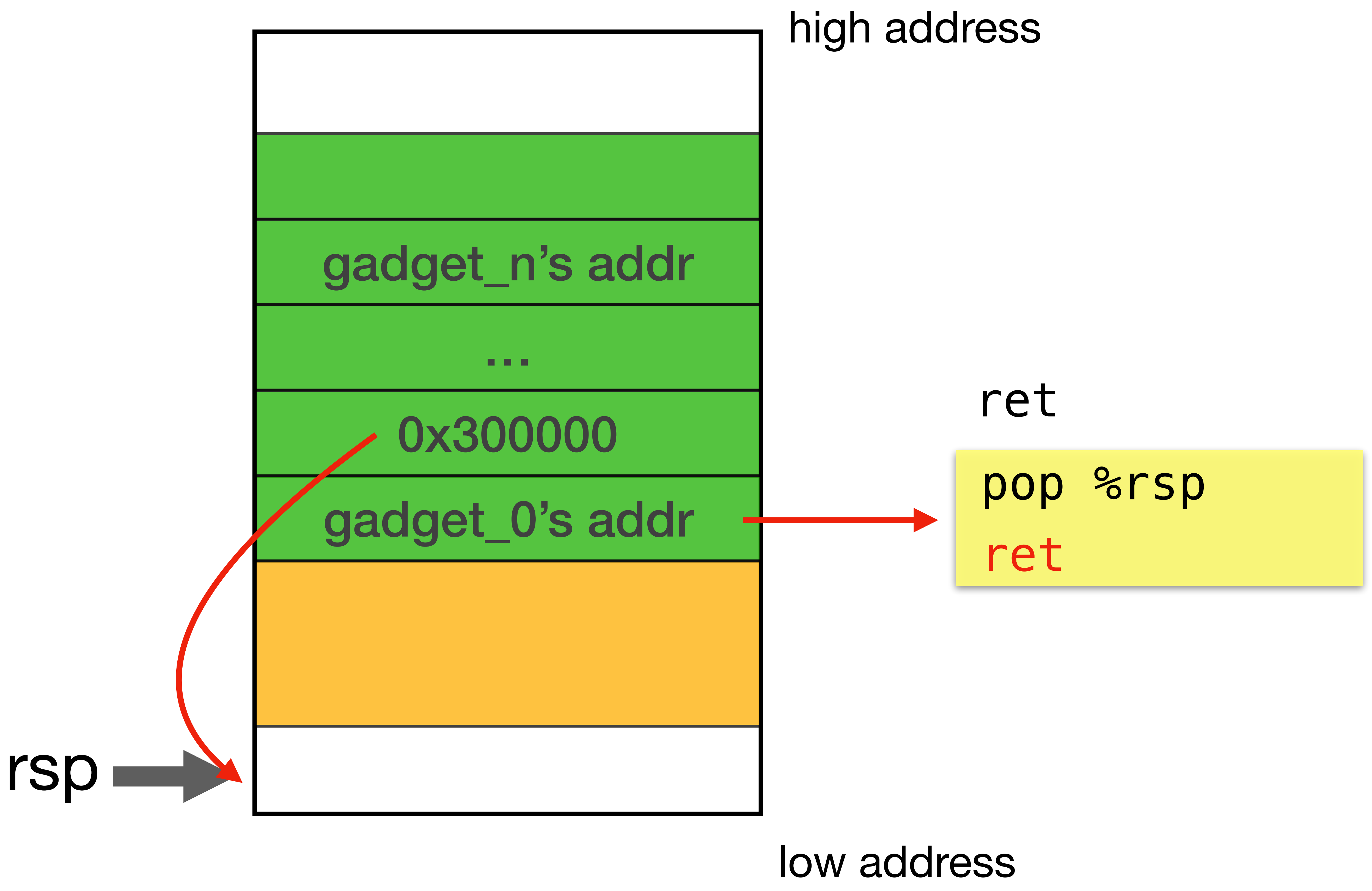


rax	0xdeadbeef
rsp	0x40018
rip	gadget_1’s address

Stack Pivoting

How to enable a larger “stack”?

- Pop the target address (e.g. `0x300000`) to `rsp`, and `ret`.



rsp	0x300000
rip	gadget ret’s address

`pop %rsp` is special:

- `rsp` gets incremented by one word.
- Data pointed by old `rsp` is loaded to `rsp`.

Finding ROP Gadgets

ROP gadgets: Instructions sequences ending with a `ret`.

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000001170 <+0>:     push   %rbp
   0x0000000000001171 <+1>:     mov    %rsp,%rbp
   0x0000000000001174 <+4>:     sub    $0x10,%rsp
   0x0000000000001178 <+8>:     movl   $0x0,-0x4(%rbp)
   0x000000000000117f <+15>:    movl   $0x0,-0x8(%rbp)
   0x0000000000001186 <+22>:    mov    $0x1,%edi
   0x000000000000118b <+27>:    mov    $0x2,%esi
   0x0000000000001190 <+32>:    call   0x1150 <foo>
   0x0000000000001195 <+37>:    movl   $0x1,-0x8(%rbp)
   0x000000000000119c <+44>:    mov    -0x8(%rbp),%esi
   0x000000000000119f <+47>:    lea    0xe5e(%rip),%rdi
   0x00000000000011a6 <+54>:    mov    $0x0,%al
   0x00000000000011a8 <+56>:    call   0x1030 <printf@plt>
   0x00000000000011ad <+61>:    xor    %eax,%eax
   0x00000000000011af <+63>:    add    $0x10,%rsp
   0x00000000000011b3 <+67>:    pop    %rbp
   0x00000000000011b4 <+68>:    ret
```

```
Dump of assembler code for function foo:
   0x0000000000001150 <+0>:     push   %rbp
   0x0000000000001151 <+1>:     mov    %rsp,%rbp
   0x0000000000001154 <+4>:     sub    $0x20,%rsp
   0x0000000000001158 <+8>:     mov    %edi,-0x4(%rbp)
   0x000000000000115b <+11>:    mov    %esi,-0x8(%rbp)
   0x000000000000115e <+14>:    lea    -0x14(%rbp),%rdi
   0x0000000000001162 <+18>:    mov    $0x0,%al
   0x0000000000001164 <+20>:    call   0x1040 <gets@plt>
   0x0000000000001169 <+25>:    add    $0x20,%rsp
   0x000000000000116d <+29>:    pop    %rbp
   0x000000000000116e <+30>:    ret
```

How many gadgets can you find in these two functions?

ROP Gadgets Are Abundant

ROP gadgets: Instructions sequences ending with a `ret`.

- Linked libraries provide a plethora of instructions.
- x86 ISA uses variable-length instructions.
  - Allows *unintended* instruction sequences

ret is encoded as 0xc3 in hexadecimal format.

```
f7 c7 07 00 00 00      test $0x00000007, %edi
0f 95 45 c3            setnzb -61(%ebp)
```

Starting one byte later, the attacker instead obtains

```
c7 07 00 00 00 0f      movl $0x0f000000, (%edi)
95                     xchg %ebp, %eax
45                     inc %ebp
c3                     ret
```

ROP Thesis

The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86), by Hovav Shacham.

“In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake ***arbitrary*** *computation.”*

☹️ Also true in almost all other major architectures.

Find ROP Gadgets

ROPGagdet: A tool that examines binaries to find code-reuse gadgets.

```c
void foo(int a, int b) {
      char buffer[12];
      gets(buffer);
      return;
}

int main() {
      int x;
      x = 0;
      foo(1,2);
      x = 1;
      printf("%d\n",x);
      return 0;
}
```

How many ret gadgets in this program?

```
$ ROPgadget --binary demo | grep ret
4:0x00000000000010b3 : add byte ptr [rax], 0 ; add byte ptr [rax], al ; ret
7:0x00000000000010b4 : add byte ptr [rax], al ; add byte ptr [rax], al ; ret
8:0x0000000000001130 : add byte ptr [rax], al ; add dword ptr [rbp - 0x3d], ebx ; nop dword ptr [rax] ; ret
13:0x00000000000010b6 : add byte ptr [rax], al ; ret
17:0x00000000000010f5 : add byte ptr [rax], r8b ; ret
18:0x0000000000001131 : add byte ptr [rcx], al ; pop rbp ; ret
20:0x0000000000001132 : add dword ptr [rbp - 0x3d], ebx ; nop dword ptr [rax] ; ret
21:0x000000000000112e : add eax, 0x100002f ; pop rbp ; ret
23:0x00000000000011b0 : add esp, 0x10 ; pop rbp ; ret
24:0x000000000000116a : add esp, 0x20 ; pop rbp ; ret
25:0x0000000000001017 : add esp, 8 ; ret
26:0x00000000000011af : add rsp, 0x10 ; pop rbp ; ret
27:0x0000000000001169 : add rsp, 0x20 ; pop rbp ; ret
28:0x0000000000001016 : add rsp, 8 ; ret
31:0x00000000000011bb : cli ; sub rsp, 8 ; add rsp, 8 ; ret
42:0x00000000000010f1 : loopne 0x1159 ; nop dword ptr [rax + rax] ; ret
43:0x000000000000112c : mov byte ptr [rip + 0x2f05], 1 ; pop rbp ; ret
44:0x00000000000010f3 : nop dword ptr [rax + rax] ; ret
45:0x00000000000010b1 : nop dword ptr [rax] ; ret
46:0x00000000000010f2 : nop word ptr [rax + rax] ; ret
47:0x00000000000010ef : or bh, bh ; loopne 0x1159 ; nop dword ptr [rax + rax] ; ret
48:0x0000000000001133 : pop rbp ; ret
51:0x000000000000101a : ret
52:0x0000000000001011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
53:0x00000000000011bd : sub esp, 8 ; add rsp, 8 ; ret
54:0x00000000000011bc : sub rsp, 8 ; add rsp, 8 ; ret
61:0x00000000000011ad : xor eax, eax ; add rsp, 0x10 ; pop rbp ; ret
```

Integer Overflows

An integer overflow occurs when an integer is increased beyond its maximum value or decreased beyond its minimum value.

- Standard integer types (signed)
  - signed char, short int, int, long int, long long int
- Signed overflow vs unsigned overflow
  - A signed overflow occurs when a value is carried over to the sign bit.
  - An unsigned overflow occurs when the underlying representation can no longer represent an integer value.

Integer Overflow Examples

```
unsigned int ui;
signed int si;
ui = UINT_MAX; // 2^32 - 1 = 4,294,967,295
ui++;
printf("ui = %u\n", ui);

si = INT_MAX; // 2^31 - 1 = 2,147,483,647
si++;
printf("si = %d\n", si);
```

What does it print?

0

What does it print?

-2^31 = -2,147,483,648

Integer Overflow Examples

```
unsigned int ui;
signed int si;
ui = 0;
ui--;
printf("ui = %u\n", ui);

si = INT_MIN; // -2^31 = -2,147,483,648
si--;
printf("si = %d\n”, si);
```

What does it print?

2^32 - 1 = 4,294,967,295

What does it print?

2^31 - 1= 2,147,483,647

Buffer Overflows

- Stack overflow: overflowing a memory region on the stack (e.g., overwriting a return address)
- Heap overflow: overflowing a memory region dynamically allocated on the heap

Overflowing Heap Critical User Data

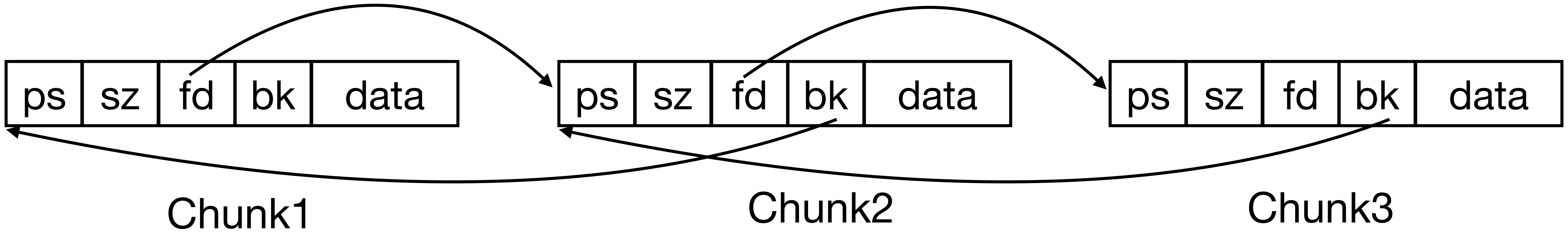
```c
typedef struct chunk {
    char inp[64];            /* vulnerable input buffer */
    void (*process)(char *); /* pointer to function */
} chunk_t;

void showlen(char *buf) {
    int len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[]) {
    chunk_t *next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

Overflow the buffer on the heap to set the function pointer to an arbitrary address.

Overflow Heap Metadata

- Heap allocators (i.e., heap memory managers)
  - What regions have been allocated and their sizes
  - What regions are available for allocation
- Heap allocators maintain metadata such as chunk size, previous, and next pointers to other chunks.
  - Metadata are adjusted during heap-management functions.
    - `malloc()`, `callaoc()`, `realloc()`, etc. and `free()`
  - Heap metadata are often adjacent to heap user data

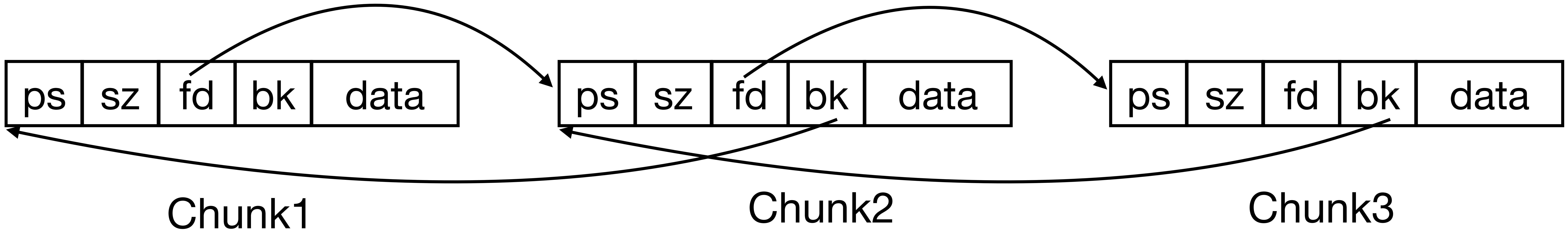
Example Heap Allocator

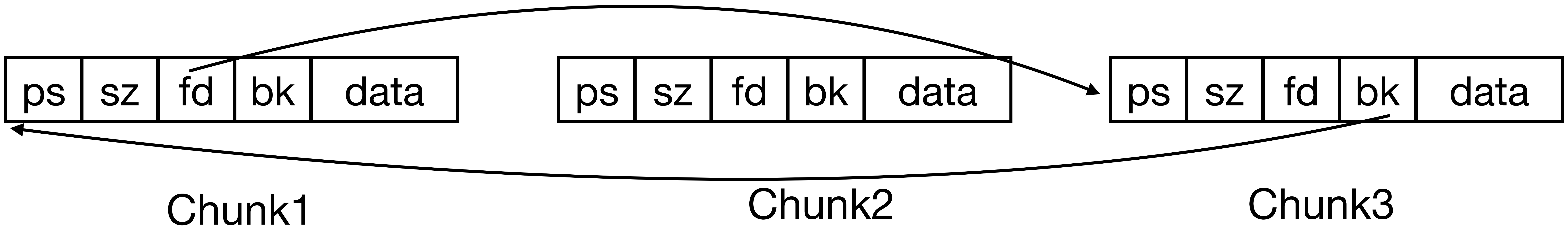


- ps: prev_size
- sz: size
- fd: forward pointer
- bk: backward pointer
- data: allocated space for user data

```
struct chunk {
    ……  // Other fields
    size_t prev_size;   // Size of the previous chunk
    size_t size;        // Size of the current chunk
    struct chunk *fd;   // Pointer to the next chunk
    struct chunk *bk;   // Pointer to the previous chunk
}
```

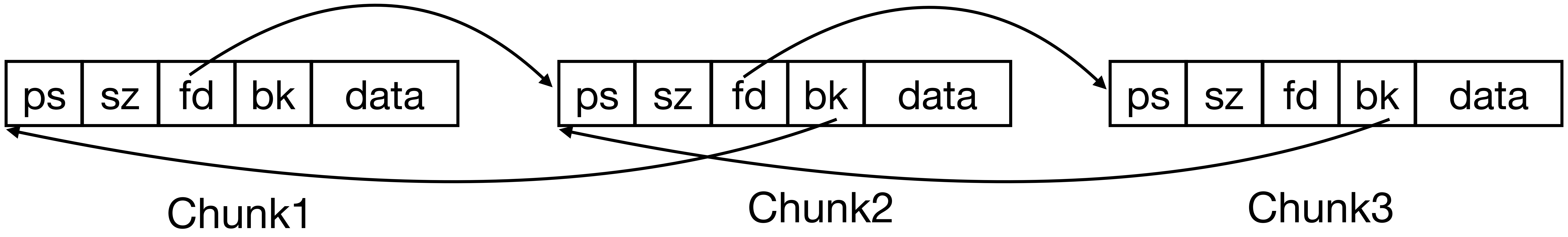
Example Heap Allocator





- `malloc()` removes a chunk from free list
  - `chunk2->bk->fd = chunk2->fd`
  - `chunk2->fd->bk = chunk2->bk`

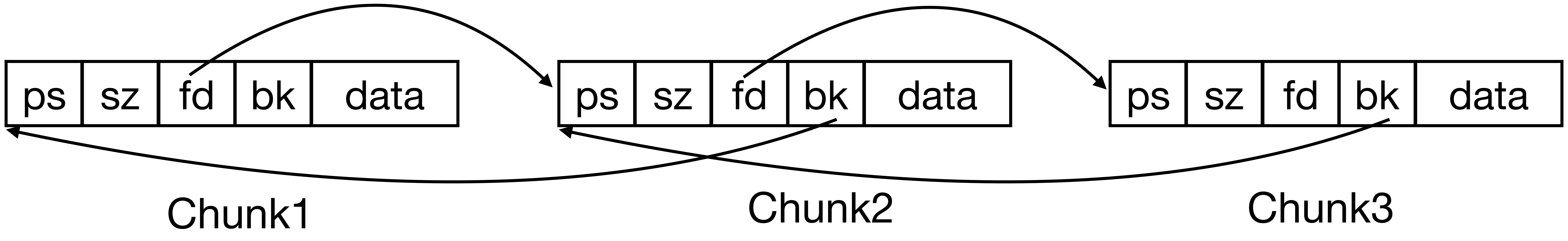
Attacking the Example Heap Allocator



- `malloc()` removes a chunk from allocated list
  - `chunk2->bk->fd = chunk2->fd`
  - `chunk2->fd->bk = chunk2->bk`
- By overflowing `chunk2`, attacker controls `bk` and `fd` of `chunk2`

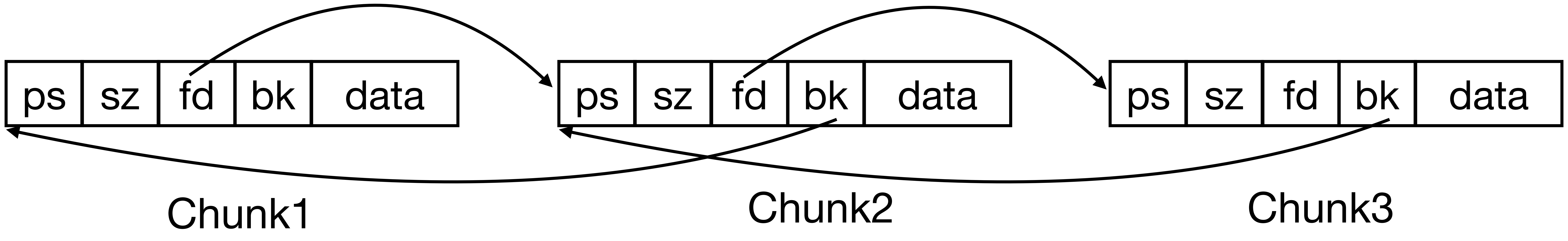
How to exploit this vulnerability for arbitrary writes (write-where-what vul.)?

Attacking the Example Heap Allocator



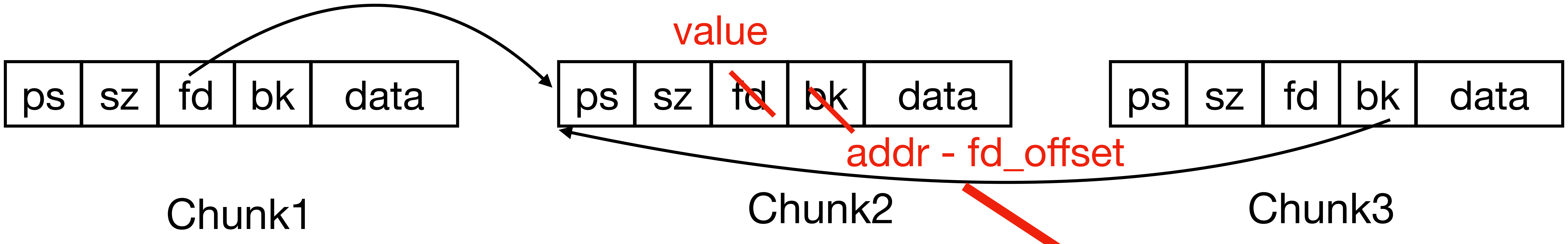
- `malloc()` removes a chunk from allocated list
  - `chunk2->bk->fd = chunk2->fd`
  - `chunk2->fd->bk = chunk2->bk`
- By overflowing `chunk2`, attacker controls `bk` and `fd` of `chunk2`
- Suppose the attacker wants to write `value` to memory address `addr`
  - Set `chunk2->fd` to be `value`
  - Set `chunk2->bk` to be `addr - fd_offset`, where `fd_offset` is the offset of the `fd` field in the chunk structure.

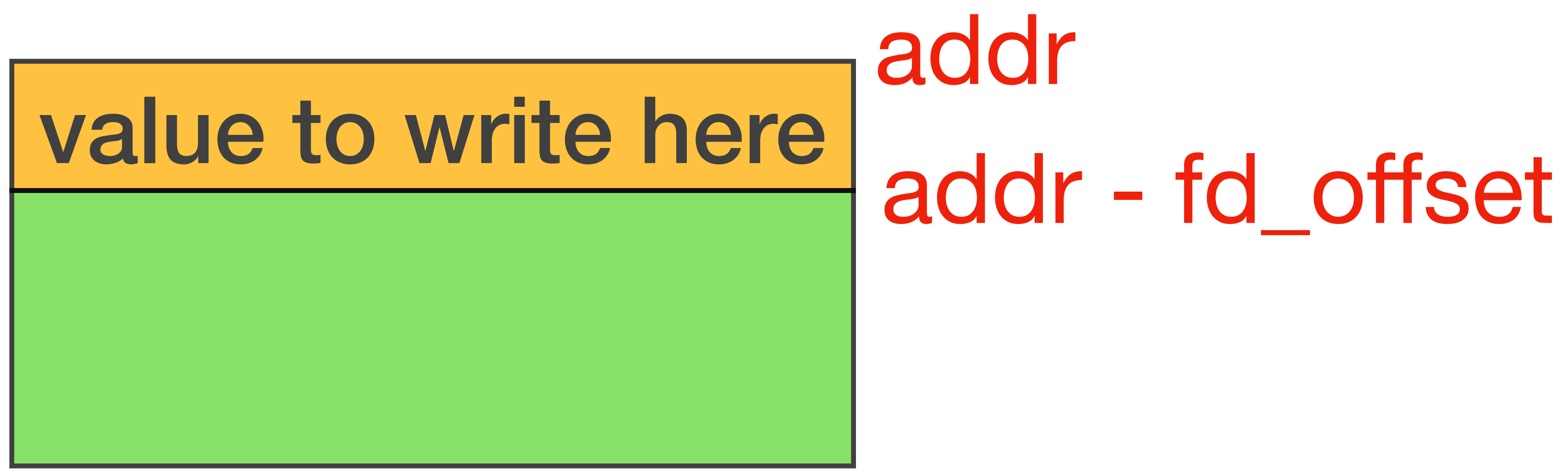
Attacking the Example Heap Allocator



- `malloc()` removes a chunk from allocated list
  - `chunk2->bk->fd = chunk2->fd`
  - `chunk2->fd->bk = chunk2->bk`
- By overflowing `chunk2`, attacker controls `bk` and `fd` of `chunk2`
- `malloc()` changes the program as follows:
  - `(addr - fd_offset)->fd = value`, the same as `(*addr) = value`
  - `value->bk = addr - offset`

Attacking the Example Heap Allocator





- `malloc()` removes a chunk from allocated list
  - `chunk2->bk->fd = chunk2->fd`
  - `chunk2->fd->bk = chunk2->bk`
- By overflowing `chunk2`, attacker controls `bk` and `fd` of `chunk2`
- `malloc()` changes the program as follows:
  - `(addr - fd_offset)->fd = value`, the same as `(*addr) = value`
  - `value->bk = addr - offset`

Enables arbitrary memory write!

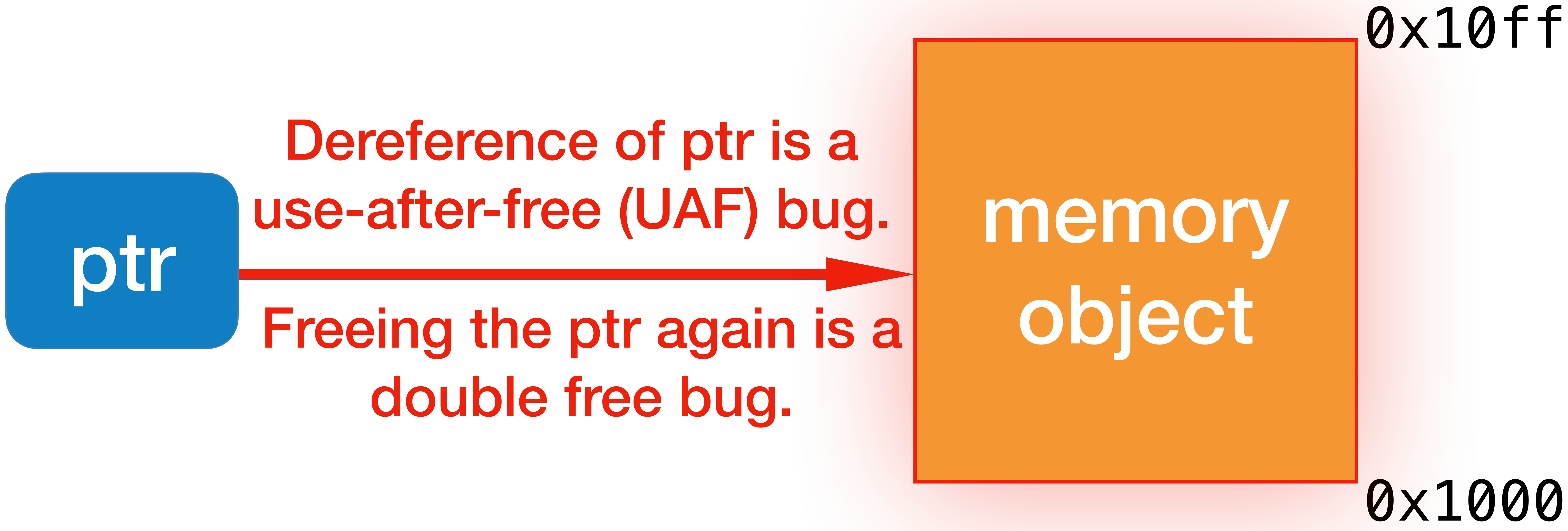
Temporal Memory Safety

Memory Management

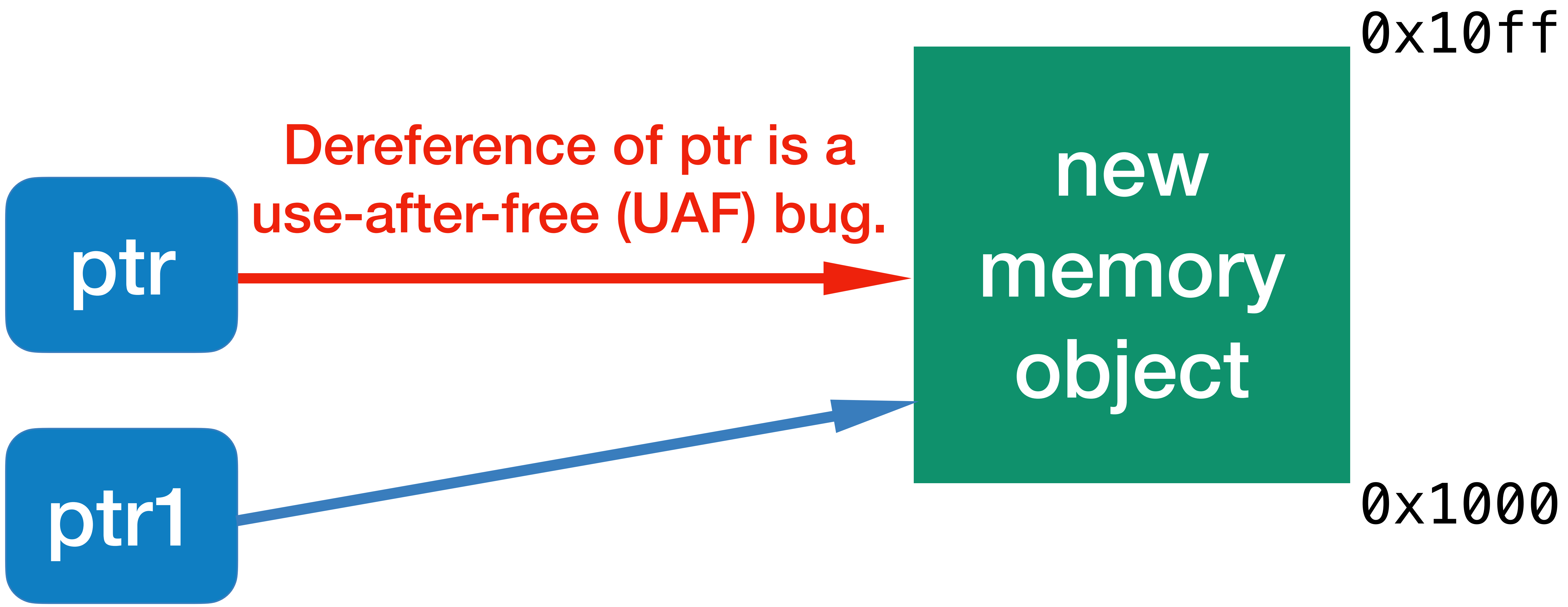
How does C/C++ manages memory?

- Global data: reserve space during program initialization; never free
- Stack: automatically allocated/deallocated at function start/end
- Heap: manual management, i.e., explicit allocations/deallocations
    - C++ supports partially automatic memory management (RAII)

Temporal Memory Safety Bugs



Temporal Memory Safety Bugs



Security risks

- ⚠ Information leaking
- ⚠ Data corruption
- ⚠ Denial of service

Temporal Memory Safety Vulnerabilities are Severe

WebKit Process Model

Available for: iPhone 11 and later, iPad Pro 12.9-inch 3rd generation and later, iPad Pro 11-inch 1st generation and later, iPad Air 3rd generation and later, iPad 8th generation and later, and iPad mini 5th generation and later

Impact: Processing maliciously crafted web content may lead to an unexpected Safari crash

Description: A use-after-free issue was addressed with improved memory management.

WebKit Bugzilla: 296276

CVE-2025-43368: Pawel Wylecial of REDTEAM.PL working with Trend Micro Zero Day Initiative

https://support.apple.com/en-us/125108

Temporal Memory Safety Vulnerabilities are Severe

Kernel

Available for: Mac Studio (2022 and later), iMac (2019 and later), Mac Pro (2019 and later) …

Impact: An app may be able to execute arbitrary code with kernel privileges

Description: A use-after-free issue was addressed with improved memory management.

CVE-2023-41995: Certik Skyfall Team, and pattern-f (@pattern_F_) of Ant Security Light-Year Lab

CVE-2023-42870: Zweig of Kunlun Lab

https://support.apple.com/en-us/HT213940

Use After Free

cwe.mitre.org/top25/archive/2024/2024_kev_list.html

Common Weakness Enumeration

A community-developed list of SW & HW weaknesses that can become vulnerabilities

Top 25

Home > CWE Top 25 > 2024 CWE Top 10 KEV Weaknesses

Home | About ▼ | Learn ▼ | Access Content ▼ | Community ▼ | Search ▼

2024 CWE Top 10 KEV Weaknesses

Top 25 Home | Share via: X | View in table format | KEV Key Insights | KEV Methodology

1 Out-of-bounds Write
CWE-787 | CVEs in KEV: 18 | Rank Last Year: 3 (up 2) ▲

2 Access of Resource Using Incompatible Type ('Type Confusion')
CWE-843 | CVEs in KEV: 6 | Rank Last Year: 8 (up 6) ▲

3 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
CWE-78 | CVEs in KEV: 6 | Rank Last Year: 5 (up 2) ▲

4 Improper Control of Generation of Code ('Code Injection')
CWE-94 | CVEs in KEV: 7 | Rank Last Year: 33 (up 29) ▲

5 Deserialization of Untrusted Data
CWE-502 | CVEs in KEV: 5 | Rank Last Year: 6 (up 1) ▲

6 Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
CWE-22 | CVEs in KEV: 5 | Rank Last Year: 9 (up 3) ▲

7 Missing Authentication for Critical Function
CWE-306 | CVEs in KEV: 6 | Rank Last Year: 10 (up 3) ▲

8 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-89 | CVEs in KEV: 4 | Rank Last Year: 11 (up 3) ▲

9 Use After Free
CWE-416 | CVEs in KEV: 5 | Rank Last Year: 1 (down 8) ▼

10 Improper Neutralization of Special Elements used in a Command ('Command Injection')
CWE-77 | CVEs in KEV: 4 | Rank Last Year: 15 (up 5) ▲

Common Weakness Enumeration

A Community-Developed List of Software & Hardware Weakness Types

Home > CWE Top 25 > 2023 CWE Top 10 KEV Weaknesses

Home | About

2023 CWE Top 10 KEV Weaknesses

Top 25 Home | Share via: | View in table format | KEV Key Insights | KEV Methodology

1 Use After Free
CWE-416 | Analysis score: 73.99 | # CVE Mappings in KEV: 44 | Avg. CVSS: 8.54

2 Heap-based Buffer Overflow
CWE-122 | Analysis score: 56.56 | # CVE Mappings in KEV: 32 | Avg. CVSS: 8.79

3 Out-of-bounds Write
CWE-787 | Analysis score: 51.96 | # CVE Mappings in KEV: 34 | Avg. CVSS: 8.19

4 Improper Input Validation
CWE-20 | Analysis score: 51.38 | # CVE Mappings in KEV: 33 | Avg. CVSS: 8.27

5 Improper Neutralization of Special Elements used in an OS Command
CWE-78 | Analysis score: 49.44 | # CVE Mappings in KEV: 25 | Avg. CVSS: 9.36

Use-After-Free (UAF)

Program frees memory then references that memory as if it were still valid.

- Adversaries can control data written using the freed pointer.
- AKA, use of *dangling pointers*

Use-After-Free (UAF)

```
int main(int argc, char **argv) {
    char *buf1, *buf2;

    buf1 = (char *) malloc(BUFSIZE1);
    free(buf1);
    buf2 = (char *) malloc(BUFSIZE2);
    strncpy(buf1, argv[1], BUFSIZE1-1);
    …
}
```

What is wrong with this program?

- When the first buffer is freed, that memory is available for reuse right away.
- Then, the following buffers are possibly allocated within that memory region.
- Finally, the write using the freed pointer may overwrite `buf2` (and its metadata).

Use-After-Free (UAF)

- Most effective attacks exploit data of another type.

```
struct A {
    void (*fnptr)(char *arg);
    char *buf;
};

struct B {
    long int B1;
    long int B2;
    char info[32];
};
```

```
x = (struct A *)malloc(sizeof(struct A));
free(x);
y = (struct B *)malloc(sizeof(struct B));
```

Free A, and allocate B.
What might happen?

Overflowing Heap Critical User Data

```
typedef struct chunk {
    char inp[64];            /* vulnerable input buffer */
    void (*process)(char *); /* pointer to function */
} chunk_t;

void showlen(char *buf) {
    int len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[]) {
    chunk_t *next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

Overflow the buffer on the heap to set the function pointer to an arbitrary address.

Use-After-Free (UAF)

- Most effective attacks exploit data of another type.

```
struct A {
    void (*fnptr)(char *arg);
    char *buf;
};

struct B {
    long int B1;
    long int B2;
    char info[32];
};
```

```
x = (struct A *)malloc(sizeof(struct A));
free(x);
y = (struct B *)malloc(sizeof(struct B));
```

```
y->B1 = 0xDEADBEEF;
x->fnptr(x->buf);
```

- Assume that
  - Attackers control what to write to y->B1
  - A later UAF that performs a call using `x->fnptr`
- One of the most commonly exploited patterns.

Vulnerabilities Caused By UAF

- General pattern of UAF vulnerabilities:
  - A new heap object N is allocated over the heap location previously occupied by an freed object O.
  - Pointer p points to and is used to access N.
  - Pointer q points to and was used to access O.
  - p and q are both used to access the same memory region.
    - Attackers control q to access N.
    - Attackers control N and wait for q to access N.
- Consequences
  - Arbitrary code execution
  - Information leak
  - Data corruption

Vulnerabilities Caused By UAF

```
1 struct N { long usr; long pwd; int (*fn)(void); };
2 struct O { int (*oper)(void); long u1; long u2; };
3
4 void foo(long uid, long secret) {
5     struct N *p = malloc(sizeof(struct N));
6     p->fn = __safe_function_1;
7     p->usr = uid;
8     p->pwd = secret;
9     p->fn();
10 }
11
12 void bar(long user1, long user2) {
13     struct O *x = malloc(sizeof(struct O));
14     x->oper = __safe_function_2;
15     struct O *q = x;
16     free(x);
17     q->oper();
18     q->u1 = user1;
19     q->u2 = user2;
20     reply("Users: %l | %l", q->u1, q->u2);
21     free(q);
22 }
```

- Write through p and read through q leads to arbitrary code execution.
  - exploit path: 16->5->7->17

Vulnerabilities Caused By UAF

```
1 struct N { long usr; long pwd; int (*fn)(void); };
2 struct O { int (*oper)(void); long u1; long u2; };
3
4 void foo(long uid, long secret) {
5     struct N *p = malloc(sizeof(struct N));
6     p->fn = __safe_function_1;
7     p->usr = uid;
8     p->pwd = secret;
9     p->fn();
10 }
11
12 void bar(long user1, long user2) {
13     struct O *x = malloc(sizeof(struct O));
14     x->oper = __safe_function_2;
15     struct O *q = x;
16     free(x);
17     q->oper();
18     q->u1 = user1;
19     q->u2 = user2;
20     reply("Users: %l | %l", q->u1, q->u2);
21     free(q);
22 }
```

- Read through q leads to information leak.
    - exploit path: 16->5->6->8->20

Vulnerabilities Caused By UAF

```
1 struct N { long usr; long pwd; int (*fn)(void); };
2 struct O { int (*oper)(void); long u1; long u2; };
3
4 void foo(long uid, long secret) {
5     struct N *p = malloc(sizeof(struct N));
6     p->fn = __safe_function_1;
7     p->usr = uid;
8     p->pwd = secret;
9     p->fn();
10 }
11
12 void bar(long user1, long user2) {
13     struct O *x = malloc(sizeof(struct O));
14     x->oper = __safe_function_2;
15     struct O *q = x;
16     free(x);
17     q->oper();
18     q->u1 = user1;
19     q->u2 = user2;
20     reply("Users: %l | %l", q->u1, q->u2);
21     free(q);
22 }
```

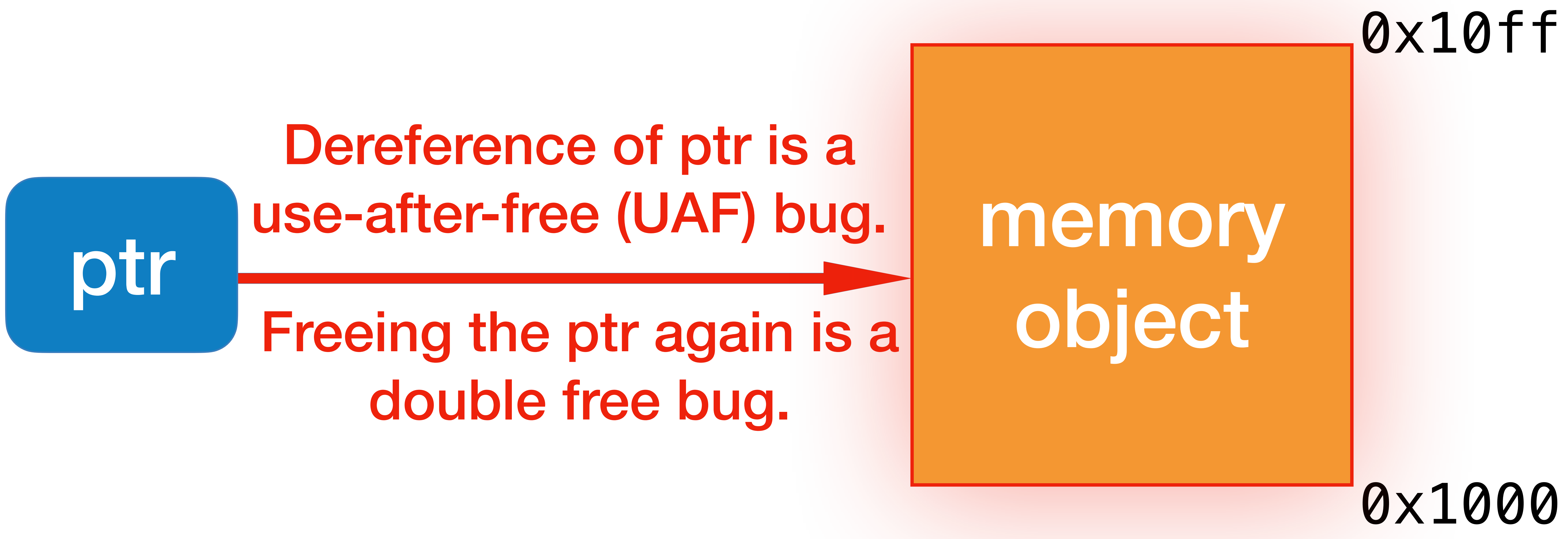
- Write through q and then read through p leads to arbitrary code execution.
    - exploit path: 16->5->19->9

Vulnerabilities Caused By UAF

```
1 struct N { long usr; long pwd; int (*fn)(void); };
2 struct O { int (*oper)(void); long u1; long u2; };
3
4 void foo(long uid, long secret) {
5     struct N *p = malloc(sizeof(struct N));
6     p->fn = __safe_function_1;
7     p->usr = uid;
8     p->pwd = secret;
9     p->fn();
10 }
11
12 void bar(long user1, long user2) {
13     struct O *x = malloc(sizeof(struct O));
14     x->oper = __safe_function_2;
15     struct O *q = x;
16     free(x);
17     q->oper();
18     q->u1 = user1;
19     q->u2 = user2;
20     reply("Users: %l | %l", q->u1, q->u2);
21     free(q);
22 }
```

- Free through q corrupts N's state.
  - exploit path: 16->5->21

Temporal Memory Safety Bugs



Double Free

```
int main(int argc, char **argv) {
    char *buf1, *buf2;
    buf1 = (char *) malloc(BUFSIZE1);
    free(buf1);
    buf2 = (char *) malloc(BUFSIZE2);
    strncpy(buf1, argv[1], BUFSIZE1-1);
    free(buf1);
    free(buf2);
}
```

What happens here?

Overflow Heap Metadata

- Heap allocators (i.e., heap memory managers)
  - What regions have been allocated and their sizes
  - What regions are available for allocation
- Heap allocators maintain metadata such as chunk size, previous, and next pointers to other chunks.
  - **Metadata are adjusted during heap-management functions.**
    - `malloc()`, `callaoc()`, `realloc()`, etc. and `free()`
  - Heap metadata are often adjacent to heap user data

Double Free

```
int main(int argc, char **argv) {
    char *buf1, *buf2;
    buf1 = (char *) malloc(BUFSIZE1);
    free(buf1);
    buf2 = (char *) malloc(BUFSIZE2);
    strncpy(buf1, argv[1], BUFSIZE1-1);
    free(buf1);
    free(buf2);
}
```

What happens here?

- Free `buf1`, then allocate `buf2`
  - `buf2` may occupy the same memory space of `buf1`.
- `buf2` gets user-supplied data
- Free `buf1` again
  - Which may use some `buf2` data as metadata
  - And may mess up `buf2`'s metadata
- Then free `buf2`, which uses really messed up metadata

Vulnerabilities Caused By UAF

```
1 struct N { long usr; long pwd; int (*fn)(void); };
2 struct O { int (*oper)(void); long u1; long u2; };
3
4 void foo(long uid, long secret) {
5     struct N *p = malloc(sizeof(struct N));
6     p->fn = __safe_function_1;
7     p->usr = uid;
8     p->pwd = secret;
9     p->fn();
10 }
11
12 void bar(long user1, long user2) {
13     struct O *x = malloc(sizeof(struct O));
14     x->oper = __safe_function_2;
15     struct O *q = x;
16     free(x);
17     q->oper();
18     q->u1 = user1;
19     q->u2 = user2;
20     reply("Users: %l | %l", q->u1, q->u2);
21     free(q);
22 }
```

- Double free
  - exploit path: 16->21

Pitfalls of `realloc`

```
void *realloc(void *ptr, size_t size);
```

- Change the size of object pointed by `ptr` to size

```
/* p is a pointer to dynamically allocated memory. */
void func(void *p, size_t size) {
  void *p2 = realloc(p, size);
  if (p2 == NULL) {
    free(p);
    return;
  }
}
```

When `size == 0`, `realloc()` frees p.

Vulnerabilities Caused By UAF

- General pattern of UAF vulnerabilities:
  - A new heap object N is allocated over the heap location previously occupied by an freed object O.
  - Pointer p points to and is used to access N.
  - Pointer q points to and was used to access O.
  - p and q are both used to access the same memory region.
    - Attackers control q to access N.
    - **Attackers control N and wait for q to access N.**
- Consequences

Precisely controlling victim memory can be challenging.

  - Arbitrary code execution
  - Information leak
  - Data corruption

Vulnerabilities Caused By UAF

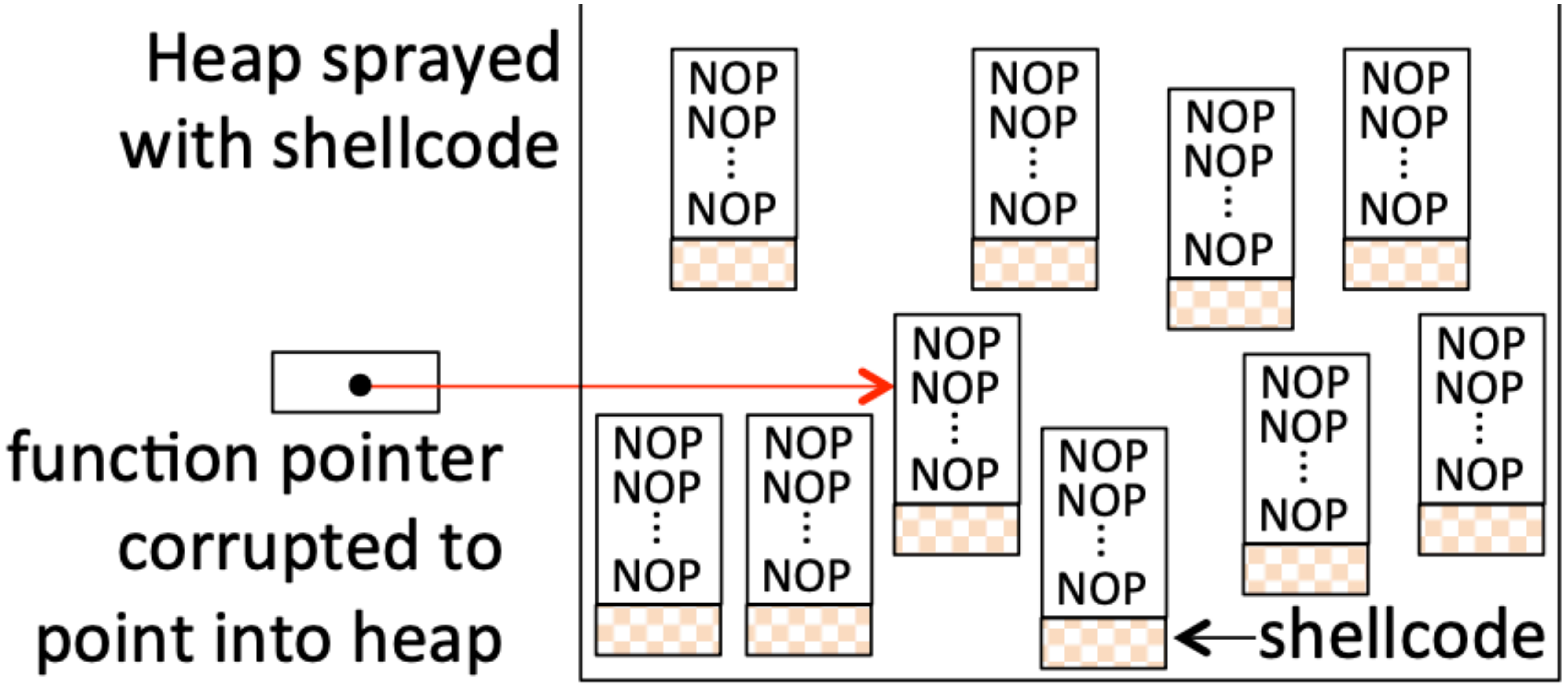
```
 1 struct N { long usr; long pwd; int (*fn)(void); };
 2 struct O { int (*oper)(void); long u1; long u2; };
 3
 4 void foo(long uid, long secret) {
 5     struct N *p = malloc(sizeof(struct N));
 6     p->fn = __safe_function_1;
 7     p->usr = uid;
 8     p->pwd = secret;
 9     p->fn();
10 }
11
12 void bar(long user1, long user2) {
13     struct O *x = malloc(sizeof(struct O));
14     x->oper = __safe_function_2;
15     struct O *q = x;
16     free(x);
17     q->oper();
18     q->u1 = user1;
19     q->u2 = user2;
20     reply("Users: %l | %l", q->u1, q->u2);
21     free(q);
22 }
```

- Write through p and read through q leads to arbitrary code execution.
    - exploit path: 16->5->7->17

Assume the attacker controls N. “Tricking” `bar()` to execute line 17 that calls a function whose address falls exactly to the first field of N can be challenging.

Heap Spraying

An exploitation technique that attempts to put a sequence of bytes on the heap to increase the likelihood of the victim program using these attacker-supplied bytes.



Root Cause

Manual **Memory Management**

thttpd: A Lightweight HTTP Server Written in C

SLOC: 8,360

Call Graph of `thttpd`

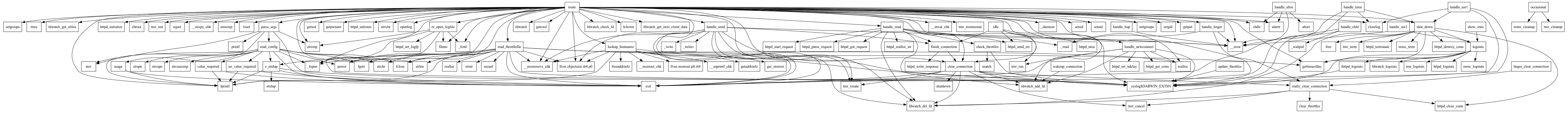
thttpd: A Lightweight HTTP Server Written in C

SLOC: 8,360

Call Graph of thttpd

`thttpd`: A Lightweight HTTP Server Written in C

SLOC: 8,360



Call Graph of `thttpd`

Manually manage memory?

Prevent Temporal Memory Safety Bugs

- Difficult to detect because these often occur in complex runtime states
  - Allocate in one function
  - Free in another function
  - Use in a third function
- It is not fun to check source code for all possible pointers.
  - Are all uses accessing valid (not freed) references?
  - In all possible runtime states!

Prevent Temporal Memory Safety Bugs

- Static and dynamic analysis to detect bugs
- Invalidate dangling pointers
- Minimize reuse of memory
- Runtime check on every memory dereference

Format String Vulnerabilities

Background: Variadic Functions

A function that accepts a *variable* number of arguments.

- Notable examples include `printf` family of functions in libc.
  - `printf`, `fprintf`, `sprintf`, `vprintf`, etc.
- Libc provides facilities to define your own variadic functions, which set of arguments followed by an optional list of additional arguments.
  - `va_list`: a special type that acts like a pointer/cursor to walk through args.
  - `va_start()`: initializes va_list to point to the first arg after the fixed args
  - `va_arg()`: fetches the next argument in the list
  - `va_end()`: signals that there are no more arguments.

Background: Variadic Functions

```
#include <stdarg.h>
#include <stdio.h>

double average(int count, ...) {
    va_list ap;
    double sum = 0;

    va_start(ap, count);
    for (int j = 0; j < count; ++j) {
        sum += va_arg(ap, int); /* Increments ap to the next argument. */
    }
    va_end(ap);

    return sum / count;
}

int main(int argc, char* argv[]) {
    printf("%f\n", average(3, 1, 2, 3));
    return 0;
}
```

What if a wrong type is provided?

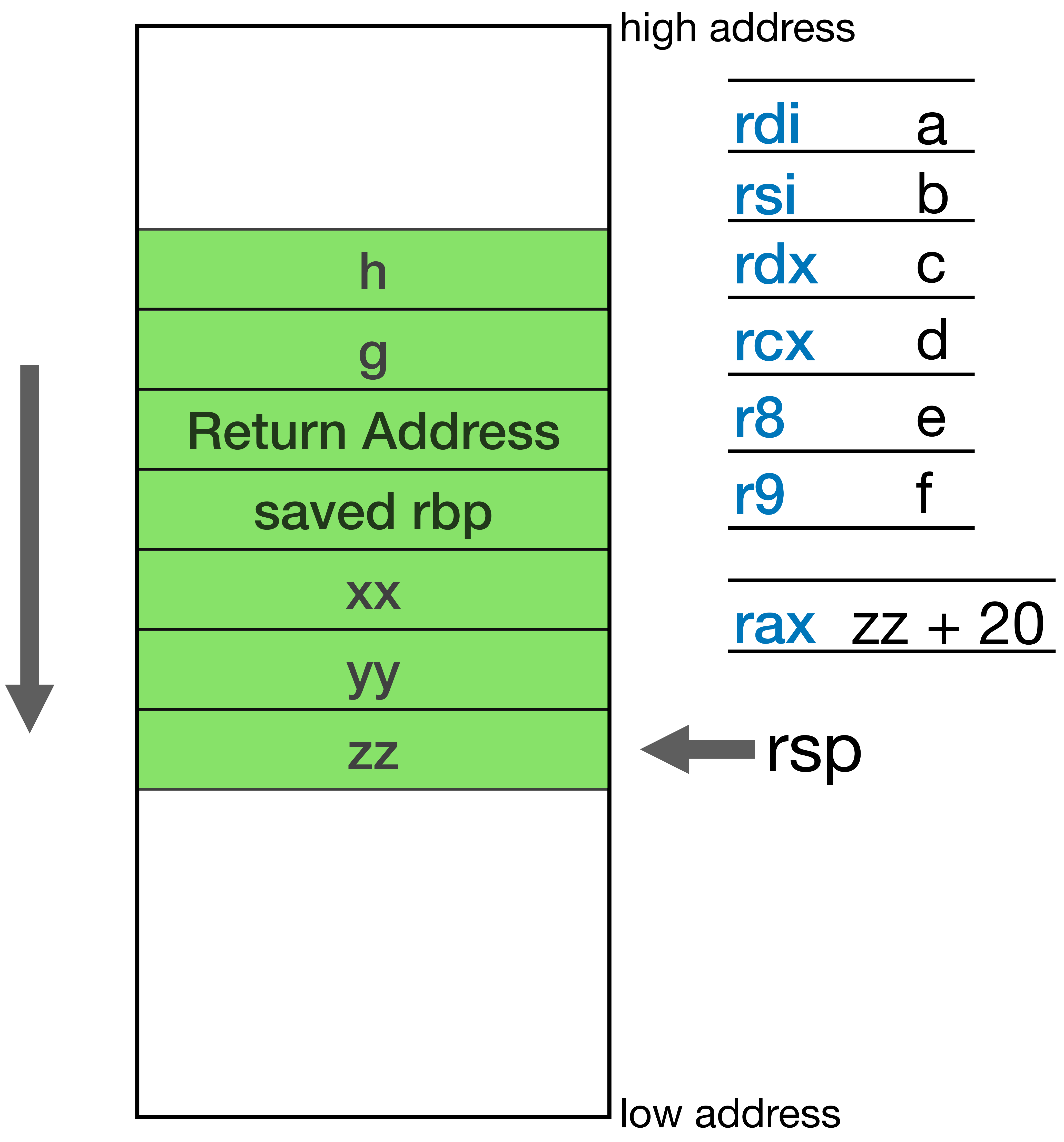
https://en.wikipedia.org/wiki/Variadic_function

Format String Attacks

- Public since 1999
  - First thought of as harmless programming errors
- Format string refers to the argument that specifies the format of a string to functions like `printf`.
  - e.g., `printf (“i = %d with address %08x\n", i, &i);`
- Functions taking format strings are commonly used.
  - `printf/sprintf/fprintf/snprintf/vprintf`, etc.
  - `scanf/fscanf/sscanf`
  - `syslog/vsyslog`
  - `warn()` and `err()` family of functions

x86-64/AMD64 Calling Convention

```
void foo() {
    ...
    bar(a, b, c, d, e, f, g, h);
    ...
}

long bar(long a, long b, long c, long d,
         long e, long f, long g, long h) {
   long xx = a * b * c * d * e * f * g * h;
   long yy = a + b + c + d + e + f + g + h;
   long zz = utilfunc(xx, yy, xx % yy);
   return zz + 20;
}
```

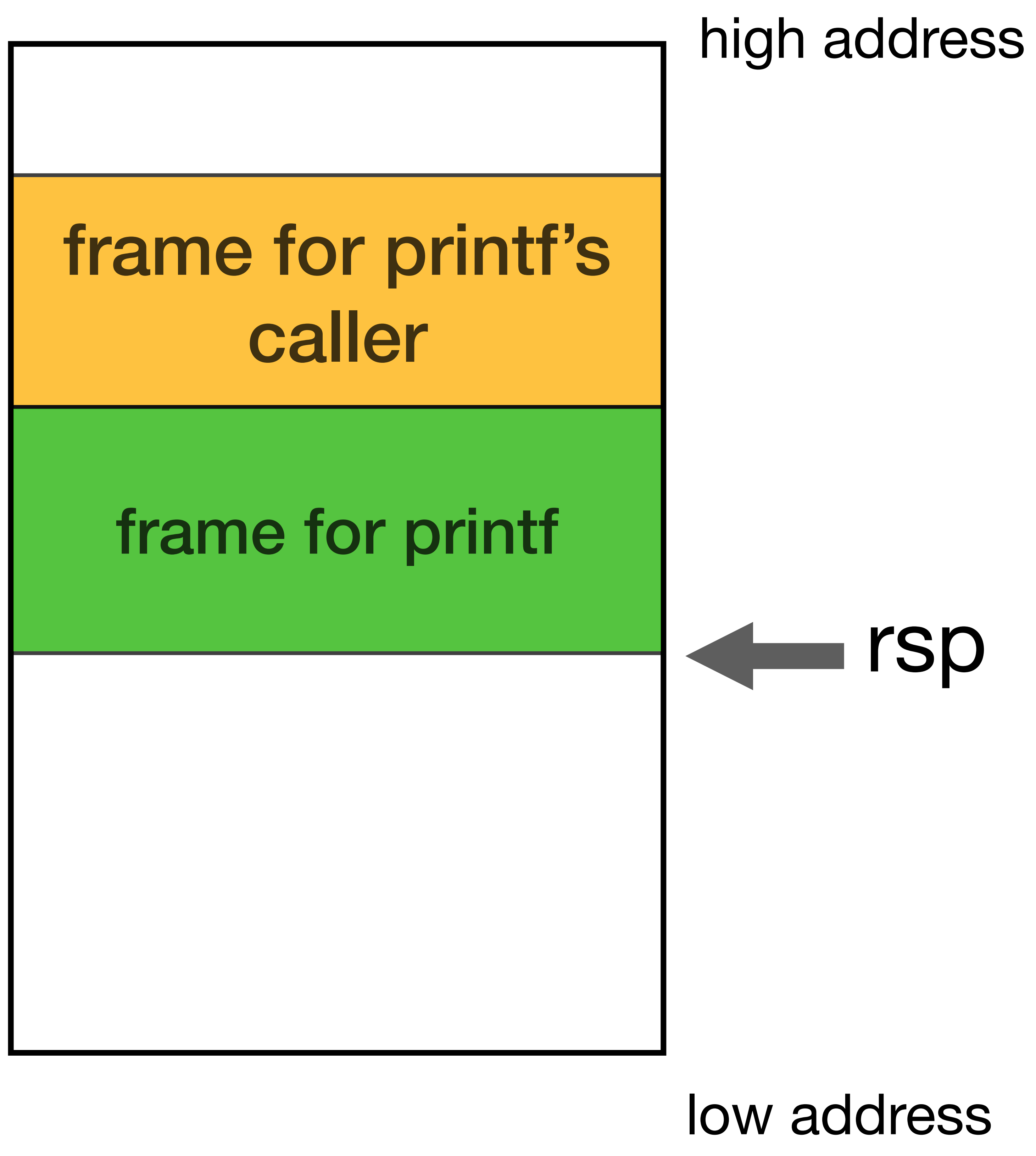


How Does `printf` Work in C?

```
printf (“i = %d with address %08x\n", i, &i);
```

- Prepare the *three* arguments: string address pointer, `i`, and `&i`
  - through rdi, rsi, rdx on x86-64
  - through stack on x86-32
- Invoke `printf`
- When control is inside `printf`, the function looks for arguments in registers/stack.

rdi	str’s addr
rsi	i
rdx	i’s addr



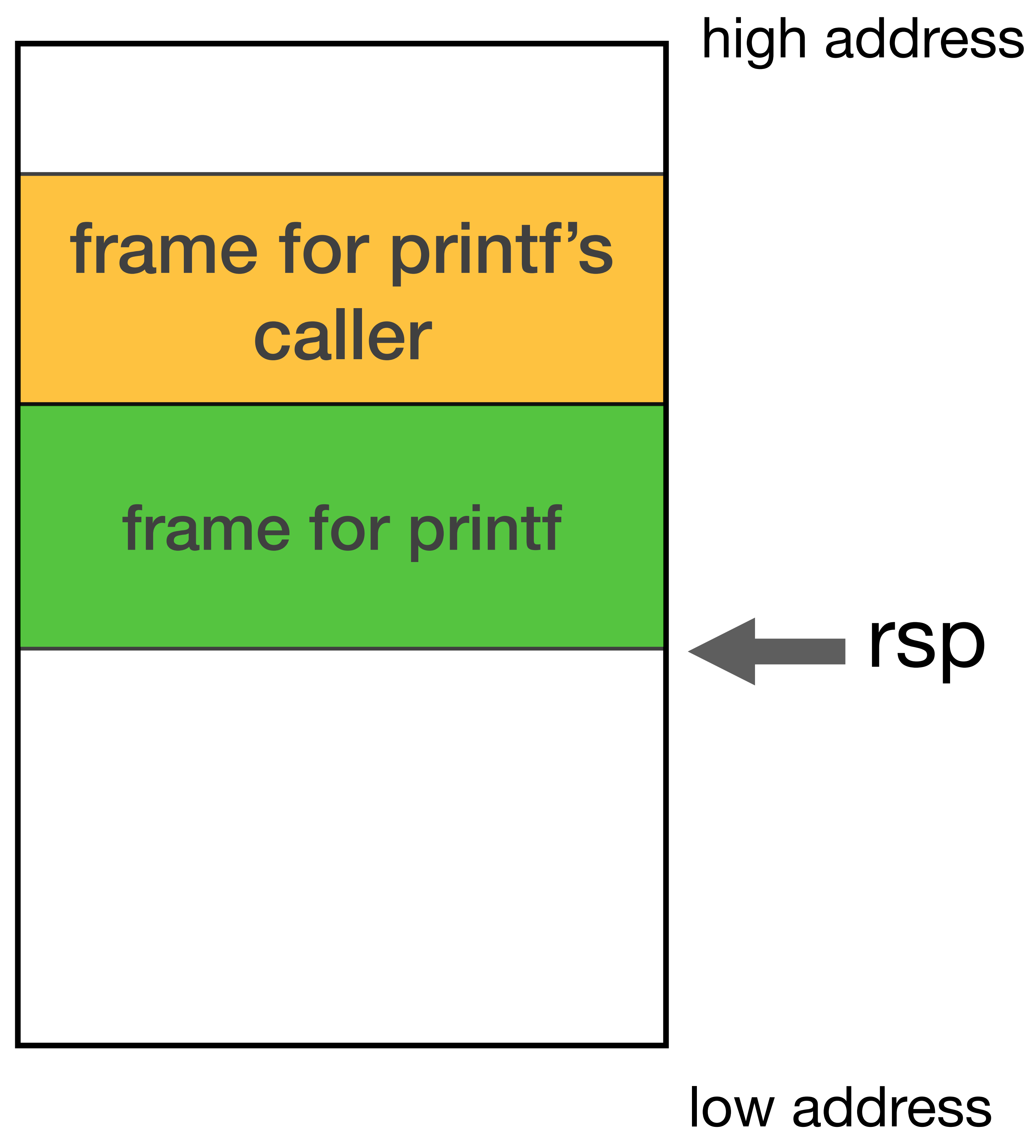
How Does `printf` Work in C?

- What happens for the following `printf`

```
printf ("i = %d with address %08x\n");
```

- The compiler may warn but still accept the program.
  - Pretending that the required arguments were in the right place.

rdi	str's addr
rsi	???
rdx	???



Format String Attacks

- What about the following simple program for echoing user input?

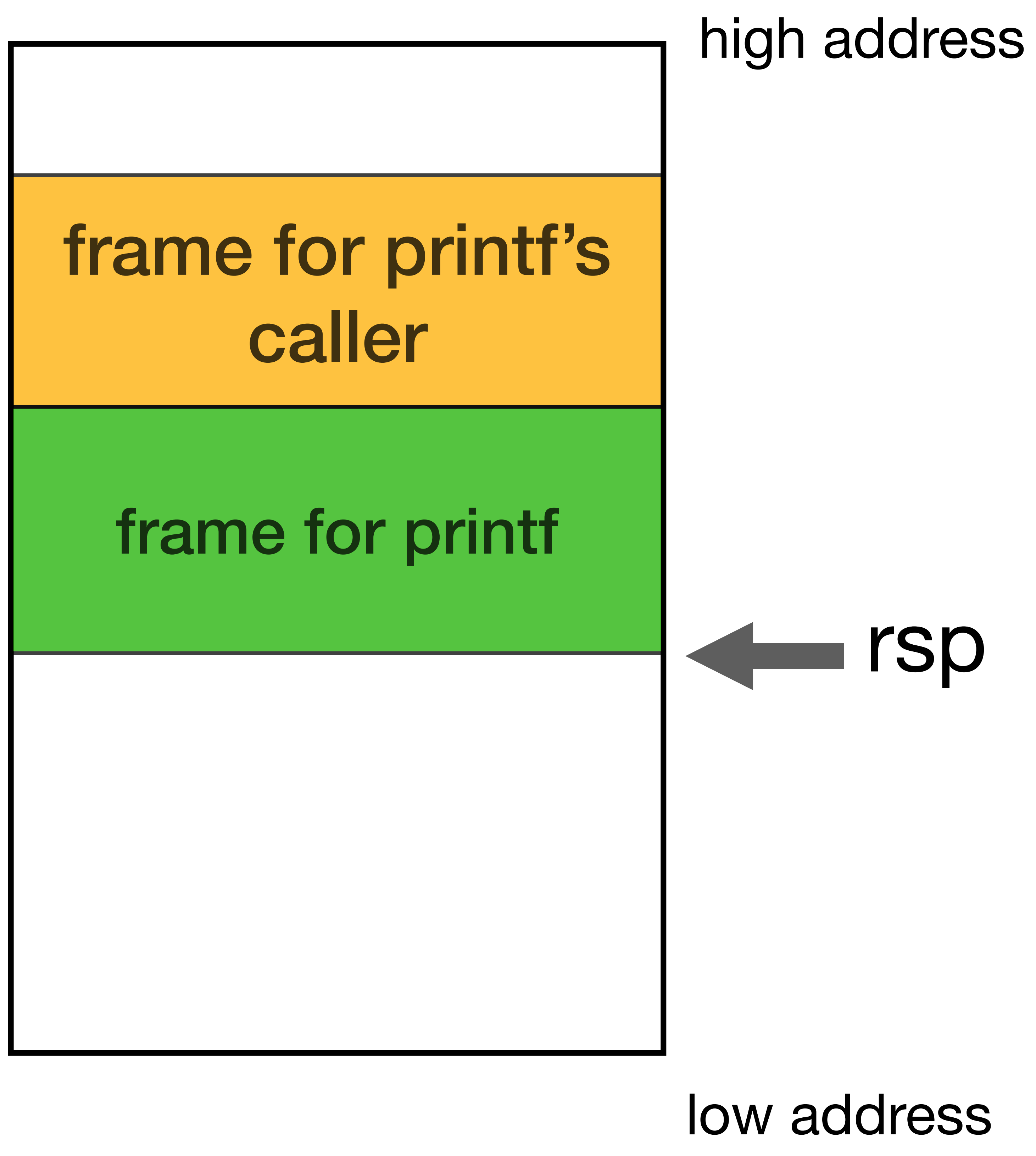
```
int main(int argc, char *argv[]) {
    if (argc > 1) {
        printf(argv[1]);
    }
}
```

- Appears to be normal
- However, what would happen if the input is "`hello%d%d%d%d%d%d%d`"?
  - i.e. `printf("hello%d%d%d%d%d%d%d");`
  - It would print numbers from five registers *and* the stack.
    - Allows attackers to peak unintended data confidentiality vulnerability
- What if `arg[1]` is "`hello%s`"?
  - Likely a segmentation fault availability vulnerability

How to Leak Data In An Arbitrary Address

1. Put the target address in a location controlled by attackers
2. Trick the program to use (load) the target address

```
buf = “target_addr%c%c…%s";
printf(buf);
```

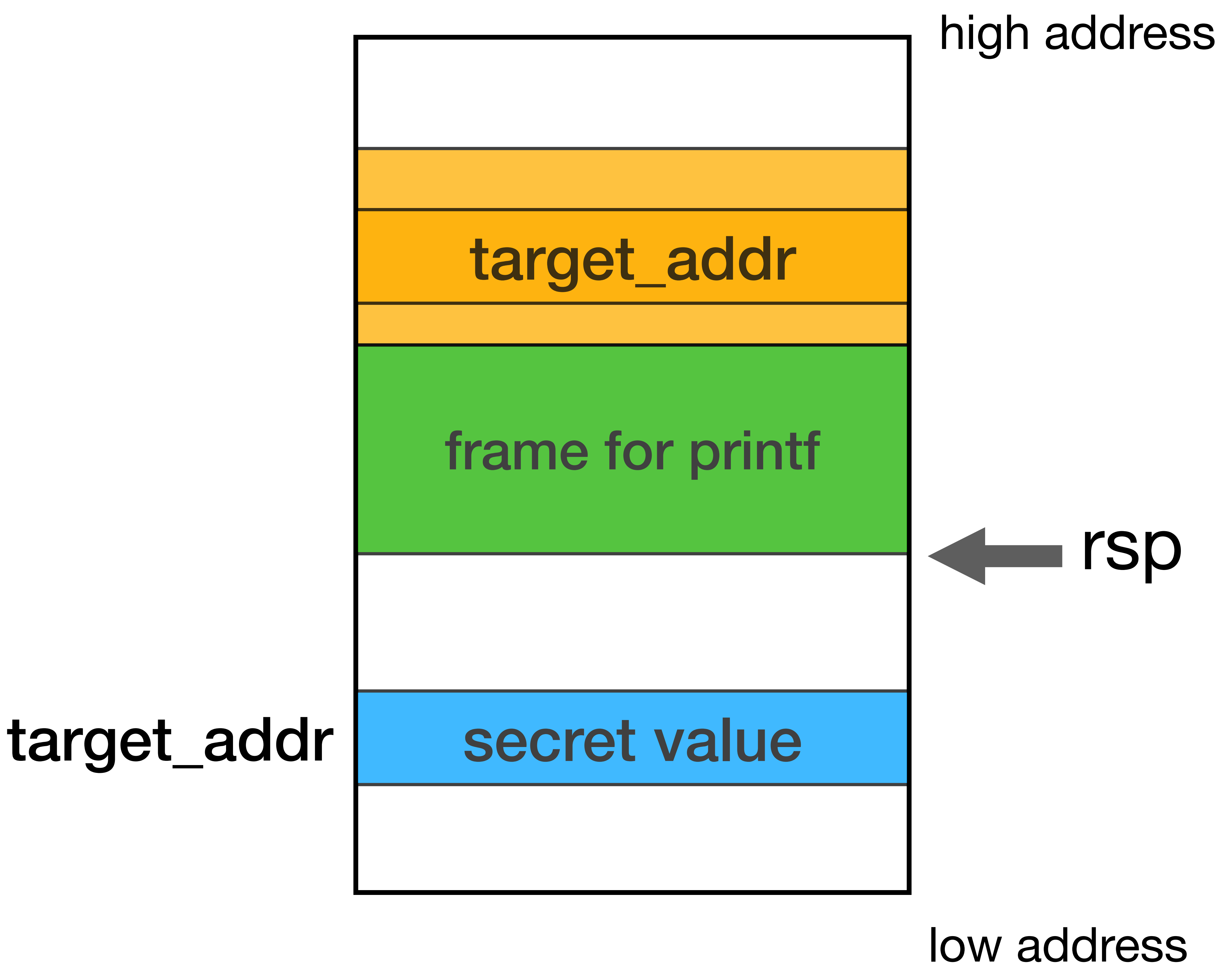


How to Leak Data In An Arbitrary Address

1. Put the target address in a location controlled by attackers
2. Trick the program to use (load) the target address

```
buf = “target_addr%c%c…%s";
printf(buf);
```

If `buf` lives on the stack of the caller of `printf`, and it is controlled by attackers, `target_addr` can be set.



Format String Attacks: Data Integrity Vulnerabilities

- There is a “%n” specifier for format strings.
  - Writes the number of bytes already printed into a variable of the programmer’ choice.

```
int i;
printf (“foobar%n\n", &i);
printf ("i = %d\n", i);
```

- `i` was assigned 6.
- “%n” has variants:
  - “%hn”: `short*`
  - “%hhn”: `signed char*`

Format String Attacks: Data Integrity Vulnerabilities

```c
int main(int argc, char *argv[]) {
    if (argc > 1) {
        printf(argv[1]);
    }
}
```

- What if the user input is “`foobar%n`”?
  - `printf(“foobar%n”);`

Format String Attacks: Data Integrity Vulnerabilities

- There is a “%n” specifier for format strings.
  - Writes the number of bytes already printed into a variable of the programmer’ choice.

```
int i;
printf (“foobar%n\n", &i);
printf ("i = %d\n", i);
```

- `i` was assigned 6.

Format String Attacks: Data Integrity Vulnerabilities

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

- What if the user input is "`foobar%n`"?
    - Will take the data in `rsi`, interpreted as an address, and write 6 to the memory location of that address.
- What about "`foobar%10c%n`"?
    - Write 16 to a memory location

Format String Attacks: Data Integrity Vulnerabilities

- How to write to an arbitrary address?
  - Put the target address at the right place (register/stack).
- An attacker can possibly update any memory with arbitrary contents.
  - e.g., overwriting a function pointer and hijacking the control flow

Format String Attacks

```
int main(int argc, char *argv[]) {
    char buf[512];
    fgets(buf, sizeof(buf), stdin);


    printf("The input is:");
    printf(buf);
    return 0;
}
```

No buffer overflows here

But format string vulnerabilities

- Attackers can possibly
  - View/change any part of the memory
  - Execute arbitrary code

Format String Attacks: Fixes

- Most of time: quite easy to fix:

```
int main(int argc, char *argv[]) {
    if (argc > 1) {
        printf(argv[1]);     printf(“%s”, argv[1]);
    }
}
```

- But not always so obvious
  - e.g., when the format string is constructed on the fly, we have to make sure that format string cannot be influenced by input controllable by the attacker.

```
printf("hello, %d, %d", 10);

char *format = "hello, %d, %d";
printf(format, 10);
```

compiler warning

no compiler warning

Prevent Format String Vulnerabilities

- Limit the ability of adversaries to control the format string
  - Hard-code format string
  - Do not use "`%n`"
  - Be careful with other specifiers, e.g., `%s` and `sprintf` may cause data disclosure.
  - Compiler support: Match arguments with format string.
    - Do not ignore compiler warnings!
    - Use extra security checking flags, i.e. "`–Wformat*`" series of flags