Exam Review

Changelog

Corrections made in this version not in first posting:

- 3 April 2017: Add corrected verison of format string exploit stack picture (slide 5), in addition to marking old segfaulting version (slide 4)
- 3 April 2017: Fix ROP with VTable overwrite example (slide 11) to use %rsi instead of %rdi. I somehow thought *(%rdi) was looking for a pointer to pointer when it certainly does not

general format

similar to last midterm

short answer and multiple choice

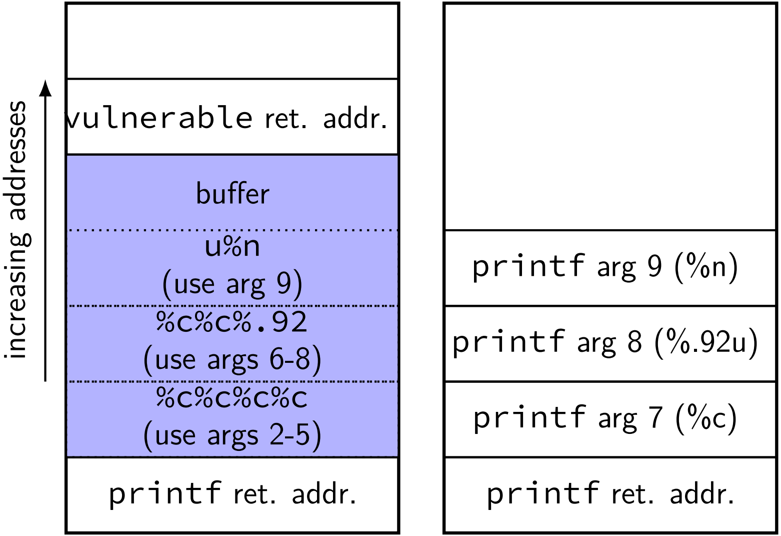
often reading vulnerable code and answering questions

less multiple multiple choice

topics

- memory error vulnerabilities
  - stack smashing, pointer subterfuge, heap smashing, double-free
  - format string exploits
  - use after free
  - integer overflows and buffers
- function pointers to overwrite
  - return addresses, stubs, VTables
- mitigations
  - ASLR, write XOR execute, stack canaries, guard pages, bounds checking
- return- and jump-oriented programming

format string segfault



increasing addresses

vulnerable ret. addr.

buffer

u%n
(use arg 9)

%c%c%.92
(use args 6-8)

%c%c%c%c
(use args 2-5)

printf ret. addr.

printf arg 9 (%n)

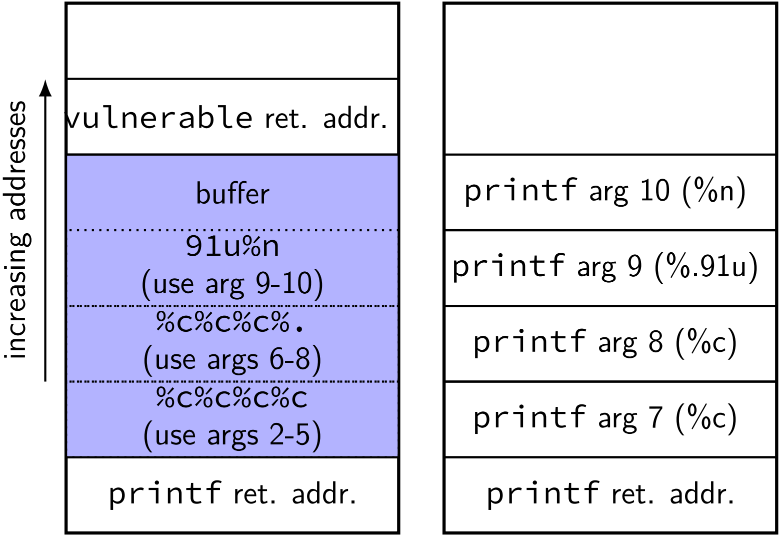
printf arg 8 (%.92u)

printf arg 7 (%c)

printf ret. addr.

```
void vulnerable() {
    char buffer[32];
    fgets(buffer,
          sizeof(buffer),
          stdin);
    printf(buffer);
}

// input:
// "%c%c%c%c%c%c%.92u%n"
```

format string exploit



increasing addresses

vulnerable ret. addr.

buffer

91u%n
(use arg 9-10)

%c%c%c%.
(use args 6-8)

%c%c%c%c
(use args 2-5)

printf ret. addr.

printf arg 10 (%n)

printf arg 9 (%.91u)

printf arg 8 (%c)

printf arg 7 (%c)

printf ret. addr.

```
void vulnerable() {
    char buffer[32];
    fgets(buffer,
          sizeof(buffer),
          stdin);
    printf(buffer);
}

// input:
// "%c%c%c%c%c%c%.92u%n"
```

format string overwrite: setup

```
/* advance through 5 registers, then
 * 5 * 8 = 40 bytes down stack, outputting
 * 4916157 + 9 characters before using
 * %ln to store a long.
 */
fputs("%c%c%c%c%c%c%c%c%c%.4196157u%ln", stdout);
/* include 5 bytes of padding to make current location
 * in buffer match where on the stack printf will be reading.
 */
fputs("?????", stdout);
void *ptr = (void*) 0x601038;
/* write pointer value, which will include \0s */
fwrite(&ptr, 1, sizeof(ptr), stdout);
fputs("\n", stdout);
```

stack smashing: the tricky parts

construct machine code that works in any executable

- same tricks as writing relocatable virus code
- usual idea: just execute shell (command prompt)

construct machine code that's valid input

- machine code usually flexible enough

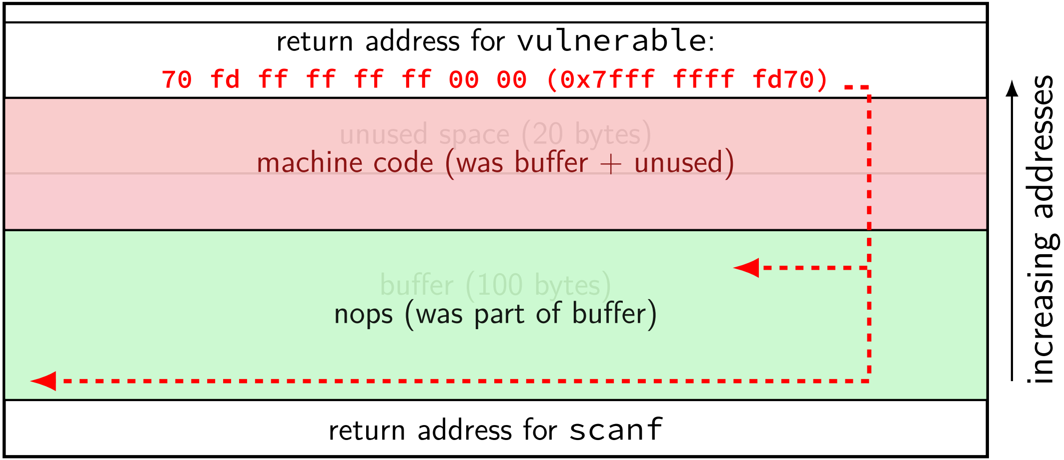
finding location of return address

- fixed offset from buffer

finding location of inserted machine code

guessed return-to-stack

highest address (stack started here)



return address for `vulnerable`:

`70 fd ff ff ff ff 00 00 (0x7fff ffff fd70)`

unused space (20 bytes)

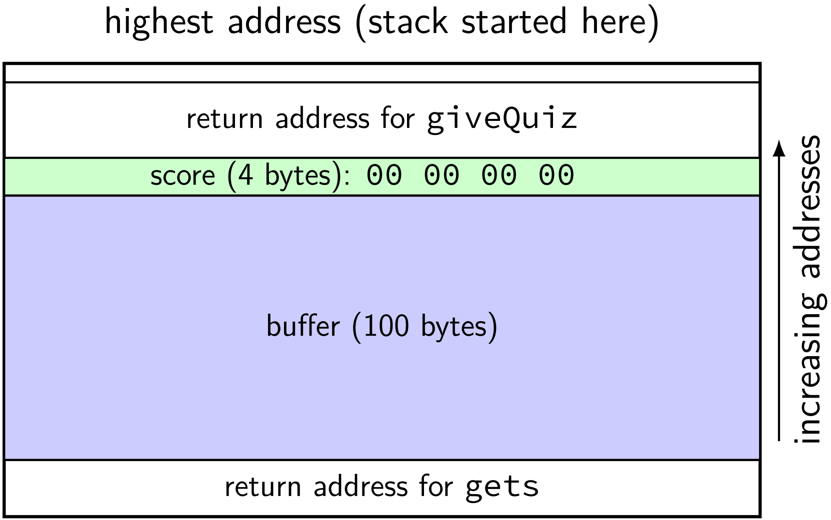
machine code (was buffer + unused)

buffer (100 bytes)

nops (was part of buffer)

return address for `scanf`

simpler overflow: stack



highest address (stack started here)

return address for `giveQuiz`

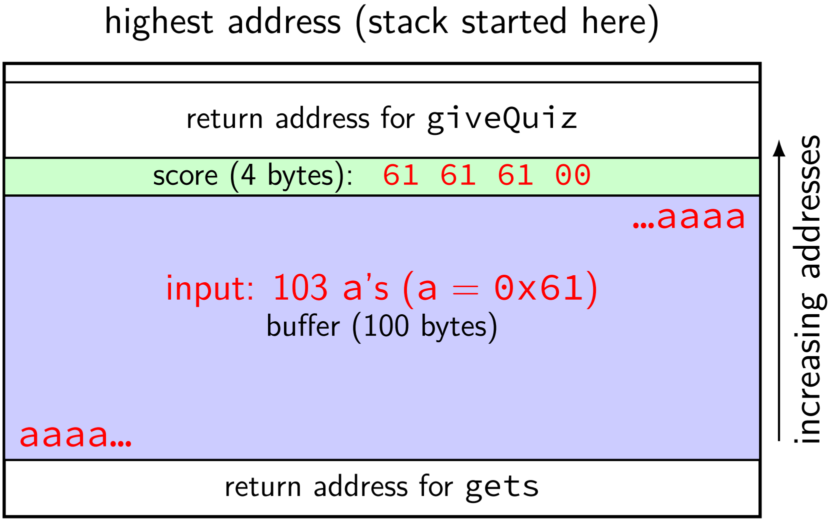
score (4 bytes): `00 00 00 00`

buffer (100 bytes)

return address for `gets`

increasing addresses

simpler overflow: stack



highest address (stack started here)

return address for `giveQuiz`

score (4 bytes): 61 61 61 00

…aaaa

input: 103 a's (a = 0x61)

buffer (100 bytes)

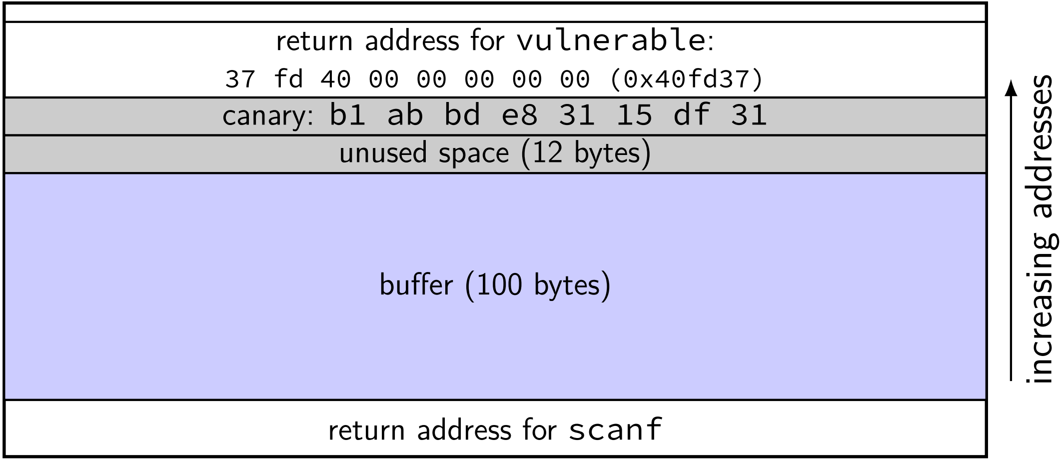
aaaa…

return address for `gets`

increasing addresses

stack canary

highest address (stack started here)



return address for `vulnerable`:

`37 fd 40 00 00 00 00 00 (0x40fd37)`

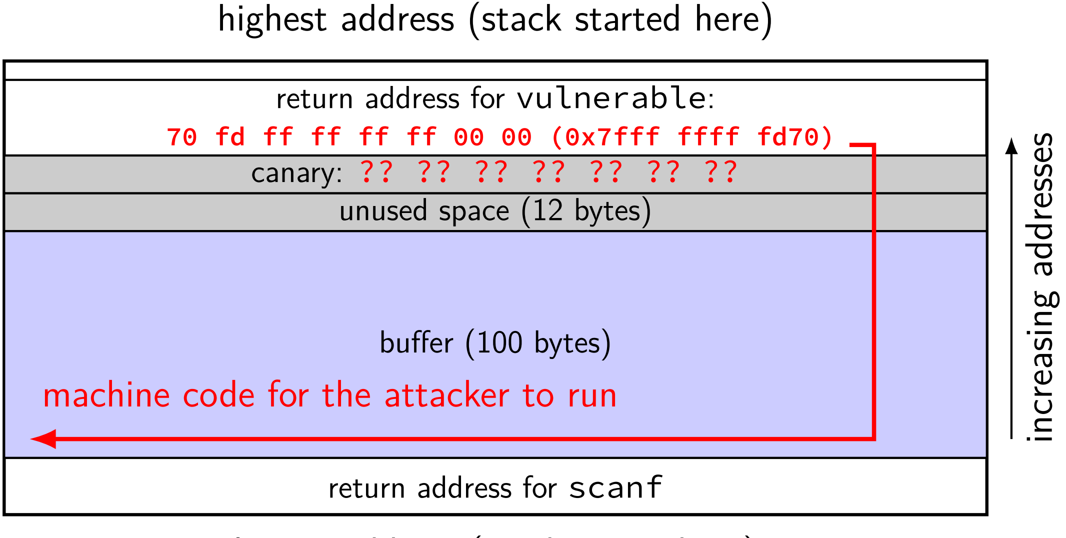
canary: `b1 ab bd e8 31 15 df 31`

unused space (12 bytes)

buffer (100 bytes)

return address for `scanf`

stack canary



highest address (stack started here)

return address for `vulnerable`:

`70 fd ff ff ff ff 00 00 (0x7fff ffff fd70)`

canary: `?? ?? ?? ?? ?? ?? ??`

unused space (12 bytes)

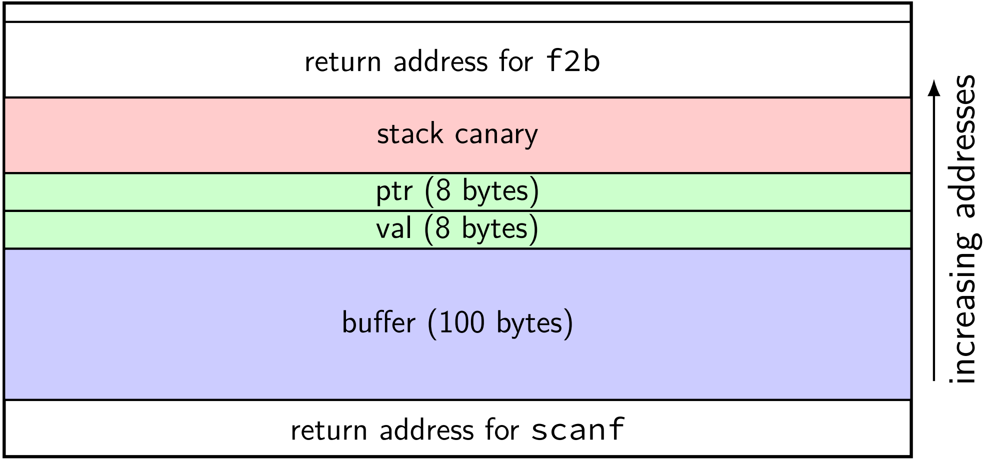
buffer (100 bytes)

machine code for the attacker to run

return address for `scanf`

skipping the canary

highest address (stack started here)



return address for `f2b`

stack canary

ptr (8 bytes)

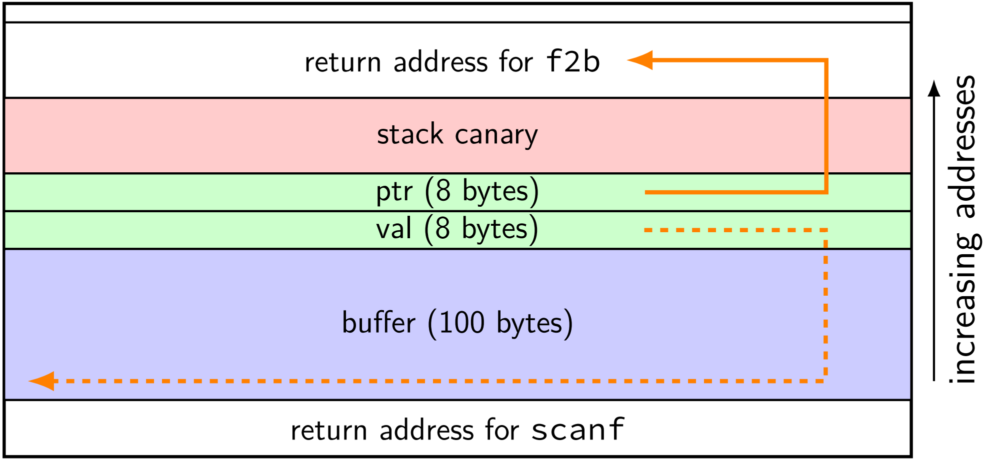
val (8 bytes)

buffer (100 bytes)

return address for `scanf`

skipping the canary

highest address (stack started here)



return address for `f2b`

stack canary

ptr (8 bytes)

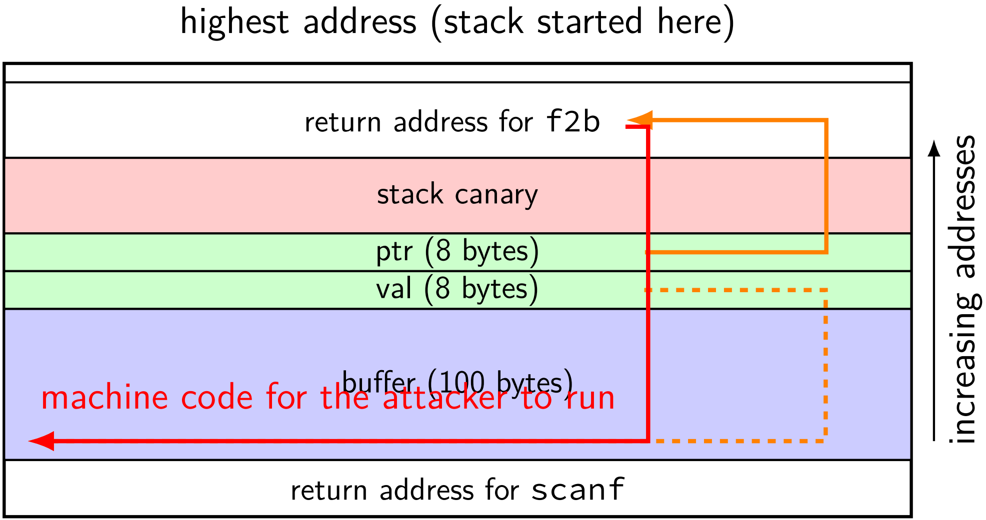
val (8 bytes)

buffer (100 bytes)

return address for `scanf`

increasing addresses

skipping the canary



highest address (stack started here)

return address for `f2b`

stack canary

ptr (8 bytes)

val (8 bytes)

buffer (100 bytes)

machine code for the attacker to run

return address for `scanf`

increasing addresses

pointer subterfuge

```
void f2b(void *arg, size_t len) {
    char buffer[100];
    long val = ...; /* assume on stack */
    long *ptr = ...; /* assume on stack */
    memcpy(buff, arg, len); /* overwrite ptr? */
    *ptr = val; /* arbitrary memory write! */
}
```

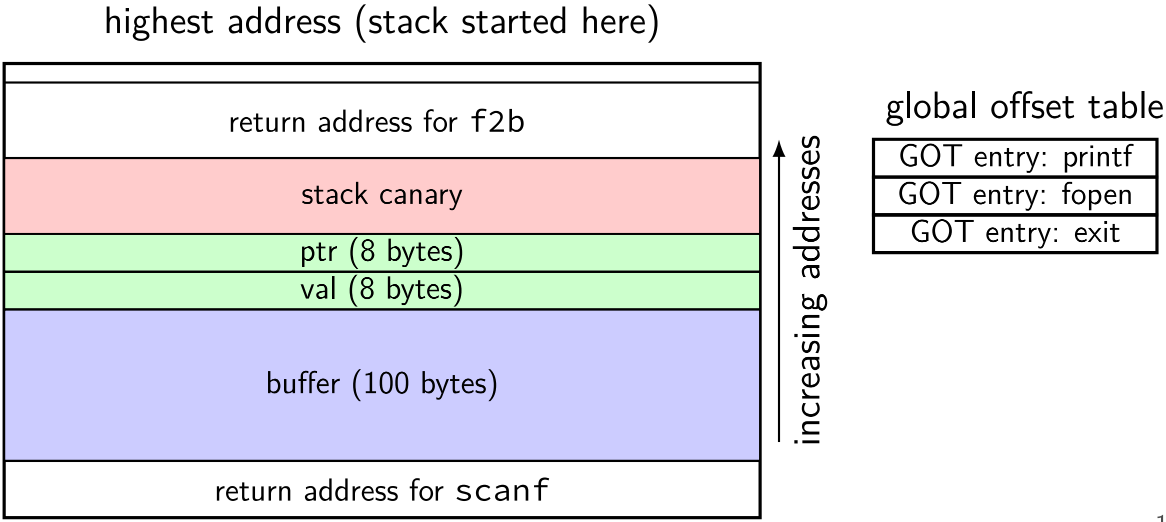
adapted from Pincus and Baker, Figure 2

pointer subterfuge

```
void f2b(void *arg, size_t len) {
    char buffer[100];
    long val = ...; /* assume on stack */
    long *ptr = ...; /* assume on stack */
    memcpy(buff, arg, len); /* overwrite ptr? */
    *ptr = val; /* arbitrary memory write! */
}
```

adapted from Pincus and Baker, Figure 2

attacking the GOT



highest address (stack started here)

return address for `f2b`

stack canary

ptr (8 bytes)

val (8 bytes)

buffer (100 bytes)

return address for `scanf`

increasing addresses

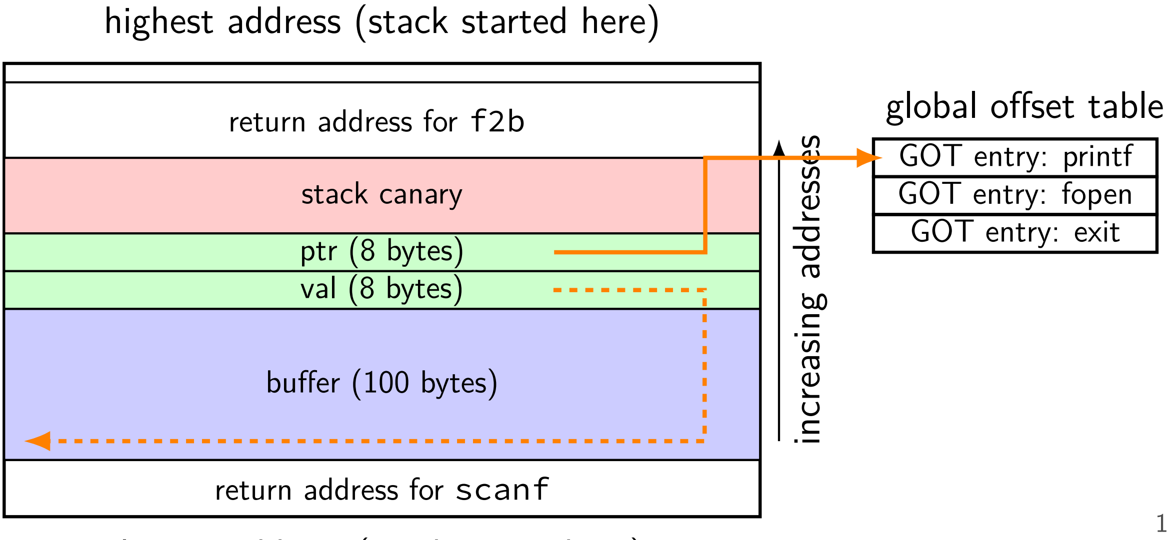
global offset table

GOT entry: printf

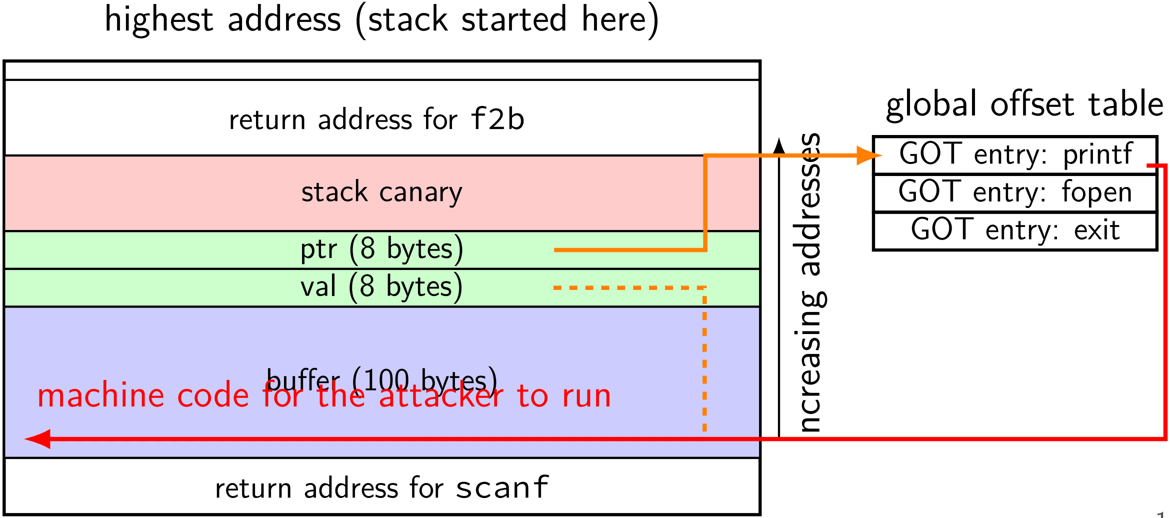
GOT entry: fopen

GOT entry: exit

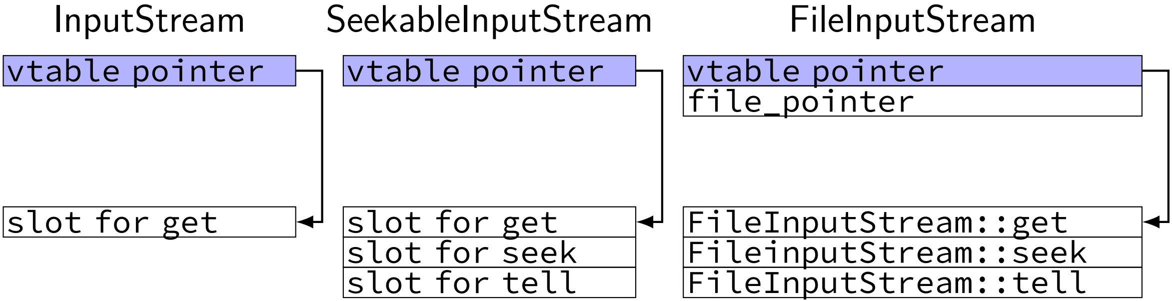
attacking the GOT



attacking the GOT



C++ inheritence: memory layout



InputStream

`vtable pointer`

`slot for get`

SeekableInputStream

`vtable pointer`

`slot for get`

`slot for seek`

`slot for tell`

FileInputStream

`vtable pointer`

`file_pointer`

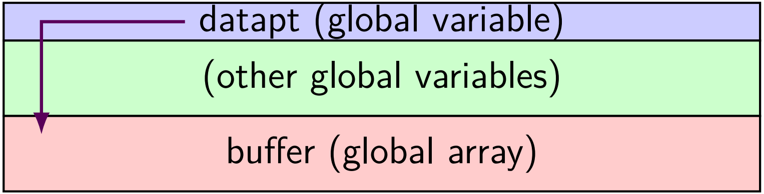
`FileInputStream::get`

`FileinputStream::seek`

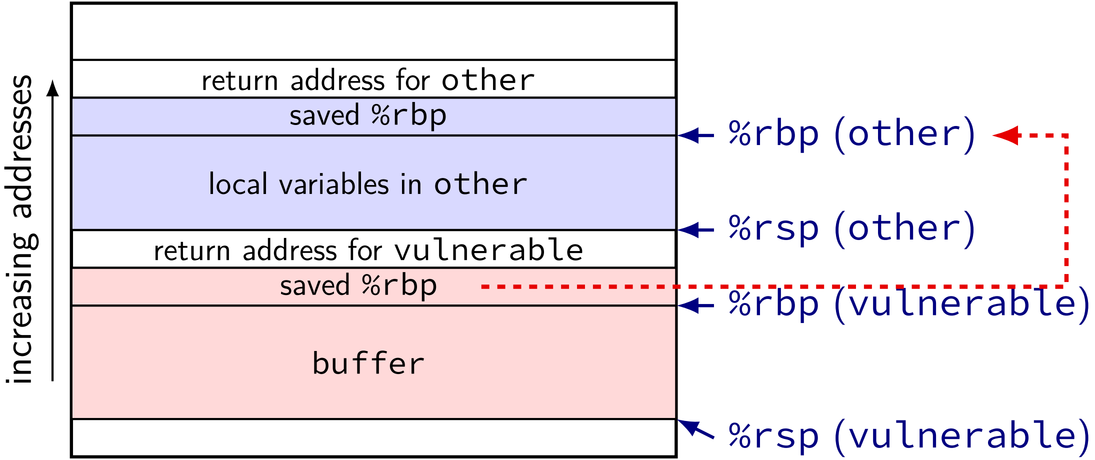
`FileInputStream::tell`

NTP exploit picture

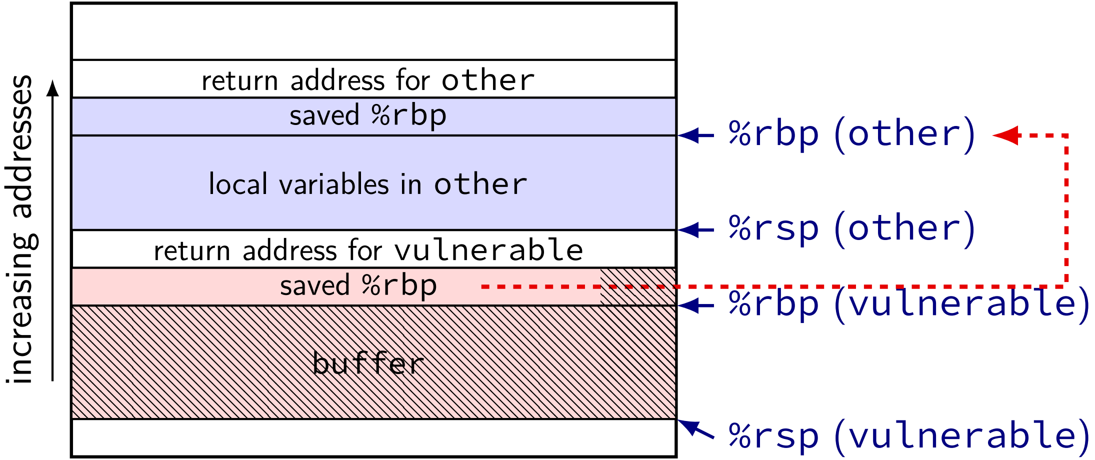
```
memmove((char *)datapt, dp, (unsigned)dlen);
```



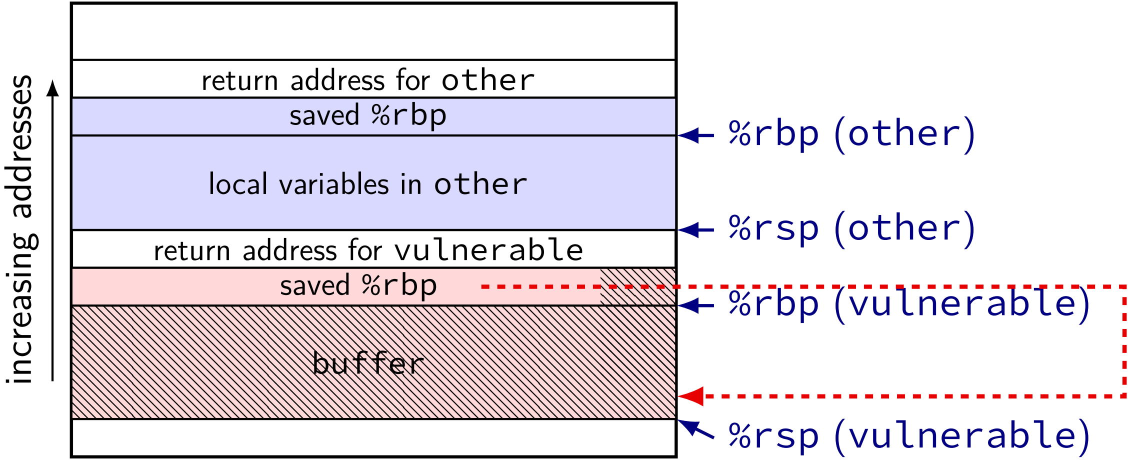
vulnerable stack layout



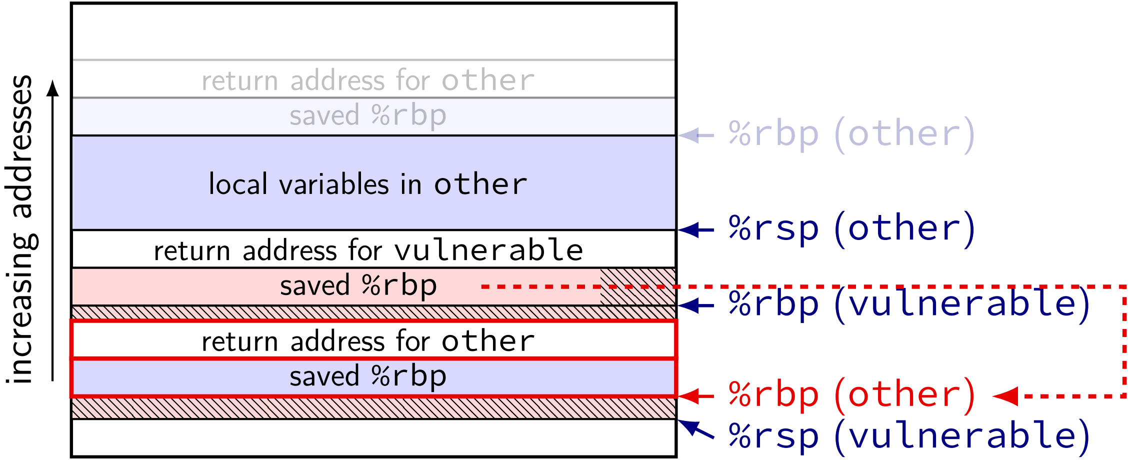
vulnerable stack layout



vulnerable stack layout

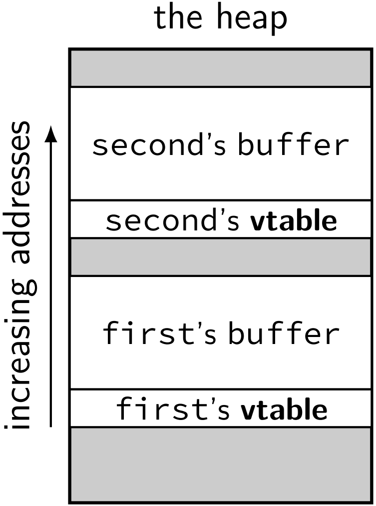


vulnerable stack layout



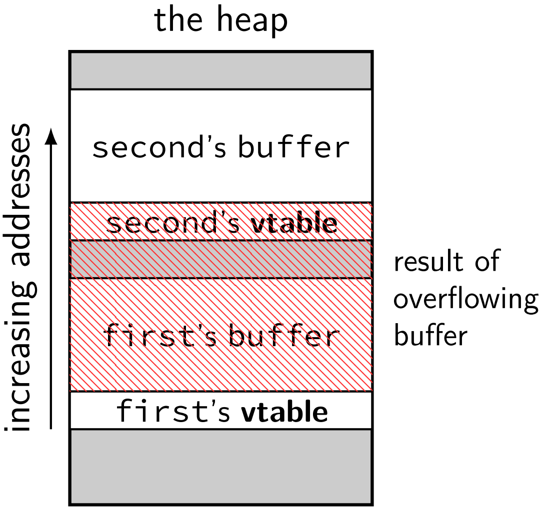
heap overflow: adjacent allocations

```
class V {
  char buffer[100];
public:
  virtual void ...;
  ...
};
...
V *first = new V(...);
V *second = new V(...);
strcpy(first->buffer,
       attacker_controlled);
```



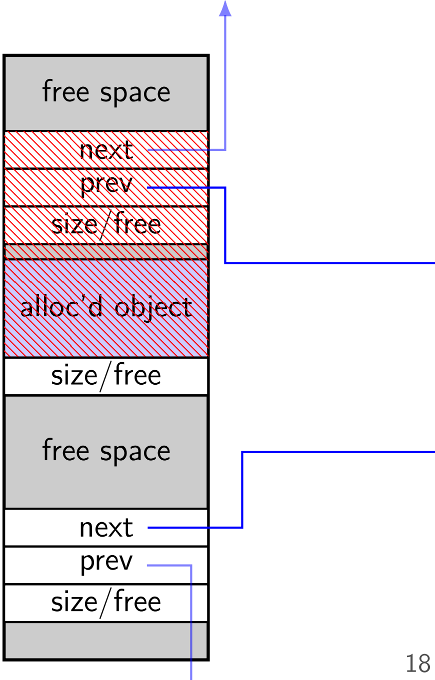
heap overflow: adjacent allocations

```
class V {
  char buffer[100];
public:
  virtual void ...;
  ...
};
...
V *first = new V(...);
V *second = new V(...);
strcpy(first->buffer,
       attacker_controlled);
```



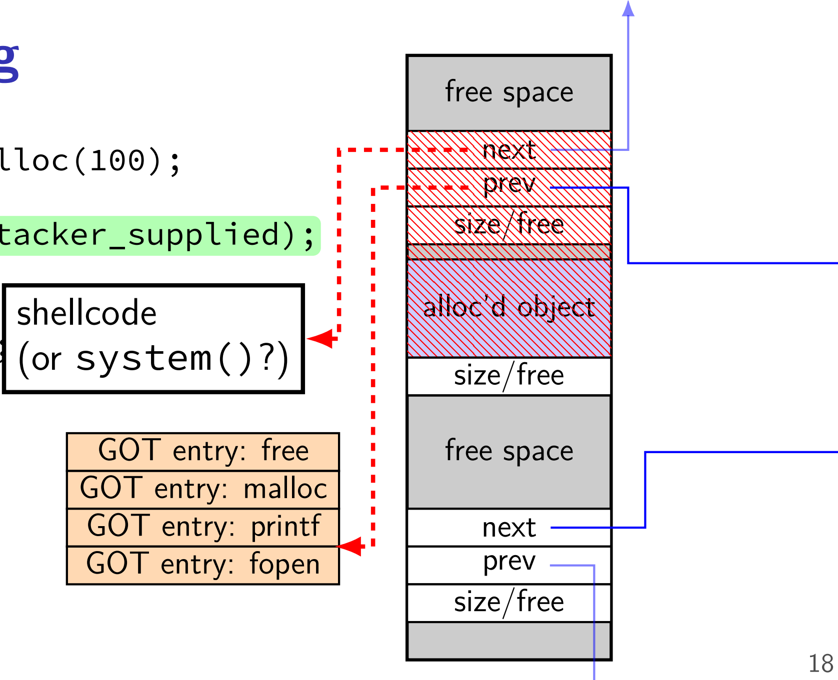
heap smashing

```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```



heap smashing

```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```

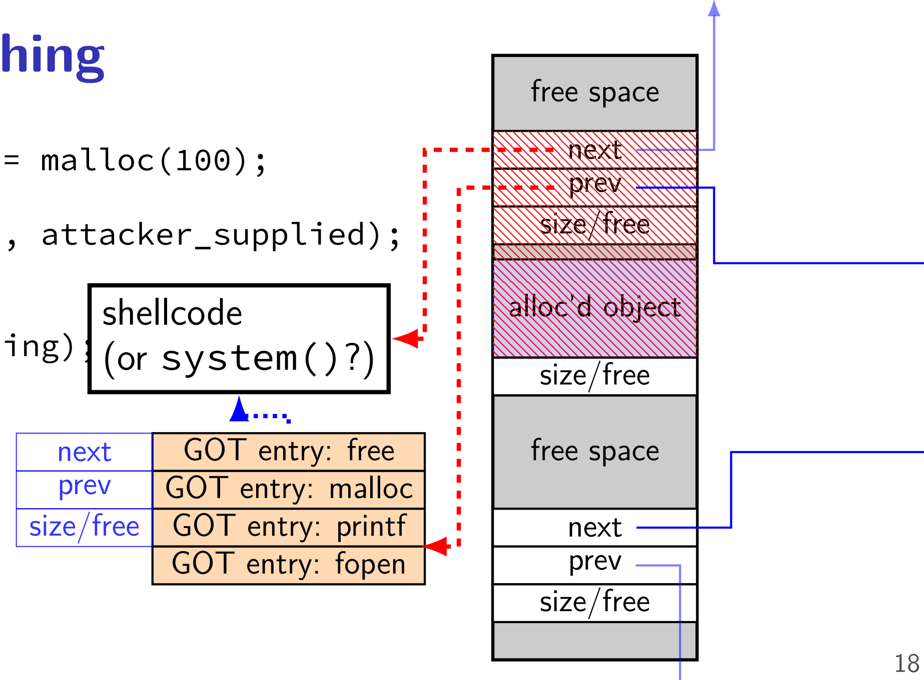


shellcode
(or `system()`?)

GOT entry: free
GOT entry: malloc
GOT entry: printf
GOT entry: fopen

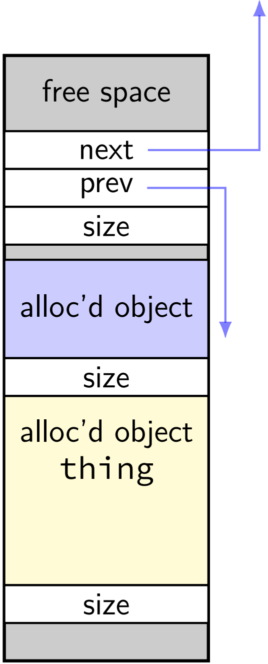
heap smashing

```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```



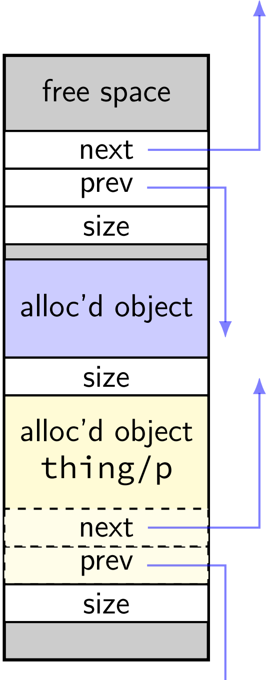
double-frees

```
free(thing);
free(thing);
char *p = malloc(...);
// p  points to next/prev
//     on list of avail.
//     blocks
strcpy(p, attacker_controlled);
malloc(...);
char *q = malloc(...);
// q  points to attacker-
//     chosen address
strcpy(q, attacker_controlled2);
...
```



double-frees

```
free(thing);
free(thing);
char *p = malloc(...);
// p  points to next/prev
//      on list of avail.
//      blocks
strcpy(p, attacker_controlled);
malloc(...);
char *q = malloc(...);
// q  points to attacker-
//      chosen address
strcpy(q, attacker_controlled2);
...
```

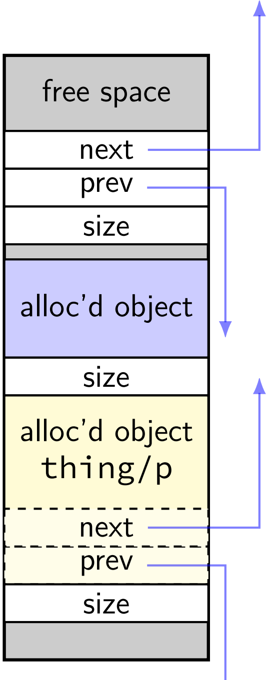


double-frees

```
free(thing);
free(thing);
char *p = malloc(...);
// p  points to next/prev
//      on list of avail.
//      blocks
strcpy(p, attacker_controlled);
```

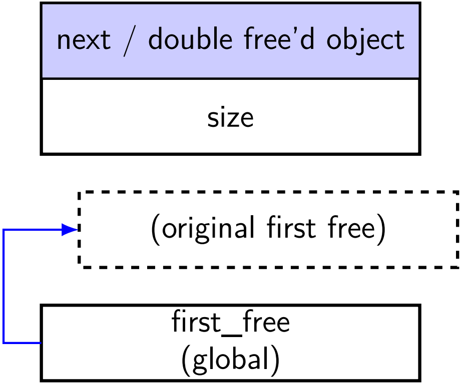
malloc returns something still on free list
because double-free made loop in linked list

```
// q  points to attacker-
//      chosen address
strcpy(q, attacker_controlled2);
...
```



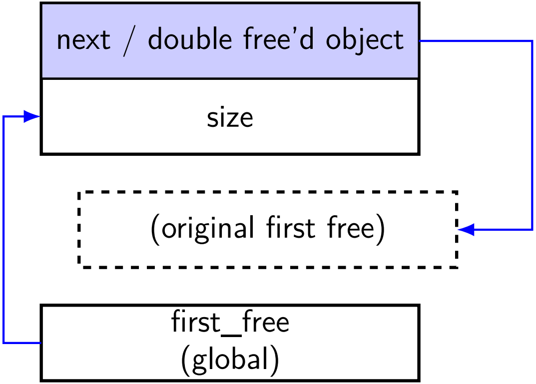
double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



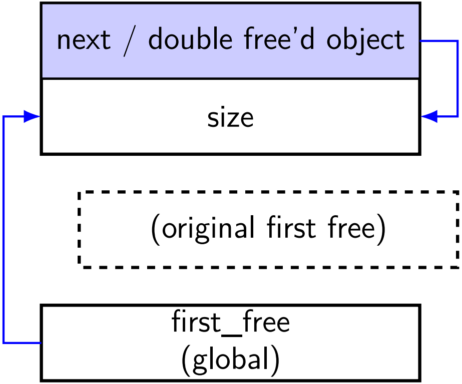
double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



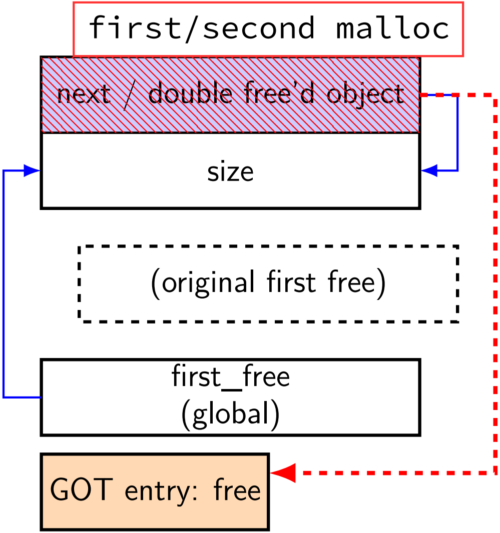
double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



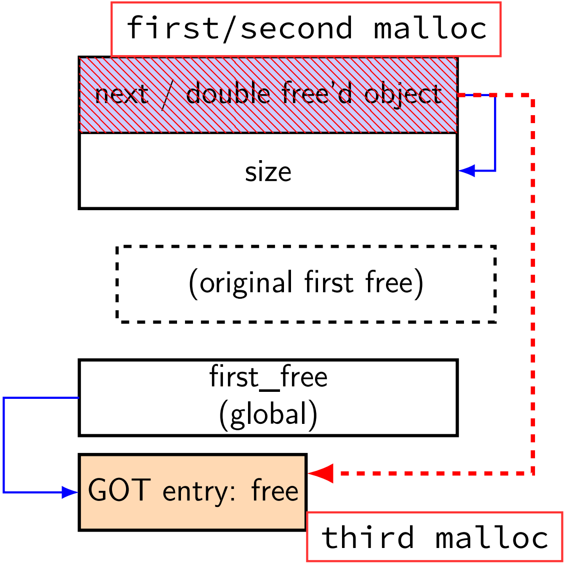
double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



use-after-free

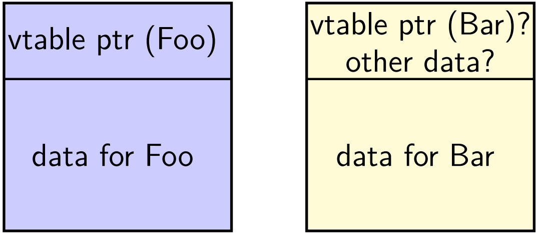
```
class Foo {
    ...
};
Foo *the_foo;
the_foo = new Foo;
...
delete the_foo;
...
something_else = new Bar(...);
the_foo->something();
```

`something_else` likely where `the_foo` was

use-after-free

```
class Foo {
    ...
};
Foo *the_foo;
the_foo = new Foo;
...
delete the_foo;
...
something_else = new Bar(...);
the_foo->something();
```

`something_else` likely where `the_foo` was



vtable ptr (Foo)

data for Foo

vtable ptr (Bar)?
other data?

data for Bar

integer overflow example

```
item *load_items(int len) {
  int total_size = len * sizeof(item);
  if (total_size >= LIMIT) {
    return NULL;
  }
  item *items = malloc(total_size);
  for (int i = 0; i < len; ++i) {
    int failed = read_item(&items[i]);
    if (failed) {
      free(items);
      return NULL;
    }
  }
  return items;
}
```

`len` = `0x4000 0001`
`sizeof(item)` = `0x10`

`total_size` =
`0x4 0000 0010`

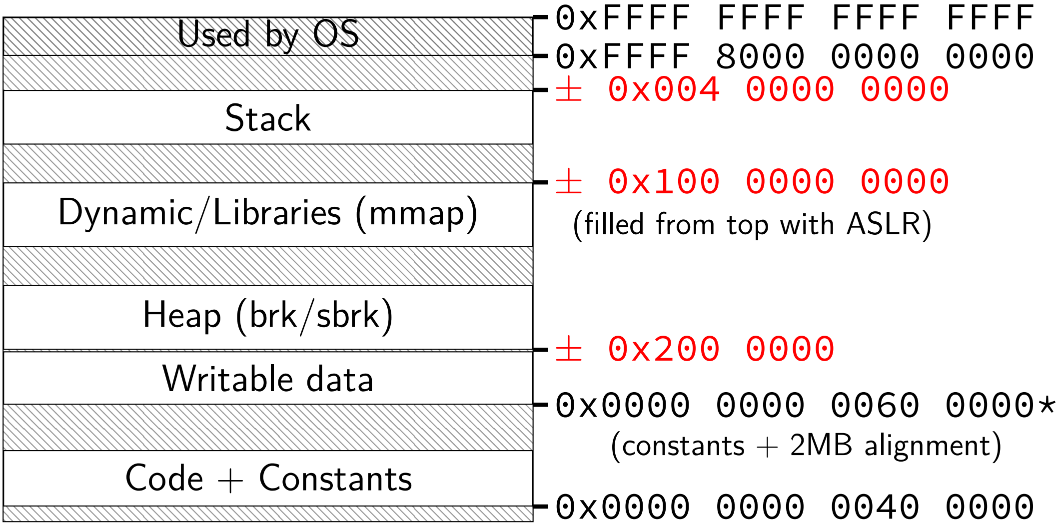
integer overflow example

```
item *load_items(int len) {
  int total_size = len * sizeof(item);
  if (total_size >= LIMIT) {
    return NULL;
  }
  item *items = malloc(total_size);
  for (int i = 0; i < len; ++i) {
    int failed = read_item(&items[i]);
    if (failed) {
      free(items);
      return NULL;
    }
  }
  return items;
}
```

len = 0x4000 0001
sizeof(item) = 0x10

total_size =
0x4 0000 0010

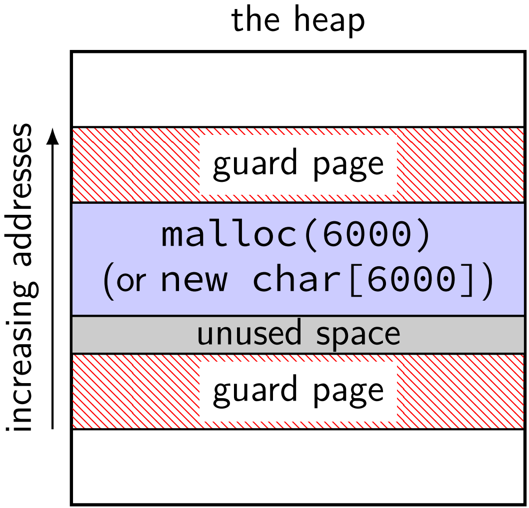
program memory (x86-64 Linux; ASLR)



the mapping (set by OS)

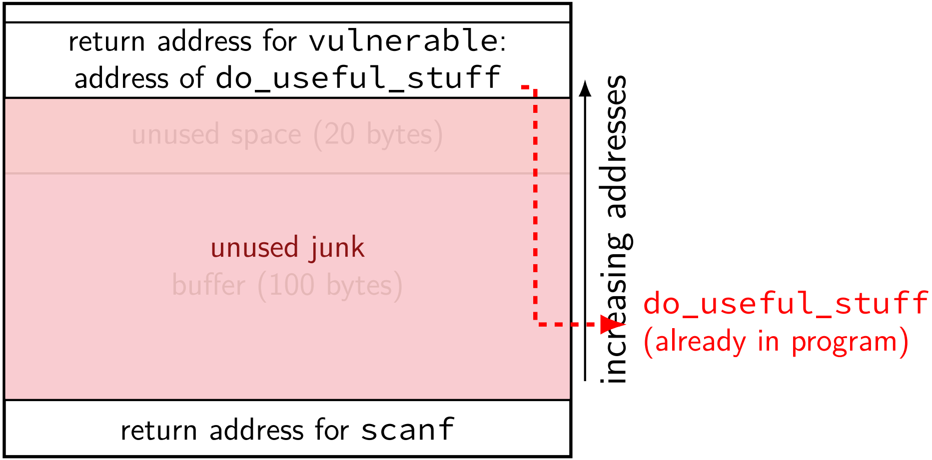
program address range	read?	write?	exec?	real address
`0x0000 --- 0x0FFF`	no	no	no	---
`0x1000 --- 0x1FFF`	no	no	no	---
...				
`0x40 0000 --- 0x40 0FFF`	yes	no	yes	`0x...`
`0x40 1000 --- 0x40 1FFF`	yes	no	yes	`0x...`
`0x40 2000 --- 0x40 2FFF`	yes	no	yes	`0x...`
...				
`0x60 0000 --- 0x60 0FFF`	yes	yes	no	`0x...`
`0x60 1000 --- 0x60 1FFF`	yes	yes	no	`0x...`
...				
0x7FFF FF00 0000 — 0x7FFF FF00 0FFF	yes	yes	no	`0x...`
0x7FFF FF00 1000 — 0x7FFF FF00 1FFF	yes	yes	no	`0x...`
...				

malloc/new guard pages



return-to-somewhere

highest address (stack started here)



return address for `vulnerable`:
address of `do_useful_stuff`

unused space (20 bytes)

unused junk
buffer (100 bytes)

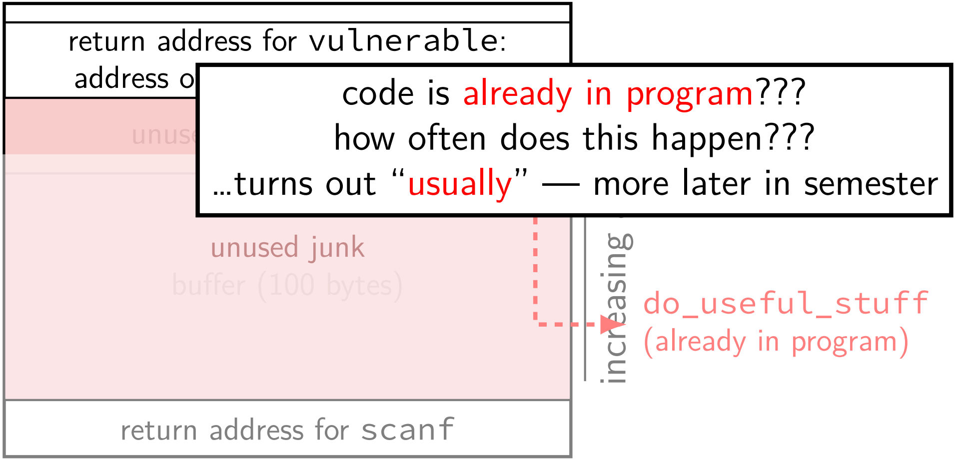
return address for `scanf`

increasing addresses

`do_useful_stuff`
(already in program)

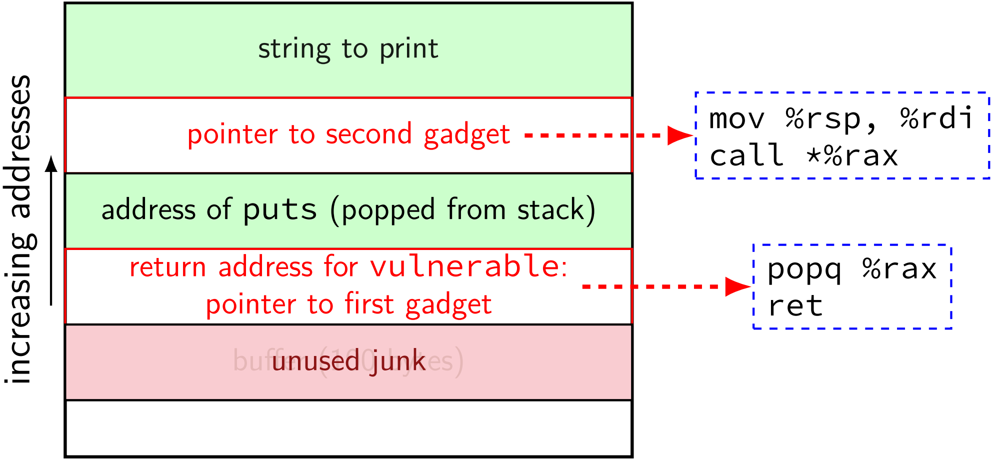
return-to-somewhere

highest address (stack started here)



code is already in program???
how often does this happen???
...turns out “usually” — more later in semester

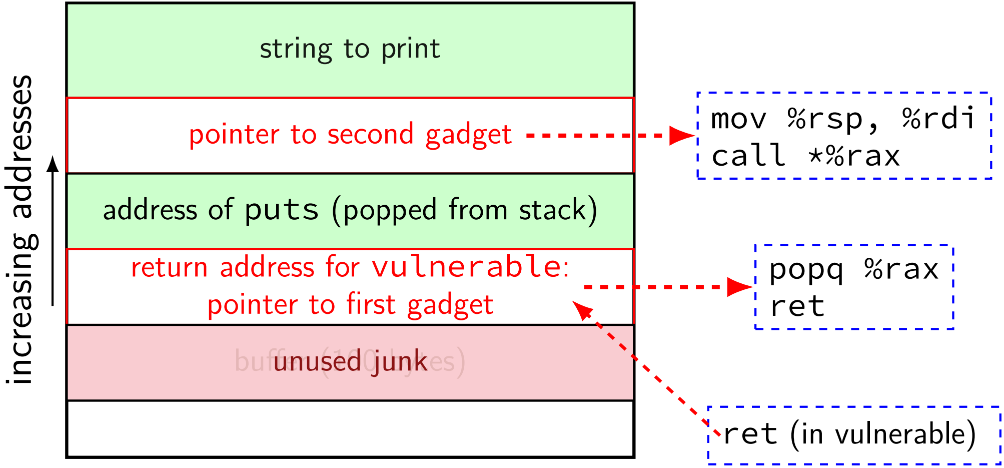
ROP chain



```
mov %rsp, %rdi
call *%rax
```

```
popq %rax
ret
```

ROP chain



increasing addresses

string to print

pointer to second gadget

```
mov %rsp, %rdi
call *%rax
```

address of `puts` (popped from stack)

return address for `vulnerable`:

pointer to first gadget

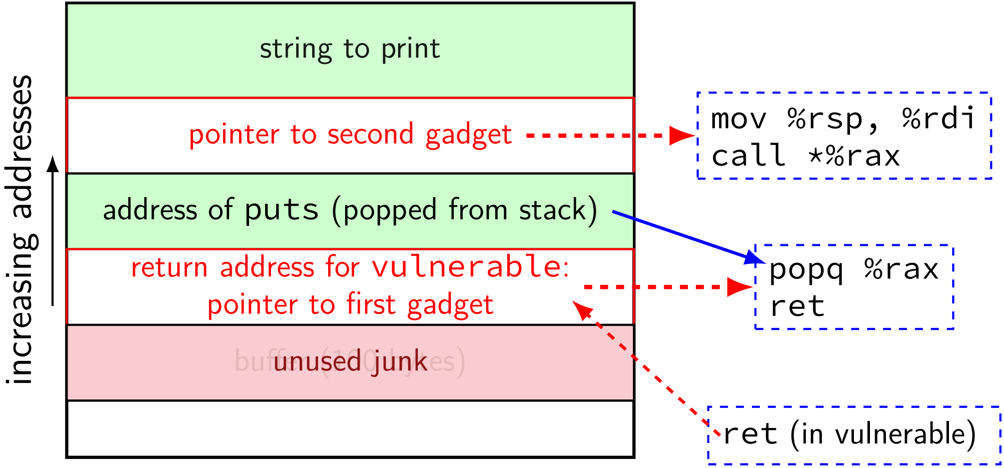
```
popq %rax
ret
```

buffer (100 bytes)

unused junk

`ret` (in vulnerable)

ROP chain



increasing addresses

string to print

pointer to second gadget

address of `puts` (popped from stack)

return address for `vulnerable`:
pointer to first gadget

buffer (100 bytes)

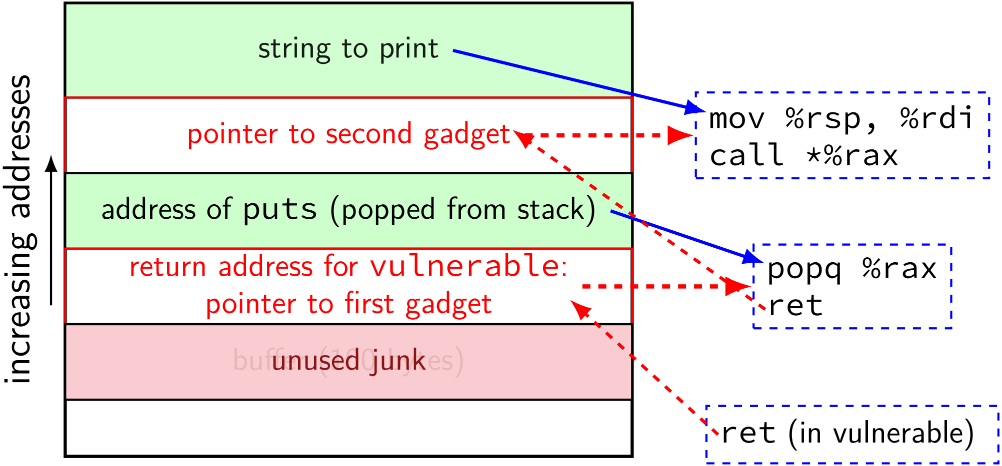
unused junk

```
mov %rsp, %rdi
call *%rax
```

```
popq %rax
ret
```

`ret` (in vulnerable)

ROP chain



increasing addresses

string to print

pointer to second gadget

address of `puts` (popped from stack)

return address for `vulnerable`:
pointer to first gadget

buffer (100 bytes)

unused junk

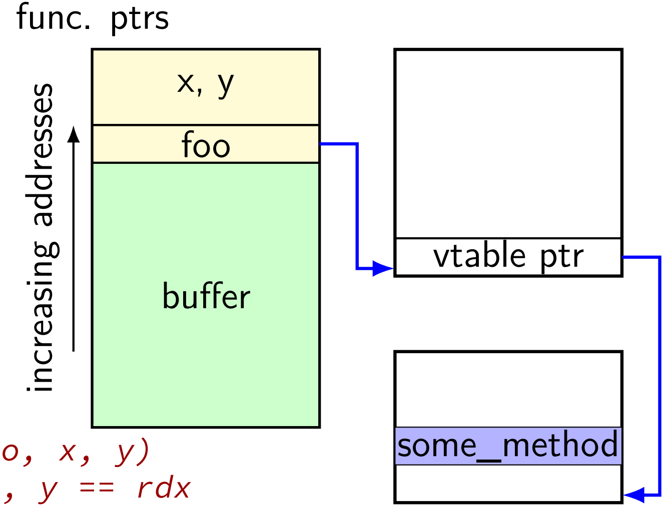
```
mov %rsp, %rdi
call *%rax
```

```
popq %rax
ret
```

`ret` (in vulnerable)

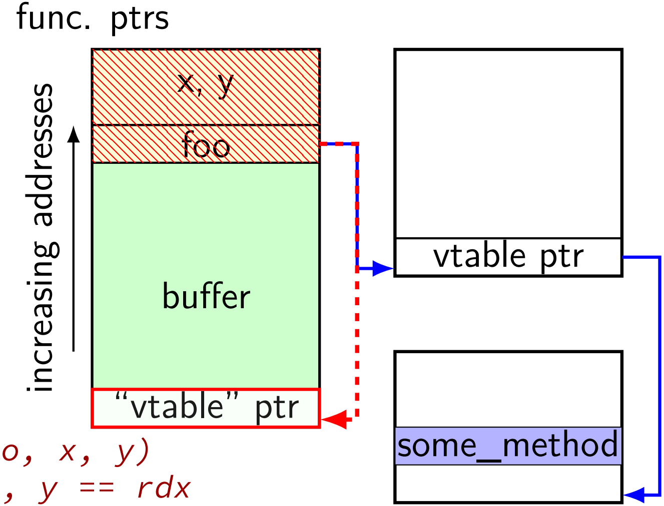
VTable overwrite with gadget

```
class Bar {
  char buffer[100];
  Foo *foo;
  int x, y;
  ...
};

void Bar::vulnerable() {
  gets(buffer);
  foo->some_method(x, y);
  // (*foo->vtable[K])(foo, x, y)
  // foo == rdi, x == rsi, y == rdx
}
```



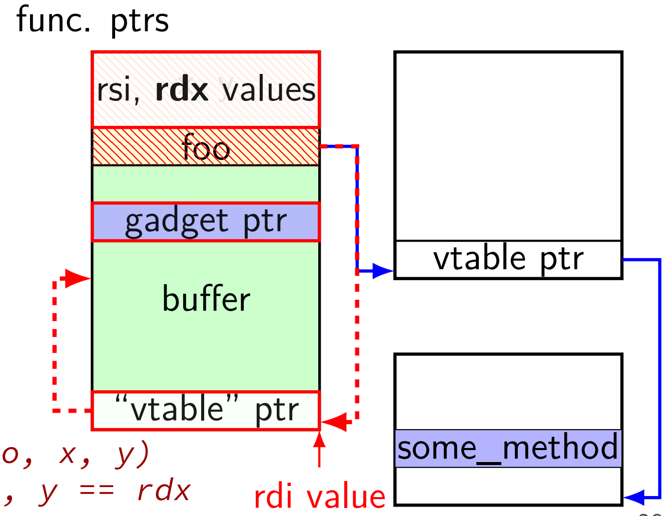
VTable overwrite with gadget

```
class Bar {
  char buffer[100];
  Foo *foo;
  int x, y;
  ...
};

void Bar::vulnerable() {
  gets(buffer);
  foo->some_method(x, y);
  // (*foo->vtable[K])(foo, x, y)
  // foo == rdi, x == rsi, y == rdx
}
```



VTable overwrite with gadget

```
class Bar {
  char buffer[100];
  Foo *foo;
  int x, y;
  ...
};

void Bar::vulnerable() {
  gets(buffer);
  foo->some_method(x, y);
  // (*foo->vtable[K])(foo, x, y)
  // foo == rdi, x == rsi, y == rdx
}
```

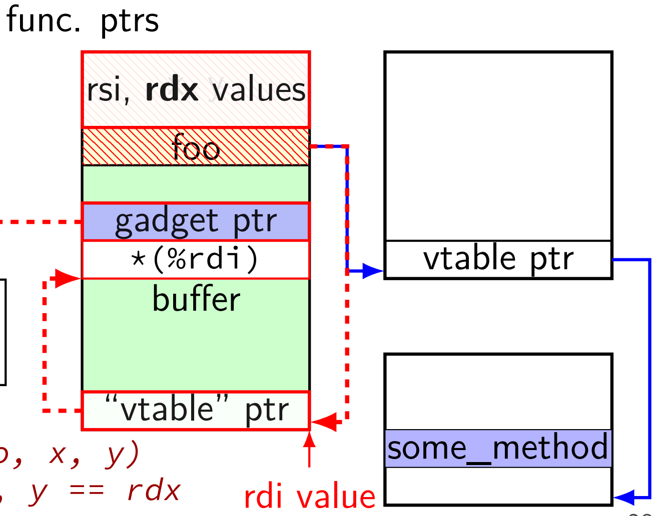


VTable overwrite with gadget

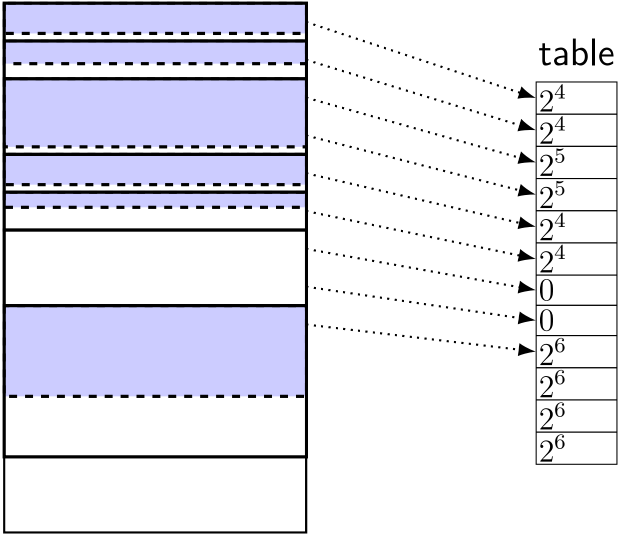
```
class Bar {
   char buffer[100];
   Foo *foo;
   int x, y;
   ...
};
   gets(buffer);
   foo->some_method(x, y);
   // (*foo->vtable[K])(foo, x, y)
   // foo == rdi, x == rsi, y == rdx
}
```

gadget:

```
push %rdx; jmp *(%rdi)
```



allocations and lookup table



object allocated in power-of-two 'slots'

table stores sizes for each 16 bytes

addresses multiples of size (may require padding)

sizes are powers of two (may require padding)