More Buffer Overflows

on the homework

due Friday + 1 week

questions?

big hint in assignment: `gets` is what does buffer overflow

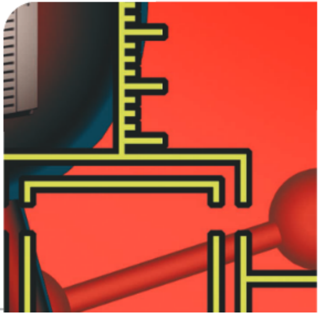
reading the assembly should be fairly straightforward

- probably easiest strategy in this case

debugger can find stack addresses you need

Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns

This article describes three powerful general-purpose families of exploits for buffer overruns: arc injection, pointer subterfuge, and heap smashing. These new techniques go beyond the traditional "stack smashing" attack and invalidate traditional assumptions about buffer overruns.



techniques from Pincus and Baker

arc injection AKA return-oriented programming

more detail (+ assignment) later in semester

overwriting data pointers

overwriting function pointers

overwriting pointers to function pointers

(on heap) overwriting `malloc`'s data structures

other buffer overflows?

examples last time:

luck: "`score`" for quiz on stack next to answer

"arc injection" — return to existing code

data pointer on stack

techniques from Pincus and Baker

arc injection AKA return-oriented programming

more detail (+ assignment) later in semester

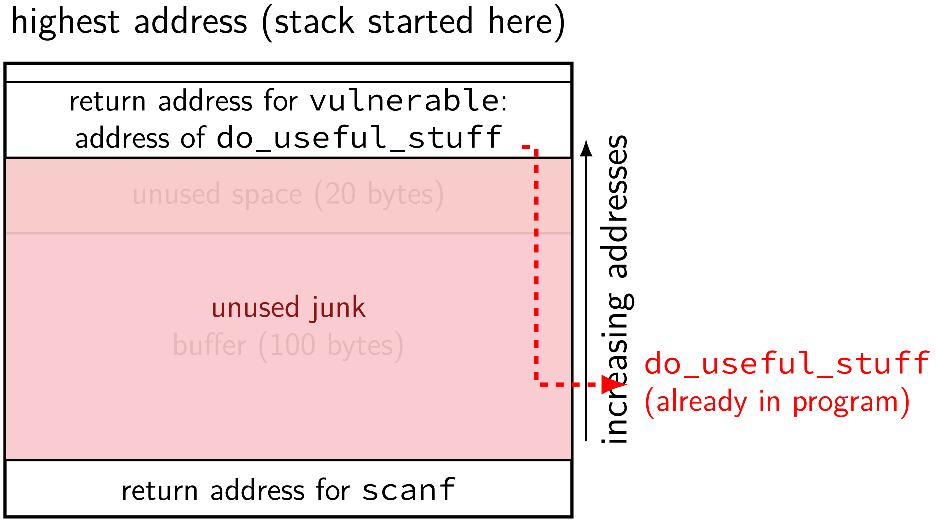
overwriting data pointers

overwriting function pointers

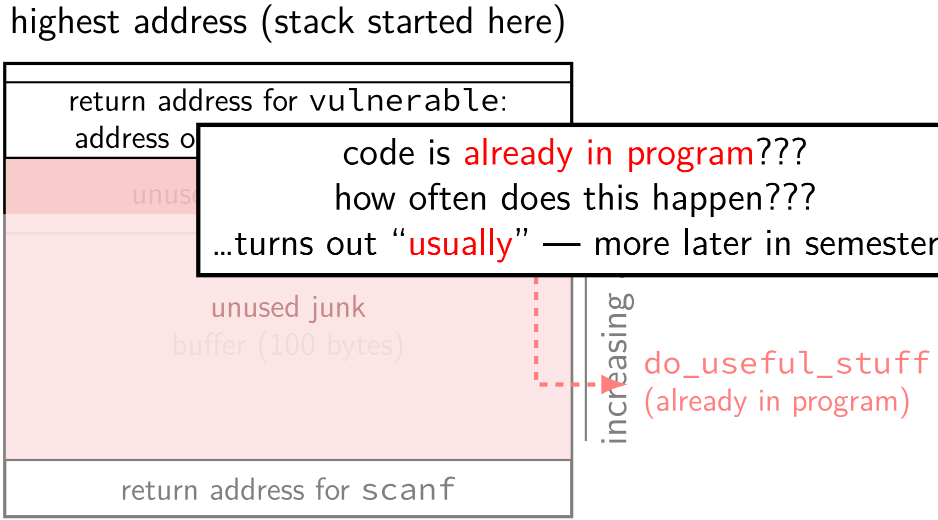
overwriting pointers to function pointers

(on heap) overwriting `malloc`'s data structures

return-to-somewhere



return-to-somewhere



code is already in program???

how often does this happen???

...turns out "usually" — more later in semester

techniques from Pincus and Baker

arc injection AKA return-oriented programming

more detail (+ assignment) later in semester

overwriting data pointers

overwriting function pointers

overwriting pointers to function pointers

(on heap) overwriting `malloc`'s data structures

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

arbitrary memory write

bunch of scenarios that lead to single arbitrary memory write

how can attacker exploit this?

arbitrary memory write

bunch of scenarios that lead to single arbitrary memory write

how can attacker exploit this?

overwrite return address directly

overwrite other function/code address pointer?

overwrite existing machine code (insert jump?)

overwrite another data pointer — copy more?

arbitrary memory write

bunch of scenarios that lead to single arbitrary memory write

how can attacker exploit this?

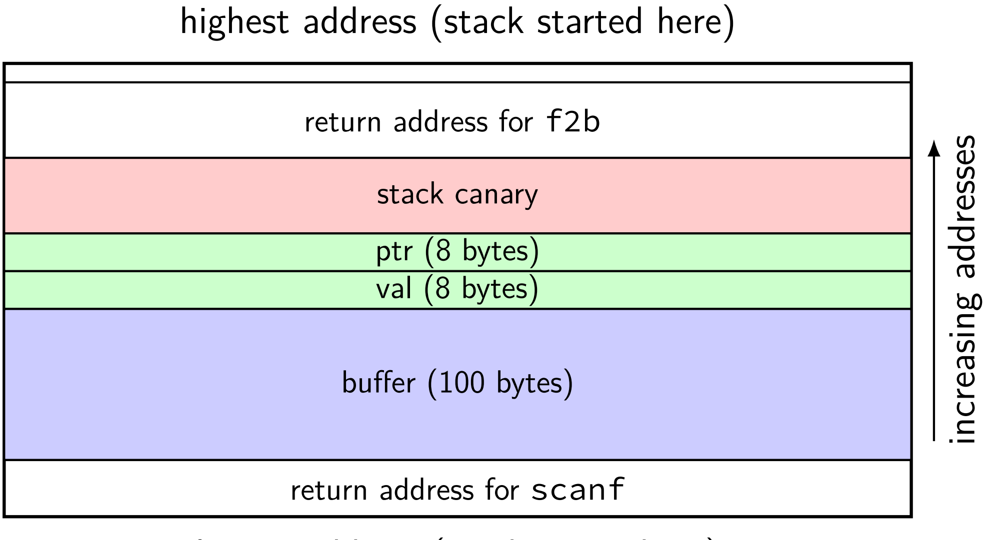
overwrite return address directly

overwrite other function/code address pointer?

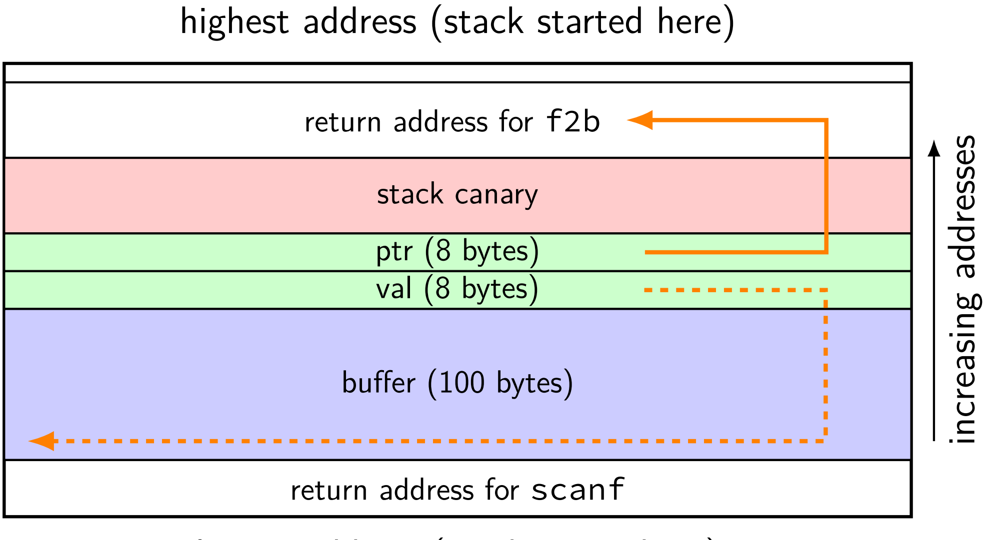
overwrite existing machine code (insert jump?)

overwrite another data pointer — copy more?

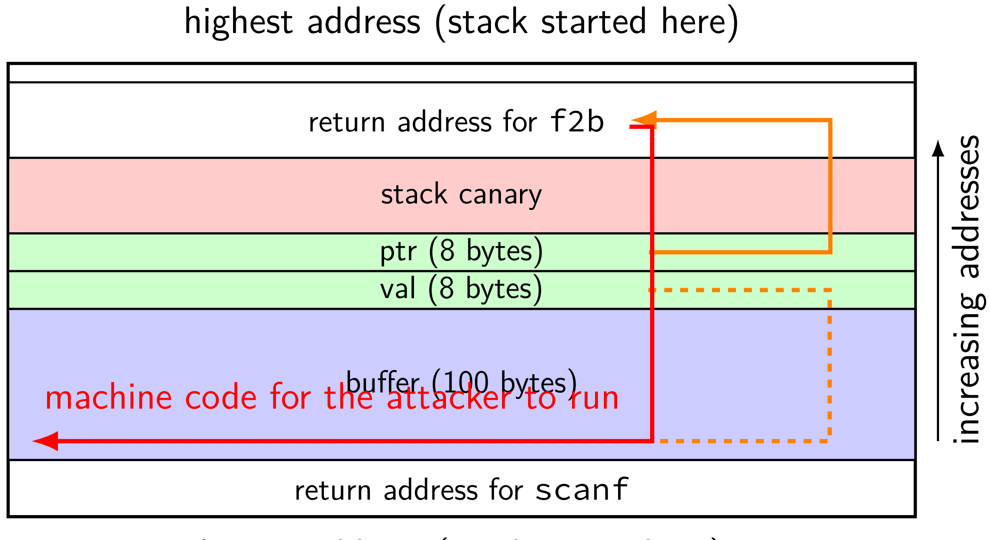
skipping the canary



skipping the canary



skipping the canary



fragility

problem: need to know exact address of return address

discussed how stack location varies — this is tricky/unreliable

arbitrary memory write

bunch of scenarios that lead to single arbitrary memory write

how can attacker exploit this?

overwrite return address directly

overwrite other function/code address pointer?

overwrite existing machine code (insert jump?)

overwrite another data pointer — copy more?

function pointers?

```
int (*compare)(char *, char *);

if (sortCaseSensitive) {
    compare = compareStringsExactly;
} else {
    compare = compareStringsInsensitive;
}
...
if ((*compare)(string1, string2) == CMP_LESS) {
    ...
}
```

function pointers are common?

used in dynamic linking (stubs!)

in large C projects

used to implement C++ virtual functions

dynamic linking stubs

```
0000000000004004a0 <__printf_chk@plt>:
  4004a0:       ff 25 82 0b 20 00       jmpq   *0x200b82(%rip)
                                             # 601028 <_GLOBAL_OFFSET_TABLE_+0x28>
  4004a6:       68 02 00 00 00          pushq  $0x2
  4004ab:       e9 c0 ff ff ff          jmpq   400470 <_init+0x28>
```

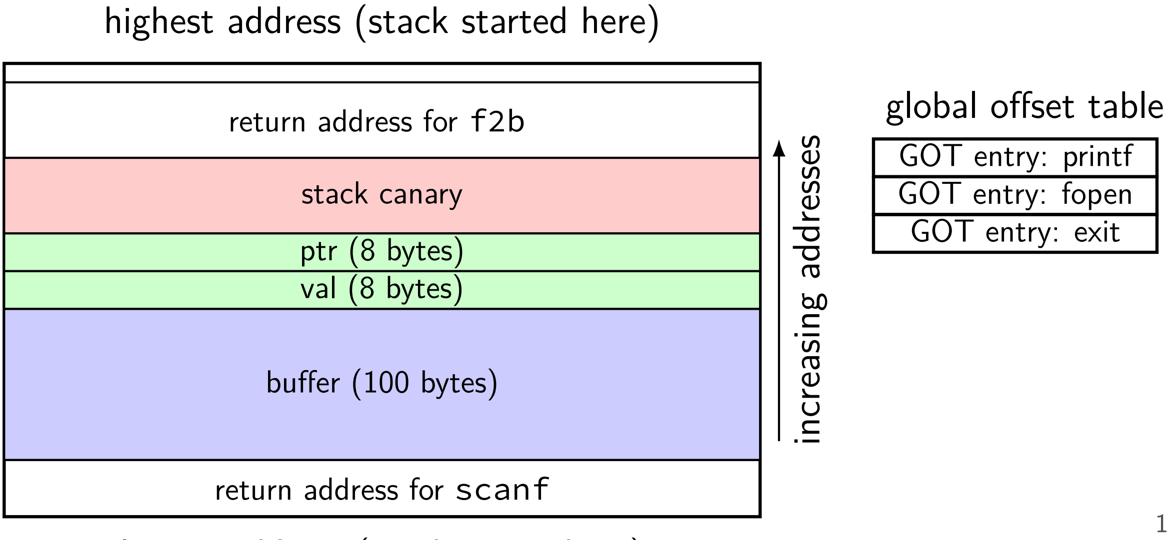
jumps to `_GLOBAL_OFFSET_TABLE_[5]`

`_GLOBAL_OFFSET_TABLE_[5]` always at address `0x601028`

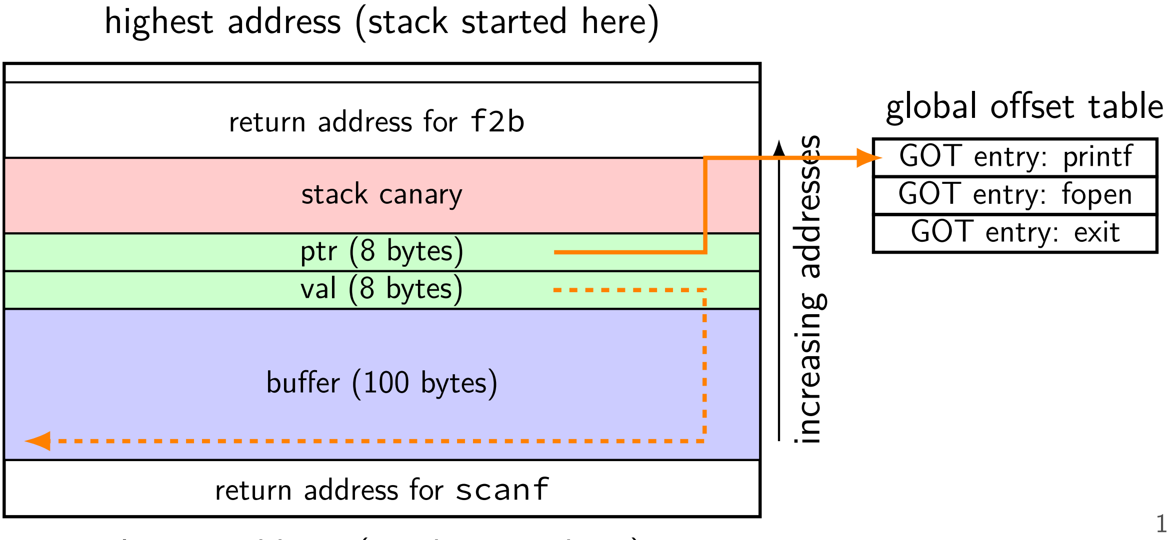
`_GLOBAL_OFFSET_TABLE_[5]` is probably writeable

- if lazy binding — normally updated first time printf called

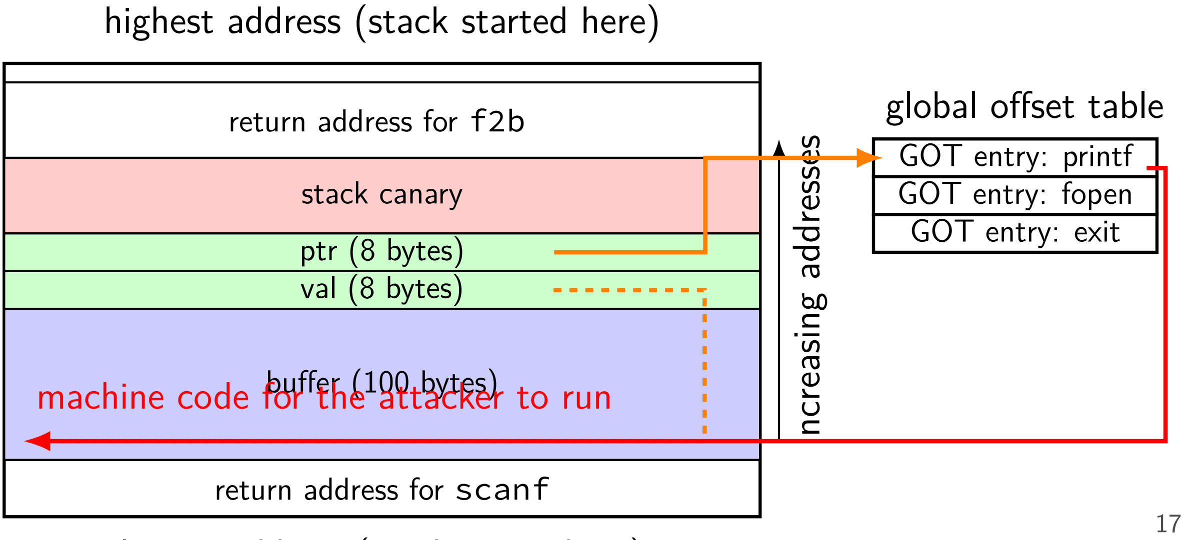
attacking the GOT



attacking the GOT



attacking the GOT



function pointers are common?

used in dynamic linking (stubs!)

in large C projects

used to implement C++ virtual functions

function pointer tables: Linux kernel (1)

```
struct file {
    union {
        struct llist_node       fu_llist;
        struct rcu_head         fu_rcuhead;
    } f_u;
    struct path             f_path;
    struct inode            *f_inode;       /* cached value */
    const struct file_operations    *f_op;

    /*
     * Protects f_ep_links, f_flags.
     * Must not be taken from IRQ context.
     */
    spinlock_t              f_lock;
```

function pointer tables: Linux kernel (2)

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *,
                     size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *,
                      size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iterate) (struct file *, struct dir_context *);
    ...
};
```

function pointers are common?

used in dynamic linking (stubs!)

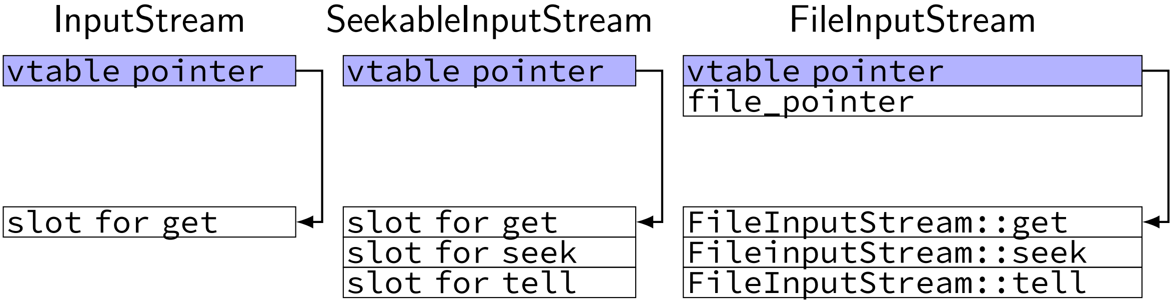
in large C projects

used to implement C++ virtual functions

C++ inheritence

```
class InputStream {
public:
    virtual int get() = 0;
    // Java: abstract int get();
    ...
};
class SeekableInputStream : public InputStream {
public:
    virtual void seek(int offset) = 0;
    virtual int tell() = 0;
};
class FileInputStream : public InputStream {
public:
    int get();
    void seek(int offset);
    int tell();
    ...
};
```

C++ inheritence: memory layout



C++ implementation (pseudo-code)

```
struct InputStream_vtable {
    int (*get)(InputStream* this);
};

struct InputStream {
    InputStream_vtable *vtable;
};

...

    InputStream *s = ...;
    int c = (s->vtable->get)(s);
```

C++ implementation (pseudo-code)

```
struct SeekableInputStream_vtable {
    struct InputStream_vtable as_InputStream;
    void (*seek)(SeekableInputStream* this, int offset);
    int (*tell)(SeekableInputStream* this);
};

struct FileInputStream {
    SeekableInputStream_vtable *vtable;
    FILE *file_pointer;
};

...

    FileInputStream file_in = { the_FileInputStream_vtable,  ... };
    InputStream *s = (InputStream*) &file_in;
```

C++ implementation (pseudo-code)

```
SeekableInputStream_vtable the_FileInputStream_vtable = {
    &FileInputStream_get,
    &FileInputStream_seek,
    &FileInputStream_tell,
};

...

    FileInputStream file_in = { the_FileInputStream_vtable,  ... };
    InputStream *s = (InputStream*) &file_in;
```

attacking function pointer tables

option 1: overwrite table entry directly

- required/easy for Global Offset Table — fixed location
- usually not possible for VTables — read-only memory

option 2: create table in buffer (big list of pointers to shellcode), point to buffer

- useful when table pointer next to buffer
- (e.g. C++ object on stack next to buffer)

case study (simplified)

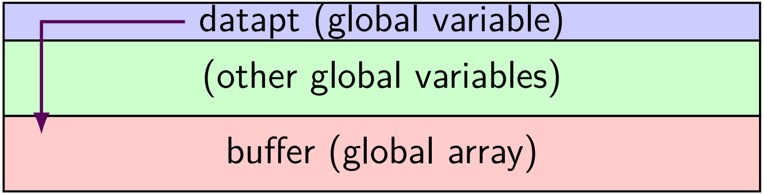
bug in NTPd (Network Time Protocol Daemon)

via Stepher Röttger, “Finding and exploiting ntpd vulnerabilities”

```
static void
ctl_putdata(
  const char *dp,
  unsigned int dlen,
  int bin   /* set to 1 when data is binary */
  ) {
    ...
    memmove((char *)datapt, dp, (unsigned)dlen);
    datapt += dlen;
    datalinelen += dlen;
}
```

the target

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

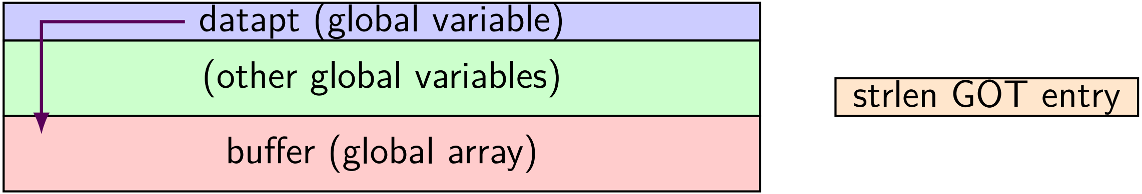


more context

```
memmove((char *)datapt, dp, (unsigned)dlen);
...
...
strlen(some_user_supplied_string)
/* calls strlen@plt
   looks up global offset table entry! */
```

the target

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



overall exploit

overwrite `datapt` to point to strlen GOT entry

overwrite value of strlen GOT entry

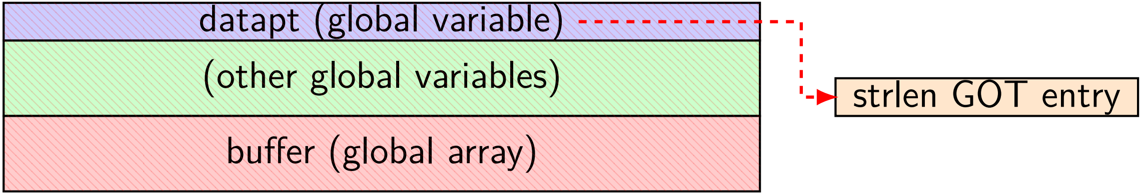
example target: `system` function

- executes command-line command specified by argument

supply string to provide argument to "`strlen`"

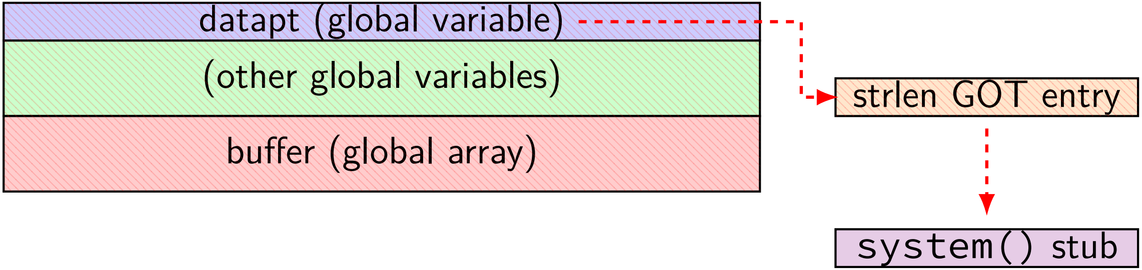
the target

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



the target

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



overall exploit: reality

real exploit was more complicated

needed to defeat more mitigations

needed to deal with not being able to write `\0`

actually tricky to send things that trigger buffer write
(meant to be local-only)

beyond normal buffer overflows

pretty much every memory error is a problem

will look at exploiting:

off-by-one buffer overflows (!)

heap buffer overflows

double-frees

use-after-free

integer overflows in size calculations

beyond normal buffer overflows

pretty much every memory error is a problem

will look at exploiting:

off-by-one buffer overflows (!)

heap buffer overflows

double-frees

use-after-free

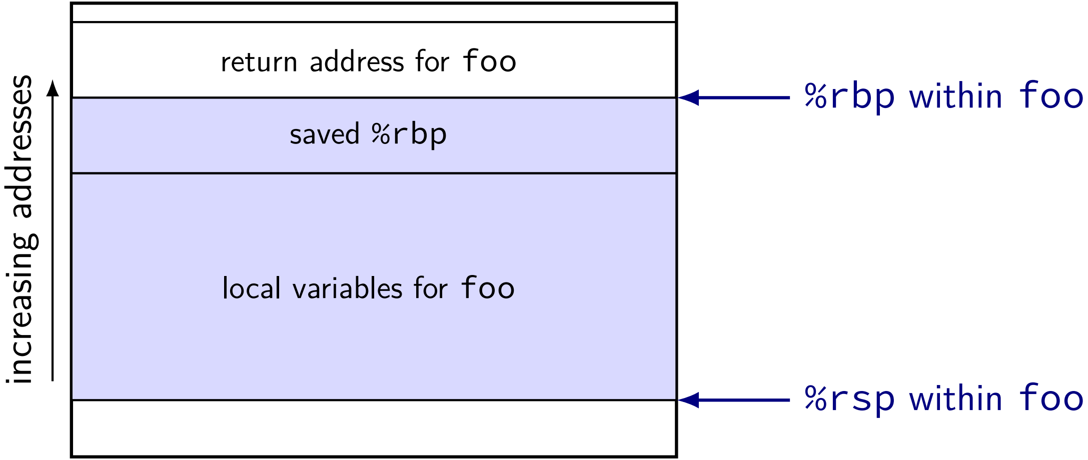
integer overflows in size calculations

preliminaries

frame pointers are commonly used in addition to stack pointers

not something we've seen in x86-64 assembly

frame pointers



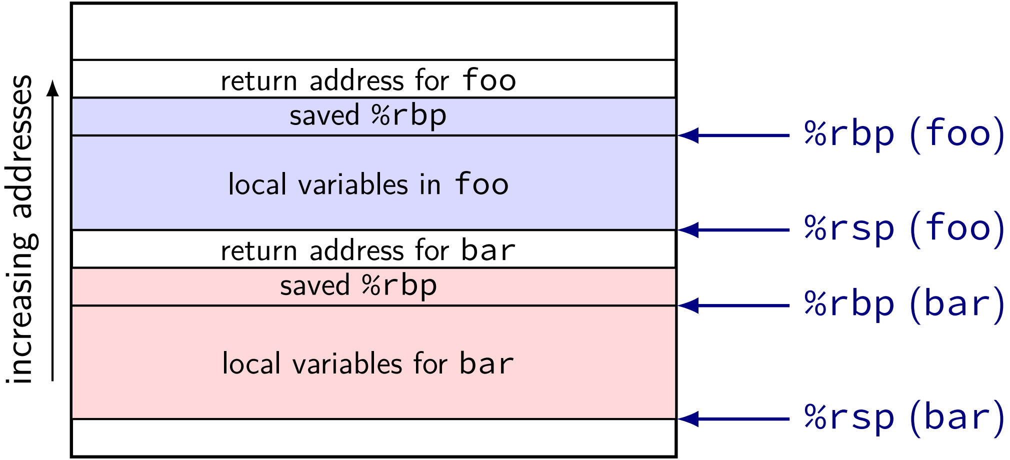
frame pointer code

```
foo:
  // prologue
  pushq %rbp
  enter $120, $1
  ...
  ...
  ...
  leave
  ret
```

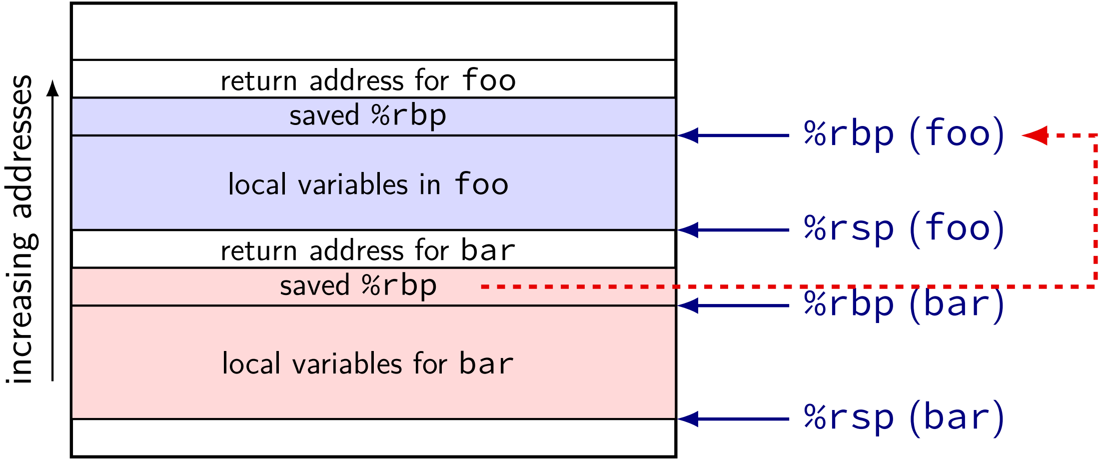
```
foo:
  // prologue
  pushq %rbp
  movq %rsp, %rbp
  subq $120, %rsp
  ...
  ...
  ...
  movq %rbp, %rsp
  popq %rbp
  ret
```

```
foo:
  // prologue
  sub $120, %rsp
  ...
  ...
  ...
  add $120, %rsp
  ret
```

stack layout: two functions



stack layout: two functions



why frame pointers?

makes writing debuggers easier

- otherwise: need table of info about stack allocations
  (just to get a stack trace)

easier for manual assembly writing

- no need to track how large stack frame is

allows 'dynamic' allocation in middle of function

why not frame pointers?

wastes a register

debugging information is more sophisticated

compiler has no trouble matching sizes in prologue/epilogue

we use the heap, not the stack for dynamic allocations

GCC option:

```
    -fomit-frame-pointer
    -fno-omit-frame-pointer
```

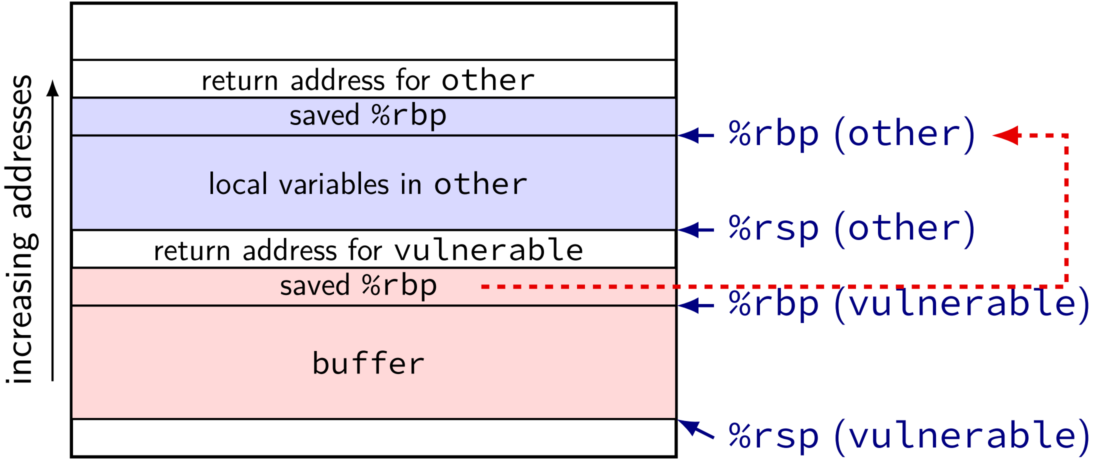
off-by-one-byte

```
int vulnerable(
        const char *attacker_controlled,
        int len) {
    char buffer[100];
    for (int i = 0; i <= 100 && i <= len; ++i) {
        buffer[i] = attacker_controlled[i];
    }
}

int other() {
    ...
    vulnerable(...);
}
```

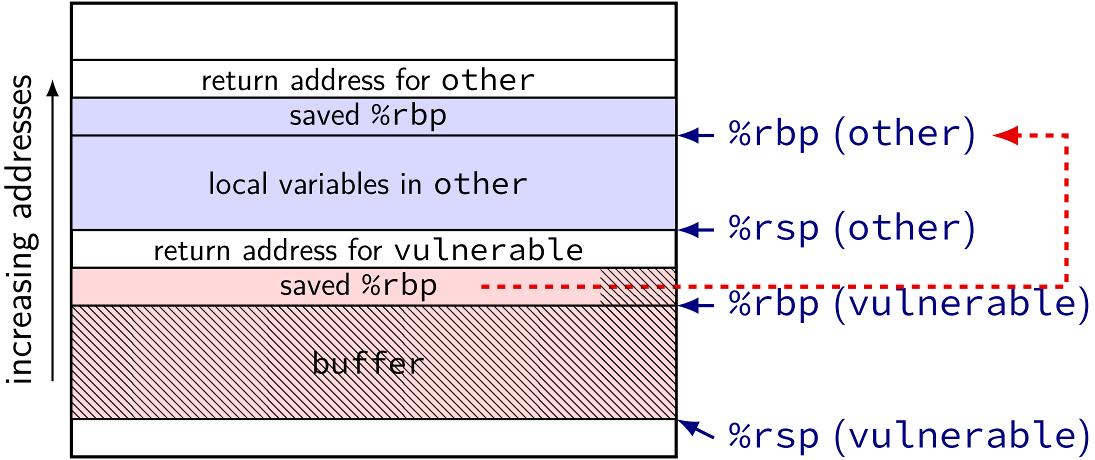
off-by-one-byte

```
int vulnerable(
        const char *attacker_controlled,
        int len) {
    char buffer[100];
    for (int i = 0; i <= 100 && i <= len; ++i) {
        buffer[i] = attacker_controlled[i];
    }
}

int other() {
    ...
    vulnerable(...);
}
```

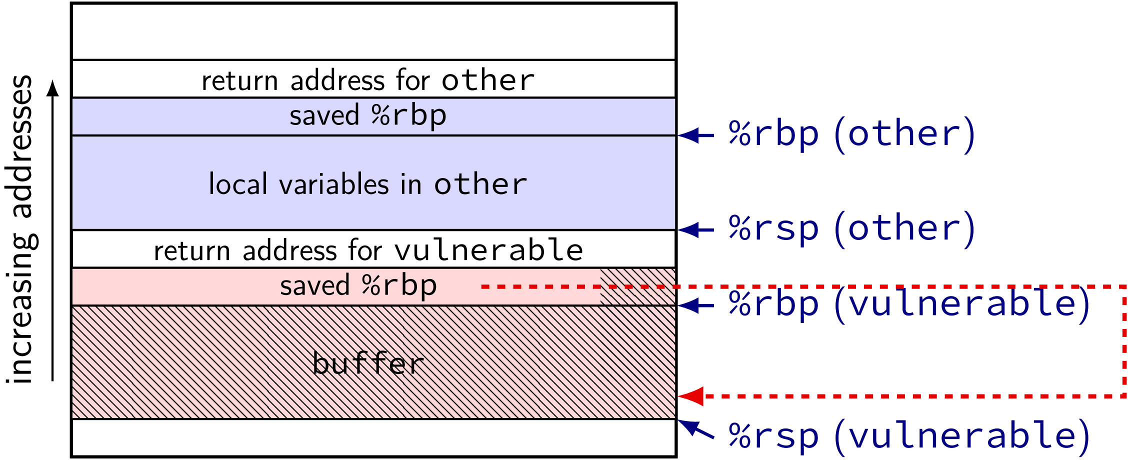
vulnerable stack layout



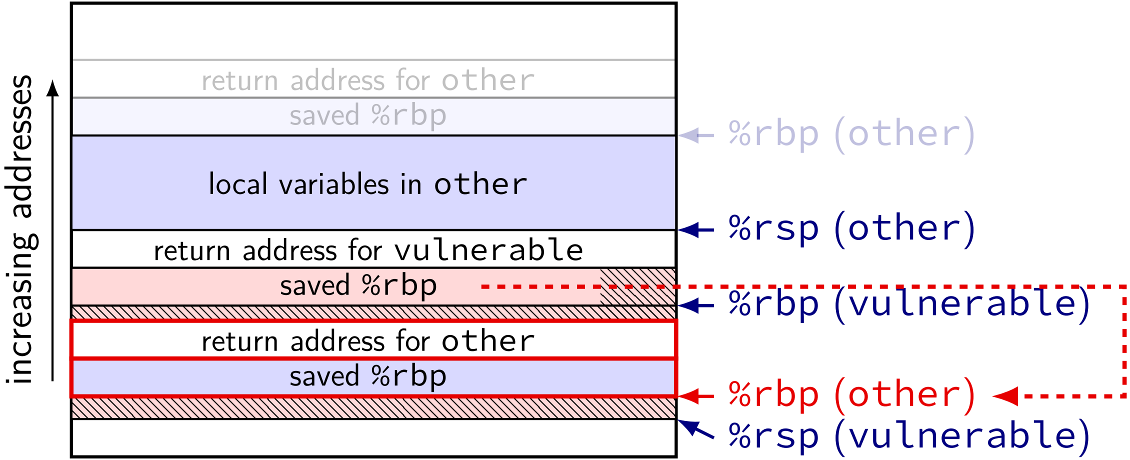
vulnerable stack layout



vulnerable stack layout



vulnerable stack layout



off-by-one frame pointer

little endian: change least sig. bit of frame pointer

off-by-one byte: max adjustment 256

- question: is that attacker controlled space?

what if you can only write `0` to last byte?

- moves frame pointer to lower address
- often attacker-controlled address!

frame pointer control

after controlling frame pointer, set return address of other

then same idea as stack smashing — point to attacker controlled machine code

can also control local variables of calling function

- potentially useful even with stack canaries/no info. disclosure

vulnerable code (real)

realpath — ../foo → /home/cr4bd/foo

remotely exploitable in wu-FTPd (File Transfer Protocol server)

bad length check — accounted for extra "/" wrong

```
char resolved[MAXPATHLEN];
...
if (strlen(resolved) + strlen(wbuf) + rootd + 1 > MAXPATHLEN) {
    errno = ENAMETOOLONG;
    goto err1;
}
if (rootd == 0)
    (void) strcat(resolved, "/");
(void) strcat(resolved, wbuf);
...
```

vulnerable code (real)

`realpath` — `../foo` → `/home/cr4bd/foo`

remotely exploitable in wu-FTPd (File Transfer Protocol server)

bad length check — accounted for extra “/” wrong

```
char resolved[MAXPATHLEN];
...
if (strlen(resolved) + strlen(wbuf) + rootd + 1 > MAXPATHLEN) {
    errno = ENAMETOOLONG;
    goto err1;
}
if (rootd == 0)
    (void) strcat(resolved, "/");
(void) strcat(resolved, wbuf);
...
```

beyond normal buffer overflows

pretty much every memory error is a problem

will look at exploiting:

off-by-one buffer overflows (!)

heap buffer overflows

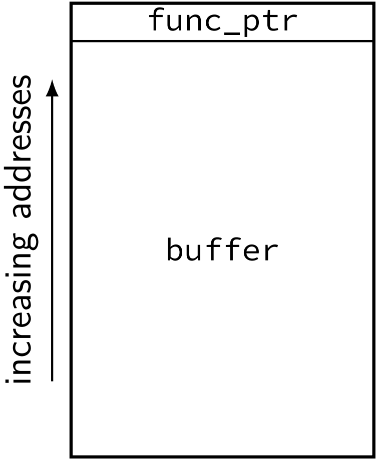
double-frees

use-after-free

integer overflows in size calculations

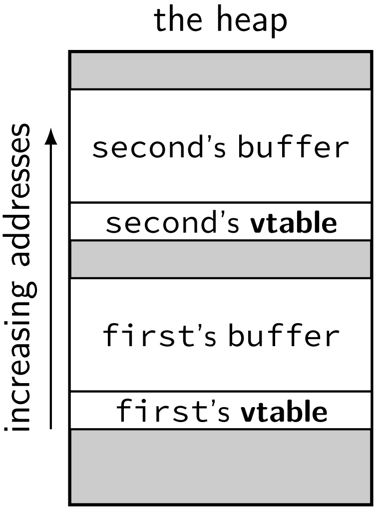
easy heap overflows

```
struct foo {
    char buffer[100];
    void (*func_ptr)(void);
};
```



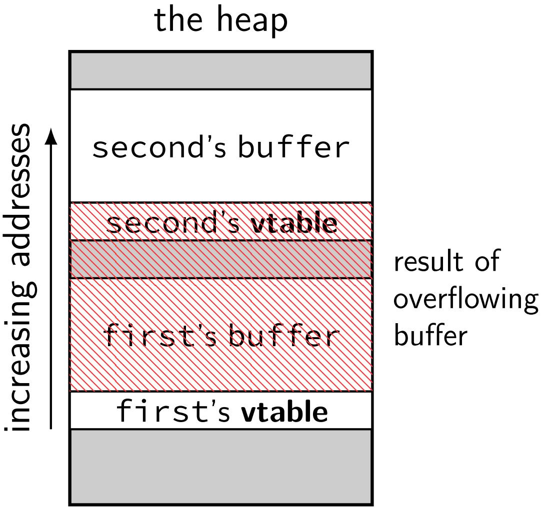
heap overflow: adjacent allocations

```
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

“lucky” adjancent objects

same things possible on stack

but stack overflows had nice generic “stack smashing”

is there an equivalent for the heap?

yes (mostly)

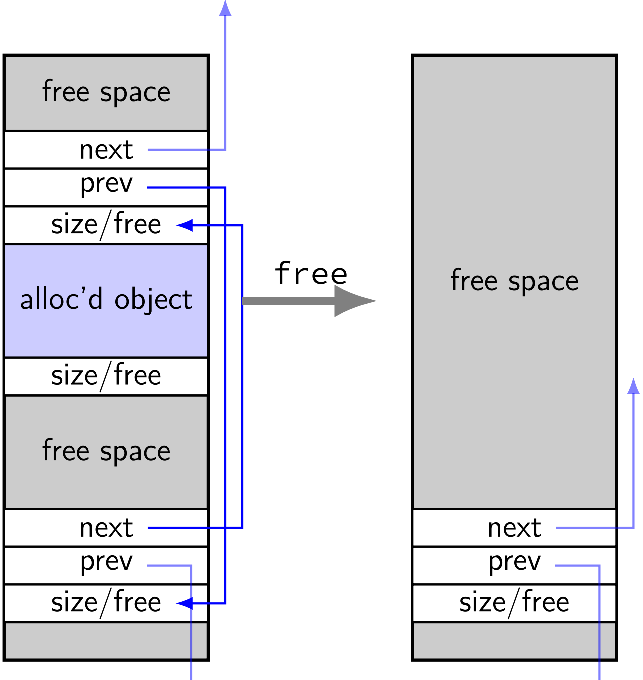
diversion: implementing malloc/new

many ways to implement malloc/new

we will talk about one common technique +

heap object

```
struct AllocInfo {
    bool free;
    int size;
    AllocInfo *prev;
    AllocInfo *next;
};
```



implementing free()

```
int free(void *object) {
    ...
    if (block_after->free) {
        /* unlink from list */
        new_block->size += block_after->size;
        block_after->prev->next = block_after->next;
        block_after->next->prev = block_after->prev;
    }
    ...
}
```

implementing free()

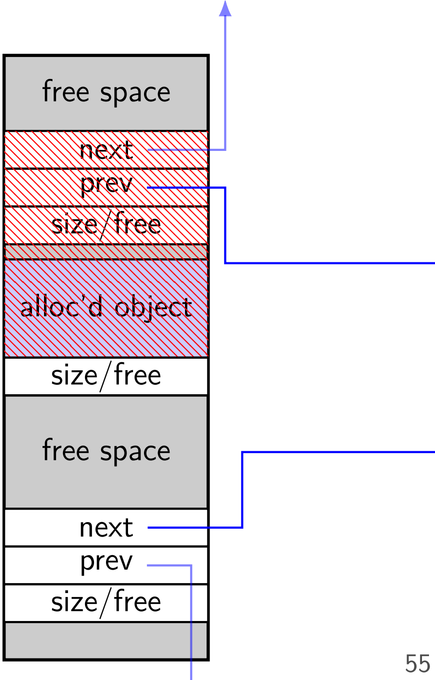
```
int free(void *object) {
    ...
    if (block_after->free) {
        /* unlink from list */
        new_block->size += block_after->size;
        block_after->prev->next = block_after->next;
        block_after->next->prev = block_after->prev;
    }
    ...
}
```

arbitrary memory write

also other list management operations

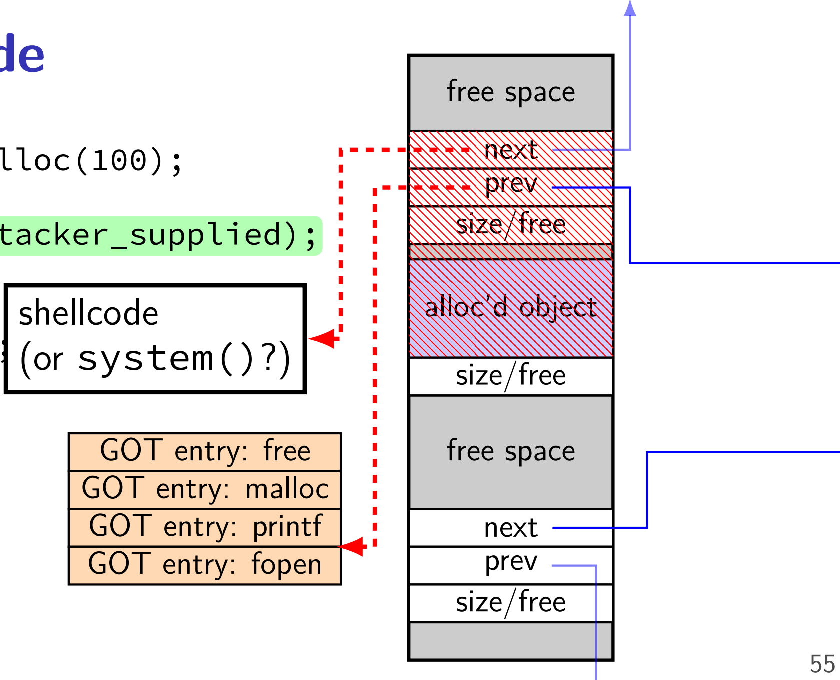
vulnerable code

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



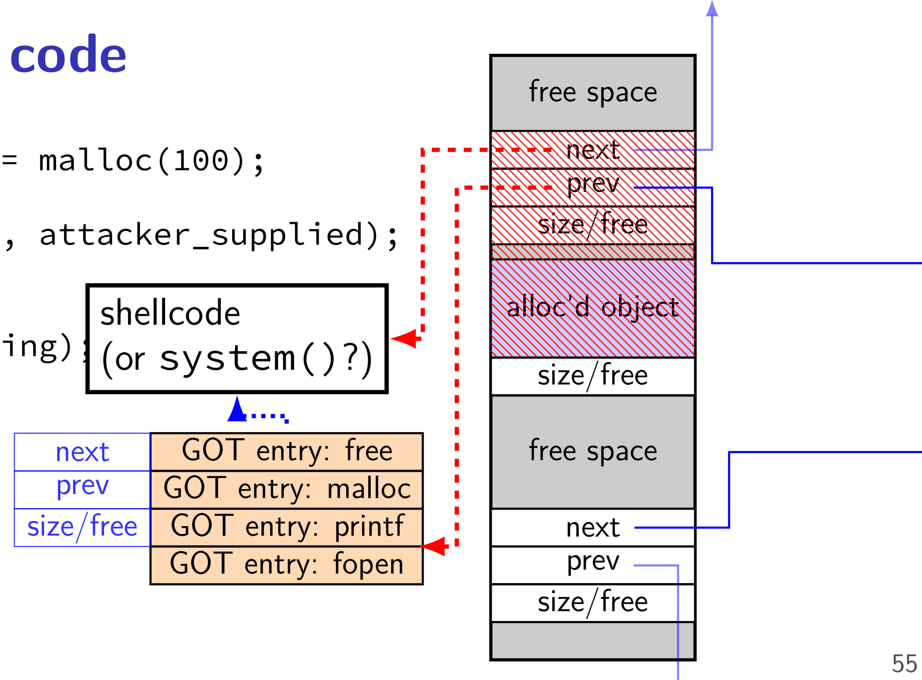
vulnerable code

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



vulnerable code

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



beyond normal buffer overflows

pretty much every memory error is a problem

will look at exploiting:

off-by-one buffer overflows (!)

heap buffer overflows

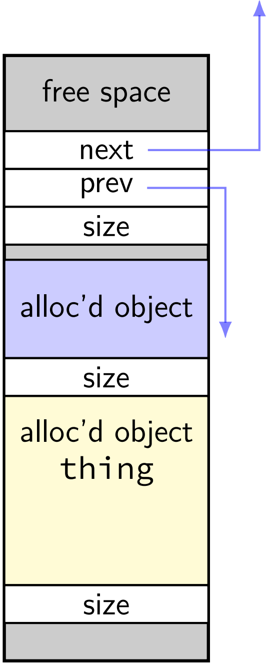
double-frees

use-after-free

integer overflows in size calculations

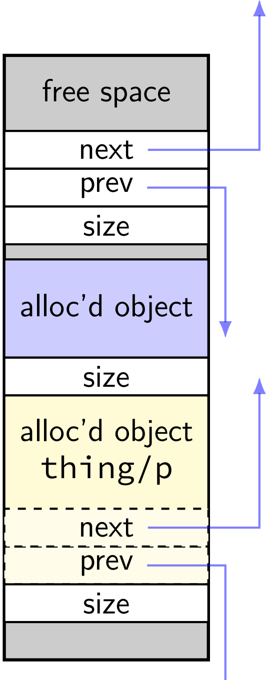
double-frees

```
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);
...
```

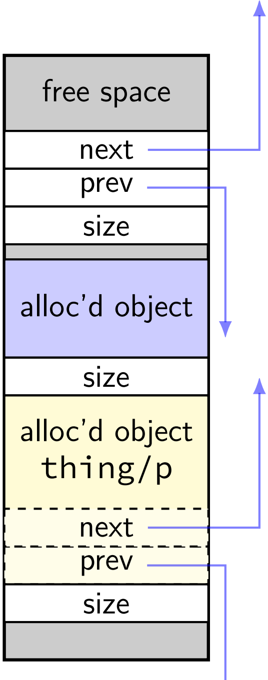


double-frees

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

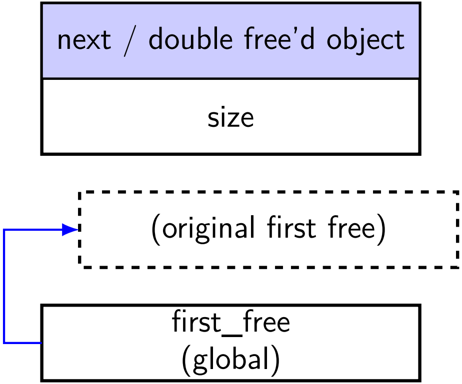
malloc returns something still on free list
because double-free made loop in linked list

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



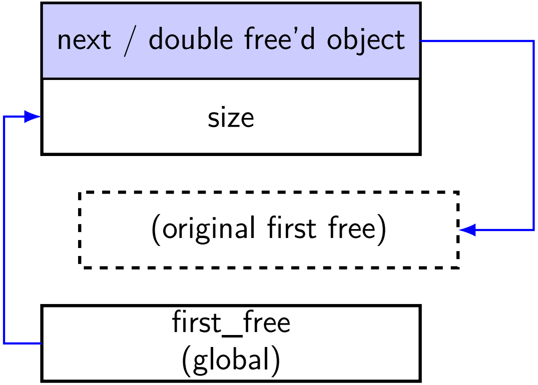
double-free expansion

```
// 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, ...);
```



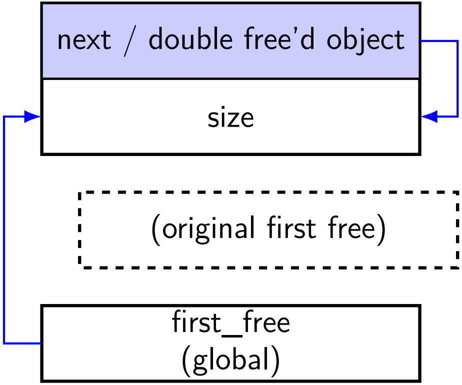
double-free expansion

```
// 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, ...);
```



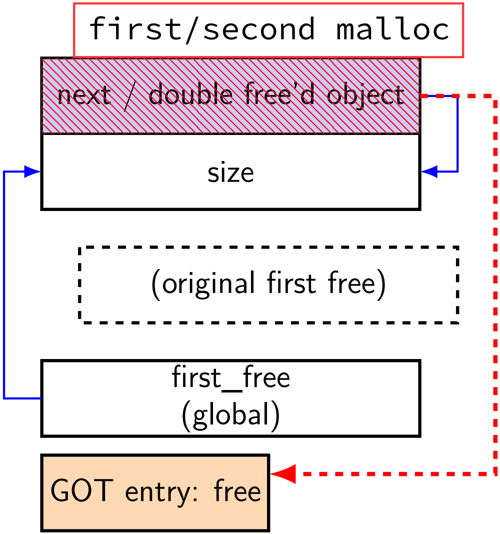
double-free expansion

```
// 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, ...);
```



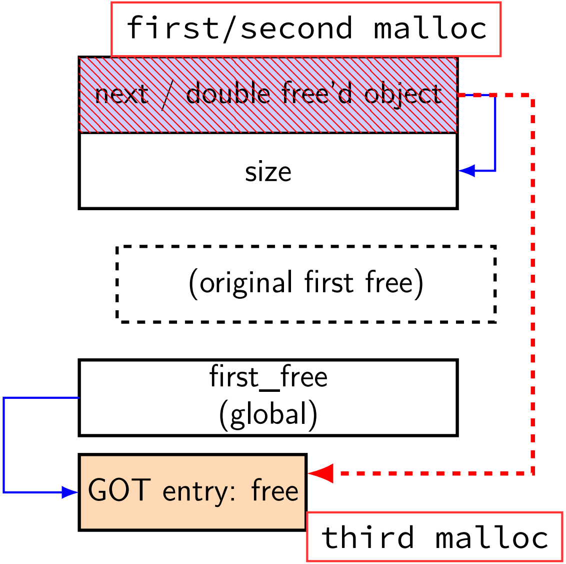
double-free expansion

```
// 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, ...);
```



double-free notes

this attack has apparently not been possible for a while

most malloc/new's check for double-frees explicitly

(e.g., look for a bit in `size` data)

prevents this issue — also catches programmer errors

pretty cheap

beyond normal buffer overflows

pretty much every memory error is a problem

will look at exploiting:

off-by-one buffer overflows (!)

heap buffer overflows

double-frees

use-after-free

integer overflows in size calculations

vulnerable code

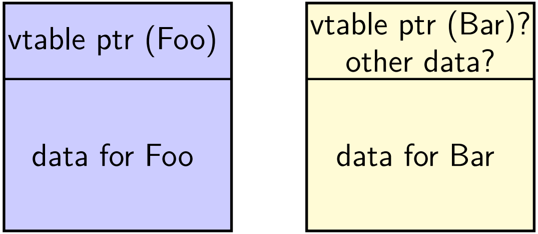
```
class Foo {
    ...
};
Foo *the_foo;
the_foo = new Foo;
...
delete the_foo;
...
something_else = new Bar(...);
the_foo->something();
```

something_else likely where the_foo was

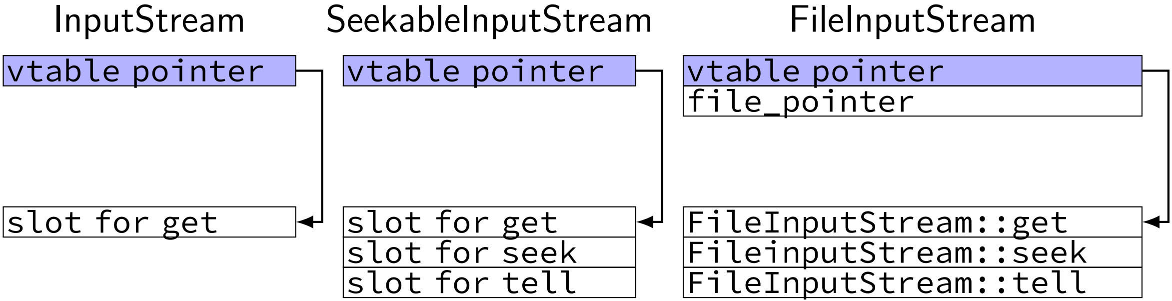
vulnerable code

```
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



C++ inheritence: memory layout



exploiting use after-free

trigger many “bogus” frees; then

allocate many things of same size with “right” pattern

- pointers to shellcode?
- pointers to pointers to `system()`?
- objects with something useful in VTable entry?

trigger use-after-free thing

use-after-free easy cases

common problem for JavaScript implementations

use-after-free'd object often some complex C++ object

- example: representation of video stream

exploits can choose type of object that replaces

- allocate that kind of object in JS

can often arrange to read/write vtable pointer

- depends on layout of thing created
- easy examples: string, array of floating point numbers

beyond normal buffer overflows

pretty much every memory error is a problem

will look at exploiting:

off-by-one buffer overflows (!)

heap buffer overflows

double-frees

use-after-free

integer overflows in size calculations

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

making this reliable

run program with `malloc`, `free` that output parameters

knowledge of how `malloc`/etc. handles different sized objects

"heap spray"

- 32-bit systems — just have your shellcode or target address everywhere
- hope "random" address matches

global variables (fixed addresses) — good place for shellcode

control hijacking generally

usually: need to know/guess program addresses

usually: need to insert executable code

usually: need to overwrite code addresses

next topic: countermeasures against these

later topic: defeating those

later later topic: secure programming languages

first mitigation: stack canaries

saw: stack canaries

tries to stop:

- overwriting code addresses
- (as long it's return addresses)

by assuming:

- compile-in protection
- attacker can't read off the stack
- attacker can't "skip" parts of the stack

second mitigation: address space randomization

problem for the stack smashing assignment

tries to stop:

- know/guess programming addresses

by assuming:

- program doesn't "leak" addresses
- relevant addresses can be changed (not hard-coded in progrma)

next time

comparing mitigations

- what do they assume the attacker can do?
- effect on performance?
- recompilation? rewriting code?