last time

bootstrapping ROP

- look for gadget that sets RSP
- use to point to ROP chain

jump-oriented programming

- looking for “dispatcher gadget” (incr pointer + jump using it)
- form loop returning back to dispatcher gadget

ASLR

- how much randomness; limits with 32-bit systems
- keeping executables/libraries together?

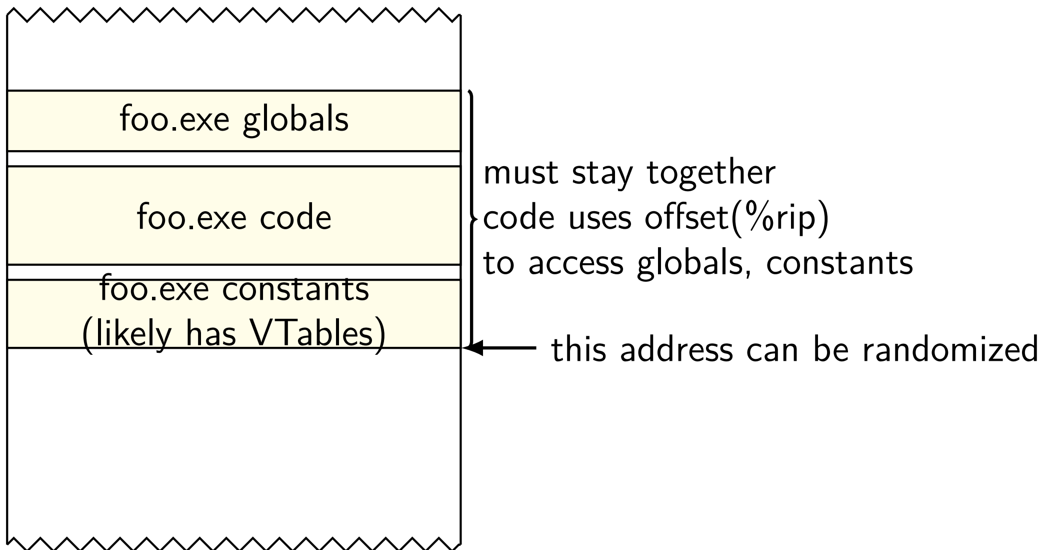
scheduling note

no class Monday

week's quiz due Wednesday

ROP assignment will be ready by Friday (probably by tomorrow morning)

exes, libraries stay together



relocating: Windows

Windows will edit code to relocate

- not everything uses a GOT-like lookup table

typically one fixed location per program/library **per boot**

- same address used across all instances of program/library
- still allows sharing memory

fixup once per program/library per boot

- before ASLR: code could be pre-relocated

Windows + Visual Studio had 'full' ASLR by default since 2010

Windows ASLR limitation

same address in all programs — not very useful against local exploits

PIC: Linux, OS X

Linux, OS X: position-independent code

allows libraries code pages to be shared

...even if loaded at different addresses

avoids per-boot randomization of Windows, but...

exercise: avoiding absolute addresses

```
foo:
        movl    $3, %eax
        cmpq    $5, %rdi
        ja      defaultCase
        jmp     *lookupTable(,%rdi,8)
returnOne:
        movl    $1, %eax
        ret
returnTwo:
        movl    $2, %eax
defaultCase:
        ret
```

```
lookupTable:
    .quad returnOne
    .quad returnTwo
    .quad returnOne
    .quad returnTwo
    .quad returnOne
    .quad returnOne
```

exercise: rewrite this without absolute addresses

but fast

PIE jump-table

```
foo:
  movl    $3, %eax
  cmpq    $5, %rdi
  ja      retDefault
  leaq    jumpTable(%rip),%rax
  movslq (%rax,%rdi,4),%rdx
  addq    %rdx, %rax
  jmp    *%rax
returnTwo:
  movl   $2, %eax
  ret
returnOne:
  movl   $1, %eax
defaultCase:
  ret
```

```
  .section       .rodata
jumpTable:
  .long returnOne—jumpTable
  .long returnTwo—jumpTable
  .long returnOne—jumpTable
  .long returnTwo—jumpTable
  .long returnOne—jumpTable
  .long returnOne—jumpTable
```

PIE jump-table

```
foo:
    movl    $3, %eax
    cmpq    $5, %rdi
    ja      retDefault
    leaq    jumpTable(%rip),%rax
    movslq  (%rax,%rdi,4),%rdx
    addq    %rdx, %rax
    jmp    *%rax
returnTwo:
    movl   $2, %eax
    ret
returnOne:
    movl   $1, %eax
defaultCase:
    ret
```

```
    .section       .rodata
jumpTable:
    .long returnOne−jumpTable
    .long returnTwo−jumpTable
    .long returnOne−jumpTable
    .long returnTwo−jumpTable
    .long returnOne−jumpTable
    .long returnOne−jumpTable
```

PIE jump-table

```
00000000000007ab <foo>:
b8 03 00 00 00          mov    $0x3,%eax
48 83 ff 05             cmp    $0x5,%rdi
77 1b                   ja     7d0 <foo+0x25>
48 8d 05 ab 00 00 00    lea    0xab(%rip),%rax       # 868
48 63 14 b8             movslq (%rax,%rdi,4),%rdx
48 01 d0                add    %rdx,%rax
ff e0                   jmpq   *%rax
b8 02 00 00 00          mov    $0x2,%eax
c3                      retq
b8 01 00 00 00          mov    $0x1,%eax
c3                      retq
...
@ 868: -156 /* offset */
@ 870: -162
...
```

PIE jump-table

```
00000000000007ab <foo>:
b8 03 00 00 00              mov    $0x3,%eax
48 83 ff 05                 cmp    $0x5,%rdi
77 1b                       ja     7d0 <foo+0x25>
48 8d 05 ab 00 00 00        lea    0xab(%rip),%rax          # 868
48 63 14 b8                 movslq (%rax,%rdi,4),%rdx
48 01 d0                    add    %rdx,%rax
ff e0                       jmpq   *%rax
b8 02 00 00 00              mov    $0x2,%eax
c3                          retq
b8 01 00 00 00              mov    $0x1,%eax
c3                          retq
...
@ 868: -156 /* offset */
@ 870: -162
...
```

PIE jump-table

```
00000000000007ab <foo>:
b8 03 00 00 00          mov     $0x3,%eax
48 83 ff 05             cmp     $0x5,%rdi
77 1b                   ja      7d0 <foo+0x25>
48 8d 05 ab 00 00 00    lea     0xab(%rip),%rax        # 868
48 63 14 b8             movslq  (%rax,%rdi,4),%rdx
48 01 d0                add     %rdx,%rax
ff e0                   jmpq    *%rax
b8 02 00 00 00          mov     $0x2,%eax
c3                      retq
b8 01 00 00 00          mov     $0x1,%eax
c3                      retq
...
@ 868: -156 /* offset */
@ 870: -162
...
```

added cost

replace `jmp *jumpTable(,%rdi,8)`

with:

`lea` (get table address — with relative offset)

`movslq` (do table lookup of offset)

`add` (add to base)

`jmp` (to computed base)

32-bit x86 is worse

no relative addressing for `mov`, `lea`, …

even changes "stubs" for printf:

```
// BEFORE: (fixed addresses)
08048310 <__printf_chk@plt>:
 8048310: ff 25 10 a0 04 08   jmp     *0x804a010
    /* 0x804a010 == global offset table entry */

// AFTER: (position-independent)
00000490 <__printf_chk@plt>:
 490:   ff a3 10 00 00 00     jmp     *0x10(%ebx)
    /* %ebx --- address of global offset table */
    /* needs to be set by caller */
```

32-bit x86 is worse

no relative addressing for `mov`, `lea`, …

even changes "stubs" for printf:

```
// BEFORE: (fixed addresses)
08048310 <__printf_chk@plt>:
 8048310: ff 25 10 a0 04 08   jmp    *0x804a010
    /* 0x804a010 == global offset table entry */

// AFTER: (position-independent)
00000490 <__printf_chk@plt>:
 490:   ff a3 10 00 00 00    jmp    *0x10(%ebx)
    /* %ebx --- address of global offset table */
    /* needs to be set by caller */
```

32-bit x86 is worse

no relative addressing for `mov`, `lea`, …

even changes "stubs" for printf:

```
// BEFORE: (fixed addresses)
08048310 <__printf_chk@plt>:
 8048310: ff 25 10 a0 04 08   jmp     *0x804a010
    /* 0x804a010 == global offset table entry */

// AFTER: (position-independent)
00000490 <__printf_chk@plt>:
 490:   ff a3 10 00 00 00     jmp     *0x10(%ebx)
    /* %ebx --- address of global offset table */
    /* needs to be set by caller */
```

PIE

position-independent executables (PIE)

- no hardcoded addresses

alternative: edit code (not global offset table) at load time

- Windows solution

GCC: `-pie -fPIE`

- `-pie` is linking option
- `-fPIE` is compilation option
- related option: `-fPIC` (position independent code)
  - used to compile runtime-loaded libraries

hard-coded addresses? (64-bit)

```c
int foo(long n) {
    switch (n) {
    case 0:
    case 2:
    case 4:
    case 5:
        return 1;
    case 1:
    case 3:
        return 2;
    default:
        return 3;
    }
}
```

```
foo:
        movl    $3, %eax
        cmpq    $5, %rdi
        ja      defaultCase
        jmp     *lookupTable(,%rdi,8)
        /* code for defaultCase, returnOne,
        ...
        .section        .rodata
lookupTable: /* read-only pointers: */
        .quad   returnOne
        .quad   returnTwo
        .quad   returnOne
        .quad   returnTwo
        .quad   returnOne
        .quad   returnOne
```

hard-coded addresses? (64-bit)

```
int foo(long n) {
    switch (n) {
    case 0:
    case 2:
    case 4:
    case 5:
        return 1;
    case 1:
    case 3:
        return 2;
    default:
        return 3;
    }
}
```

```
400570 <foo>:
b8 03 00 00 00      mov    $0x3,%eax
48 83 ff 05         cmp    $0x5,%rdi
        /* jump to defaultCase: */
77 12               ja     0x40058d
        /* lookup table jump: */
ff 24 fd
18 06 40 00         jmpq   *0x400618(,%rdi,8)
...
 /* lookupTable @ 0x400618 */
@ 400618: 0x400588 /* returnOne */
@ 400620: 0x400582 /* returnTwo */
@ 400628: 0x400588
@ 400630: 0x400582
```

hard-coded addresses? (64-bit)

```c
int foo(long n) {
    switch (n) {
    case 0:
    case 2:
    case 4:
    case 5:
        return 1;
    case 1:
    case 3:
        return 2;
    default:
        return 3;
    }
}
```

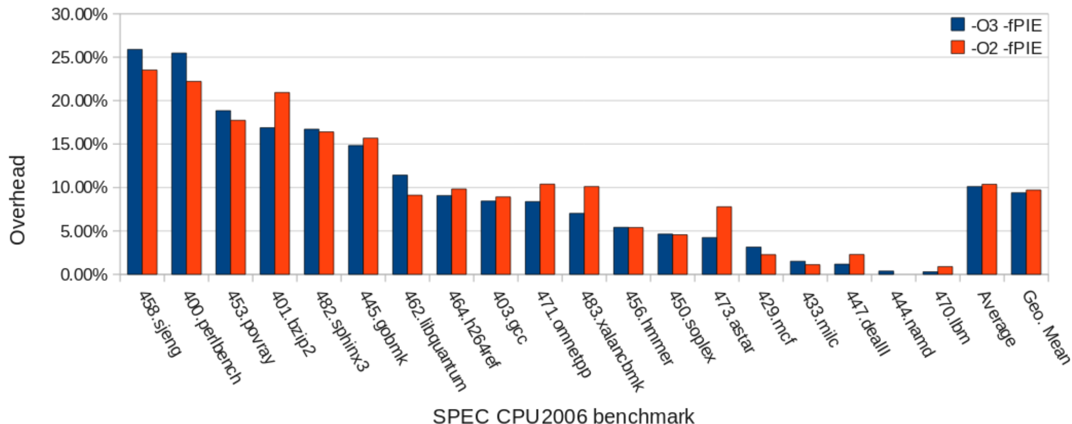
```
400570 <foo>:
b8 03 00 00 00      mov    $0x3,%eax
48 83 ff 05         cmp    $0x5,%rdi
            /* jump to defaultCase: */
77 12              ja     0x40058d
            /* lookup table jump: */
ff 24 fd
18 06 40 00        jmpq   *0x400618(,%rdi,8)
...
 /* lookupTable @ 0x400618 */
@ 400618: 0x400588 /* returnOne */
@ 400620: 0x400582 /* returnTwo */
@ 400628: 0x400588
@ 400630: 0x400582
```

hard-coded addresses? (64-bit)

```
int foo(long n) {
    switch (n) {
    case 0:
    case 2:
    case 4:
    case 5:
        return 1;
    case 1:
    case 3:
        return 2;
    default:
        return 3;
    }
}
```

```
400570 <foo>:
b8 03 00 00 00    mov    $0x3,%eax
48 83 ff 05       cmp    $0x5,%rdi
      /* jump to defaultCase: */
77 12             ja     0x40058d
      /* lookup table jump: */
ff 24 fd
18 06 40 00       jmpq   *0x400618(,%rdi,8)
...
 /* lookupTable @ 0x400618 */
@ 400618: 0x400588 /* returnOne */
@ 400620: 0x400582 /* returnTwo */
@ 400628: 0x400588
@ 400630: 0x400582
```

position independence cost (32-bit)



Payer, “Too much PIE is bad for performance”, ETH Zurich Tech Report

position independence cost: Linux

geometric mean of SPECcpu2006 benchmarks on x86 Linux

with particular version of GCC, etc., etc.

64-bit: 2-3% (???)

"preliminary result"; couldn't find reliable published data

32-bit: 9-10%

depends on compiler, ...

position independence: deployment

common for a very long time in dynamic libraries

default for all executables in...

Microsoft Visual Studio 2010 and later

`DYNAMICBASE` linker option

OS since 10.7 (2011)

Fedora 23 (2015) and Red Hat Enterprise Linux 8 (2019) and later

default for "sensitive" programs earlier

Ubuntu 16.10 (2016) and later (for 64-bit), 17.10 (2017) and later (for 32-bit)

default for "sensitive" programs earlier

sudo exploit

this writeup: summary from `https://www.openwall.com/lists/oss-security/2021/01/26/3`

from group at Qualys

sudo bug

the bug:

```
for (size = 0, av = NewArgv + 1; *av; av++)
     size += strlen(*av) + 1;
if (size == 0 || (user_args = malloc(size)) == NULL) { ... }
...
for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
while (*from) {
  if (from[0] == '\\' && !isspace((unsigned char)from[1]))
    from++;
  *to++ = *from++;
...
```

can skip `\0` if prefixed with backslash

but `strlen` used to allocate buffer

disagreement about copied string length

heap overflow!

brute-forcing?

method: tried to lots of buffer overflows, get crashes

looked at them by hand, found interesting ones...

one crash

```
0x000056291a25d502 in process_hooks_getenv (name=name@...ry=0x7f4a6d7dc046 "SYSTEMD_BYPASS_US

=> 0x56291a25d502 <process_hooks_getenv+82>:    callq  *0x8(%rbx)

108         rc = hook->u.getenv_fn(name, &val, hook->closure);
```

they overwrote a function pointer on the heap!

next inquiry: where did that usually point?

sudoers.so

```
    *** interesting standard library function: ***
0000000000008a00 <execv@plt>:
    8a00:       endbr64
    8a04:       bnd jmpq *0x55565(%rip)        # 5df70 <execv@GLIBC_2
    8a0b:       nopl   0x0(%rax,%rax,1)
...
    *** usual value of function pointer: ***
000000000000ea00 <sudoers_hook_getenv>:
    ea00:       endbr64
    ea04:       xor    %eax,%eax
    ea06:       cmpb   $0x0,0x51d36(%rip)        # 60743 <sudoers_pol
    ea0d:       jne    eaf8 <freeaddrinfo@plt+0x60a8>
    ea13:       cmpq   $0x0,0x51d45(%rip)        # 60760 <sudoers_pol
```

sudoers.so

```
    *** interesting standard library function: ***
0000000000008a00 <execv@plt>:
    8a00:       endbr64
    8a04:       bnd jmpq *0x55565(%rip)        # 5df70 <execv@GLIBC_2
    8a0b:       nopl   0x0(%rax,%rax,1)
...
    *** usual value of function pointer: ***
000000000000ea00 <sudoers_hook_getenv>:
    ea00:       endbr64
    ea04:       xor    %eax,%eax
    ea06:       cmpb   $0x0,0x51d36(%rip)       # 60743 <sudoers_pol
    ea0d:       jne    eaf8 <freeaddrinfo@plt+0x60a8>
    ea13:       cmpq   $0x0,0x51d45(%rip)       # 60760 <sudoers_pol
```

observations (that hold true even with ASLR):

addr(`execv@plt`) - addr(`sudoers_hook_getenv`) = `-0x6000`

last 12 bits of execv@plt always `a00` (page alignment)

changing pointer (part one)

suppose hook_getenv pointer is `0xabcdef8a00`

as bytes: `00 8a ef cd ab 00 00 00`

then execv@plt pointer is `0xabcdef3a00`

as bytes: `00 3a ef cd ab 00 00 00`

only need to change the last two bytes

also: same change would work if pointer had different high bits

changing pointer (part one)

suppose hook_getenv pointer is `0xabcdef8a00`

as bytes: `00 8a ef cd ab 00 00 00`

then execv@plt pointer is `0xabcdef3a00`

as bytes: `00 3a ef cd ab 00 00 00`

only need to change the last two bytes

also: same change would work if pointer had different high bits

only four bits of random data from ASLR!

changing pointer (part two)

solution: guess hook_getenv pointer at `0x` (unknown) `8a00`

overwrite last two bytes with `00 3a`

if right: will execute your program

if wrong: will crash

changing pointer (part two)

solution: guess hook_getenv pointer at `0x (unknown) 8a00`

overwrite last two bytes with `00 3a`

if right: will execute your program

if wrong: will crash

what if crashes? try again!

- would work about once every 16 tries...
- but actual exploit needed to write a 00 byte at the end (strcpy)
- so worked 'only' about once every 4096 tries

into exploit

make SYSTEMD_BYPASS_USERDB program in current directory

run sudo, triggering buffer overflow to change

```
sudoers_hook_getenv("SYSTEMD_BYPASS_USERDB", ...)
```

into

```
execv(SYSTEMD_BYPASS_USERDB, ...)
```

(well, try to change — it won't always work)

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)

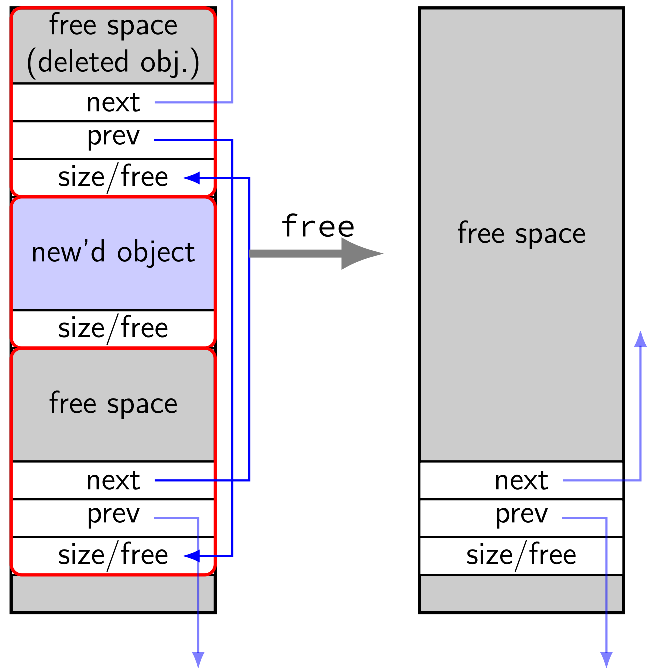
diversion: implementing malloc/new

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) {
    ...
    block_after = object + object_size;
    if (block_after->free) {
        /* unlink from list, about to merge with previous bloc
        new_block->size += block_after->size;
        block_after->prev->next = block_after->next;
        block_after->next->prev = block_after->prev;
    }
    ...
}
```

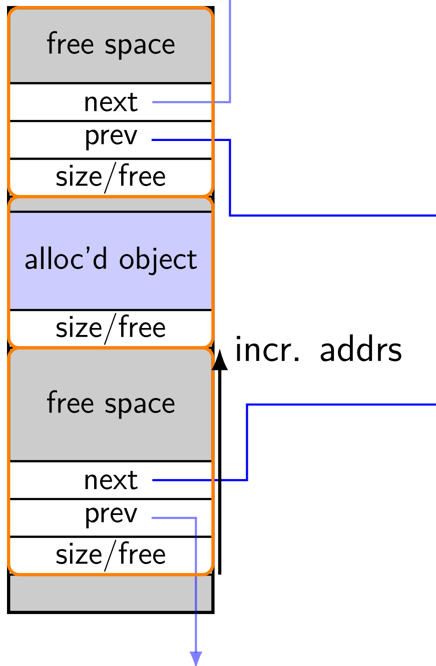
implementing free()

```
int free(void *object) {
    ...
    block_after = object + object_size;
    if (block_after->free) {
        /* unlink from list, about to merge with previous bloc
        new_block->size += block_after->size;
        block_after->prev->next = block_after->next;
        block_after->next->prev = block_after->prev;
    }
    ...
}
```

arbitrary memory write

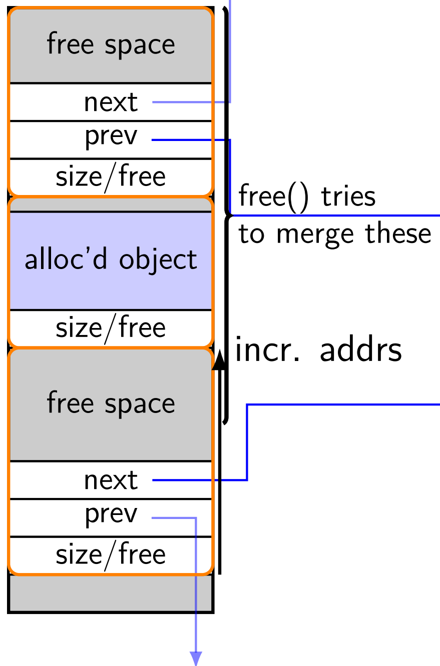
vulnerable code

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



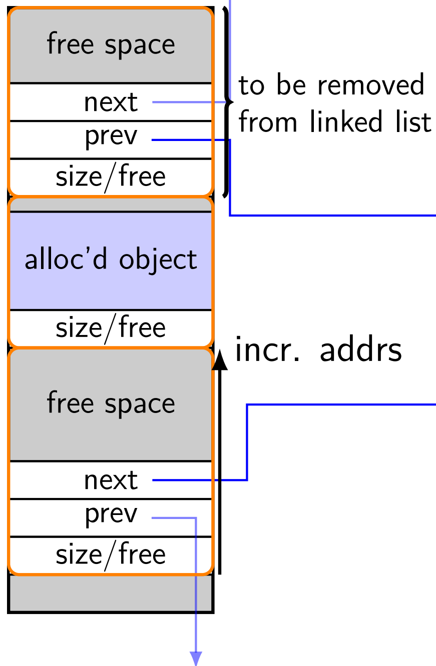
vulnerable code

```
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);
...
```

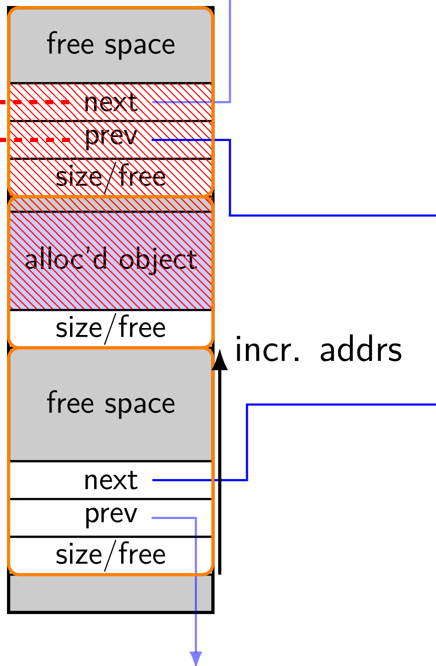


vulnerable code

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

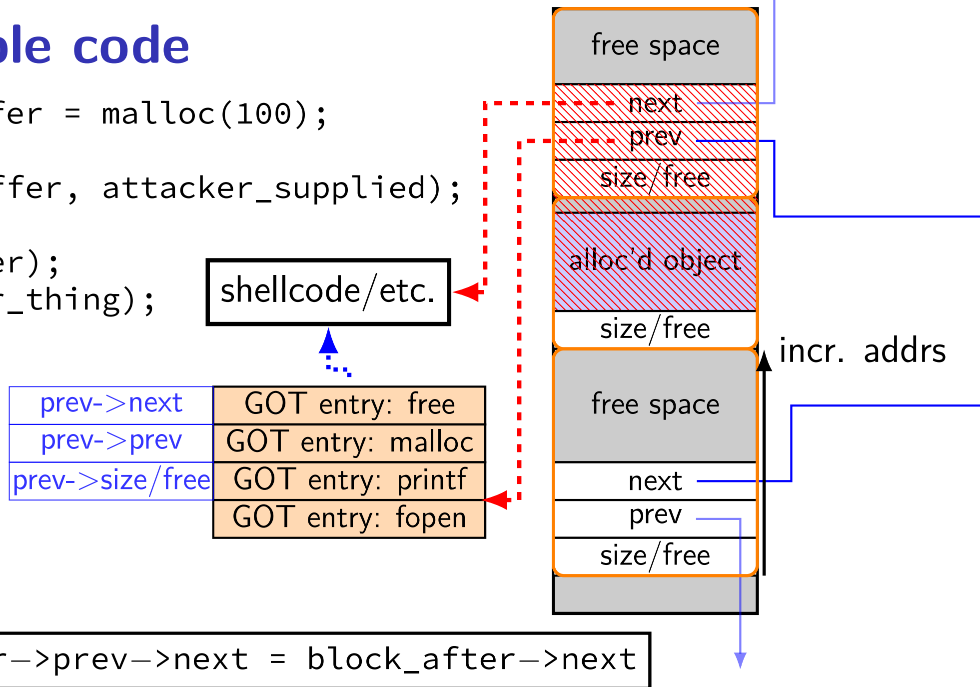
shellcode/etc.

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



vulnerable code

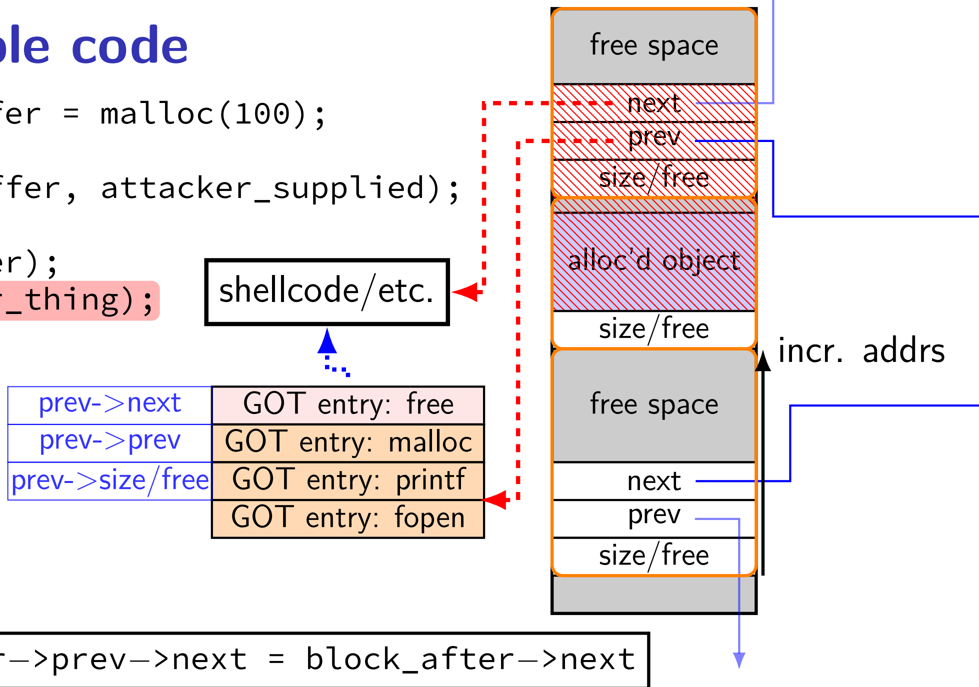
```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```




```
block_after->prev->next = block_after->next
```

vulnerable code

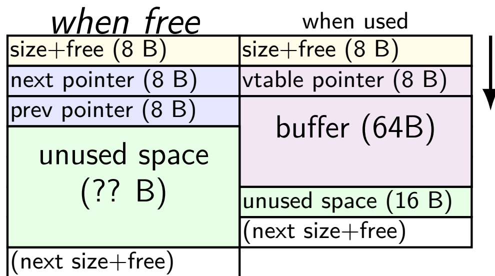
```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```




```
block_after->prev->next = block_after->next
```

heap overflow exercise

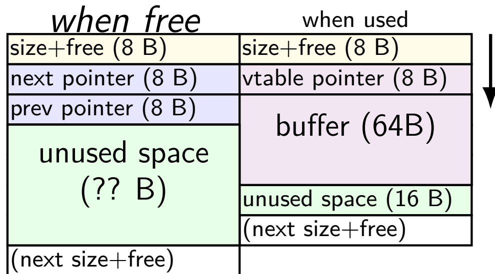
```
void operator delete(void *p) {
    ...
    block_after->prev->next = block_after->next;
    ...
}
...
class MyBuffer : public GenericMyBuffer {
public:
    virtual void store(const char *p) override {
        strcpy(buffer, p);
    }
private:
    char buffer[64];
};
...
    GenericMyBuffer *a = new MyBuffer;
    ...
    a->store(attacker_controlled);
    ...
    delete a;
    ...
```



exercise 1:
to attack this buffer overflow
by overwriting the heap data structures
does it matter if space after a
is already free or not?

heap overflow exercise

```
void operator delete(void *p) {
    ...
    block_after->prev->next = block_after->next;
    ...
}
...
class MyBuffer : public GenericMyBuffer {
public:
    virtual void store(const char *p) override {
        strcpy(buffer, p);
    }
private:
    char buffer[64];
};
...
    GenericMyBuffer *a = new MyBuffer;
    ...
    a->store(attacker_controlled);
    ...
    delete a;
    ...
```



exercise 2: if a at address 0x10000,
and attacker wants to overwrite
value at address 0x20000 with 0x30000,
where should attacker put 0x20000, 0x30000
in `attacker_controlled`?

other malloc designs?

there are a lot of different malloc/new implementations

often multiple free lists

free block list might not be kept with linked list

some place metadata next to allocations like this

some keep it separate

usually performance determines which is chosen

vulnerable code

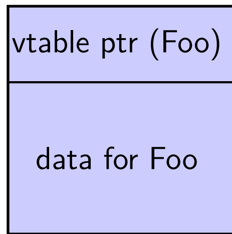
```
class Foo {
    ...
};
Foo *the_foo;
the_foo = new Foo;
...
delete the_foo;
...
something_else = new Bar(...);
the_foo->something();
```

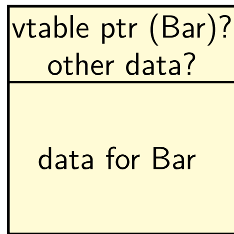
`something_else` likely where `the_foo` was

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





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

backup slides

exericse: using a leak (1)

```
class Foo {
    virtual const char *bar() { ... }
};
...
Foo *f = new Foo;
printf("%s\n", f);
```

Part 1: What address is most likely leaked by the above?

A. the location of the Foo object allocated on the heap
B. the location of the first entry in Foo's VTable"
C. the location of the first instruction of Foo::Foo() (Foo's compiler-generated constructor)"
D. the location of the stack pointer

exercise: using a leak (2)

```
class Foo {
    virtual const char *bar() { ... }
};
...
Foo *f = new Foo;
char *p = new char[1024];
printf("%s\n", f);
```

if leaked value was 0x822003 and in a debugger (with **different randomization**):

- stack pointer was 0x7ffff000
- Foo::bar's address was 0x400000
- f's address was 0x900000
- f's Vtable's address was 0x403000
- a "gadget" address from the main executable was 0x401034
- a "gadget" address from the C library was 0x2aaaa40034
- p's address was 0x901000

which of the above can I compute based on the leak?

using function pointer overwrite (1)

```
struct Example {
    char input[1000];
    void (*process_function)(Example *, long, char *);
};
void vulnerable(struct Example *e) {
    long index;
    char name[1000];
    gets(e->input); /* can overwrite process_function */
    scanf("%ld,%s", &index, &name[0]); /* expects <decimal number>,<string> */
    (e->process_function)(e /* rdi */, index /* rsi */, name /* rdx */);
}
```

if we overwrite process_function's address with the address of the gadget `mov %rsi, %rsp; ret`, then the beginning of the input should contain...

A. the shellcode to run
B. an ROP chain to run
C. the address of shellcode (or existing function) in decimal
D. the address of the ROP chain to run written out in decimal
E. the address of a RET instruction written out in decimal

explanation

```
gets(e->input); /* can overwrite process_function */
scanf("%ld,%s", &index, &name[0]); /* expects <decimal number>,<string> */
(e->process_function)(e /* rdi */, index /* rsi */, name /* rdx */);
```

`"1234,FOO........"` + addr of `mov %rsi, %rsp, ret`

arguments setup registers for gadget:

- %rdi (irrelevant) is "1234,FOO…" (copy in e)
- %rsi is 1234 (from scanf)
- %rdx (irrelevant) is "FOO…" (pointer to name)

mov in gadget: %rsi (1234) becomes %rsp

ret in gadget: read pointer at 1234, set %rsp to 1234 + 8

- jump to next gadget (whose address should be stored at 1234)
- if that gadget returns, will read new return address from 1238

using function pointer overwrite (2)

```
struct Example {
    char input[1000];
    void (*process_function)(Example *, long, char *);
};
void vulnerable(struct Example *e) {
    long index;
    char name[1000];
    gets(e->input); /* can overwrite process_function */
    scanf("%ld,%s", &index, &name[0]); /* expects <decimal number>,<string> */
    (e->process_function)(e /* rdi */, index /* rsi */, name /* rdx */);
}
```

if we overwrite process_function's address with the address of the gadget `push %rdx; jmp *(%rdi)`, then the beginning of the input should contain...

A. the shellcode to run
B. an ROP chain to run
C. the address of shellcode (or existing function)
D. the address of the ROP chain
E. the address of a RET instruction

explanation (one option)

```
gets(e->input); /* can overwrite process_function */
scanf("%ld,%s", &index, &name[0]); /* expects <decimal number>,<string> */
(e->process_function)(e /* rdi */, index /* rsi */, name /* rdx */);
```

`"FOOBARBAZ........"` + addr of `push %rdx; jmp *(%rdi)`

arguments setup registers for gadget:

- %rdi is "FOOBARBAZ...." (copy in e)
- %rsi (irrelevant) is uninitialized? (scanf failed)
- %rdx (irrelevant) is uninitialized? (scanf failed)

push in gadget: top of stack becomes copy of uninit. value

jmp in gadget

- interpret "FOOBARBA" as 8-byte address
- jump to that address

explanation (unlikely alternative?)

```
gets(e->input); /* can overwrite process_function */
scanf("%ld,%s", &index, &name[0]); /* expects <decimal number>,<string> */
(e->process_function)(e /* rdi */, index /* rsi */, name /* rdx */);
```

`"1234567890,FOO........"` + addr of `push %rdx; jmp *(%rdi)`

arguments setup registers for gadget:

- %rdi is address of string "12345678,FOO..." (copy in e)
- %rsi is 12345678
- %rdx is address of string "FOO..." (copy in name)

push in gadget: top of stack becomes address of "FOO..."

jmp in gadget

- interpret *ASCII encoding of "12345678"* (???) as 8-byte address
- jump to that address