Project Zero

News and updates from the Project Zero team at Google

Thursday, August 22, 2019

The Many Possibilities of CVE-2019-8646

Posted by Natalie Silvanovich, Project Zero

CVE-2019-8646 is a somewhat unusual vulnerability I reported in iMessage. It has a number of consequences, including information leakage and the ability to remotely read files on a device. This blog post discusses the ways that an attacker could use this bug. It is a good example of how the large number of classes available for `NSKeyedArchiver` deserialization can make a bug more versatile. It's also a good example of how minor functional bugs can make a vulnerability more useful.

Please note that this blog post assumes some familiarity with `NSKeyedArchiver` deserialization. If you haven't read our general post on iMessage, I'd recommend reading that first.

The Bug

The bug described in CVE-2019-8646 is that an unsafe class, `_NSDataFileBackedFuture`, can be deserialized by iMessage in a remote context. It was introduced in iOS 12.1. This class is a subclass of `NSData` that initializes a buffer with the contents of a file at the time the buffer is used. When this class is deserialized, it decodes the length of the buffer, a string file name and a few other objects. It then initializes the instance with the length and filename. Then when `[_NSDataFileBackedFuture length]` is called, it returns the deserialized length. When `[_NSDataFileBackedFuture bytes]` is called, the file is opened and loaded into a buffer into memory, and the buffer is returned. The buffer is also cached for future

Search This Blog

Search

Pages

- Working at Project Zero
- 0day "In the Wild"
- Vulnerability Disclosure FAQ

Archives

2019

- A very deep dive into iOS Exploit chains found in ... (Aug)
- In-the-wild iOS Exploit Chain 1 (Aug)
- In-the-wild iOS Exploit Chain 2 (Aug)
- In-the-wild iOS Exploit Chain 3 (Aug)
- In-the-wild iOS Exploit Chain 4 (Aug)
- In-the-wild iOS Exploit Chain 5 (Aug)
- Implant Teardown (Aug)
- JSC Exploits (Aug)
- The Many Possibilities of CVE-2019-8646 (Aug)

calls to the method.

There are two immediate problems with being able to deserialize this class in an untrusted context. One is that it has the potential to allow a process to access a file that it is not authorized to access, because the process doing the deserialization is the one that loads the file. When I reported this bug, I thought that this was more likely to be a concern for deserialization that occurs locally via IPC as opposed to deserialization that occurs on a remote target like iMessage. The second is that this class violates one of the guarantees that the `NSData` class makes, that the `length` property will always return the length of the `bytes` property. This is because the length of the buffer returned by `[_NSDataFileBackedFuture bytes]` is the length of the loaded file, and has no relationship to the deserialized length returned by `[_NSDataFileBackedFuture length]`.

The original proof-of-concept (PoC) attached to the bug report is a simple out-of-bounds read. The payload includes a serialized instance of class `ACZeroingString`, which is a subclass of `NSString`. Its `initWithCoder` method deserializes an instance of class `NSData`, as well as a length that must be half the `[NSData length]` that it uses to initialize the contents of the string. If the `NSData` instance is of subclass `_NSDataFileBackedFuture`, the `length` property of the instance can be longer than its internal data, causing the PoC to return a string that contains the contents of unallocated memory, or cause a crash.

Accessing a Remote URL

At this point, this bug didn't seem that useful for a remote attack, so I wondered if it would be possible for it to access a remote URL instead of a local file. The URL is accessed by calling [`[NSData initWithContentsOfURL:options:error:]`](), which can initialize a buffer from any type of URL, including HTTP URLs, however the `_NSDataFileBackedFuture` class contains some checks to prevent this.

There are no checks to the URL on initialization, but there are some checks when the URL is accessed in `[_NSDataFileBackedFuture fileURL]`. Specifically, it calls `[NSURL path]` on the URL, and then calls `[NSFileManager fileExistsAtPath:]` on that path. This does not check that the URL is a file URL before checking the path. So it is possible to bypass this check by using a URL that has a path component that resolves to an existing file. I used: `http://natashenka.party//System/Library/ColorSync/Resources/ColorTables.data`.

Creating a URL with Unprintable Characters

The ability to make a request to a URL created an interesting possibility. Maybe it was possible to use the URL to leak data remotely. Since the original PoC created a string that contained leaked data, and the `NSURL` class is deserialized using a string, it didn't seem like it would be that difficult. It turned out there was a problem using the `NSURL` class though. An `NSURL` instance has very strict limitations on the characters it can contain. This class is mostly open-source, so the exact limitations can be seen in the [`_CFStringIsLegalURLString`] method. This is a very robust method, and I did not find any ways to get around the limitations. I did notice that after the method succeeds, the method `_CFURLInit` is called, which calls `CFStringCreateCopy` on the input string, so it caches a copy of the validated string to use as the URL later.

One idea I had was to change the string after the URL was created, because the URL string is only validated once. In the absence of bugs, this shouldn't be possible. `CFStringCreateCopy` calls `[NSString copy]` on most string objects, and for a mutable string, this should copy the string, so that any future changes to the string do not affect the copy. For a non-mutable string, it sometimes just increases the retain count on the string, but that also shouldn't be a problem, because the contents of a mutable string can't change.

I looked through the subclasses of `NSString` that can be deserialized in iMessage to see if there were any that didn't follow the mutable copy rules described above. There were a few, but the most promising was the class `INDeferredLocalizedString`. This class is technically immutable (in that it extends `NSString` instead of `NSMutableString`), and it implements copy by adding a reference. But the value of an `INDeferredLocalizedString` instance can change. Its deserialization implementation in pseudocode is as follows.

```
INDeferredLocalizedString *__cdecl -[INDeferredLocalizedString
initWithCoder:](INDeferredLocalizedString *self, id decoder)
{
self->_formatKey = [decoder decodeObjectOfClass:[NSString class]
forKey:@"_formatKey"];
self->_table = [decoder decodeObjectOfClass:[NSString class]
forKey:@"_table"];
self->_arguments = [decoder decodeObjectOfClasses:@[[NSString
class], [NSArray class]] forKey:@"_arguments"];
self->_bundleIdentifier = [decoder decodeObjectOfClass:[NSString
class] forKey:@"_bundleIdentifier"];
```


- Heap Feng Shader: Exploiting SwiftShader in Chrome... (Oct)
- Deja-XNU (Oct)
- Injecting Code into Windows Protected Processes us... (Oct)
- 365 Days Later: Finding and Exploiting Safari Bugs... (Oct)
- A cache invalidation bug in Linux memory managemen... (Sep)
- OATmeal on the Universal Cereal Bus: Exploiting An... (Sep)
- The Problems and Promise of WebAssembly (Aug)
- Windows Exploitation Tricks: Exploiting Arbitrary ... (Aug)
- Adventures in vulnerability reporting (Aug)
- Drawing Outside the Box: Precision Issues in Graph... (Jul)
- Detecting Kernel Memory Disclosure – Whitepaper (Jun)
- Bypassing Mitigations by Attacking JIT Server in M... (May)
- Windows Exploitation Tricks: Exploiting Arbitrary ... (Apr)
- Reading privileged memory with a side-channel (Jan)

2017

- aPAColypse now: Exploiting Windows 10 in a Local N... (Dec)
- Over The Air - Vol. 2, Pt. 3: Exploiting The Wi-Fi... (Oct)
- Using Binary Diffing to Discover Windows Kernel Me... (Oct)
- Over The Air - Vol. 2, Pt. 2: Exploiting The Wi-Fi... (Oct)
- Over The Air - Vol. 2, Pt. 1: Exploiting The Wi-Fi... (Sep)

```
self->_bundleURL = [decoder decodeObjectOfClass:[NSURL class]
forKey:@"_bundleURL"];
self->_cachedLocalization = [decoder decodeObjectOfClass:[NSString
class] forKey:@"_cachedLocalization"];
return self;
}
```

It deserializes many properties, including a bundle URL that can be used for localization, a format string with a corresponding array of localized strings and a cached string. When an `INDeferredLocalizedString` instance is accessed, its value is determined by calling `[INDeferredLocalizedString localizeForLanguage:]`, which generates the string based on these values and the device's language settings. The deserialized properties have a precedence. For example, the class would prefer to fetch a string from a bundle as opposed to generating it from the format string.

Even with these properties, the string would only change if the device's language changed, however, it is possible to make the string change due to the issues with cycling in `NSKeyedArchiver` deserialization described in this post. The highest precedence property of the class, the `_cachedLocalization` string is deserialized last, meanwhile a lower precedence property `_formatKey` is serialized earlier. The bundle URL is deserialized in the middle of these two. So if an instance of class `INDeferredLocalizedString` has a valid `_formatKey`, and then the bundle URL's string is a reference to the string itself, the URL will validate the `_formatKey` when it is being created. Initialization of the `INDeferredLocalizedString` instance will then continue, and the `_cachedLocalization` string will be deserialized and set as a property. After the `INDeferredLocalizedString` deserialization is complete, the URL will be available in the `NSKeyedUnarchiver` decoder's cache. When another class, such as `_NSDataFileBackedFuture`, uses it, the string value will now be generated based on the `_cachedLocalization` property, which is the unvalidated string.

This behavior allowed me to create a message that would leak memory and send it to a remote server as a URL parameter. A sample message with this behavior is available here. That said, the parameter is only read up to the first null character, so this PoC usually only sends a few bytes. This is probably enough to leak a single pointer to break ASLR with enough tries, but not good for much else.

Concatenating and Encoding Leaked Data

This limitation also prevents a more interesting attack: remotely leaking a file. Since the `_NSDataFileBackedFuture` class can load a file into a buffer, and also send the contents to a remote URL, is a possible attack, but the prevalence of null characters in file format headers limits its usefulness.

There is also a more subtle problem preventing the PoC above from being immediately repurposed to leak a file. The PoC works by creating a `_NSDataFileBackedFuture` instance with contents that are smaller than its length, then using that instance to create an `ACZeroingString` instance, which in a roundabout way becomes the string of a URL. That URL is then used as the URL of another `_NSDataFileBackedFuture` instance. But what is the string value of the URL of the first `_NSDataFileBackedFuture` instance? I used another remote URL which responded with a buffer containing another partial URL (`http://natashenka.party//System/Library/ColorSync/Resources/ColorTables.data?val=`). So when the `_NSDataFileBackedFuture` buffer is read out of bounds when creating the `ACZeroingString` instance, the leaked data continues the URL. This is not possible when accessing a file with `_NSDataFileBackedFuture`, because the file contents are set and generally are not in the format of a URL. So in order to leak a file, I also need to be able to concatenate strings.

The `INDeferredLocalizedString` class has functionality that is helpful in getting around both of these limitations. Two properties that can be decoded during deserialization are the string `_formatKey` and an array of strings, `_argument`. If an `INDeferredLocalizedString` instance has only these properties, it will generate its value using the first property as a format string, and the second property as its parameter.

(You might be wondering at this point whether this behaviour is a vulnerability in itself because an attacker can control both a format string and its parameters. It's not, because the class uses a 'fake' format string implementation that is based on regular expressions. The implementation searches for instances of "%@" or similar in the format string, and then replaces them sequentially with values from the array).

This format string behaviour allows the `ACZeroingString` instance to be inserted into a string containing a URL, and it also helps with the issue of null characters. When an `ACZeroingString` instance is formatted with "%@", non-printable characters are escaped in the format "\UXXXX". Single null bytes will be added to the string as a part of a character in this way, however if there are two null bytes in a row, this character will be omitted. This type of encoding is useful in some contexts (for example, leaking the SMS database, where there are a lot of null characters, but only the string are relevant), but is not enough to completely leak a full file.

Looking at the 'fake' format string function a bit more, it calls `description` on every member of the arguments array before inserting it into the format string. This would be very useful behavior if the arguments array wasn't limited to containing string types. Calling `description` on an instance of class `NSURL` URL encodes the URL string before it is inserted. So if it was possible to put the URL containing the leaked bytes into the arguments array, it could be encoded, which would allow the entire file to be sent as a part of a URL.

There is a problem in `NSKeyedUnarchiverSerialization` which can allow objects that are not included in the allow list to be returned when an instance of class `NSArray` or `NSDictionary` is deserialized. The first time an array or dictionary is deserialized, every element, key or value that is deserialized as a part of the object's contents is checked against the allow list. But if the object has already been deserialized, the object is returned from the `NSKeyedUnarchiverSerialization` instance's object cache, and only the object, and not its elements, keys or values are checked against the allow list. So the `_arguments` array could contain a URL, so long as the array had already been deserialized elsewhere.

It was a bit of a challenge finding somewhere an array containing a URL could be deserialized in iMessage. The top level allow list does not include class `NSArray`, and I could not find a class with an `initWithCoder:` implementation that contained a deserialization call that allows both arrays and URLs. I eventually implemented it so that it uses the bug twice. First, a dictionary containing the URL is decoded, which is allowed at the top level. Then, an instance of `__NSLocalizedString` is decoded, which decodes a property `NS.configDict`, which allows arrays and dictionaries, but not URLs, but because of the bug, a dictionary containing a URL is okay. Then, the bug is used again when initializing the `_arguments` array of the `INDeferredLocalizedString` instance, which is allowed because it only checks that the referenced array is an instance of `NSArray`. When this object is formatted into a string, it will contain some extra characters due to the dictionary, but otherwise will still be encoded.

Leaking a File

Putting this all together allowed for a file to be read remotely from an iPhone. There are a few limitations to this attack. First, it is very memory intensive, the largest memory hog being the 'fake' format string function that needs to handle a very long string. SpringBoard can crash due to memory limits if the file is too long. The limit appears to be around 40kB to 100kB depending on device memory, though it's likely this could be increased with enough effort. It is possible to fetch the beginning of a larger file within this limit, and also reduce memory usage a bit by using escape ("\U") encoding instead of URL encoding in situations where

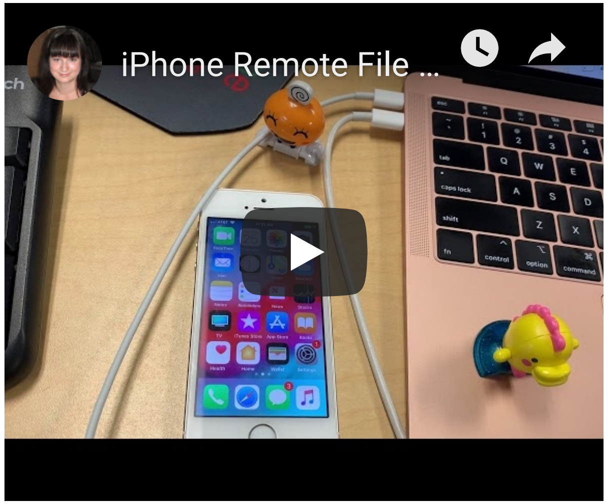
- Three bypasses and a fix for one of Flash's Vector... (Aug)
- Attacking ECMAScript Engines with Redefinition (Aug)
- One font vulnerability to rule them all #3: Window... (Aug)
- One font vulnerability to rule them all #2: Adobe ... (Aug)
- One font vulnerability to rule them all #1: Introd... (Jul)
- One Perfect Bug: Exploiting Type Confusion in Flas... (Jul)
- Significant Flash exploit mitigations are live in ... (Jul)
- From inter to intra: gaining reliability (Jul)
- When 'int' is the new 'short' (Jul)
- What is a "good" memory corruption vulnerability? (Jun)
- Analysis and Exploitation of an ESET Vulnerability... (Jun)
- Owning Internet Printing - A Case Study in Modern ... (Jun)
- Dude, where's my heap? (Jun)
- In-Console-Able (May)
- A Tale of Two Exploits (Apr)
- Taming the wild copy: Parallel Thread Corruption (Mar)
- Exploiting the DRAM rowhammer bug to gain kernel p... (Mar)
- Feedback and data-driven updates to Google's discl... (Feb)
- (^Exploiting)\s*(CVE-2015-0318)\s*(in)\s*(Flash$) (Feb)
- A Token's Tale (Feb)
- Exploiting NVMAP to escape the Chrome sandbox - CV... (Jan)
- Finding and exploiting ntpd vulnerabilities (Jan)

stripping null characters are okay.

The encoding of the URL returned to the remote server is also quite complex. The URL is escaped for a few characters up until the end of where the URL schema would be is reached, and then it moves into URL encoding. The URL coding is also escaped though, so it needs to be unescaped before it is URL decoded., The escaped characters are in UTF-16 meanwhile the URL encoded characters are in UTF-8, complicating matters. Then there can be a third section of the URL that is just escaped, which occurs because when the valid characters are switched for the invalid characters when creating the URL it retains the length of the valid characters. If the URL is too short, the extra characters will only be escaped.

There's another problem with encoding, which is that sometimes printable characters are duplicated in the URL encoded string. It's not clear why this happens, it could be a bug in `[NSURL description]` or the URL encoder. It is fairly easy to programmatically recognize and correct these duplications, but there is always the possibility that a file contains these exact patterns, at which case the file could be read with errors. A python script that decodes all the iterations of encoding in a returned URL and outputs a file is available here. I have not seen any files that contain errors after being processed with this script, but there is a small probability that this could occur.

The following video shows this vulnerability being used to access a photo from a remote device's memory. First the sms.db file is accessed to get the URL of the photo, and then the photo is accessed.



2014

Conclusion

CVE-2019-8646 is a vulnerability in iMessage that can allow memory to be leaked and files to be read remotely from a device. The bug was fixed on July 23, 2019. This fix requires the class to be explicitly allowed for deserialization, as opposed to being allowed in any situation that permits `NSData` deserialization.

There were several factors that caused this bug to be exploitable in this way. One is the large number of classes available for deserialization in SpringBoard. Without the `ACZeroingString`, `INDeferredLocalizedString` and `__NSLocalizedString` object being available, this bug would be less useful to an attacker.

Also, there were three small bugs that contributed to this bug's capabilities. First, the error in `[INDeferredLocalizedString copy]` is a bug that would usually just lead to occasional crashes when the device's language was changed, but in this situation, it turned out to be exactly the bug that was needed to circumvent the character restrictions of the `NSURL` class. Likewise, the error in `NSKeyedUnarchiver` that allows arrays and dictionaries containing any type of object to be returned if they are already decoded would usually only cause exceptions related to typing, but in the case allows for a URL to be encoded. Finally, the ability of the `_NSDataFileBackedFuture` class to access remote URLs was also a small bug in URL filtering. This shows that there is a security benefit to avoiding and fixing bugs, even if they don't have an obvious security impact. Alone, none of these bugs, including the vulnerability were that serious, but together they allow a user's data to be accessed remotely.

Posted by Tim at 12:49 PM

No comments:

Post a Comment

Enter your comment...

Comment as: Google Accoun

Publish Preview