Project Zero

News and updates from the Project Zero team at Google

Wednesday, August 7, 2019

The Fully Remote Attack Surface of the iPhone

Posted by Natalie Silvanovich, Project Zero

While there have been several rumours and reports of fully remote vulnerabilities affecting the iPhone being used by attackers in the last couple of years, limited information is available about the technical details of these vulnerabilities, as well as the underlying attack surface they occur in. I investigated the remote, interaction-less attack surface of the iPhone, and found several serious vulnerabilities.

Vulnerabilities are considered ‘remote’ when the attacker does not require any physical or network proximity to the target to be able to use the vulnerability. Remote vulnerabilities are described as ‘fully remote’, ‘interaction-less’ or ‘zero click’ when they do not require any physical interaction from the target to be exploited, and work in real time. I focused on the attack surfaces of the iPhone that can be reached remotely, do not require any user interaction and immediately process input.

There are several attack surfaces of the iPhone that have these qualities, including SMS, MMS, VVM, Email and iMessage.

SMS

SMS seemed like a good starting point, as I had looked at SMS on Android in the past. Unlike Android, SMS messages are processed in native code by the iPhone, which increases the likelihood of memory corruption

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)

vulnerabilities. SMS Packet Data Units (PDUs) are parsed by the CommCenter binary using the method `sms::Controller::parseRawBytes` which creates an instance of class `sms::Model` containing details of the message. This instance is eventually processed by `sms::Controller::processReceivedSms_sync` which does additional processing and sends the message on to other processes that handle them. I reviewed these two methods, but did not find any vulnerabilities.

I also noticed that CommCenter contained an SMS simulator that can be triggered via XPC. This tool processes SMS deliver PDUs as if they arrived over the network. The simulator was missing a library that likely exists on internal test devices, so I wrote a library that implements the needed functionality to make the simulator work. This tool is available here. Fuzzing SMS with this tool did not uncover any vulnerabilities.

MMS

MMS messages are also processed by CommCenter, and the bulk of the processing is performed in the method `MmsOperation::decodeMessage`. I reviewed this method using IDA, and also fuzzed it by writing an application that called into this method in iOS. There were no vulnerabilities discovered with either method.

Visual Voicemail

While reviewing `sms::Controller::processReceivedSms_sync`, I noticed that this method forwards many specially formatted SMS messages on to other processes. One area that looked interesting was Visual Voicemail (VVM), which I had reviewed previously on Android. VVM is a feature that allows voicemail messages to be viewed in a visual format similar to how emails are displayed.

VVM works by fetching voicemail messages from an IMAP server maintained by the device's carrier. The server URL and credentials for this server are provided to the device by the carrier over SMS. The iPhone uses a different format for VVM SMS messages than the publicly documented format, so I determined the contents of an incoming VVM SMS by putting a breakpoint in CommCenter where SMS PDUs are received. The following is an example of an incoming VVM message.

```
STATE?
state=Active;server=vvm.att.com;port=143;pw=asdf;name=5556667777@at
t.com
```

- Down the Rabbit-Hole... (Aug)
- The Fully Remote Attack Surface of the iPhone (Aug)
- Trashing the Flow of Data (May)
- Windows Exploitation Tricks: Abusing the User-Mode... (Apr)
- Virtually Unlimited Memory: Escaping the Chrome Sa... (Apr)
- Splitting atoms in XNU (Apr)
- Windows Kernel Logic Bug Class: Access Mode Mismat... (Mar)
- Android Messaging: A Few Bugs Short of a Chain (Mar)
- The Curious Case of Convexity Confusion (Feb)
- Examining Pointer Authentication on the iPhone XS (Feb)
- voucher_swap: Exploiting MIG reference counting in... (Jan)
- Taking a page from the kernel&#39;s book: A TLB issue ... (Jan)

2018

- On VBScript (Dec)
- Searching statically-linked vulnerable library fun... (Dec)
- Adventures in Video Conferencing Part 5: Where Do ... (Dec)
- Adventures in Video Conferencing Part 4: What Didn... (Dec)
- Adventures in Video Conferencing Part 3: The Even ... (Dec)
- Adventures in Video Conferencing Part 2: Fun with ... (Dec)
- Adventures in Video Conferencing Part 1: The Wild ... (Dec)
- Injecting Code into Windows Protected Processes us... (Nov)

I tried sending this message on an Android device that had been modified to send raw PDUs, and found that the logs showed an additional query had been made to the server. I tried changing the server to a server I controlled, and after several attempts, I was able to send a message that changed a target device's VVM server, with the following limitations:

- VVM must be configured (a greeting message recorded) on the device
- The PID field of the SMS must be set to the VVM value for the target device's carrier (this can be easily determined if you have a SIM from that carrier, but is different for each carrier)
- Some carriers block VVM IMAP requests to external servers, in which case this won't work for that carrier remotely. It's possible that an attacker could get around this using a base station in proximity of the target device, but I didn't look into this attack.

This was enough for VVM IMAP to be a viable attack surface for most carriers, and I thought it was reasonably likely to contain bugs, as VVM uses the same IMAP library as Email on iOS. IMAP servers are usually hardened against attacks from untrusted email clients, because it is common for a malicious client to attack the server in an attempt to access other users' emails. It is far less common, however, for a client to connect to a malicious server, as users need to enter these by hand, and typically only enter servers they trust. This means that the server to client attack surface is likely less well-tested, as it is not a realistic attack surface from the perspective of Email. VVM changes this, as it allows a device to be connected to a malicious IMAP server without user interaction. I wondered if the IMAP library had been adequately reviewed when its attack surface was drastically changed by the VVM implementation.

I looked at the IMAP library in IDA, but didn't find any bugs, so I set up fuzzing. I wrote a fake IMAP server that returned malformed responses to every request, and used the SMS simulator from the section above to constantly send VVM SMS messages, triggering the device to query the server. This uncovered one vulnerability, CVE-2019-8613, in the implementation. This bug is a use-after-free of an `NSString` that occurs due to incorrect handling of the `NAMESPACE` IMAP command. When an IMAP server sets up a connection, it first sends a `LIST` command to the client to get the mailbox separator string, and then sends a `NAMESPACE` command to get the mailbox prefix. In the iOS IMAP implementation, the separator string is freed if the server encounters an error, but the code that calls the `NAMESPACE` command does not check whether the command has succeeded, so it continues even if the separator has been freed.

Email

- 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)

Looking at the IMAP implementation, I noticed several code paths in MIME that are not used by VVM, but are used by the email client when processing messages. One of these had an obvious and unusual vulnerability in it.

The method `[MFMimePart _contents:toOffset:resultOffset:downloadIfNecessary:asHTML:isComplete:]` processes incoming MIME messages, and sends them to specific decoders based on the MIME type. Unfortunately, the implementation did this by appending the MIME type string from an incoming message to the string ‘decode’ and calling the resulting method. This meant that an unintended selector could be called, leading to memory corruption.

I found this vulnerability in version 11.3.1 of iOS, but it was clearly unexploitable in iOS 12 due to changes to the functionality of the unintended selectors that could be called. These changes did not appear to be security related. This issue could still cause a crash though, and was resolved as CVE-2019-8626.

While email is a potential remote attack surface for the iPhone, it is unclear how serious it is. To start, some users install third-party clients instead of using the native email client, and some email providers also filter incoming messages and remove malformed MIME components that are needed to reach a vulnerability. While the above bug worked on Gmail signed in on the native email client, it is not clear how common this configuration is, or whether provider filtering could be a problem in reaching similar bugs.

iMessage

iMessage is the native messaging client on iOS and Mac devices. It supports sending and receiving messages with a variety of formatting options, and also supports extensions, which allow custom message types to be sent and received by the device. Extensions can be written by both Apple and third parties. Samuel Groß and I reviewed iMessage and its extensions that are installed by default on the iPhone.

To start off the project, Samuel wrote tools that can send and dump iMessage messages on a Mac. They work by hooking code in iMessage that sends or receives messages with Frida, and either writing the message to the console in the case of dumping, or replacing it with a different message in the case of sending. The following is a sample message dumped with these tools.

```
to: mailto:TARGET@gmail.com
from: tel:+15556667777
{
```

- The Great DOM Fuzz-off of 2017 (Sep)
- Bypassing VirtualBox Process Hardening on Windows (Aug)
- Windows Exploitation Tricks: Arbitrary Directory C... (Aug)
- Trust Issues: Exploiting TrustZone TEEs (Jul)
- Exploiting the Linux kernel via packet sockets (May)
- Exploiting .NET Managed DCOM (Apr)
- Exception-oriented exploitation on iOS (Apr)
- Over The Air: Exploiting Broadcom’s Wi-Fi Stack (P... (Apr)
- Notes on Windows Uniscribe Fuzzing (Apr)
- Pandavirtualization: Exploiting the Xen hypervisor... (Apr)
- Over The Air: Exploiting Broadcom’s Wi-Fi Stack (P... (Apr)
- Project Zero Prize Conclusion (Mar)
- Attacking the Windows NVIDIA Driver (Feb)
- Lifting the (Hyper) Visor: Bypassing Samsung’s Rea... (Feb)

2016

- Chrome OS exploit: one byte overflow and symlinks (Dec)
- BitUnmap: Attacking Android Ashmem (Dec)
- Breaking the Chain (Nov)
- task_t considered harmful (Oct)
- Announcing the Project Zero Prize (Sep)
- Return to libstagefright: exploiting libutils on A... (Sep)
- A Shadow of our Former Self (Aug)

```
    gid = "FAA29682-27A6-498D-8170-CC92F2077441";
    gv = 8;
    p =     (
        "tel:+15556667777",
        "mailto:TARGET@gmail.com"
    );
    pv = 0;
    r = "68DF1E20-9ABB-4413-B86B-02E6E6EB9DCF";
    t = "Hello World";
    v = 1;
}
```

It is a binary plist containing several fields. The following is a table of interesting fields.

t	**Plain text message content**
x	**XML message content**
bid	**“Balloon identifier” for plugin**
bp	**Plugin data**
ati	**Attribution info**
p	**Participants**

We noticed that several of these fields contain binary data that is deserialized with the `NSKeyedUnarchiver` class. (Fields can also be optionally compressed with gzip. To get around this, we wrote a program that calls `[NSData _FTOptionallyDecompressData]` to decompress them on the Mac command line.) Specifically, the **bp** field is deserialized in SpringBoard for the purpose of notifications, which makes deserialization a fully remote attack surface. SpringBoard also does not have any sandboxing on iOS. This field is also deserialized by the MobileSMS process, but this requires one click. The **ati** field is also decoded without user interaction in the imagent process, though it is more restricted in what it can decode than the **bp** field.

- A year of Windows kernel font fuzzing #2: the tech... (Jul)
- How to Compromise the Enterprise Endpoint (Jun)
- A year of Windows kernel font fuzzing #1: the resu... (Jun)
- Exploiting Recursion in the Linux Kernel (Jun)
- Life After the Isolated Heap (Mar)
- Race you to the kernel! (Mar)
- Exploiting a Leaked Thread Handle (Mar)
- The Definitive Guide on Win32 to NT Path Conversio... (Feb)
- Racing MIDI messages in Chrome (Feb)
- Raising the Dead (Jan)

2015

- FireEye Exploitation: Project Zero’s Vulnerability... (Dec)
- Between a Rock and a Hard Link (Dec)
- Windows Sandbox Attack Surface Analysis (Nov)
- Hack The Galaxy: Hunting Bugs in the Samsung Galax... (Nov)
- Windows Drivers are True’ly Tricky (Oct)
- Revisiting Apple IPC: (1) Distributed Objects (Sep)
- Kaspersky: Mo Unpackers, Mo Problems. (Sep)
- Stagefrightened? (Sep)
- Enabling QR codes in Internet Explorer, or a story... (Sep)
- Windows 10^H^H Symbolic Link Mitigations (Aug)
- One font vulnerability to rule them all #4: Window... (Aug)

`NSKeyedArchiver` serialization encodes `NSObject` instances in the plist format. Below is an example portion of a serialized object that includes an instance of `NSURL`.

```
 <dict>
  <key>$class</key>
  <dict>
   <key>CF$UID</key>
   <integer>7</integer>
  </dict>
  <key>NS.base</key>
  <dict>
   <key>CF$UID</key>
   <integer>0</integer>
  </dict>
  <key>NS.relative</key>
  <dict>
   <key>CF$UID</key>
   <integer>6</integer>
  </dict>
 </dict>
 <string>http://www.google.com</string>
 <dict>
  <key>$classes</key>
  <array>
   <string>NSURL</string>
   <string>NSObject</string>
  </array>
  <key>$classname</key>
  <string>NSURL</string>
 </dict>
```

The fields `NS.base` and `NS.relative` are objects that will be used to construct the `NSURL` instance. The `NS.relative` field references the string ‘http://www.google.com’, which represents the URL location. The dictionary below that, with the `$classes` and `$classname` fields, describes the class of the instance, which is referenced by the `$class` field of the first dictionary. When deserializing this instance, the decoder will call `[NSURL initWithCoder:]` which contains code that will deserialize the `NS.base` and `NS.relative` fields and use them to initialize the `NSURL` instance.

- Three bypasses and a fix for one of Flash&#39;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 &quot;good&quot; 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)

NSKeyedArchiver serialization can serialize or deserialize any Objective-C class that implements initWithCoder:, but it has a security feature called NSSecureCoding that allows developers to limit what is decoded. First, classes that implement initWithCoder: must also implement requiresSecureCoding for deserialization to be enabled when NSSecureCoding is enabled. This prevents deserialization code from being accidentally exposed by developers in secure contexts. Secondly, NSSecureCoding requires that all deserialization calls provide a list of allowed classes that can be deserialized, and other classes are not allowed. It's important to note though, that the list of allowed classes is not a complete list of what initWithCoder: methods can be called during serialization. For example, a pseudo-code representation of [NSURL initWithCoder:] with some omissions is as follows.

```
[NSURL initWithCoder:](NSURL *u, id decoder){
 NSData* book = [decoder decodeObjectOfClass:[NSData class]
forKey:@"NS.minimalBookmarkData"];
 if(book)
  return [URLByResolvingBookmarkData:data];
 NSString* base = [decoder decodeObjectOfClass:[NSString class]
forKey:@"NS.base"];
 NSString* relative = [decoder decodeObjectOfClass:[NSString class]
forKey:@"NS.relative"];
 return [NSURL initWithString:base relativeToURL:relative];
}
```

For a URL that is not a bookmark, the method will need to deserialize an instance of class NSString for the NS.relative and NS.base fields, and the NSString class will be allowed in that deserialization. Likewise, if the serialized data contains the NS.minimalBookmarkData field, it will deserialize an instance of NSData. So while the class limits on deserialization limit what class will be returned, they do not limit the attack surface to just that class. They still reduce the attack surface somewhat though.

There are several methods that can be used to create an NSKeyedUnarchiver instance or deserialize an object, and not all of them enable NSSecureCoding by default. The following methods enable it by default:

```
initForReadingFromData:
unarchivedObjectOfClasses:fromData:error:
```

The following methods do not:

2014

- Internet Explorer EPM Sandbox Escape CVE-2014-6350... (Dec)
- pwn4fun Spring 2014 - Safari - Part II (Nov)
- Project Zero Patch Tuesday roundup, November 2014 (Nov)
- Did the "Man With No Name" Feel Insecure? (Oct)
- More Mac OS X and iPhone sandbox escapes and kerne... (Oct)
- Exploiting CVE-2014-0556 in Flash (Sep)
- The poisoned NUL byte, 2014 edition (Aug)
- What does a pointer look like, anyway? (Aug)
- Mac OS X and iPhone sandbox escapes (Jul)
- pwn4fun Spring 2014 - Safari - Part I (Jul)
- Announcing Project Zero (Jul)

```
initWithData:
unarchiveObjectWithData:error
initForReadingWithData:
```

The names of these methods do not make it especially clear whether `NSSecureCoding` is enabled, especially the very similarly named `initForReadingFromData:` and `initForReadingWithData:`. Our first attempt at finding bugs was looking for a place in iMessage that performed deserialization without `NSSecureCoding`. The hope was to be able to use this to deserialize a WebKit instance, and find a way to get it to load a webpage containing a WebKit vulnerability, as there are many such vulnerabilities found on a regular basis, and their exploitability is well understood. Unfortunately, we did not find any deserialization without `NSSecureCoding`.

Next, we looked at extensions. Extensions are a fairly new feature, so we hoped to find a bug in how extensions process the serialized data in the **bp** field. This processing can sometimes be performed without user interaction. Extensions can support previews, in which case SpringBoard will call `previewSummary` in the extension without user interaction. Some versions of iOS also process the entire input by calling `initWithPluginPayload:` without user interaction, but it is inconsistent based on version. This occured on 12.1.2 but not later versions while testing.

We found one bug in the Digital Touch extension, CVE-2019-8624. This extension allows users to send messages containing drawings and other visual elements. Extensions are allowed to use custom encoding so long as they signal to SpringBoard not to attempt to decode the **bp** field, and Digital Touch uses protobuf to decode its payload. It decodes several byte arrays, and in one case incorrectly checks the length of a byte array before copying it, leading to an out-of-bounds read. This issue is very likely not exploitable, but the path to the bug is interesting.

The Link Presentation extension displays link previews when a link is sent in a message. It works by loading the link in WebKit on the sender device and generating a preview text and image that is then sent to the destination device. We looked at this extension in great detail, looking for a way to spawn a WebKit instance on the receiving device, but did not find any. The WebKit processing always appears to be done by the sender.

We then decided to look for bugs in the `initWithCoder:` methods of the classes that are allowed to be deserialized by SpringBoard when generating a message preview. Ian Beer has found several issues of this type allowing privilege escalation in the past. The classes permitted when SpringBoard decodes the **bp** field are: `NSDictionary`, `NSString`, `NSData`, `NSNumber`, `NSURL`, `NSUUID` and `NSValue`. Subclasses of these classes with any level of inheritance are also allowed, as is always the case with `NSKeyedUnarchiver` deserialization. We reviewed the `initWithCoder:` implementations of these classes and their subclasses that are imported by SpringBoard. This analysis resulted in three vulnerabilities.

CVE-2019-8663 is a vulnerability in deserializing the `SGBigUTF8String` class, which is a subclass of `NSString`. The `initWithCoder:` implementation of this class deserializes a byte array that is then treated as a UTF-8 string with a null terminator, even if it does not have one. This can lead to a string that contains out-of-bounds memory being created.

CVE-2019-8661 is a vulnerability in `[NSURL initWithCoder:]` that affects Mac only. When a URL is deserialized, usually an instance of class `NSString` is decoded, but it is also possible for an `NSData` instance to be deserialized, which is then treated as a bookmark. On Mac only, it is possible for this bookmark to be in ‘alis’ alias format, which was deprecated in 2012. This format is processed by a Framework called CarbonCore, which processes the alias file using many safe and unsafe string handling functions. The vulnerability is caused by heap corruption due to an unsafe call to `strcat`. It is important to note that this bookmarking functionality is never legitimately used by iMessage. It is present because `NSURL` deserialization is universal across the system, and so the `initWithCoder` implementation has to support all input possibilities, even ones that will never be encountered in normal use on a specific attack surface.

CVE-2019-8646 is a vulnerability in deserializing a subclass of `NSData`, `_NSDataFileBackedFuture`. This class allows a buffer containing the contents of a file to be created, but it does not load the file until the data is accessed. Deserialization is implemented so that the buffer length is deserialized from the input data, as is the filename, and the implementation never checks that the deserialized length is consistent with the length of the file that is eventually loaded. This violates a basic guarantee that the `NSData` class makes, that the `length` property will be the correct length of the `bytes` property. This can cause a variety of problems, including memory corruption, that will be explored in a future blog post. It is also interesting to note that the `_NSDataFileBackedFuture` class is a hidden class. Classes do not need to be public or exported to be available for deserialization.

After reviewing all the `initWithCoder` implementations, we started to wonder what happens if a subclass of an allowed class does not implement `initWithCoder`. It turns out that it follows normal inheritance rules. This means that it will use the `initWithCoder` implementation for its superclass, but then any method that the class has overridden will be called for the subclass. This turned out to be possible for many subclasses of the allowed classes, for example `initWithCapacity:` is a common method to implement and call. Some classes have checks that prevent inheritance, or more commonly, require direct inheritance (i.e. a subclass that overrides all needed methods is allowed, while a subclass that relies on some superclass implementations is not). This is something that needs to be reviewed on a class by class basis. We determined the classes available by loading the dyld_shared_cache into IDA, and running a script that inspects the Objective C metadata.

One vulnerability we found is CVE-2019-8647. This vulnerability occurs when deserializing class `_PFArray`, which extends `NSArray` and implements `[_PFArray initWithObjects:count:]`, which is called by `[NSArray initWithCoder:]`. This method assumes that all the objects in the array have references to them, which is likely the case for the intended use of this class, but is not the case during deserialization. This means that the array that contains objects that have already been freed can be created and used. It is likely that this class was never intended to be deserialized, and the availability of the method `initWithObjects:count:` for deserialization whenever its containing library is imported by a process that deserializes arrays is behaviour that its developer did not expect.

We reported a similar vulnerability, CVE-2019-8662 in a class that as far as we know is not imported into iMessage, but is likely imported by other applications that use deserialization.

Another interesting question about `NSKeyedArchiver` serialization is what happens if a serialized object contains a cycle. Fundamentally, the `NSKeyedArchiver` format is a plist file containing numeric references, so an object can reference itself, or there can be cycles involving multiple objects. Looking at the source in IDA, deserialization of an object works roughly as follows.

```
    if(temp_dict[key])
      return [temp_dict[key] copy];
     if(obj_dict[key])
      return [obj_dict[key] copy];
     NSObject* a = [NSSomeClass alloc];
     temp_dict[key] = a; //No references!!
     NSObject* obj = [a initWithCoder:];
     temp_dict[key] = NIL;
     obj_dict[key] = obj;
     return obj;
```

So the first time an object is deserialized, `alloc` is called on its class, and then the object returned by alloc is stored in a temporary dictionary that does not retain references to the object. Then `initWithCoder` is called on the allocated object. When that call is finished, the allocated object is removed from the temporary dictionary, and the object returned by `initWithCoder` is added to the permanent object dictionary, which does add a reference to the object.

There are a couple of problems with this scheme. First, there is no guarantee that `initWithCoder` returns the `this` object that it is called with, in fact, the documentation states specifically that this is not guaranteed to be the case. Moreover, `initWithCoder` is responsible for releasing the `this` object in the case where it is not returned. So theoretically, an `initWithCoder` implementation could free the object returned by `alloc` and then deserialize a field that could be a reference to that same object, which would lead to the reference that is returned being an invalid reference to freed memory. We looked, and did not find any `initWithCoder` implementations that have this problem in SpringBoard, but it is possible they exist in other applications, as this does not violate any documented restrictions on the behavior of `initWithCoder` implementations.

Another problem is if an `initWithCoder` implementation ends up deserializing itself and then using the object, it can use the object before the call is completed. This can cause problems if certain methods assume that an object is complete or will not change (because it could continue to change as the `initWithCoder` call completes).

We looked for vulnerabilities involving cycles and found two such bugs. The first is CVE-2019-8641, which we are not yet disclosing, because its fix did not fully remediate the issue.

Another issue that involves cycles in serialized objects was CVE-2019-8660. This vulnerability was in another subclass of `NSDictionary`, `NSKnownKeysDictionary1`. This is another optimized dictionary class that requires that keys be provided up front. In this case, the keys are provided as an instance of class `NSKnownKeysMappingStrategy1`, which deserializes the number of keys separately from the array containing the keys. The deserialized number of keys is checked to be consistent with the key array length after the keys are deserialized, so if a key is another instance of `NSKnownKeysDictionary1`, it can use the `NSKnownKeysMappingStrategy1` instance before the key number has been checked. This allows for an integer overflow leading to memory corruption in `[NSKnownKeysDictionary1 initWithCoder:]` to be reachable when it otherwise wouldn't be.

The nature of `NSKeyedArchiver` serialization makes it extremely difficult to secure. Even if `NSSecureCoding` is enabled, `NSKeyedArchiver` serialization can unintentionally create extremely large attack surfaces. To give an example, what is the attack surface of the following call, if secure coding is enabled?

```
[NSKeyedUnarchiver unarchivedObjectOfClasses:@[NSURL] fromData:mydata
error:NIL];
```

Clearly, it includes the URL deserializer, `[NSURL initWithCoder:]` as well as any subclass deserializers implemented in the calling application, for example `[NSMyURLSubClass initWithCoder:]`.

But it also includes any subclasses of `NSURL` in libraries that were imported by the application. For example, let's say that this application imports the UserNotifications framework. In that case, `[UNSecurityScopedURL initWithCoder:]`, a subclass of `NSURL` would also be a part of the attack surface, even if the library was not imported for the purposes of serialization.

Let's look a bit more at `[NSURL initWithCoder:]`, which was discussed earlier.

```
[NSURL initWithCoder:](NSURL *u, id decoder){
 NSData* book = [decoder decodeObjectOfClass:[NSData class]
forKey:@"NS.minimalBookmarkData"];
 if(book)
  return [URLByResolvingBookmarkData:data];
 NSString* base = [decoder decodeObjectOfClass:[NSString class]
forKey:@"NS.base"];
 NSString* relative = [decoder decodeObjectOfClass:[NSString class]
forKey:@"NS.relative"];
 return [NSURL initWithString:base relativeToURL:relative];
}
```

It contains three calls to `decodeObjectOfClass:forKey:`, decoding object of classes `NSString`, `NSData` and `NSURL` respectively. So `[NSString initWithCoder:]` and `[NSData initWithCoder:]` are now part of the attack surface. The `[NSURL initWithCoder:]` implementation will also parse the provided `NSData` object as a bookmark if it exists, so that is in the attack surface as well.

The attack surface now also includes subclasses of `NSString` and `NSData` too. Assuming that this application only imports the UserNotifications framework as well as Foundation and CoreFoundation which are typically mandatory, the following deserialization functions are now in the attack surface as they are subclasses of those two classes: `[_NSDispatchData initWithCoder:]`, `[__NSLocalizedString initWithCoder:]`, `[NSLocalizableString initWithCoder:]` and `[UNLocalizedString initWithCoder:]`.

Looking at these, there are two methods which allow even more classes. `[UNLocalizedString initWithCoder:]` deserializes an `NSArray` instance, meanwhile `[__NSLocalizedString initWithCoder:]` decodes objects of class `NSDictionary`, `NSNumber` and `NSDate`. We won't investigate the subclasses of these classes for the purposes of this example, but it's clear that they will allow even more classes, and increase the attack surface even more, as will the subclasses of those classes, and so on.

This is just considering the `initWithCoder` methods of the subclasses. Considering that any subclass of the allowed classes could be part of deserialization due to inheritance, the attack surface could be even larger. For example, `[NSString initWithCoder:]` can call `[NSString initWithString:]` or `[NSString initWithBytes:length:encoding:]` depending on what fields are deserialized, so both of those methods of subclasses are part of the attack surface. In this case, this includes `[NSBigMutableString initWithString:]`, `[NSDebugString initWithString:]`, `[NSPlaceholderMutableString initWithBytes:length:encoding:]` and `[NSPlaceholderString initWithBytes:length:encoding:]`. All the other allowed classes have a similar increase in attack surface due to inheritance.

This is an extremely large attack surface for decoding a URL which is probably just a string, and it is an attack surface that gets exponentially larger as an application grows. For example, imagine the impact of importing a few extra libraries on this attack surface. Or adding a few extra classes to the allow list. It's also relevant that classes from different frameworks that were never intended to be used together can be combined during serialization. For example, a URL can be decoded with a string subclass from another framework that contains a data object from another framework, and the resulting object can contain many properties that are of classes that were never intended or expected. The expansive attack surface of deserialization, as well as the many degrees of freedom that are available when deserializing objects from many frameworks are the reasons we found so many vulnerabilities in iMessage.

Conclusion

We investigated the remote attack surface of the iPhone, and reviewed SMS, MMS, VVM, Email and iMessage. Several tools which can be used to further test these attack surfaces were released. We reported a total of 10 vulnerabilities, all of which have since been fixed. The majority of vulnerabilities occurred in iMessage due to its broad and difficult to enumerate attack surface. Most of this attack surface is not part of normal use, and does not have any benefit to users. Visual Voicemail also had a large and unintuitive attack surface that likely led to a single serious vulnerability being reported in it. Overall, the number and severity of the remote vulnerabilities we found was substantial. Reducing the remote attack surface of the iPhone would likely improve its security.

Posted by Ben at 3:21 PM

No comments:

Post a Comment

Enter your comment...

Comment as: Google Accoun ▾

Publish Preview

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)