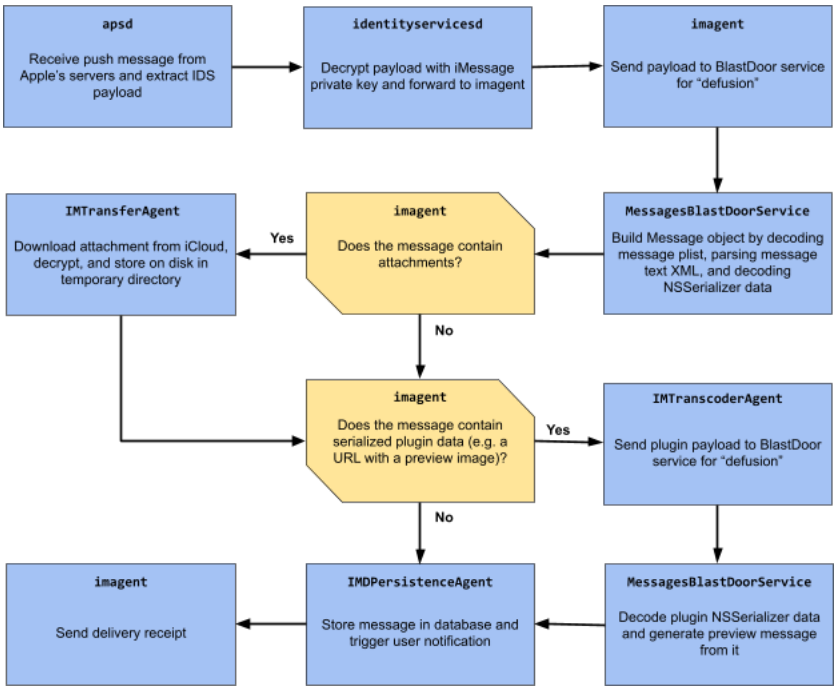
A Look at iMessage in iOS 14

googleprojectzero.blogspot.com/2021/01/a-look-at-imessage-in-ios-14.html



Posted By Samuel Groß, Project Zero

On December 20, Citizenlab published "The Great iPwn", detailing how "Journalists [were] Hacked with Suspected NSO Group iMessage 'Zero-Click' Exploit". Of particular interest is the following note: "We do not believe that [the exploit] works against iOS 14 and above, which includes new security protections". Given that it is also now almost exactly one year ago since we published the Remote iPhone Exploitation blog post series, in which we described how an iMessage 0-click exploit can work in practice and gave a number of suggestions on how similar attacks could be prevented in the future, now seemed like a great time to dig into the security improvements in iOS 14 in more detail and explore how Apple has hardened their platform against 0-click attacks.

The content of this blog post is the result of a roughly one-week reverse engineering project, mostly performed on a M1 Mac Mini running macOS 11.1, with the results, where possible, verified to also apply to iOS 14.3, running on an iPhone XS. Due to the nature of this project and the limited timeframe, it is possible that I have missed some relevant changes or made mistakes interpreting some results. Where possible, I've tried to describe the steps necessary to verify the presented results, and would appreciate any corrections or additions.

The blog post will start with an overview of the major changes Apple implemented in iOS 14 which affect the security of iMessage. Afterwards, and mostly for the readers interested in the technical details, each of the major improvements is described in more detail while also providing a walkthrough of how it was reverse engineered. At least for the technical details, it is recommended to briefly review the blog post series from last year for a basic introduction to iMessage and the exploitation techniques used to attack it.

Overview

Memory corruption based 0-click exploits typically require at least the following pieces:

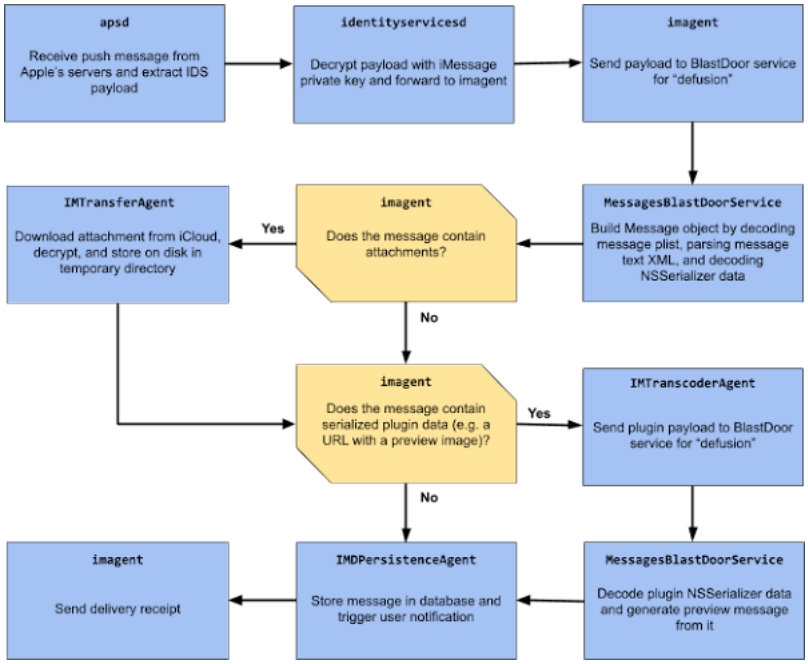
1. A memory corruption vulnerability, reachable without user interaction and ideally without triggering any user notifications
2. A way to break ASLR remotely
3. A way to turn the vulnerability into remote code execution
4. (Likely) A way to break out of any sandbox, typically by exploiting a separate vulnerability in another operating system component (e.g. a userspace service or the kernel)

With iOS 14, Apple shipped a significant refactoring of iMessage processing, and made all four parts of the attack harder. This is mainly due to three central changes:

1. The BlastDoor Service

One of the major changes in iOS 14 is the introduction of a new, tightly sandboxed "BlastDoor" service which is now responsible for almost all parsing of untrusted data in iMessages (for example, NSKeyedArchiver payloads). Furthermore, this service is written in Swift, a (mostly) memory safe language which makes it significantly harder to introduce classic memory corruption vulnerabilities into the code base.

The following diagram shows the rough new iMessage processing pipeline, with the name of the respective service process shown at the top of each box.



As can be seen, the majority of the processing of complex, untrusted data has been moved into the new BlastDoor service. Furthermore, this design with its 7+ involved services allows fine-grained sandboxing rules to be applied, for example, only the IMTransferAgent and apsd processes are required to perform network operations. As such, all services in this pipeline are now properly sandboxed (with the BlastDoor service arguably being sandboxed the strongest).

2. Re-randomization of the Dyld Shared Cache Region

Historically, ASLR on Apple's platforms had one architectural weakness: the shared cache region, containing most of the system libraries in a single prelinked blob, was only randomized per boot, and so would stay at the same address across all processes. This turned out to be especially critical in the context of 0-click attacks, as it allowed an attacker, able to remotely observe process crashes (e.g. through timing of automatic delivery receipts), to infer the base address of the shared cache and as such break ASLR, a prerequisite for subsequent exploitation steps.

However, with iOS 14, Apple has added logic to specifically detect this kind of attack, in which case the shared cache is re-randomized for the targeted service the next time it is started, thus rendering this technique useless. This should make bypassing ASLR in a 0-click attack context significantly harder or even impossible (apart from brute force) depending on the concrete vulnerability.

3. Exponential Throttling to Slow Down Brute Force Attacks

To limit an attacker's ability to retry exploits or brute force ASLR, the BlastDoor and imagent services are now subject to a newly introduced exponential throttling mechanism enforced by launchd, causing the interval between restarts after a crash to double with every subsequent crash (up to an apparent maximum of 20 minutes). With this change, an exploit that relied on repeatedly crashing the attacked service would now likely require in the order of multiple hours to roughly half a day to complete instead of a few minutes.

The remainder of this blog post will now look at each of these three changes in greater depths.

The BlastDoor Service

The new BlastDoor service and its role in the processing of iMessages can be studied by following the flow of an incoming iMessage. On the wire, a simple text iMessage would look something like this, encoded as binary plist:

```
{
    // Group UUID
    gid = "008412B9-A4F7-4B96-96C3-70C4276CB2BE";
    // Group protocol version
    gv = 8;
```

```
    // Chat participants
    p =     (
        "mailto:sender@foo.bar",
        "mailto:receiver@foo.bar"
    );
    // Participants version
    pv = 0;
    // Message being replied to, usually the last message in the chat
    r = "6401430E-CDD3-4BC7-A377-7611706B431F";
    // The plain text content
    t = "Hello World!";
    // Probably some other version number
    v = 1;
    // The rich text content
    x = "<html><body>Hello World!</body></html>";
}
```

As such, the minimal steps required to parse it are:

1. If necessary, decompress the binary data
2. Decode the plist from its binary serialization format
3. Extract its various fields and ensure they have the correct type
4. Decode the `x` key if present, using an XML decoder

Previously, all of this work happened in imagent. With iOS 14, however, it all moved into the new BlastDoor service. While the main processing flow still starts in imagent, which receives the raw but unencrypted payload bytes from identityservicesd (part of the IDS framework) in -[IMDiMessageIDSDelegate service:account:incomingTopLevelMessage:fromID:messageContext:], messages are then more or less immediately forwarded to the BlastDoor service through +[IMBlastdoor sendDictionary:withCompletionBlock:] which creates the reply handler block and then calls -[IMMessagesBlastDoorInterface diffuseTopLevelDictionary:resultHandler:]. At that point processing ends up in Swift code that deserializes the binary payload and sends it to the BlastDoor service over XPC.

Inside BlastDoor, the work mostly happens in BlastDoor.framework and MessagesBlastDoorService. As most of it is written in Swift, it is fairly unpleasant to statically reverse engineer it (no symbols, many virtual calls, swift runtime code sprinkled all over the place), but fortunately, that is also not really necessary for the purpose of this blog post. However, it is worth noting that while the high level control flow logic is written in Swift, some of the parsing steps still involve the existing ObjectiveC or C implementations. For example, XML is being parsed by libxml, and the NSKeyedArchiver payloads by the ObjectiveC implementation of NSKeyedUnarchiver.

The responses from BlastDoor can be seen by breaking on the reply handler function in imagent (the function can be found in +[IMBlastdoor sendDictionary:withCompletionBlock:] or by searching for XREFs to the string “Blastdoor response %p received (command: %hhu, guid: %@)” in IMDaemonCore.framework). A typical BlastDoor response for a simple text message is shown below:

```
(lldb) po $x2
TextMessage(
    metadata: BlastDoor.Metadata(
        messageGUID: D391CC96-9CC6-44C6-B827-1DEB0F252529,
        timestamp: Optional(1610108299117662350),
        wantsDeliveryReceipt: true,
        wantsCheckpointing: false,
        storageContext: BlastDoor.Metadata.StorageContext(
```

```
            isFromStorage: false, isLastFromStorage: false
        )
    ),
    messageSubType: MessageType.textMessage(BlastDoor.Message(
        plainTextBody: Optional("Hello World"),
        plainTextSubject: nil,
        content: Optional(BlastDoor.AttributedString(
            attributes: [
                BlastDoor.BaseWritingDirectionAttribute(
                    range: Range(0..<11), direction: WritingDirection.natural
                ),
                BlastDoor.MessagePartAttribute(
                    range: Range(0..<11), partNumber: 0
                )
            ],
            string: "Hello World"
        )),
        _participantDestinationIdentifiers: [
            "mailto:sender@foo.bar",
            "mailto:receiver@foo.bar"
        ],
        attributionInfo: []
    )),
    encryptionType: BlastDoor.TextMessage.EncryptionType.pair_ec,
    replyToGUID: Optional(6401430E-CDD3-4BC7-A377-7611706B431F),
    _threadIdentifierGUID: nil,
    _expressiveSendStyleIdentifier: nil,
    _groupID: Optional("008412B9-A4F7-4B96-96C3-70C4276CB2BE"),
    currentGroupName: nil,
    groupParticipantVersion: Optional(0),
    groupProtocolVersion: Optional(8),
    groupPhotoCreationTime: nil,
    messageSummaryInfo: nil,
    nicknameInformation: nil,
    truncatedNicknameRecordKey: nil
)
```

One can roughly associate every field in this data structure with parts of the on-wire iMessage format. For example, the plainTextBody field contains the content of the `t` field, while the content field corresponds to the content of the `x` field.

Besides simple text messages, iMessages can additionally contain attachments (essentially arbitrary files which are encrypted and temporarily uploaded to iCloud) as well as rather complex serialized NSKeyedArchiver archives, which have been the source of bugs in the past.

For these types of iMessages, the following additional parsing steps are necessary:

1. Unpack attachment metadata (NSKeyedArchiver format)
2. Download attachments from iCloud server
3. Deserialize NSKeyedArchiver plugin archives and generate a preview for the notification

As an example, consider what happens when a user sends a link to a website over iMessage. In that case, the sending device will first render a preview of the webpage and collect some metadata about it (such as the title and page description), then pack those fields into an NSKeyedArchiver archive. This archive is then encrypted with a temporary key and uploaded to the iCloud servers. Finally, the link as well as the decryption key are sent to the receiver as part of the iMessage. In order to create a useful user notification about the incoming iMessage, this data has to be processed by the receiver on a 0-click code path. As that again involves a fair amount of complexity, it is also done inside BlastDoor: after receiving the BlastDoor reply from above and realizing that the message contains an attachment, imagent first instructs IMTransferAgent to download and decrypt the iCloud attachment. Afterwards, it will call into -[IMTranscodeController decodeiMessageAppPayload:bundleID:completionBlock:blockUntilReply:] which forwards the relevant data to the IMTranscoderAgent, which then proceeds into +[IMAttachmentBlastdoor sendBalloonPluginPayloadData:withBundleIdentifier:completionBlock:] and finally calls -[IMMessagesBlastDoorInterface defuseBalloonPluginPayload:withIdentifier:resultHandler:].

In the BlastDoor service, the plugin data decoding is then again performed in Swift, and dispatched to the corresponding plugin type, as determined by the plugin id. For RichLinks (plugin id com.apple.messages.URLBalloonProvider), processing ends up in LinkPresentation.MessagesPayload.init(dataRepresentation:), which deserializes the NSKeyedArchiver payload and to extract the preview image and URL metadata from it in order to generate a preview message.

Sandboxing

The sandbox profile can be found in System/Library/Sandbox/Profiles/blastdoor.sb and is also attached at the end of this blog post. It appears to be identical on iOS and macOS. The profile can be studied statically, and for that purpose is attached at the bottom of this blogpost, or dynamically, for example by using the sandbox-exec tool:

```
> echo "(allow process-exec (literal \"$(pwd)/test\"))" >> ./blastdoor.sb

> clang -o test test.c   # try to open files, network connections, etc.

> sandbox-exec -f ./blastdoor.sb ./test
```

The sandbox profile states:

```
;;; This profile contains the rules necessary to make BlastDoor as close to

;;; compute-only as possible, while still remaining functional.
```

And indeed, the sandbox profile is quite tight:

- only a handful of local IPC services, namely diagnosticd, logd, opendirectoryd, syslogd, and notifyd, can be reached
- almost all file system interaction is blocked
- any interaction with IOKit drivers (historically a big source of vulnerabilities) is forbidden
- outbound network access is denied

Furthermore, the profile makes use of syscall filtering to restrict the interactions with the core kernel. However, as of now the syscall filter seems to be in “permissive” mode:

```
;; To be uncommented once the system call whitelist is complete...

;; (deny syscall-unix (with send-signal SIGKILL))
```

As such, the BlastDoor service is still allowed to perform any syscall, but it is to be expected that the syscall filtering will soon be put into “enforcement mode”, which would further boost its effectiveness.

Crash Monitoring?

An interesting side effect of the new processing pipeline is that imagent is now able to detect when an incoming message caused a crash in BlastDoor (it will receive an XPC error). Even more interesting is the fact that imagent appears to be informing Apple’s servers about such events, as can be seen by setting a breakpoint on -[APSConnectionServer handleSendOutgoingMessage:] in apsd, the daemon responsible for implementing Apple’s push services (on top of which iMessage is built). Displaying the outgoing message will show the following:

```
(lldb) po [$x2 dictionaryRepresentation]
```

```
{
    APSCritical = 1;
    APSMessageID = 543;
    APSMessageIdentifier = 1520040396;
    APSMessageTopic = "com.apple.madrid";
    APSMessageUserInfo =     {
        c = 115;
        fR = 13500;
        fRM = "c-100-BlastDoor.Explosion-1-com.apple.BlastDoor.XPC-ServiceCrashed";
        fU = {length = 16, bytes = 0x3a4912626c9645f98cb26c7c2d439520};
        i = 1520040396;
        nr = 1;
        t = {length = 32, bytes = ... };
        ua = "[macOS,11.1,20C69,Macmini9,1]";
        v = 7;
    };
    APSOutgoingMessageSenderTokenName = 501;
    APSPayloadFormat = 1;
    APSTimeout = 120;
    APSTimestamp = "2021-01-06 19:52:10 +0000";
}
```

As can be seen, imagent is apparently informing the iMessage servers that the message with the UUID 0x3a4912626c9645f98cb26c7c2d439520 (fU key) has caused a crash in BlastDoor.

It is unclear what the purpose of this is without access to the server's code. While these notifications may simply be used for statistical purposes, they would also give Apple a fairly clear signal about attacks against iMessage involving brute-force and a somewhat weaker signal about any failed exploits against the BlastDoor service.

In my experiments, after observing one of these crash notifications, the server would start directly sending delivery receipts to the sender for messages that hadn't actually been processed by the receiver yet. Possibly this is another, independent effort to break the crash oracle technique by confusing the sender, but that is hard to verify without access to the code running on the server. In any case, it is worth noting that this "spoofing" of delivery receipts by the server is generally possible as the message UUID, which is more or less the only content of a delivery receipt, is part of the non-end2end encrypted payload and is thus known to the server (break on -[APSConnectionServer handleSendOutgoingMessage:] and inspect outgoing iMessages to verify this, the UUID will be in the U key, while the e2e-encrypted data will be in the P key). This is most likely necessary so the server can track which messages have already been delivered and which ones it still needs to keep around for delivery in the future.

Shared Cache Resliding

Previously, when exploiting an iMessage memory corruption bug, a "crash oracle" could be used to reveal the location of the shared cache region in memory: the attacker would trigger the memory corruption bug in a way that would cause an access to a memory location somewhere in the region 0x180000000 - 0x280000000 (where the shared cache can be mapped). If the memory was valid, no crash would occur and imagent would then send a delivery receipt to the attacker. However, If a crash occurred, no such receipt would be delivered, informing the attacker that the address was unmapped. Through clever selection of the queried addresses, the location of the shared cache could be revealed in logarithmic time, with only about 20 messages.

However, with iOS 14 Apple has added a mechanism to re-randomize the location of the shared cache region for an "attacked" process, thus breaking a fundamental assumption of this technique and rendering it ineffective. This is significant as the crash oracle technique was one of very few, if not the only, fairly generic ASLR bypass techniques usable in 0-click iMessage attacks.

To understand how the shared cache resliding works, one can start by looking at the kernel. In iOS 14, the kernel can now have two active shared cache regions: the “regular” region and a “reslided” region. During an attack, the following then happens:

1. When an attacker attempts to use a crash-oracle-based technique, the attacked process would quickly end up accessing unmapped memory in the range 0x180000000 - 0x280000000 (where the shared cache is mapped) and crashes
2. The kernel handles the segmentation fault generated by the CPU, and sets a specific flag in the crash info that signals that the crash happened inside the shared cache region
3. At the same time, the kernel will mark the currently active reslided shared cache region (if one exists) as stale, causing it to be recreated and thus re-randomized the next time it is used
4. launchd (as the parent process of the crashed service) receives the crash info, notices the OS_REASON_FLAG_SHAREDREGION_FAULT flag, and sets the ReslideSharedCache property on the service associated with the crashed process (see `launchctl procinfo $pid` and search for `reslide shared cache = 1`)
5. The next time the service is restarted, launchd then adds the POSIX_SPAWN_RESLIDE attribute for posix_spawn due to the ReslideSharedCache property
6. In the kernel, this flag now causes the newly created process to be given the reslided shared cache image. However, as no active reslided region currently exists (the previous one was marked as stale in step 3.), a new one is created at a newly randomized address.

The result of this is that whenever an attacker attempts to use a crash-oracle to break ASLR, the attacked service would receive a different shared cache region every time it is launched, thus preventing the attack from succeeding. For the time being, this feature appears to only be active on iOS though, but it would be expected to come to macOS as well.

While this mechanism would in principle also protect 3rd party apps from similar attacks, protection for those is currently somewhat weaker, likely in order to first evaluate the real-world performance impact of this change (the shared cache is a significant performance optimization of the OS). In particular, step 3 is currently only performed if the crashing process is a platform binary (essentially binaries that ship with the OS and are directly signed by Apple) such as the services handling iMessages. However, for 3rd party processes, it would only happen if the global vm_shared_region_reslide_restrict is set to zero:

```
/*
 * Flag to control what processes should get shared cache randomize resliding
 * after a fault in the shared cache region:
 *
 * 0 - all processes get a new randomized slide
 * 1 - only platform processes get a new randomized slide
 */
```

Which is controlled by the vm_shared_region_reslide_restrict bootarg. This currently seems to be set to one. In essence, for 3rd party apps this means:

1. When the attacked process first crashes, the kernel will still set the OS_REASON_FLAG_SHAREDREGION_FAULT flag, and launchd will add the ReslideSharedCache property, but the current reslided region won’t be invalidated
2. The restarted service is then restarted and now uses the “reslided” shared cache region
3. When the service crashes the next time, and if that service is the only one currently using the reslided shared cache region (which should usually be the case, but could possibly be influenced by the attacker), the region’s refcount drops to zero, and the shared cache region is marked for removal.
4. However, removal will only actually happen after two minutes. As such, if the service is restarted within two minutes, it will receive the same shared cache region at the same location in memory.

As a result, a third-party app could still be attacked through a crash-oracle technique if it automatically sends some form of delivery receipt to the sender and restarts quickly enough after a crash. This could, however, be prevented for example by enabling ExponentialThrottling for these services. Ideally, and assuming that the performance penalty is reasonable, Apple would enable re-randomization for all apps in the future.

Exponential Throttling

Another thing we suggested back in 2019 was to limit the number of attempts an attacker gets when attempting to exploit a vulnerability. This was mostly important to defend against the crash-oracle technique, but would also help to prevent brute force attacks (e.g., given enough attempts, one could simply brute force the location of the shared cache region). The new ExponentialThrottling feature in launchd seems to achieve just that.

To use it, a system daemon or agent has to opt-in by setting "_ExponentialThrottling = 1” in its Info.plist (essentially the service metadata), as can be seen below for the BlastDoor service:

```
> plutil -p
/System/Library/PrivateFrameworks/MessagesBlastDoorSupport.framework/Versions/A/XPCServices/MessagesBlastDoorService.xpc/Contents/In

{
  "CFBundleDisplayName" => "MessagesBlastDoorService"
  "CFBundleExecutable" => "MessagesBlastDoorService"
  "CFBundleIdentifier" => "com.apple.MessagesBlastDoorService"
  ...
  "XPCService" => {
    "_ExponentialThrottling" => 1
  }
}
```

Apart from the BlastDoor service, it is also used for imagent:

```
> plutil -p /System/Library/LaunchAgents/com.apple.imagent.plist

{
  "_ExponentialThrottling" => 1
  ...
```

but doesn't appear to be used for any other service, as can, for example, be seen by looking at the output of the launchctl dumpstate command, which will only show "exponential throttling = 1" for com.apple.imagent and com.apple.MessagesBlastDoorService.

Presumably, the _ExponentialThrottling property instructs launchd (the macOS and iOS init process), to delay subsequent restarts of a crashing service. While it is somewhat challenging to statically reverse engineer launchd due to the lack of source code or binary symbols, it is fortunately fairly easy to experimentally determine the impact of the _ExponentialThrottling property, for example by installing a custom daemon that writes the current timestamp to a file before crashing. By default, so without ExponentialThrottling, one would see the following:

```
Service started on Wed Jan  6 13:56:03 2021
Service started on Wed Jan  6 13:56:13 2021
Service started on Wed Jan  6 13:56:23 2021
Service started on Wed Jan  6 13:56:33 2021
```

As can be seen, by default, a service is, at the earliest, restarted ten seconds after it was previously started. However, using the following service plist which enables ExponentialThrottling:

```
> # Start service with
> # launchctl bootstrap system /Library/LaunchDaemons/net.saelo.test.plist
> plutil -p /Library/LaunchDaemons/net.saelo.test.plist
{
  "_ExponentialThrottling" => 1
  "KeepAlive" => 1
  "Label" => "net.saelo.test"
  "POSIXSpawnType" => "Interactive"
  "Program" => "/path/to/program"
}
```

One can observe the following:

```
Service started on Wed Jan  6 10:42:43 2021
```

Service started on Wed Jan 6 10:42:53 2021 (+10s)

Service started on Wed Jan 6 10:43:03 2021 (+10s)

Service started on Wed Jan 6 10:43:13 2021 (+10s)

Service started on Wed Jan 6 10:43:33 2021 (+20s)

Service started on Wed Jan 6 10:44:13 2021 (+40s)

Service started on Wed Jan 6 10:45:33 2021 (+80s)

Service started on Wed Jan 6 10:48:13 2021 (+160s [~2.5m])

Service started on Wed Jan 6 10:53:33 2021 (+320s [~5m])

Service started on Wed Jan 6 11:04:13 2021 (+640s [~10m])

Service started on Wed Jan 6 11:24:13 2021 (+20m)

Service started on Wed Jan 6 11:44:13 2021 (+20m)

Service started on Wed Jan 6 12:04:13 2021 (+20m)

Here, the exponential increase in the time between subsequent restarts is clearly visible, and goes up to an apparent maximum of 20 minutes. And indeed, launchd does contain the following bit of code in a function presumably responsible for computing the next restart delay (search for XREFs to the string "%s: service throttled by %llu seconds"):

```
if ( delay >= 1200 )
  result = 1200LL;                // 20 minutes
else
  result = delay;
```

With this change, an exploit that relied on brute force would now only get one attempt every 20 minutes instead of every 10 seconds.

(Upcoming?) ObjectiveC ISA PAC

The PoC exploit against iMessage on iOS 12.4 relied heavily on faking ObjectiveC objects to gain a form of arbitrary code execution despite the presence of pointer authentication (PAC). This was mainly possible because the ISA field, containing the pointer to the Class object and thus making a piece of memory appear like a valid ObjectiveC object, was not protected through PAC and could thus be faked. With iOS 14, this now seems to be changing: while previously, the top 19 bits of the ISA value contained the inline refcount, it now appears that this field has been reduced to 9 bits (of which the LSB appears to be reserved for some purpose, leaving an 8-bit inline refcount, see the bit shifting logic in objc_release or objc_retain), while the freed-up bits now hold a PAC, as can be seen in objc_rootAllocWithZone in libobjc.dylib:

```
  ; Allocate the object
  BL            j__calloc_3
  CBZ           X0, loc_1953DA434
  MOV           X8, X0
  ; “Tag” the address with a constant to get a PAC modifier value
  MOVK          X8, #0x6AE1,LSL#48
  MOV           X9, X19
  ; Compute PAC of Class pointer with tagged object address as modifier
  PACDA         X9, X8
  ; Clear top 9 bits (inline refcnt) and bottom 3 bits (other bitfields)
  AND           X8, X9, #0x7FFFFFFFFFFFF8
  ; Set LSB and inline refcount to one
  MOV           X9, #0x100000000000001
```

```
ORR            X9, X8, X9

; Presumably, the refcnt isn't used for all types of classes...

TST            W20, #0x2000

CSEL           X8, X9, X8, EQ

; Store the resulting value into the ISA field

STR            X8, [X0]
```

However, currently the ISA PAC appears to never be checked, as such, it doesn't yet affect any exploits. The most likely reason for this is that the ISA PAC feature is being rolled out in multiple phases, with the current implementation meant to allow in-depth performance evaluation, in particular of the reduced size of the inline refcount, which will likely cause more objects to use the more expensive out-of-line refcounting (used once the inline refcount saturates). With that, it can be expected that, in the absence of major performance issues, future releases of iOS and macOS will use PAC for the ObjC ISA field, thus likely breaking exploits that have to rely on faking ObjectiveC objects to achieve arbitrary code execution.

Conclusion

This blog post discussed three improvements in iOS 14 affecting iMessage security: the BlastDoor service, resliding of the shared cache, and exponential throttling. Overall, these changes are probably very close to the best that could've been done given the need for backwards compatibility, and they should have a significant impact on the security of iMessage and the platform as a whole. It's great to see Apple putting aside the resources for these kinds of large refactorings to improve end users' security. Furthermore, these changes also highlight the value of offensive security work: not just single bugs were fixed, but instead structural improvements were made based on insights gained from exploit development work.

As for the alleged NSO iMessage exploit, it may have been prevented from working against iOS 14 by any of the following:

> The bug was fixed in iOS 14, for example due to the rewrite of large parts of the iMessage processing pipeline in Swift

- The mere fact that processing happens in a different process, which could for example break a heap layouting primitive
- The shared cache resliding would break their exploit if their exploit relied on some form of crash oracle to break ASLR
- The stronger sandbox of the BlastDoor service, which could prevent the exploitation of a privilege escalation vulnerability after compromising the BlastDoor process

While these are some possible scenarios, and it could be the case that the exploit "just" needs some re-engineering to function again, the fact that these security improvements were shipped is certainly a positive outcome.

Attachment 1: blastdoor.sb

```
;;; This profile contains the rules necessary to make BlastDoor as close to

;;; compute-only as possible, while still remaining functional.

;;;

;;; For all platforms:
/System/Library/PrivateFrameworks/MessagesBlastDoorSupport.framework/XPCServices/MessagesBlastDoorService.xpc/MessagesBlastDoorSe

(version 1)

;;; ----------------------------------------------------------------------------------------- ;;;

;;; Basic Rules

;;; ----------------------------------------------------------------------------------------- ;;;

;; Deny all default rules.

(deny default)

(deny file-map-executable process-info* nvram*)

(deny dynamic-code-generation)

;; Rules copied from system.sb. Ones that we've deemed overly permissive

;; or unnecessary for BlastDoor have been removed.
```

```
;; Allow read access to standard system paths.
(allow file-read*
    (require-all (file-mode #o0004)
        (require-any (subpath "/System")
            (subpath "/usr/lib")
            (subpath "/usr/share")
            (subpath "/private/var/db/dyld"))))
(allow file-map-executable
    (subpath "/System/Library/CoreServices/RawCamera.bundle")
    (subpath "/usr/lib")
    (subpath "/System/Library/Frameworks"))
(allow file-test-existence (subpath "/System"))
(allow file-read-metadata
    (literal "/etc")
    (literal "/tmp")
    (literal "/var")
    (literal "/private/etc/localtime"))
;; Allow access to standard special files.
(allow file-read*
    (literal "/dev/random")
    (literal "/dev/urandom"))
(allow file-read* file-write-data
    (literal "/dev/null")
    (literal "/dev/zero"))
(allow file-read* file-write-data file-ioctl
    (literal "/dev/dtracehelper"))
;; TODO: Don't allow core dumps to be written out unless this is on a dev
;; fused device?
(allow file-write*
    (require-all (regex #"^/cores/")
        (require-not (file-mode 0))))
;; Allow IPC to standard system agents.
(allow mach-lookup
    (global-name "com.apple.diagnosticd")
    (global-name "com.apple.logd")
    (global-name "com.apple.system.DirectoryService.libinfo_v1")
    (global-name "com.apple.system.logger")
```

```
    (global-name "com.apple.system.notification_center"))
;; Allow mostly harmless operations.
(allow signal process-info-dirtycontrol process-info-pidinfo
    (target self))
;; Temporarily allow sysctl-read with reporting to see if this is
;; used for anything.
(allow (with report) sysctl-read)
;; We don't need to post any darwin notifications.
(deny darwin-notification-post)
;; We shouldn't allow any other file operations not covered under
;; the default of deny above.
(deny file-clone file-link)
;; Don't deny file-test-existence: <rdar://problem/59611011>
;; (deny file-test-existence)
;; Don't allow access to any IOKit properties.
(deny iokit-get-properties)
(deny mach-cross-domain-lookup)
;; Don't allow BlastDoor to spawn any other XPC services other than
;; ones that we can intentionally whitelist later.
(deny mach-lookup (xpc-service-name-regex #".*"))
;; Don't allow any commands on sockets.
(deny socket-ioctl)
;; Denying this should have no ill effects for our use case.
(deny system-privilege)
;; To be uncommented once the system call whitelist is complete...
;; (deny syscall-unix (with send-signal SIGKILL))
(allow syscall-unix
    (syscall-number SYS_exit)
    (syscall-number SYS_kevent_qos)
    (syscall-number SYS_kevent_id)
    (syscall-number SYS_thread_selfid)
    (syscall-number SYS_bsdthread_ctl)
    (syscall-number SYS_kdebug_trace64)
    (syscall-number SYS_getattrlist)
    (syscall-number SYS_sigsuspend_nocancel)
    (syscall-number SYS_proc_info)
```

(syscall-number SYS___disable_threadsignal)

(syscall-number SYS___pthread_sigmask)

(syscall-number SYS___mac_syscall)

(syscall-number SYS___semwait_signal_nocancel)

(syscall-number SYS_abort_with_payload)

(syscall-number SYS_access)

(syscall-number SYS_bsdthread_create)

(syscall-number SYS_bsdthread_terminate)

(syscall-number SYS_close)

(syscall-number SYS_close_nocancel)

(syscall-number SYS_connect)

(syscall-number SYS_csops_audittoken)

(syscall-number SYS_csrctl)

(syscall-number SYS_fcntl)

(syscall-number SYS_fsgetpath)

(syscall-number SYS_fstat64)

(syscall-number SYS_fstatfs64)

(syscall-number SYS_getdirentries64)

(syscall-number SYS_geteuid)

(syscall-number SYS_getfsstat64)

(syscall-number SYS_getgid)

(syscall-number SYS_getrlimit)

(syscall-number SYS_getuid)

(syscall-number SYS_ioctl)

(syscall-number SYS_issetugid)

(syscall-number SYS_lstat64)

(syscall-number SYS_madvise)

(syscall-number SYS_mmap)

(syscall-number SYS_munmap)

(syscall-number SYS_mprotect)

(syscall-number SYS_mremap_encrypted)

(syscall-number SYS_open)

(syscall-number SYS_open_nocancel)

(syscall-number SYS_openat)

(syscall-number SYS_pathconf)

(syscall-number SYS_pread)

(syscall-number SYS_read)

```
    (syscall-number SYS_readlink)
    (syscall-number SYS_shm_open)
    (syscall-number SYS_socket)
    (syscall-number SYS_stat64)
    (syscall-number SYS_statfs64)
    (syscall-number SYS_sysctl)
    (syscall-number SYS_sysctlbyname)
    (syscall-number SYS_workq_kernreturn)
    (syscall-number SYS_workq_open)
)
;; Still allow the system call but report in log.
(allow (with report) syscall-unix)
;; For validating the entitlements of clients. This is so only entitled
;; clients can pass data into a BlastDoor instance.
(allow process-info-codesignature)
;;; ------------------------------------------------------------------------------------------ ;;;
;;; Reading Files
;;; ------------------------------------------------------------------------------------------ ;;;
;; Support for BlastDoor receiving sandbox extensions from clients to either read files, or
;; write to a target location.
;; com.apple.app-sandbox.read
(allow file-read*
    (extension "com.apple.app-sandbox.read"))
;; com.apple.app-sandbox.read-write
(allow file-read* file-write*
    (extension "com.apple.app-sandbox.read-write"))
```