Process injection: breaking all macOS security layers with a single vulnerability

sector7.computest.nl/post/2022-08-process-injection-breaking-all-macos-security-layers-with-a-single-vulnerability

August 12, 2022

August 12, 2022

If you have created a new macOS app with Xcode 13.2, you may noticed this new method in the template:

```
- (BOOL)applicationSupportsSecureRestorableState:(NSApplication *)app {
        return YES;
}
```

This was added to the Xcode template to address a process injection vulnerability we reported!

In October 2021, Apple fixed CVE-2021-30873. This was a process injection vulnerability affecting (essentially) all macOS AppKit-based applications. We reported this vulnerability to Apple, along with methods to use this vulnerability to escape the sandbox, elevate privileges to root and bypass the filesystem restrictions of SIP. In this post, we will first describe what process injection is, then the details of this vulnerability and finally how we abused it.

This research was also published at Black Hat USA 2022 and DEF CON 30.

Process injection

Process injection is the ability for one process to execute code in a different process. In Windows, one reason this is used is to evade detection by antivirus scanners, for example by a technique known as DLL hijacking. This allows malicious code to pretend to be part of a different executable. In macOS, this technique can have significantly more impact than that due to the difference in permissions two applications can have.

In the classic Unix security model, each process runs as a specific user. Each file has an owner, group and flags that determine which users are allowed to read, write or execute that file. Two processes running as the same user have the same permissions: it is assumed there is no security boundary between them. Users are security boundaries, processes are not. If two processes are running as the same user, then one process could attach to the other as a debugger, allowing it to read or write the memory and registers of that other process. The root user is an exception, as it has access to all files and processes. Thus, root can always access all data on the computer, whether on disk or in RAM.

This was, in essence, the same security model as macOS until the introduction of SIP, also known as “rootless”. This name doesn’t mean that there is no root user anymore, but it is now less powerful on its own. For example, certain files can no longer be read by the root user unless the process also has specific entitlements. Entitlements are metadata that is included when generating the code signature for an executable. Checking if a process has a certain entitlement is an essential part of many security measures in macOS. The Unix ownership rules are still present, this is an additional layer of permission checks on top of them. Certain sensitive files (e.g. the Mail.app database) and features (e.g. the webcam) are no longer possible with only root privileges but require an additional entitlement. In other words, privilege escalation is not enough to fully compromise the sensitive data on a Mac.

For example, using the following command we can see the entitlements of Mail.app:

```
$ codesign -dvvv --entitlements - /System/Applications/Mail.app
```

In the output, we see the following entitlement:

```
...
        [Key] com.apple.rootless.storage.Mail
        [Value]
                [Bool] true
...
```

This is what grants Mail.app the permission to read the SIP protected mail database, while other malware will not be able to read it.

Aside from entitlements, there are also the permissions handled by Transparency, Consent and Control (TCC). This is the mechanism by which applications can request access to, for example, the webcam, microphone and (in recent macOS versions) also files such as those in the Documents and Download folders. This means that even applications that do not use the Mac Application sandbox might not have access to certain features or files.

Of course entitlements and TCC permissions would be useless if any process can just attach as a debugger to another process of the same user. If one application has access to the webcam, but the other doesn’t, then one process could attach as a debugger to the other process and inject some code to steal the webcam video. To fix this, the ability to debug other applications has been heavily restricted.

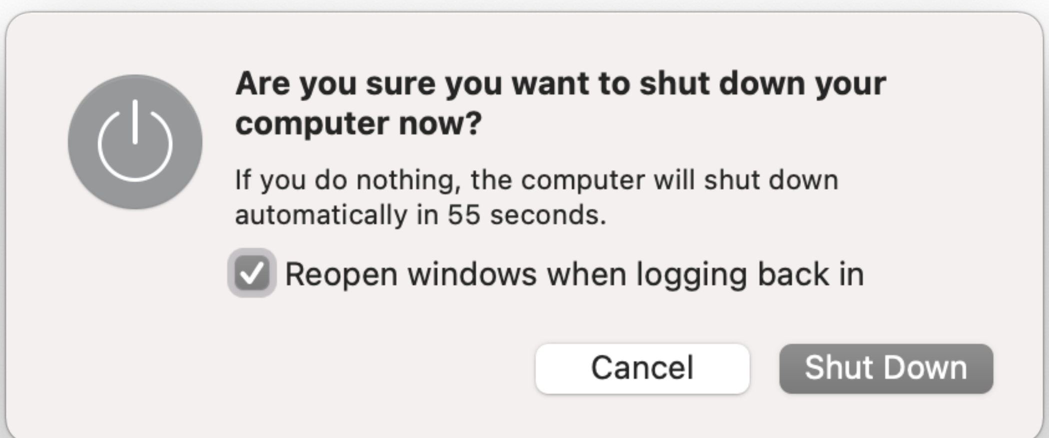
Changing a security model that has been used for decades to a more restrictive model is difficult, especially in something as complicated as macOS. Attaching debuggers is just one example, there are many similar techniques that could be used to inject code into a different process. Apple has squashed many of these techniques, but many other ones are likely still undiscovered.

Aside from Apple's own code, these vulnerabilities could also occur in third-party software. It's quite common to find a process injection vulnerability in a specific application, which means that the permissions (TCC permissions and entitlements) of that application are up for grabs for all other processes. Getting those fixed is a difficult process, because many third-party developers are not familiar with this new security model. Reporting these vulnerabilities often requires fully explaining this new model! Especially Electron applications are infamous for being easy to inject into, as it is possible to replace their JavaScript files without invalidating the code signature.

More dangerous than a process injection vulnerability in one application is a process injection technique that affects multiple, or even *all*, applications. This would give access to a large number of different entitlements and TCC permissions. A generic process injection vulnerability affecting all applications is a very powerful tool, as we'll demonstrate in this post.

The saved state vulnerability

When shutting down a Mac, it will prompt you to ask if the currently open windows should be reopened the next time you log in. This is a part of functionally called "saved state" or "persistent UI".



When reopening the windows, it can even restore new documents that were not yet saved in some applications.

It is used in more places than just at shutdown. For example, it is also used for a feature called App Nap. When application has been inactive for a while (has not been the focused application, not playing audio, etc.), then the system can tell it to save its state and terminates the process. macOS keeps showing a static image of the application's windows and in the Dock it still appears to be running, while it is not. When the user switches back to the application, it is quickly launched and resumes its state. Internally, this also uses the same saved state functionality.

When building an application using AppKit, support for saving the state is for a large part automatic. In some cases the application needs to include its own objects in the saved state to ensure the full state can be recovered, for example in a document-based application.

Each time an application loses focus, it writes to the files:

```
~/Library/Saved Application State/<Bundle ID>.savedState/windows.plist
~/Library/Saved Application State/<Bundle ID>.savedState/data.data
```

The `windows.plist` file contains a list of all of the application's open windows. (And some other things that don't look like windows, such as the menu bar and the Dock menu.)

For example, a `windows.plist` for TextEdit.app could look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
	<dict>
		<key>MenuBar AvailableSpace</key>
		<real>1248</real>
		<key>NSDataKey</key>
		<data>
		Ay1IqBriwup4bKAanpWcEw==
		</data>
		<key>NSIsMainMenuBar</key>
		<true/>
		<key>NSWindowID</key>
		<integer>1</integer>
		<key>NSWindowNumber</key>
		<integer>5978</integer>
	</dict>
	<dict>
		<key>NSDataKey</key>
		<data>
		5lyzOSsKF24yEcwAKTBSVw==
		</data>
		<key>NSDragRegion</key>
		<data>
		AAAAgAIAAADAAQAABAAAAAMAAABHAgAAxgEAAAoAAAADAAAABwAAABUAAAAb
		AAAAKQAAAC8AAAA9AAAARwIAAMcBAAAMAAAAAwAAAAcAAAAVAAAAGwAAACkA
		AAAvAAAAPQAAAAkBAABLAQAARwIAANABAAAKAAAAFQAAABsAAAApAAAALwAA
		AD0AAAAJAQAASwEAAD4CAADWAQAABgAAAAwAAAAJAQAASwEAAD4CAADXAQAA
		BAAAAAwAAAA+AgAA2QEAAAIAAAD///9/
		</data>
		<key>NSTitle</key>
		<string>Untitled</string>
		<key>NSUIID</key>
		<string>_NS:34</string>
		<key>NSWindowCloseButtonFrame</key>
		<string>{{7, 454}, {14, 16}}</string>
		<key>NSWindowFrame</key>
		<string>177 501 586 476 0 0 1680 1025 </string>
		<key>NSWindowID</key>
		<integer>2</integer>
		<key>NSWindowLevel</key>
		<integer>0</integer>
		<key>NSWindowMiniaturizeButtonFrame</key>
		<string>{{27, 454}, {14, 16}}</string>
		<key>NSWindowNumber</key>
		<integer>5982</integer>
		<key>NSWindowWorkspaceID</key>
		<string></string>
		<key>NSWindowZoomButtonFrame</key>
		<string>{{47, 454}, {14, 16}}</string>
```

```
</dict>
<dict>
	<key>CFBundleVersion</key>
	<string>378</string>
	<key>NSDataKey</key>
	<data>
	P7BYxMryj6Gae9Q76wpqVw==
	</data>
	<key>NSDockMenu</key>
	<array>
		<dict>
			<key>command</key>
			<integer>1</integer>
			<key>mark</key>
			<integer>2</integer>
			<key>name</key>
			<string>Untitled</string>
			<key>system-icon</key>
			<integer>1735879022</integer>
			<key>tag</key>
			<integer>2</integer>
		</dict>
		<dict>
			<key>separator</key>
			<true/>
		</dict>
		<dict>
			<key>command</key>
			<integer>2</integer>
			<key>indent</key>
			<integer>0</integer>
			<key>name</key>
			<string>New Document</string>
			<key>tag</key>
			<integer>0</integer>
		</dict>
	</array>
	<key>NSExecutableInode</key>
	<integer>1152921500311961010</integer>
	<key>NSIsGlobal</key>
	<true/>
	<key>NSSystemAppearance</key>
	<data>
	YnBsaXN0MDDUAQIDBAUGBwpYJHZlcnNpb25ZJGFyY2hpdmVyVCR0b3BYJG9i
	amVjdHMSAAGGoF8QD05TS2V5ZWRBcmNoaXZlctEICVRyb290gAGkCwwRElUk
	bnVsbNINDg8QViRjbGFzc18QEE5TQXBwZWFyYW5jZU5hbWWAA4ACXxAUTlNB
	cHBlYXJhbmNlTmFtZUFxdWHSExQVFlokY2xhc3NuYW1lWCRjbGFzc2VzXE5T
	QXBwZWFyYW5jZaIVF1hOU09iamVjdAgRGiQpMjdJTFFTWF5jan1/gZidqLG+
	wQAAAAAAAAEBAAAAAAAAABgAAAAAAAAAAAAAAAAAAADK
	</data>
	<key>NSSystemVersion</key>
	<array>
```

```
                <integer>12</integer>
                <integer>2</integer>
                <integer>1</integer>
            </array>
            <key>NSWindowID</key>
            <integer>4294967295</integer>
            <key>NSWindowZOrder</key>
            <array>
                <integer>5982</integer>
            </array>
        </dict>
</array>
</plist>
```

The `data.data` file contains a custom binary format. It consists of a list of records, each record contains an AES-CBC encrypted serialized object. The `windows.plist` file contains the key (`NSDataKey`) and a ID (`NSWindowID`) for the record from `data.data` it corresponds to.[1]

For example:

```
00000000  4e 53 43 52 31 30 30 30  00 00 00 01 00 00 01 b0  |NSCR1000........|
00000010  ec f2 26 b9 8b 06 c8 d0  41 5d 73 7a 0e cc 59 74  |..&.....A]sz..Yt|
00000020  89 ac 3d b3 b6 7a ab 1b  bb f7 84 0c 05 57 4d 70  |..=..z.......WMp|
00000030  cb 55 7f ee 71 f8 8b bb  d4 fd b0 c6 28 14 78 23  |.U..q.......(.x#|
00000040  ed 89 30 29 92 8c 80 bf  47 75 28 50 d7 1c 9a 8a  |..0)....Gu(P....|
00000050  94 b4 d1 c1 5d 9e 1a e0  46 62 f5 16 76 f5 6f df  |....]...Fb..v.o.|
00000060  43 a5 fa 7a dd d3 2f 25  43 04 ba e2 7c 59 f9 e8  |C..z../%C...|Y..|
00000070  a4 0e 11 5d 8e 86 16 f0  c5 1d ac fb 5c 71 fd 9d  |...]........\q..|
00000080  81 90 c8 e7 2d 53 75 43  6d eb b6 aa c7 15 8b 1a  |....-SuCm.......|
00000090  9c 58 8f 19 02 1a 73 99  ed 66 d1 91 8a 84 32 7f  |.X....s..f....2.|
000000a0  1f 5a 1e e8 ae b3 39 a8  cf 6b 96 ef d8 7b d1 46  |.Z....9..k...{.F|
000000b0  0c e2 97 d5 db d4 9d eb  d6 13 05 7d e0 4a 89 a4  |...........}.J..|
000000c0  d0 aa 40 16 81 fc b9 a5  f5 88 2b 70 cd 1a 48 94  |..@.......+p..H.|
000000d0  47 3d 4f 92 76 3a ee 34  79 05 3f 5d 68 57 7d b0  |G=O.v:.4y.?]hW}.|
000000e0  54 6f 80 4e 5b 3d 53 2a  6d 35 a3 c9 6c 96 5f a5  |To.N[=S*m5..l._.|
000000f0  06 ec 4c d3 51 b9 15 b8  29 f0 25 48 2b 6a 74 9f  |..L.Q...).%H+jt.|
00000100  1a 5b 5e f1 14 db aa 8d  13 9c ef d6 f5 53 f1 49  |.[^..........S.I|
00000110  4d 78 5a 89 79 f8 bd 68  3f 51 a2 a4 04 ee d1 45  |MxZ.y..h?Q.....E|
00000120  65 ba c4 40 ad db e3 62  55 59 9a 29 46 2e 6c 07  |e..@...bUY.)F.l.|
00000130  34 68 e9 00 89 15 37 1c  ff c8 a5 d8 7c 8d b2 f0  |4h....7.....|...|
00000140  4b c3 26 f9 91 f8 c4 2d  12 4a 09 ba 26 1d 00 13  |K.&....-.J..&...|
00000150  65 ac e7 66 80 c0 e2 55  ec 9a 8e 09 cb 39 26 d4  |e..f...U.....9&.|
00000160  c8 15 94 d8 2c 8b fa 79  5f 62 18 39 f0 a5 df 0b  |....,..y_b.9....|
00000170  3d a4 5c bc 30 d5 2b cc  08 88 c8 49 d6 ab c0 e1  |=.\.0.+....I....|
00000180  c1 e5 41 eb 3e 2b 17 80  c4 01 64 3d 79 be 82 aa  |..A.>+....d=y...|
00000190  3d 56 8d bb e5 7a ea 89  0f 4c dc 16 03 e9 2a d8  |=V...z...L....*.|
000001a0  c5 3e 25 ed c2 4b 65 da  8a d9 0d d9 23 92 fd 06  |.>%..Ke.....#...|
[...]
```

Whenever an application is launched, AppKit will read these files and restore the windows of the application. This happens automatically, without the app needing to implement anything. The code for reading these files is quite careful: if the application crashed, then maybe the state is corrupted too. If the application crashes while restoring the state, then the next time the state is discarded and it does a fresh start.

The vulnerability we found is that the encrypted serialized object stored in the `data.data` file was *not* using "secure coding". To explain what that means, we'll first explain serialization vulnerabilities, in particular on macOS.

Serialized objects

Many object-oriented programming languages have added support for binary serialization, which turns an object into a bytestring and back. Contrary to XML and JSON, these are custom, language specific formats. In some programming languages, serialization support for classes is automatic, in other languages classes can opt-in.

In many of those languages these features have lead to vulnerabilities. The problem in many implementations is that an object is created first, and *then* its type is checked. Methods may be called on these objects when creating or destroying them. By combining objects in unusual ways, it is sometimes possible to gain remote code execution when a malicious object is deserialized. It is, therefore, not a good idea to use these serialization functions for any data that might be received over the network from an untrusted party.

For Python `pickle` and Ruby `Marshall.load` remote code execution is straightforward. In Java `ObjectInputStream.readObject` and C#, RCE is possible if certain commonly used libraries are used. The ysoserial and ysoserial.net tools can be used to generate a payload depending on the libraries in use. In PHP, exploitability for RCE is rare.

Objective-C serialization

In Objective-C, classes can implement the `NSCoding` protocol to be serializable. Subclasses of `NSCoder`, such as `NSKeyedArchiver` and `NSKeyedUnarchiver`, can be used to serialize and deserialize these objects.

How this works in practice is as follows. A class that implements `NSCoding` must include a method:

```
- (id)initWithCoder:(NSCoder *)coder;
```

In this method, this object can use `coder` to decode its instance variables, using methods such as `-decodeObjectForKey:`, `-decodeIntegerForKey:`, `-decodeDoubleForKey:`, etc. When it uses `-decodeObjectForKey:`, the coder will recursively call `-initWithCoder:` on that object, eventually decoding the entire graph of objects.

Apple has also realized the risk of deserializing untrusted input, so in 10.8, the `NSSecureCoding` protocol was added. The documentation for this protocol states:

> A protocol that enables encoding and decoding in a manner that is robust against object substitution attacks.

This means that instead of creating an object first and then checking its type, a set of allowed classes needs to be included when decoding an object.

So instead of the unsafe construction:

```
id obj = [decoder decodeObjectForKey:@"myKey"];
if (![obj isKindOfClass:[MyClass class]]) { /* ...fail... */ }
```

The following must be used:

```
id obj = [decoder decodeObjectOfClass:[MyClass class] forKey:@"myKey"];
```

This means that when a secure coder is created, `-decodeObjectForKey:` is no longer allowed, but `-decodeObjectOfClass:forKey:` must be used.

That makes exploitable vulnerabilities significantly harder, but it could still happen. One thing to note here is that subclasses of the specified class are allowed. If, for example, the `NSObject` class is specified, then all classes implementing `NSCoding` are still allowed. If only `NSDictionary` are expected and an imported framework contains a rarely used and vulnerable subclass of `NSDictionary`, then this could also create a vulnerability.

In all of Apple's operating systems, these serialized objects are used all over the place, often for inter-process exchange of data. For example, `NSXPCConnection` heavily relies on secure serialization for implementing remote method calls. In iMessage, these serialized objects are even exchanged with other users over the network. In such cases it is very important that secure coding is always enabled.

Creating a malicious serialized object

In the `data.data` file for saved states, objects were stored using an `NSKeyedArchiver` without secure coding enabled. This means we could include objects of any class that implements the `NSCoding` protocol. The likely reason for this is that applications can extend the saved state with their own objects, and because the saved state functionality is older than `NSSecureCoding`, Apple couldn't just upgrade this to secure coding, as this could break third-party applications.

To exploit this, we wanted a method for constructing a chain of objects that could allows us to execute arbitrary code. However, no project similar to ysoserial for Objective-C appears to exist and we could not find other examples of abusing insecure deserialization in macOS. In

Remote iPhone Exploitation Part 1: Poking Memory via iMessage and CVE-2019-8641
Samuel Groß of Google Project Zero describes an attack against a *secure* coder by abusing a vulnerability in `NSSharedKeyDictionary`, an uncommon subclass of `NSDictionary`. As this vulnerability is now fixed, we couldn't use this.

By decompiling a large number of `-initWithCoder:` methods in AppKit, we eventually found a combination of 2 objects that we could use to call arbitrary Objective-C methods on another deserialized object.

We start with `NSRuleEditor`. The `-initWithCoder:` method of this class creates a binding to an object from the same archive with a key path also obtained from the archive.

Bindings are a reactive programming technique in Cocoa. It makes it possible to directly bind a model to a view, without the need for the boilerplate code of a controller. Whenever a value in the model changes, or the user makes a change in the view, the changes are automatically propagated.

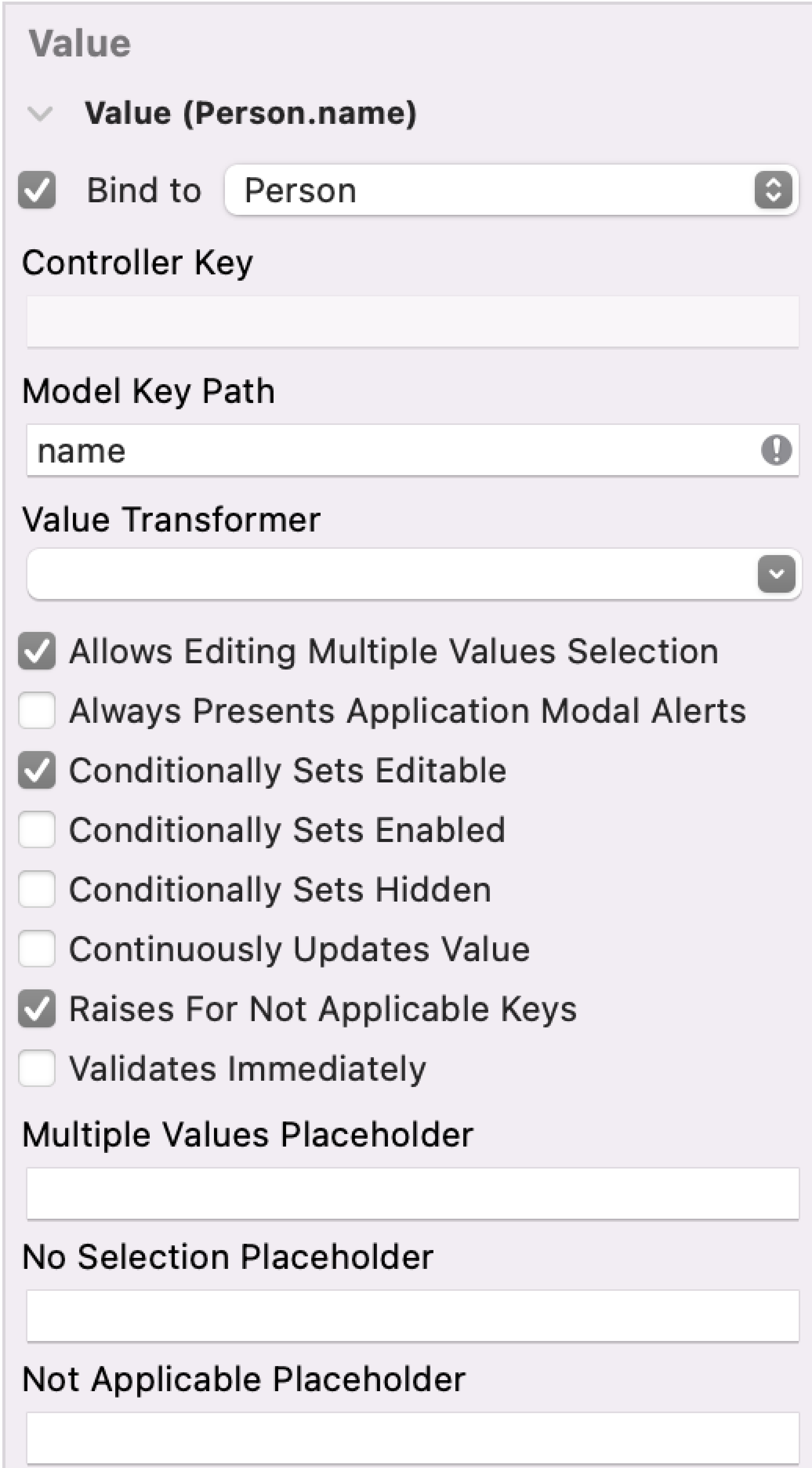
A binding is created calling the method:

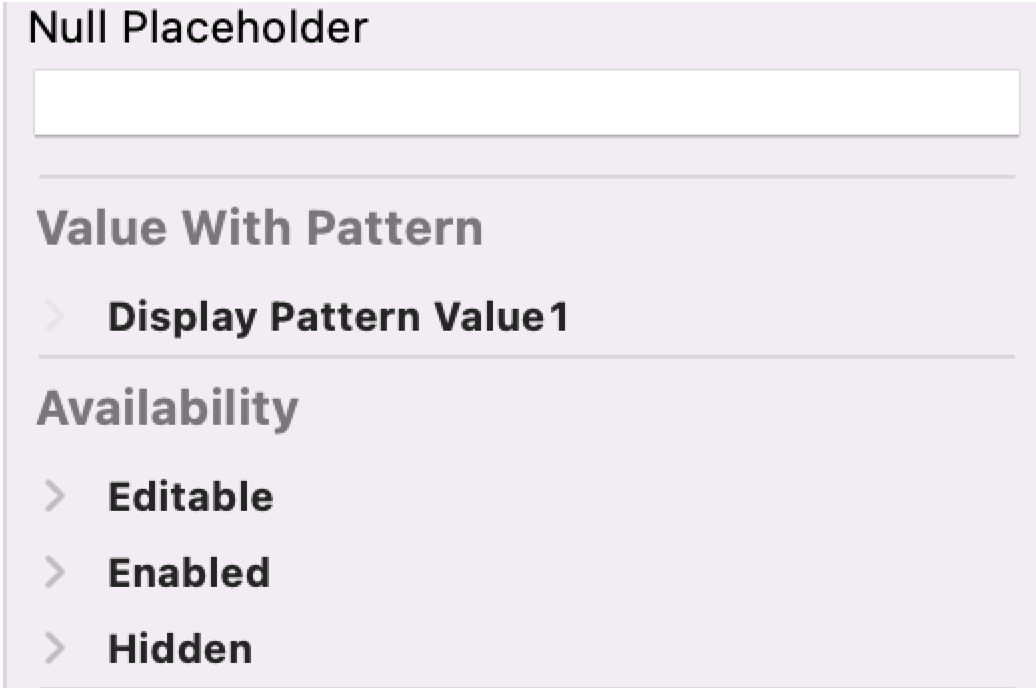
```
- (void)bind:(NSBindingName)binding
    toObject:(id)observable
 withKeyPath:(NSString *)keyPath
     options:(NSDictionary<NSBindingOption, id> *)options;
```

This binds the property `binding` of the receiver to the `keyPath` of `observable`. A *keypath* a string that can be used, for example, to access nested properties of the object. But the more common method for creating bindings is by creating them as part of a XIB file in Xcode.

For example, suppose the model is a class `Person`, which has a property `@property (readwrite, copy) NSString *name;`. Then you could bind the "value" of a text field to the "name" keypath of a Person to create a field that shows (and can edit) the person's name.

In the XIB editor, this would be created as follows:


Value
Value (Person.name)
Bind to
Person
Controller Key
Model Key Path
name
Value Transformer
Allows Editing Multiple Values Selection
Always Presents Application Modal Alerts
Conditionally Sets Editable
Conditionally Sets Enabled
Conditionally Sets Hidden
Continuously Updates Value
Raises For Not Applicable Keys
Validates Immediately
Multiple Values Placeholder
No Selection Placeholder
Not Applicable Placeholder



The different options for what a keypath can mean are actually quite complicated. For example, when binding with a keypath of “foo”, it would first check if one the methods `getFoo`, `foo`, `isFoo` and `_foo` exists. This would usually be used to access a property of the object, but this is not required. When a binding is created, the method will be called immediately when creating the binding, to provide an initial value. It does not matter if that method actually returns void. This means that by creating a binding during deserialization, we can use this to call zero-argument methods on other deserialized objects!

```
ID NSRuleEditor::initWithCoder:(ID param_1,SEL param_2,ID unarchiver)
{
        ...

        id arrayOwner = [unarchiver
decodeObjectForKey:@"NSRuleEditorBoundArrayOwner"];


        ...

        if (arrayOwner) {
          keyPath = [unarchiver decodeObjectForKey:@"NSRuleEditorBoundArrayKeyPath"];
          [self bind:@"rows" toObject:arrayOwner withKeyPath:keyPath options:nil];
        }

        ...
}
```

In this case we use it to call `-draw` on the next object.

The next object we use is an `NSCustomImageRep` object. This obtains a selector (a method name) as a string and an object from the archive. When the `-draw` method is called, it invokes the method from the selector on the object. It passes itself as the first argument:

```
ID NSCustomImageRep::initWithCoder:(ID param_1,SEL param_2,ID unarchiver)
{
        ...
        id drawObject = [unarchiver decodeObjectForKey:@"NSDrawObject"];
        self.drawObject = drawObject;
        id drawMethod = [unarchiver decodeObjectForKey:@"NSDrawMethod"];
        SEL selector = NSSelectorFromString(drawMethod);
        self.drawMethod = selector;
        ...
}

...

void ___24-[NSCustomImageRep_draw]_block_invoke(long param_1)
{
  ...
  [self.drawObject performSelector:self.drawMethod withObject:self];
  ...
}
```

By deserializing these two classes we can now call zero-argument methods and multiple argument methods, although the first argument will be an `NSCustomImageRep` object and the remaining arguments will be whatever happens to still be in those registers. Nevertheless, is a very powerful primitive. We'll cover the rest of the chain we used in a future blog post.

Exploitation

Sandbox escape

First of all, we escaped the Mac Application sandbox with this vulnerability. To explain that, some more background on the saved state is necessary.

In a sandboxed application, many files that would be stored in `~/Library` are stored in a separate container instead. So instead of saving its state in:

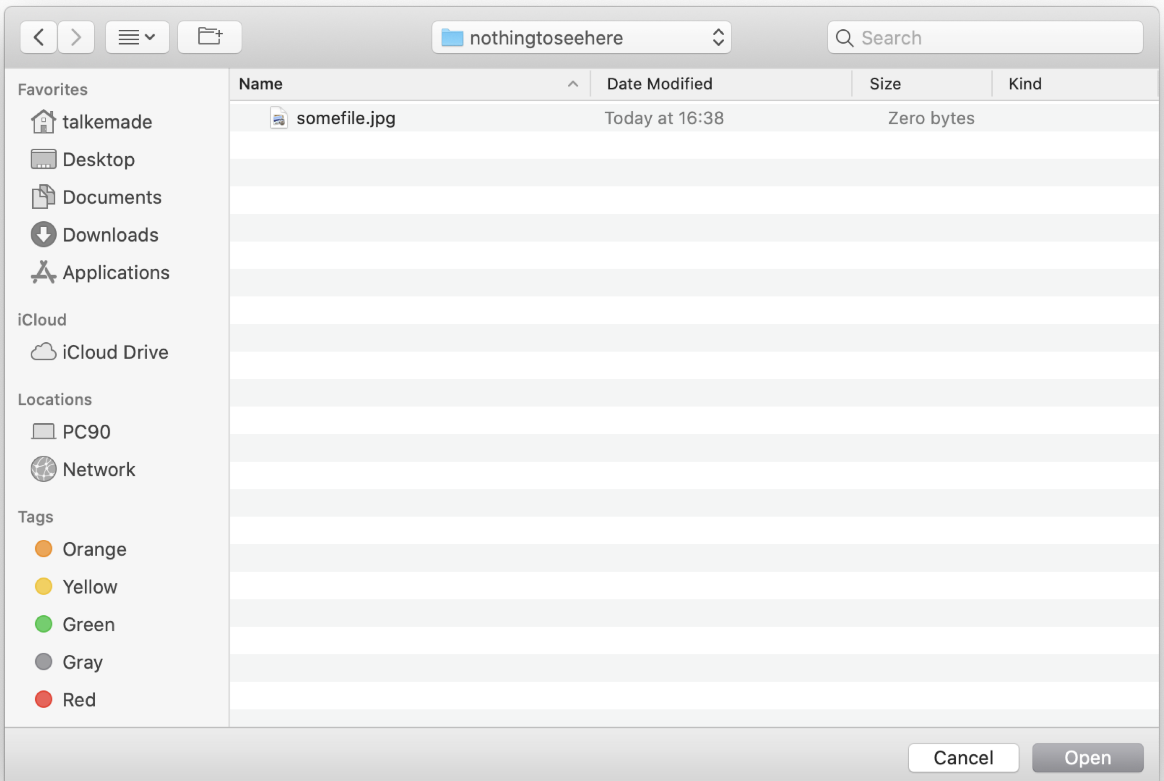
```
~/Library/Saved Application State/<Bundle ID>.savedState/
```

Sandboxed applications save their state to:

```
~/Library/Containers/<Bundle ID>/Data/Library/Saved Application State/<Bundle
ID>.savedState/
```

Apparently, when the system is shut down while an application is still running (when the prompt is shown asking the user whether to reopen the windows the next time), the first location is symlinked to the second one by `talagent`. We are unsure of why, it might have something to do with upgrading an application to a new version which is sandboxed.

Secondly, most applications do not have access to all files. Sandboxed applications are very restricted of course, but with the addition of TCC even accessing the Downloads, Documents, etc. folders require user approval. If the application would open an open or save panel, it would be quite inconvenient if the user could only see the files that that application has access to. To solve this, a different process is launched when opening such a panel: `com.apple.appkit.xpc.openAndSavePanelService`. Even though the window itself is part of the application, its contents are drawn by openAndSavePanelService. This is an XPC service which has full access to all files. When the user selects a file in the panel, the application gains temporary access to that file. This way, users can still browse their entire disk even in applications that do not have permission to list those files.



As it is an XPC service with service type Application, it is launched separately for each app.

What we noticed is that this XPC Service reads its saved state, but using the bundle ID of the app that launched it! As this panel might be part of the saved state of multiple applications, it does make some sense that it would need to separate its state per application.

As it turns out, it reads its saved state from the location *outside* of the container, but with the application's bundle ID:

`~/Library/Saved Application State/<Bundle ID>.savedState/`

But as we mentioned if the app was ever open when the user shut down their computer, then this will be a symlink to the container path.

Thus, we can escape the sandbox in the following way:

1. Wait for the user to shut down while the app is open, if the symlink does not yet exist.
2. Write malicious `data.data` and `windows.plist` files inside the app's own container.
3. Open an `NSOpenPanel` or `NSSavePanel`.

The `com.apple.appkit.xpc.openAndSavePanelService` process will now deserialize the malicious object, giving us code execution in a non-sandboxed process.

This was fixed earlier than the other issues, as CVE-2021-30659 in macOS 11.3. Apple addressed this by no longer loading the state from the same location in `com.apple.appkit.xpc.openAndSavePanelService`.

Privilege escalation

By injecting our code into an application with a specific entitlement, we can elevate our privileges to root. For this, we could apply the technique explained by A2nkF in Unauthd - Logic bugs FTW.

Some applications have an entitlement of `com.apple.private.AuthorizationServices` containing the value `system.install.apple-software`. This means that this application is allowed to install packages that have a signature generated by Apple without authorization from the user. For example, "Install Command Line Developer Tools.app" and "Bootcamp Assistant.app" have this entitlement. A2nkF also found a package signed by Apple that contains a vulnerability: `macOSPublicBetaAccessUtility.pkg`. When this package is installed to a specific disk, it will run (as root) a post-install script from that disk. The script assumes it is being installed to a disk containing macOS, but this is not checked. Therefore, by creating a malicious script at the same location it is possible to execute code as root by installing this package.

The exploitation steps are as follows:

1. Create a RAM disk and copy a malicious script to the path that will be executed by `macOSPublicBetaAccessUtility.pkg`.
2. Inject our code into an application with the `com.apple.private.AuthorizationServices` entitlement containing `system.install.apple-software` by creating the `windows.plist` and `data.data` files for that application and then launching it.

3. Use the injected code to install the `macOSPublicBetaAccessUtility.pkg` package to the RAM disk.
4. Wait for the post-install script to run.

In the writeup from A2nkF, the post-install script ran without the filesystem restrictions of SIP. It inherited this from the installation process, which needs it as package installation might need to write to SIP protected locations. This was fixed by Apple: post- and pre-install scripts are no longer SIP exempt. The package and its privilege escalation can still be used, however, as Apple still uses the same vulnerable installer package.

SIP filesystem bypass

Now that we have escaped the sandbox and elevated our privileges to root, we did want to bypass SIP as well. To do this, we looked around at all available applications to find one with a suitable entitlement. Eventually, we found something on the macOS Big Sur Beta installation disk image: “macOS Update Assistant.app” has the `com.apple.rootless.install.heritable` entitlement. This means that this process can write to all SIP protected locations (and it is heritable, which is convenient because we can just spawn a shell). Although it is supposed to be used only during the beta installation, we can just copy it to a normal macOS environment and run it there.

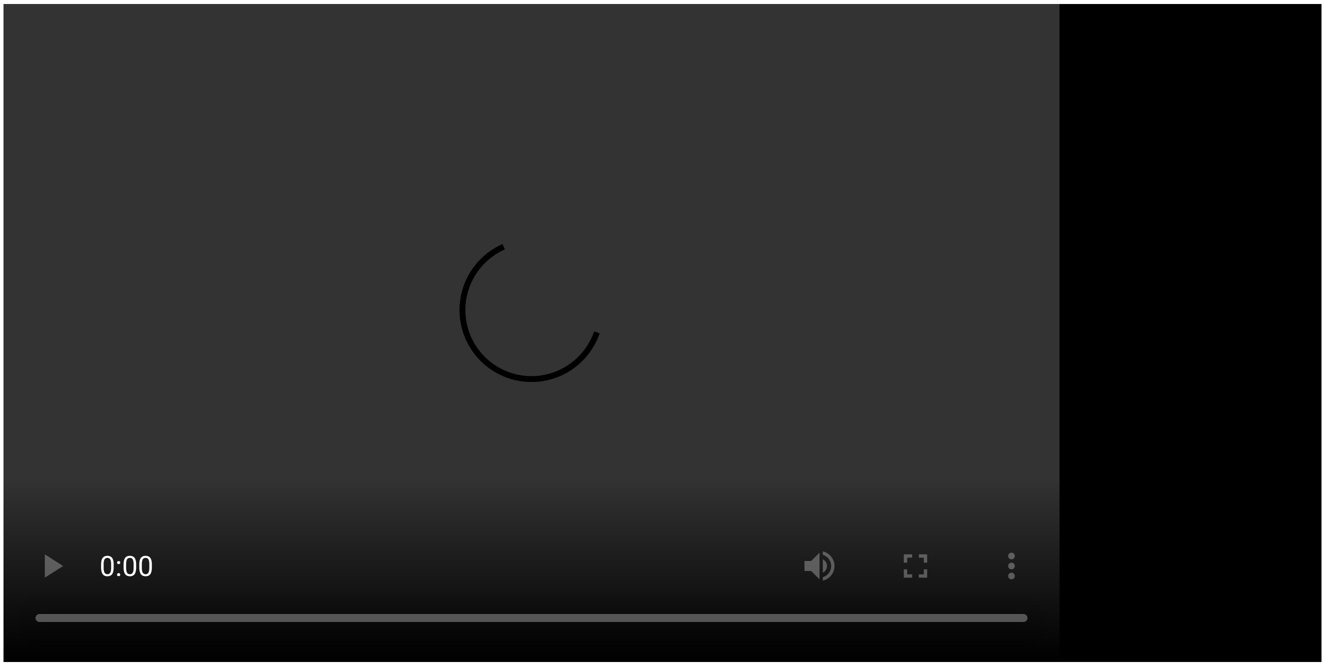
The exploitation for this is quite simple:

1. Create malicious `windows.plist` and `data.data` files for “macOS Update Assistant.app”.
2. Launch “macOS Update Assistant.app”.

When exempt from SIP’s filesystem restrictions, we can read all files from protected locations, such as the user’s Mail.app mailbox. We can also modify the TCC database, which means we can grant ourselves permission to access the webcam, microphone, etc. We could also persist our malware on locations which are protected by SIP, making it very difficult to remove by anyone other than Apple. Finally, we can change the database of approved kernel extensions. This means that we could load a new kernel extension silently, without user approval. When combined with a vulnerable kernel extension (or a codesigning certificate that allows signing kernel extensions), we would have been able to gain kernel code execution, which would allow disabling all other restrictions too.

Demo

We recorded the following video to demonstrate the different steps. It first shows that the application “Sandbox” is sandboxed, then it escapes its sandbox and launches “Privesc”. This elevates privileges to root and launches “SIP Bypass”. Finally, this opens a reverse shell that is exempt from SIP's filesystem restrictions, which is demonstrated by writing a file in `/var/db/SystemPolicyConfiguration` (the location where the database of approved kernel modules is stored):



The fix

Apple first fixed the sandbox escape in 11.3, by no longer reading the saved state of the application in `com.apple.appkit.xpc.openAndSavePanelService` (CVE-2021-30659).

Fixing the rest of the vulnerability was more complicated. Third-party applications may store their own objects in the saved state and these objects might not support secure coding. This brings us back to the method from the introduction: `-applicationSupportsSecureRestorableState:`. Applications can now opt-in to requiring secure coding for their saved state by returning `TRUE` from this method. Unless an app opts in, it will keep allowing non-secure coding, which means process injection might remain possible.

This does highlight one issue with the current design of these security measures: downgrade attacks. The code signature (and therefore entitlements) of an application will remain valid for a long time, and the TCC permissions of an application will still work if the application is downgraded. A non-sandboxed application could just silently download an older, vulnerable version of an application and exploit that. For the SIP bypass this would not work, as "macOS Update Assistant.app" does not run on macOS Monterey because certain private frameworks no longer contain the necessary symbols. But that is a coincidental fix, in many other cases older applications may still run fine. This vulnerability will therefore be present for as long as there is backwards compatibility with older macOS applications!

Nevertheless, if you write an Objective-C application, please make sure you add `-applicationSupportsSecureRestorableState:` to return `TRUE` and to adapt secure coding for all classes used for your saved states!

Conclusion

In the current security architecture of macOS, process injection is a powerful technique. A generic process injection vulnerability can be used to escape the sandbox, elevate privileges to root and to bypass SIP's filesystem restrictions. We have demonstrated how we used the use of insecure deserialization in the loading of an application's saved state to inject into any Cocoa process. This was addressed by Apple as CVE-2021-30873.

Back