

Learn XPC exploitation - Part 3: Code injections

 wojciechregula.blog/post/learn-xpc-exploitation-part-3-code-injections/

@Wojciech Reguła · Jun 29, 2020 · 5 min read

XPC Exploitation series

Learn XPC exploitation - Part 1: Broken cryptography

Learn XPC exploitation - Part 2: Say no to the PID!

Learn XPC exploitation - Part 3: Code injections

Intro

The last technique I showed in my presentation during Objective by the Sea v3 conference was abusing privileged XPC services using different code injections. In many apps I exploited, I observed that developers are aware that privileged XPC services have to verify incoming connections. Devs usually perform the validation using Apple's cryptographic APIs - what is good 👍, but... The problem is that these APIs cannot easily detect if there was any malicious code injected to the process. In this post, I'll show you techniques that may help you exploit a bit more secured XPC helpers. 😊

MacOS != Linux



Before I start the exploitation, I have to mention one more thing. macOS is not Linux. The `ptrace` syscall is not as powerful as it is in Linux. You cannot write data to the victim's process memory using the `ptrace` on macOS. For such purpose you should: **1.** get the task of the victim's app (using for instance the `task_for_pid()` function); **2.** use the `mach_vm_write` function. I used that technique when I was exploiting iExplorer. You can read more about that here.

Can I inject to other processes on macOS?

Typically if you want to use the above-mentioned `task_for_pid()` function to retrieve old/poorly secured apps' tasks, you need to have root permissions. However, there is an exception - if the application possesses a `com.apple.security.get-task-allow` entitlement set to `true`, you do not need the root permissions. This problematic entitlement is usually used for debugging purposes. For example, XCode signs apps with this entitlement because it attaches the lldb. Always look for that entitlement during audits. I exploited a few apps because of that left entitlement.

But why I wrote that you do need root to take over the old/poorly secured apps? Well, in the modern/well-secured apps, the Hardened Runtime capability is turned on. **If the application was signed with the hardened runtime flag, even if you have root permissions, you are unable to retrieve the app's task.** If you want to debug a hardened app (without the get-task-allow entitlement), you should disable the System Integrity Protection. The hardened runtime feature gives macOS some cross-app isolation.

In all new projects, the hardened runtime flag is turned on by default. In the old projects, it was enforced by Apple in the macOS Catalina. All apps that are downloaded from the Internet (outside of the App Store) are quarantined and thus need to be notarized.

Gatekeeper macOS Catalina		NEW
		
	First use, quarantined	First use, quarantined
Malicious content scan	No known malicious content	No known malicious content
Signature check	No tampering	No tampering
Local policy check	All new software requires notarization	All new software requires notarization
First launch prompt	User must approve	Users must approve software in bundles

The notarization enforces not only the `hardened runtime` but also lack of the `com.apple.security.get-task-allow` entitlement:

Prepare Your Software for Notarization

Notarization requires Xcode 10 or later. Building a new app for notarization requires macOS 10.13.6 or later. Stapling an app requires macOS 10.12 or later.

Apple's notary service requires you to adopt the following protections:

- Enable code-signing for all of the executables you distribute.
- Enable the Hardened Runtime capability for your app and command line targets, as described in [Enable hardened runtime](#).
- Use a "Developer ID" application, kernel extension, or installer certificate for your code-signing signature. (Don't use a Mac Distribution or local development certificate.) For more information, see [Create, export, and delete signing certificates](#).
- Include a secure timestamp with your code-signing signature. (The Xcode distribution workflow includes a secure timestamp by default. For custom workflows, include the `--timestamp` option when running the `codesign` tool.)
- Don't include the `com.apple.security.get-task-allow` entitlement with the value set to any variation of `true`. If your software hosts third-party plug-ins and needs this entitlement to debug the plug-in in the context of a host executable, see [Avoid the Get-Task-Allow Entitlement](#).
- Link against the macOS 10.9 or later SDK.

How can I verify if the app has the hardened runtime turned on?

Use the following command:

```
$ codesign -d -vv LuLu.app
Executable=./LuLu.app/Contents/MacOS/LuLu
Identifier=com.objective-see.lulu
Format=app bundle with Mach-O thin (x86_64)
CodeDirectory v=20500 size=1538 flags=0x12300(hard,kill,library-validation,runtime) hashes=39+5
location=embedded
Signature size=8974
Authority=Developer ID Application: Objective-See, LLC (VBG97UB4TA)
Authority=Developer ID Certification Authority
Authority=Apple Root CA
Timestamp=11 Dec 2019 at 23:49:45
Info.plist entries=24
TeamIdentifier=VBG97UB4TA
Runtime Version=10.14.0
Sealed Resources version=2 rules=13 files=12
Internal requirements count=1 size=216
```

You can see that there is a `runtime` flag set.

How can I verify if the app has `get-task-allow` entitlement set?

Also with the `codesign` command:

```
$ codesign -d --entitlements :- GetTaskAllowTest.app
Executable=./GetTaskAllowTest.app/Contents/MacOS/GetTaskAllowTest
<?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">
<dict>
  <key>com.apple.security.app-sandbox</key>
  <true/>
  <key>com.apple.security.files.user-selected.read-only</key>
  <true/>
  <key>com.apple.security.get-task-allow</key>
  <true/>
</dict>
</plist>
```

Code injection with the `DYLD_INSERT_LIBRARIES`

When the process doesn't have the `hardened runtime` capability turned on and the `com.apple.security.get-task-allow` entitlements are not set, the easiest way to inject your code is to use the `DYLD_INSERT_LIBRARIES` environment variable. There is a great article explaining how it works in details.

To simply exploit such an application, create an `exploit.m` file with the following content:

```
#import <Foundation/Foundation.h>

__attribute__((constructor)) static void pwn(int argc, const char **argv) {
    NSLog(@"[+] Dylib injected");
}
```

Compile it:

```
gcc -dynamiclib exploit.m -o exploit.dylib -framework Foundation
```

Inject to a vulnerable app:

```
DYLD_INSERT_LIBRARIES=exploit.dylib ./App.app/Contents/MacOS/App
```

What if the hardened runtime is turned on? Well, there is an entitlement that can loose the restrictions and allow the `DYLD_INSERT_LIBRARIES - com.apple.security.cs.allow-dyld-environment-variables`. But, the runtime will verify if the loaded dylib is signed with the same certificate. In order to be able to inject to a hardened application, it has to possess also the `com.apple.security.cs.disable-library-validation` entitlement.

Summing up the code injection techniques

Exploiting privileged XPC services, that cryptographically validate incoming connections without the code injection checks, can be done with following tricks:

- If the application doesn't have the hardened runtime turned on -> try `DYLD_INSERT_LIBRARIES`
- If the application has the hardened runtime turned on:
 - look also for the `com.apple.security.get-task-allow` entitlement. If it's set to inject via `task_for_pid`
 - look also for the pair of `com.apple.security.cs.allow-dyld-environment-variables` & `com.apple.security.cs.disable-library-validation` entitlements. If they are both set to true use the `DYLD_INSERT_LIBRARIES`
 - find an older version of the application without the hardened runtime and use the `DYLD_INSERT_LIBRARIES`
 - find another app signed by the same developer without the hardened runtime and use the `DYLD_INSERT_LIBRARIES`

Exploitation example

An example of an application I was able to exploit using the code injection technique is LuLu - an open-source firewall. While it had the hardened runtime turned on and was verifying the incoming XPC connections cryptographically - it wasn't detecting code injections. So, I took KextViewr (another Objective-See's application) that was signed with the same developer certificate but didn't have the hardened runtime. LuLu's validation was verifying if the incoming XPC connection was initiated by a process that was signed

with a defined certificate. So, I injected a dylib using the `DYLD_INSERT_LIBRARIES` technique to the KextViewr and established a valid connection with the privileged XPC helper. It allowed me to bypass the firewall by setting my own rule without root permissions:

The fix

LuLu mitigated the vulnerability by also verifying if the bundle identifier belongs to LuLu and if the minimum version is 1.2.0 (in that version, the hardened runtime was turned on).

```
    .↑  @@ -120,7 +120,7 @@ -(BOOL)listener:(NSXPCListener *)listener shouldAcceptNewConnection:(NSXPCConnec
120  120      NSString *requirementString = nil;
121  121
122  122      //init signing req string
123  -      requirementString = [NSString stringWithFormat:@"anchor trusted and certificate leaf [subject.CN] = \"%@\"", SIGNING_AUTH];
123  +      requirementString = [NSString stringWithFormat:@"anchor trusted and identifier \"%@" and certificate leaf [subject.CN] = \"%@" and info [CFBundleShortVersionString] >= \"1.2.0\"", INSTALLER_ID, SIGNING_AUTH];
124  124
```

BTW I wrote an open-source privileged XPC helper that may help you with securing & abusing XPC apps.

😁 You can find it here.

© Wojciech Reguła