Bypassing macOS Security & Privacy Mechanisms: From Gatekeeper to System Integrity Protection

FFRI Security, Inc.
https://www.ffri.jp

$ whoami – Koh M. Nakagawa

𝕏 @tsunek0h

- Security Researcher at FFRI Security, Inc.
- Mainly conducting vulnerability research on macOS
- Earned more than $50k in various bug bounty programs since last year
- One of MSRC Most Valuable Security Researchers 2023
- Gave talks at Black Hat Briefings (EU 2020 and Asia 2023) & CODE BLUE (2021)

Fundamental concepts of macOS security

System Integrity Protection (SIP) (a.k.a. rootless)

Introduced from OS X El Captain

Restricts some dangerous operations, such as …

- Modifying system files (e.g., files of the /bin directory)
- Loading untrusted kernel extensions
- Debugging system processes

Even the root user cannot perform these dangerous operations, unlike in a traditional *NIX security model.

Fundamental concepts of macOS security

Code signature & entitlements

macOS security & privacy mechanisms heavily rely on code signature & entitlements.

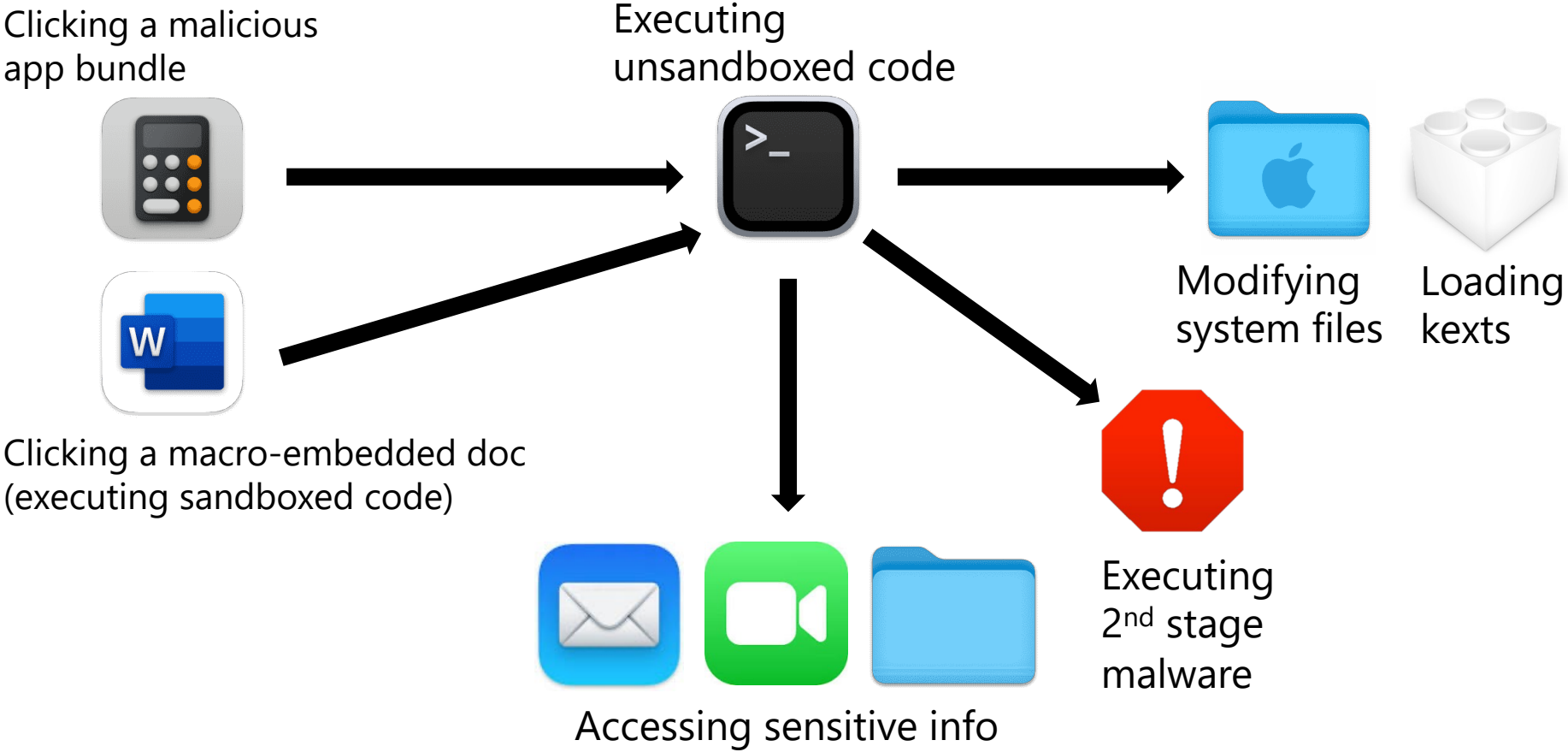
- "An entitlement is a right or privilege that grants an executable particular capabilities."

Some operations are not permitted without proper entitlements.

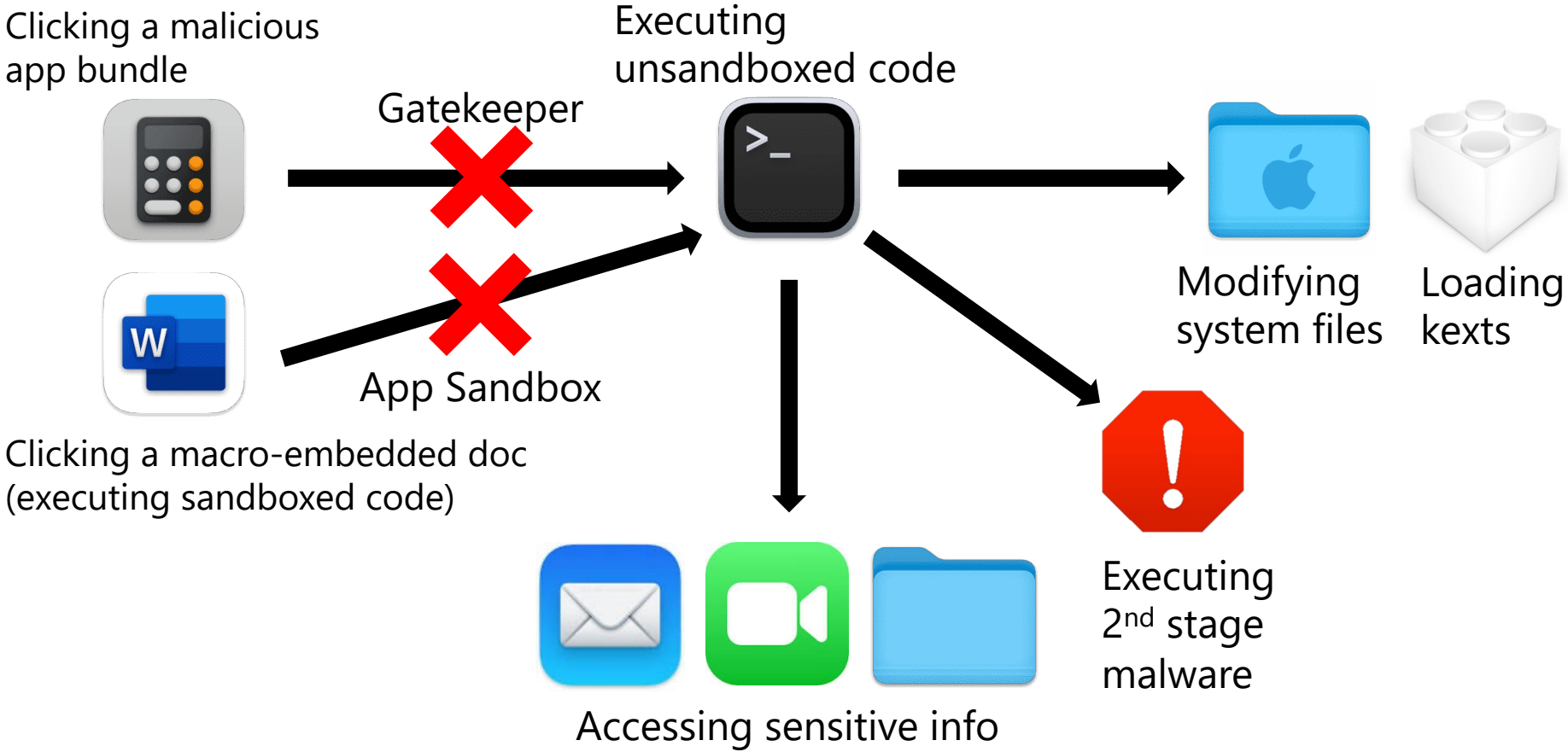
- Example: Only Apple binaries with proper private entitlements can modify SIP-protected files.

```
sh-3.2$ codesign -dv --entitlements - /usr/libexec/rootless-init
Executable=/usr/libexec/rootless-init
Identifier=com.apple.rootless-init
Format=Mach-O universal (x86_64 arm64e)
CodeDirectory v=20400 size=624 flags=0x0(none) hashes=9+7 location=embedded
Platform identifier=14
Signature size=4442
Signed Time=Apr 24, 2023 12:32:43
Info.plist=not bound
TeamIdentifier=not set
Sealed Resources=none
Internal requirements count=1 size=72
[Dict]
        [Key] com.apple.private.apfs.set-firmlink
        [Value]
                [Bool] true
        [Key] com.apple.rootless.install
        [Value]
                [Bool] true
```

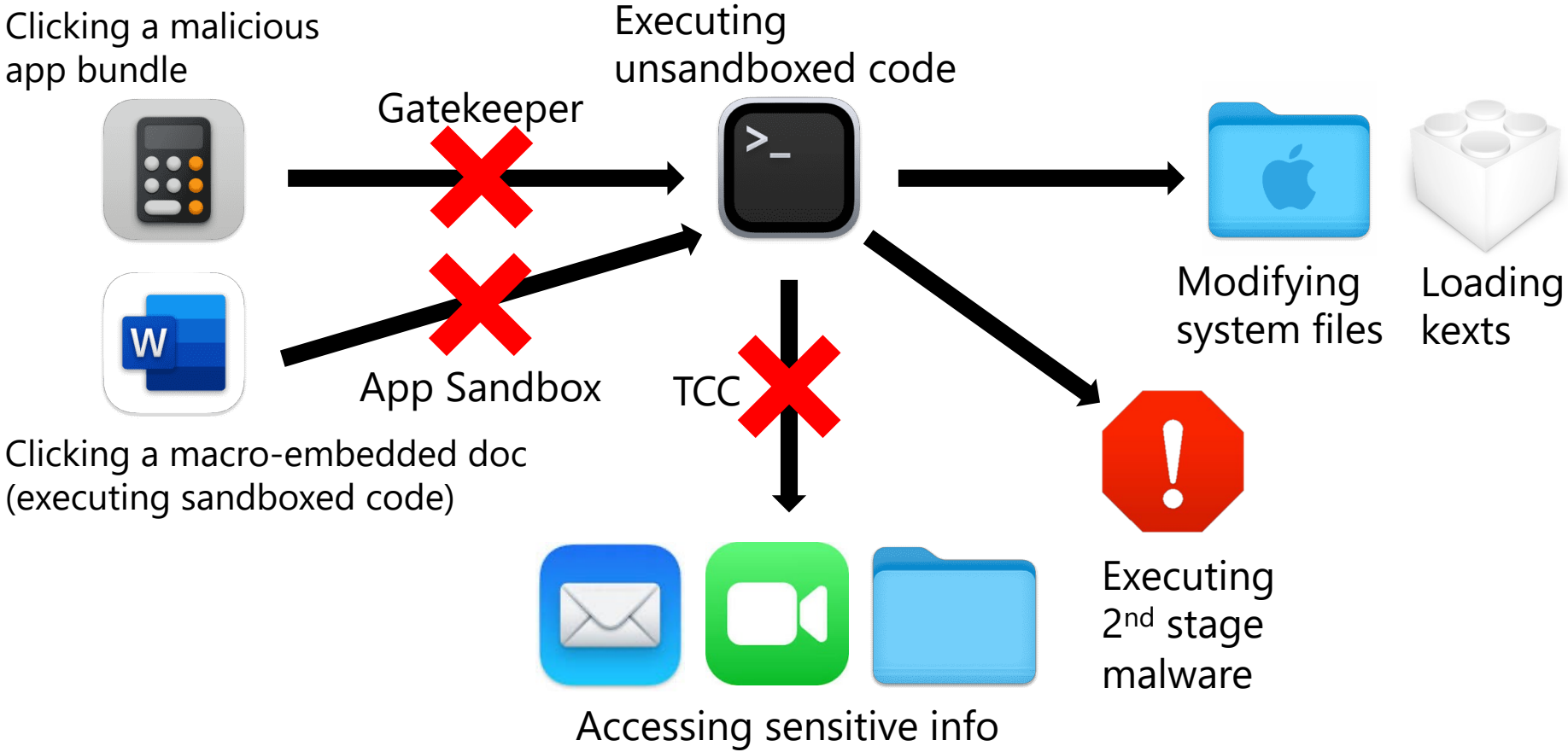
Overview of macOS security & privacy mechanisms



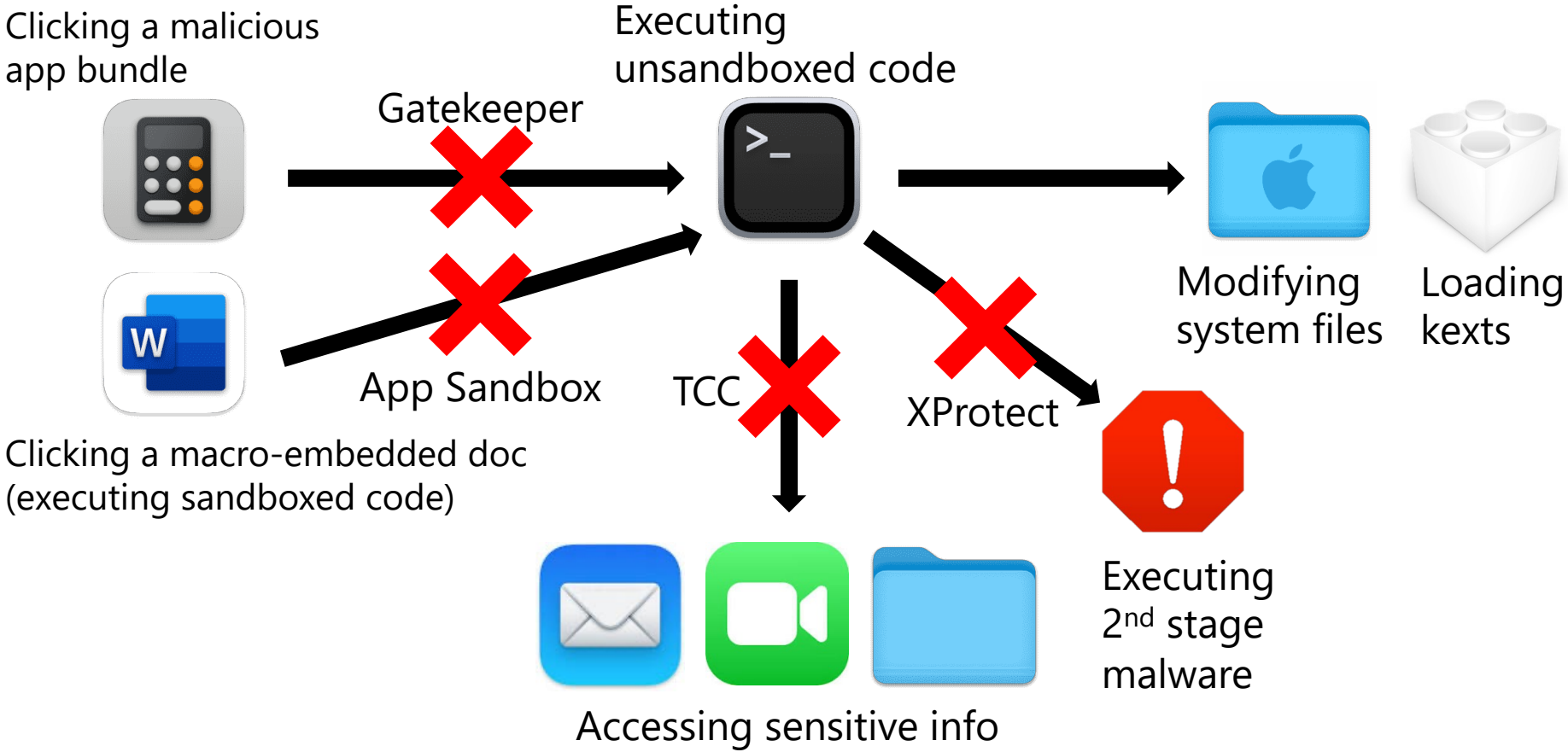
Overview of macOS security & privacy mechanisms



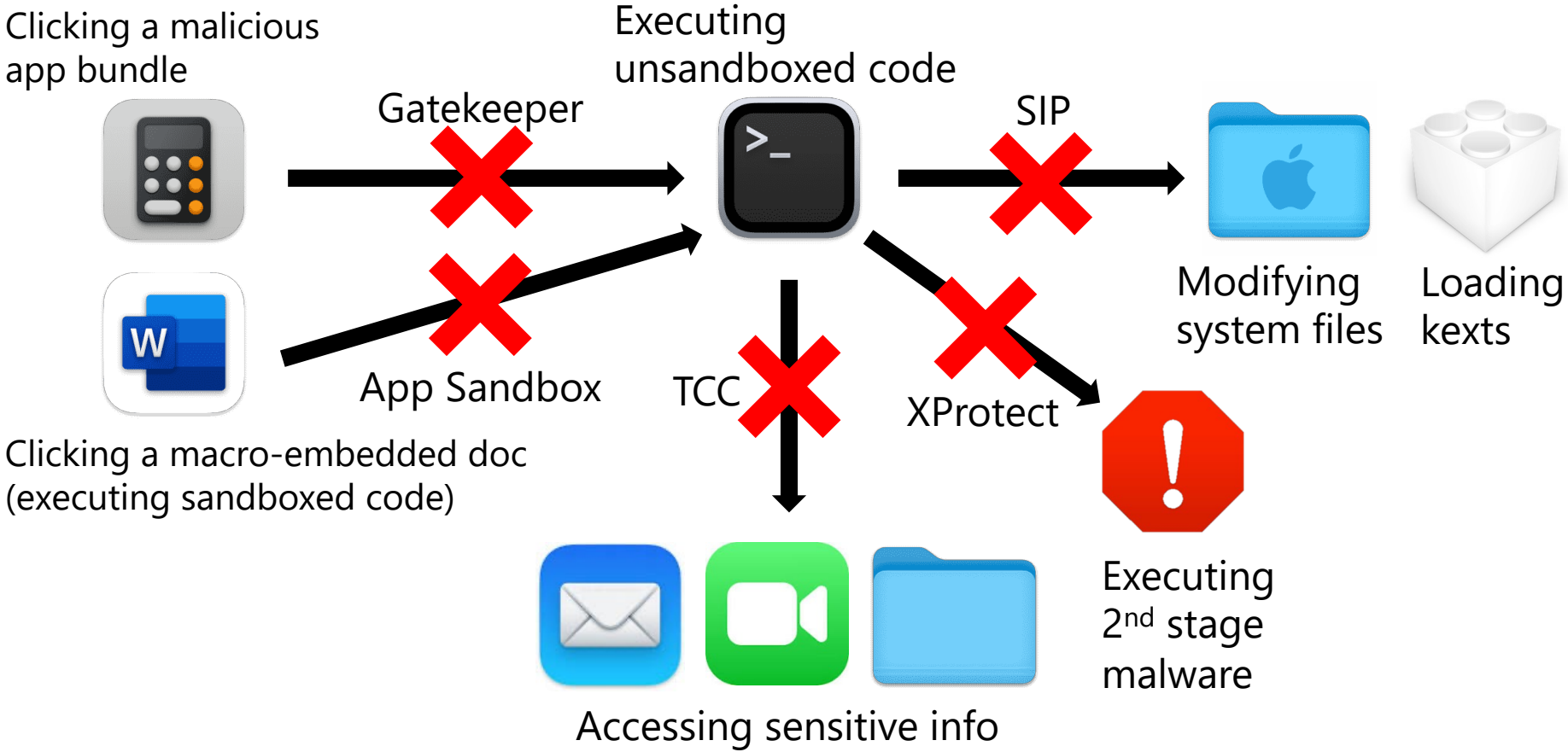
Overview of macOS security & privacy mechanisms



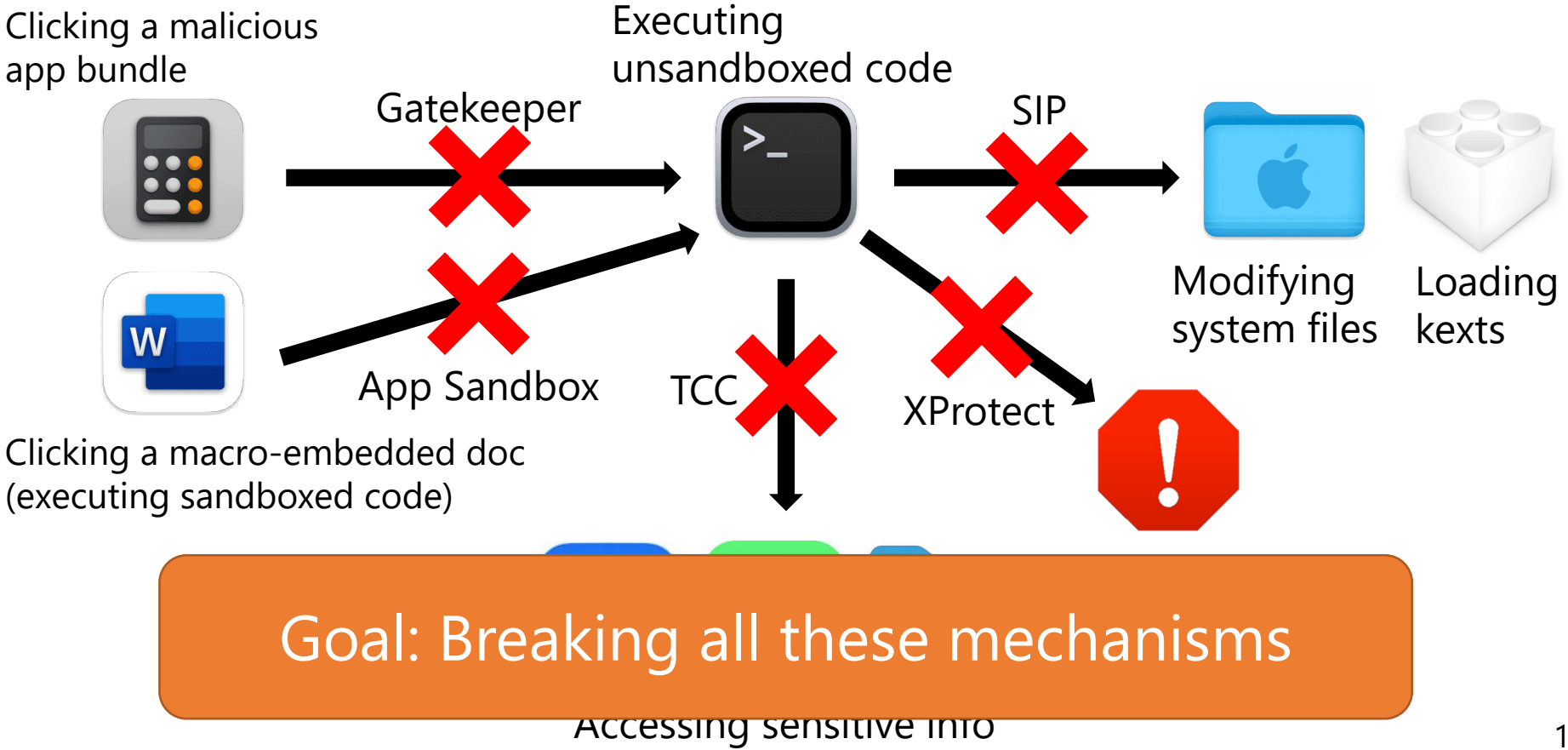
Overview of macOS security & privacy mechanisms



Overview of macOS security & privacy mechanisms



Overview of macOS security & privacy mechanisms



Goal: Breaking all these mechanisms

Gatekeeper bypass

What is Gatekeeper?

Apple Platform Security

macOS includes a security technology called ***Gatekeeper, which is designed to help ensure that only trusted software runs on a user's Mac.***

https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf

What is Gatekeeper?

For an app on the App Store, Apple reviews each app and signs it to make sure that it has not been tampered with or altered.

Gatekeeper verifies the app has been signed by the App Store.

For an app not on the App Store, Gatekeeper verifies the following:

- The app is from an identified developer (by checking the code signature).
- The app has not been altered.
- The app is “notarized” by Apple.

What is “notarized”?

Notarization

"Notarization is a malware scanning service provided by Apple."

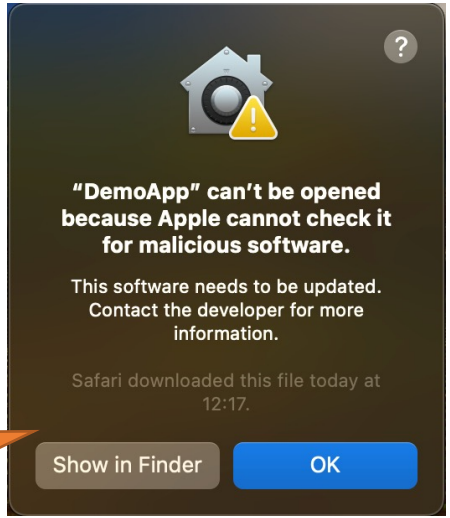
An app approved by the Notarization service is "notarized."

- This app is regarded to be free of malicious content by Apple.

App developers should submit their app to Notarization before distributing it.

- A ticket is awarded once the app is approved.
- Gatekeeper verifies the app based on the awarded ticket.
- App developers can optionally staple the ticket to the app.
  - ✓ This enables Gatekeeper to verify the app even if the user is offline.

There is no "ignore" option. A user cannot execute this app by ignoring this warning.

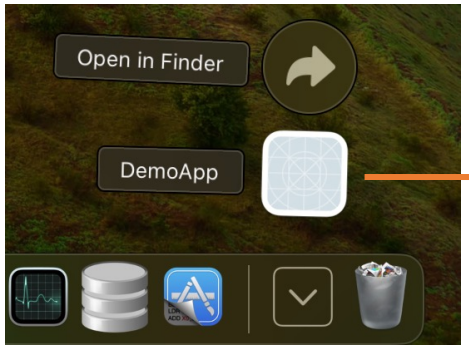


Quarantine attribute

Extended file attribute named com.apple.quarantine

Which files are quarantined?

- Downloaded files
- Files dropped by sandboxed apps



Extended attributes of the downloaded app

```
sh-3.2$ xattr -p com.apple.quarantine DemoApp.app
0083;650912b5;Safari;C7090EE3-1C1F-4D79-AC65-38516CE9B997
```

flags;timestamp;agent;UUID

Gatekeeper does not check apps without com.apple.quarantine

- Because macOS regards files without com.apple.quarantine as local ones

APIs related to the quarantine mechanism

libquarantine is the user-mode interface of the quarantine mechanism.

Two classes of functions (qtn_file_* and qtn_proc_*) are exported.

- qtn_file_* are used for dealing with the quarantine policy on a per-file basis.
  - ✓ e.g., qtn_file_apply_to_path adds the quarantine attribute to a specified path.
- qtn_proc_* are used for dealing with the quarantine policy on a per-process basis.
  - ✓ e.g., All files created by a process calling qtn_proc_apply_to_self are quarantined.

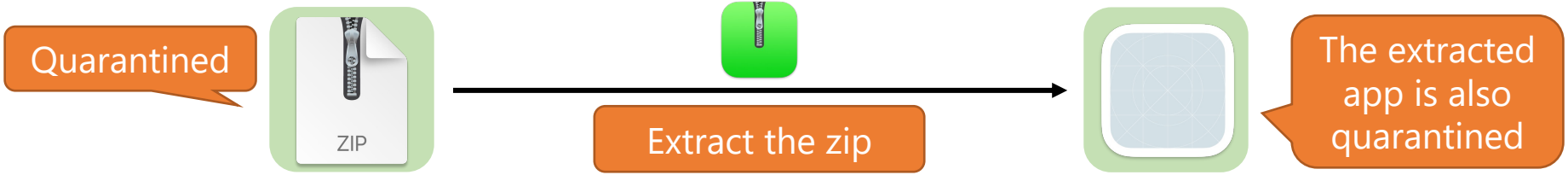
```
sh-3.2$ r2 -c "iE" -q usr/lib/system/libquarantine.dylib
[Exports]

nth paddr      vaddr          bind   type size lib name
_____________________________________________________
0   0x00001f0c 0x7ff80cd1ef0c GLOBAL FUNC 0        __qtn_error
1   0x00000ec6 0x7ff80cd1dec6 GLOBAL FUNC 0        __qtn_file_alloc
2   0x00001574 0x7ff80cd1e574 GLOBAL FUNC 0        __qtn_file_apply_to_fd
3   0x00002060 0x7ff80cd1f060 GLOBAL FUNC 0        __qtn_file_apply_to_mount_point
4   0x00000f7b 0x7ff80cd1df7b GLOBAL FUNC 0        __qtn_file_apply_to_path
5   0x00001f2c 0x7ff80cd1ef2c GLOBAL FUNC 0        __qtn_file_clone
6   0x00001244 0x7ff80cd1e244 GLOBAL FUNC 0        __qtn_file_free
```

```
23  0x00001704 0x7ff80cd1e704 GLOBAL FUNC 0        __qtn_proc_alloc
24  0x00002224 0x7ff80cd1f224 GLOBAL FUNC 0        __qtn_proc_apply_to_pid
25  0x000019d8 0x7ff80cd1e9d8 GLOBAL FUNC 0        __qtn_proc_apply_to_self
```

How to use quarantine APIs

```
char* input_file = argv[1];

// initialize quarantine info
qtn_file_t qinfo = _qtn_file_alloc();
const char* qdata =
"q/0083;60bca5e1;Safari;ED038CA1-1FD3-4A6A-B3DD-EF64B565C027";
_qtn_file_init_with_data(qinfo, qdata, strlen(qdata));

// add quarantine info to the file
_qtn_file_apply_to_path(qinfo, input_file);

// free quarantine info
_qtn_file_free(qinfo);
```

Propagating the quarantine attribute




```
sh-3.2$ xattr -p com.apple.quarantine DemoApp.zip
0083;650923f3;Safari;2D7D412C-8E85-4D7A-9DE4-5079042E170E
```

```
sh-3.2$ xattr -p com.apple.quarantine DemoApp.app/
0083;650923f3;Safari;2D7D412C-8E85-4D7A-9DE4-5079042E170E
sh-3.2$ xattr -p com.apple.quarantine DemoApp.app/Contents/MacOS/DemoApp
0083;650923f3;Safari;2D7D412C-8E85-4D7A-9DE4-5079042E170E
```

Decompile: _propagateQuarantineInformation - (x86-64-cpu0x3)

```
            (*(code *)__got::_objc_release)(local_100);
            puVar1 = __got::_objc_release;
            (*(code *)__got::_objc_release)(uVar7);
            (*(code *)puVar1)(uVar5);
            if (local_d1 != '\0') goto LAB_10001861f;
          }
          else {
LAB_100018616:
            (*(code *)__got::_objc_release)(uVar5);
LAB_10001861f:
            uVar5 = *(undefined8 *)(local_108 + _qtInfo);
            uVar3 = __stubs::_objc_retainAutorelease(uVar3);
            uVar3 = (*(code *)__got::_objc_msgSend)(uVar3,"fileSystemRepresentation");
            __stubs::__qtn_file_apply_to_path(uVar5,uVar3);
          }
          (*(code *)__got::_objc_release)(uVar4);
          uVar3 = local_e8;
          lVar9 = lVar9 + 1;
        } while (local_130 != lVar9);
```

Archive Utility calls qtn_file_apply_to_path to propagate com.apple.quarantine to the extracted files

My initial Gatekeeper bypass idea

Can we prevent Archive Utility from propagating quarantine attr?

I checked which files are not quarantined by qtn_file_apply_to_path

BSD file flag

macOS has components originating from BSD.

File flag is one of the BSD-derived features.

- Various flags can be specified to a file (https://man.freebsd.org/cgi/man.cgi?chflags(1)).
- The uchg flag captured my attention.

```
uchg, uchange, uimmutable
        set the user immutable flag (owner or super-user only)
```

- This flag is typically used for locking a file.

Can qtn_file_apply_to_path add com.apple.quarantine to a file having uchg?

Experiment

```
sh-3.2$ touch testfile
sh-3.2$ ./add_quarantine.out testfile
sh-3.2$ xattr -l testfile
com.apple.quarantine: 0083;60bca5e1;Safari;ED038CA1-1FD3-4A6A-B3DD-EF64B565C027
```

The file without uchg is quarantined

Experiment

```
sh-3.2$ touch testfile
sh-3.2$ ./add_quarantine.out testfile
sh-3.2$ xattr -l testfile
com.apple.quarantine: 0083;60bca5e1;Safari;ED038CA1-1FD3-4A6A-B3DD-EF64B565C027
sh-3.2$ rm testfile
sh-3.2$ touch testfile
sh-3.2$ chflags uchg testfile
sh-3.2$ ./add_quarantine.out testfile
sh-3.2$ xattr -l testfile
sh-3.2$
```

However, after adding uchg to this file...

...the file with uchg is not quarantined!

If we can add the uchg flag to an app bundle, we can bypass the Gatekeeper check.

How to retain uchg

Compressing an app to a ZIP file cannot retain file flags :(

The extracted files do not have uchg.

```
sh-3.2$ mkdir -p test/a
sh-3.2$ chflags uchg test/a
sh-3.2$ ls -lO test
total 0
drwxr-xr-x  2 nanoha  wheel  uchg 64 Sep 19 13:59 a
sh-3.2$ zip -r test.zip test
  adding: test/ (stored 0%)
  adding: test/a/ (stored 0%)
sh-3.2$ unzip -d tmp test.zip
Archive:  test.zip
   creating: tmp/test/
   creating: tmp/test/a/
sh-3.2$ ls -lO tmp/test
total 0
drwxr-xr-x  2 nanoha  wheel  -  64 Sep 19 13:59 a
```

uchg is missing :(

How to retain uchg

However, compressing an app to a tar.gz file can retain file flags :)

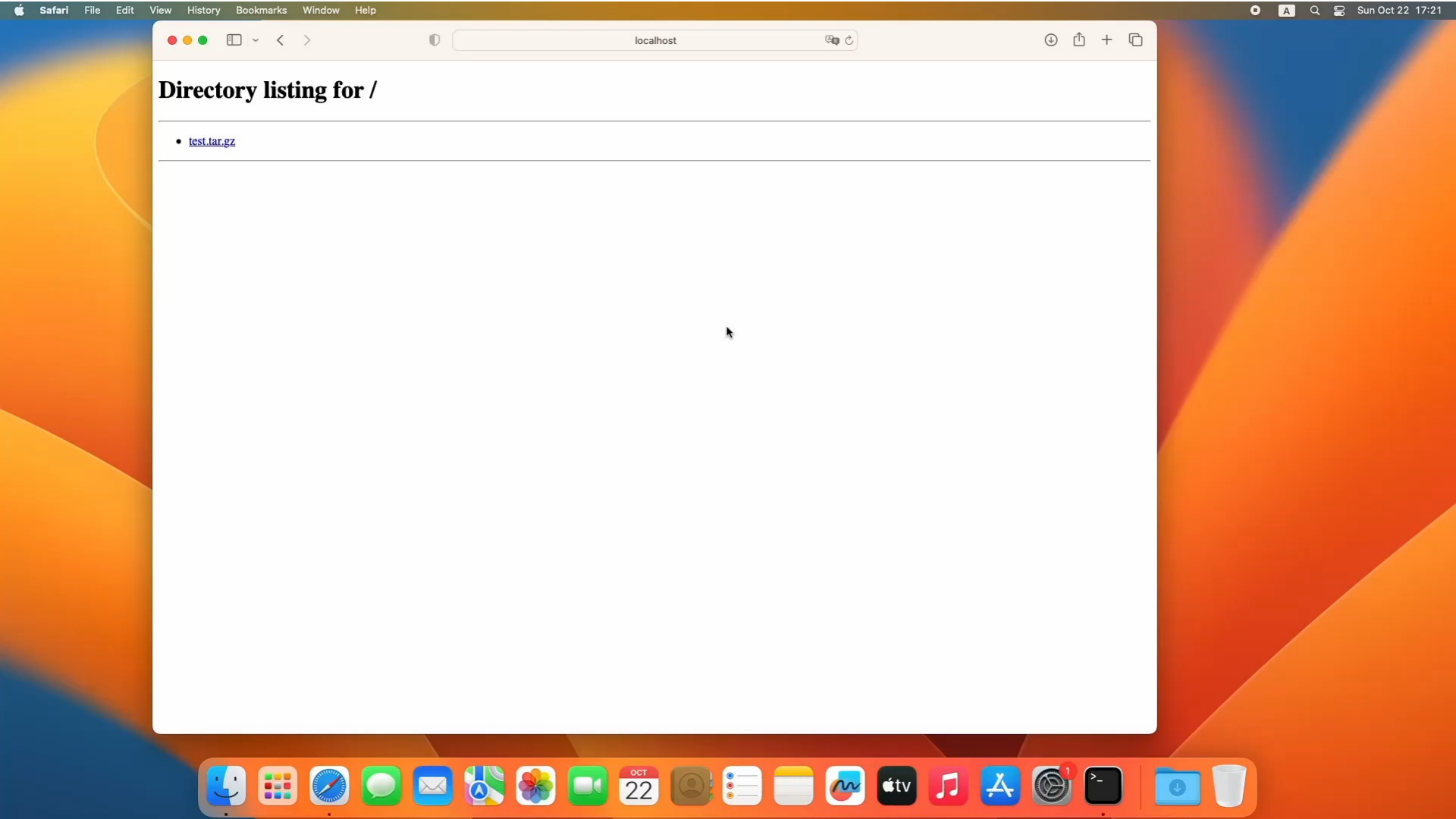
The extracted files have uchg.

```
sh-3.2$ mkdir -p test/a
sh-3.2$ chflags uchg test/a
sh-3.2$ tar czvf test.tar.gz test
a test
a test/a
sh-3.2$ open -a "Archive Utility" test.tar.gz
sh-3.2$ ls -lO test\ 2/
total 0
drwxr-xr-x  2 nanoha  staff  uchg 64 Sep 19 14:15 a
```

uchg is here :)

Steps to exploit

- Create a directory containing an app.
- Add uchg to the app.
- Compress the app to a tar.gz file.
- Send the tar.gz file to a victim.
- The victim opens the tar.gz file and runs the app.
- Gatekeeper does not check the app.


Safari File Edit View History Bookmarks Window Help
Sun Oct 22 17:21
localhost
Directory listing for /
test.tar.gz

Apple's fix

qtn_file_apply_to_path can add the quarantine attribute to a file having uchg.

```
[sh-3.2$ touch ~/a
[sh-3.2$ chflags uchg ~/a
[sh-3.2$ ls -lO ~/a
-rw-r--r--  1 kohnakagawa  staff  uchg 0 May 16 12:26 /Users/kohnakagawa/a
[sh-3.2$ ./add_quarantine.out ~/a
[sh-3.2$ xattr -l ~/a
com.apple.quarantine: 0083;60bca5e1;Safari;ED038CA1-1FD3-4A6A-B3DD-EF64B565C027
sh-3.2$
```

Apple did not assign CVE(?), but added my name to Additional Recognition.

quarantine

We would like to acknowledge Koh M. Nakagawa of FFRI Security, Inc. for their assistance.

https://support.apple.com/en-us/HT213670

Disclosure timeline

2022/11/28: I reported this vulnerability to Apple.

2022/11/30: I sent additional details.

2022/12/03: Apple validated the report.

2023/03/27: Apple fixed this vulnerability in macOS Ventura 13.3.

Bonus: App Sandbox bypass

This vulnerability can be used for bypassing App Sandbox.

Because we can prevent files dropped by sandboxed apps from adding the quarantine attribute

- Drop an app bundle and run it using the “open” command.
- The app is executed under the unsandboxed environment.

Related vulnerability: CVE-2022-42821

@yo_yo_yo_jbo at Microsoft reported a very similar vulnerability.

He abused AppleDouble & ACL to prevent Safari from adding the quarantine attr

- "Gatekeeper's Achilles heel: Unearthing a macOS vulnerability"

However, I reported the file flag trick to Apple before the disclosure of this vulnerability.

- Moreover, Apple's fix of CVE-2022-42821 was incomplete and still vulnerable to my trick in Ventura 13.
- The ultimate fix for my trick was applied in Ventura 13.3.

Typical Gatekeeper bypass vulns in 3rd party apps

Typical Gatekeeper bypass vulns in 3rd party apps

The root cause is missing LSFileQuarantineEnabled in the Info.plist file.

- This is "a Boolean value indicating whether the files this app creates are quarantined by default."

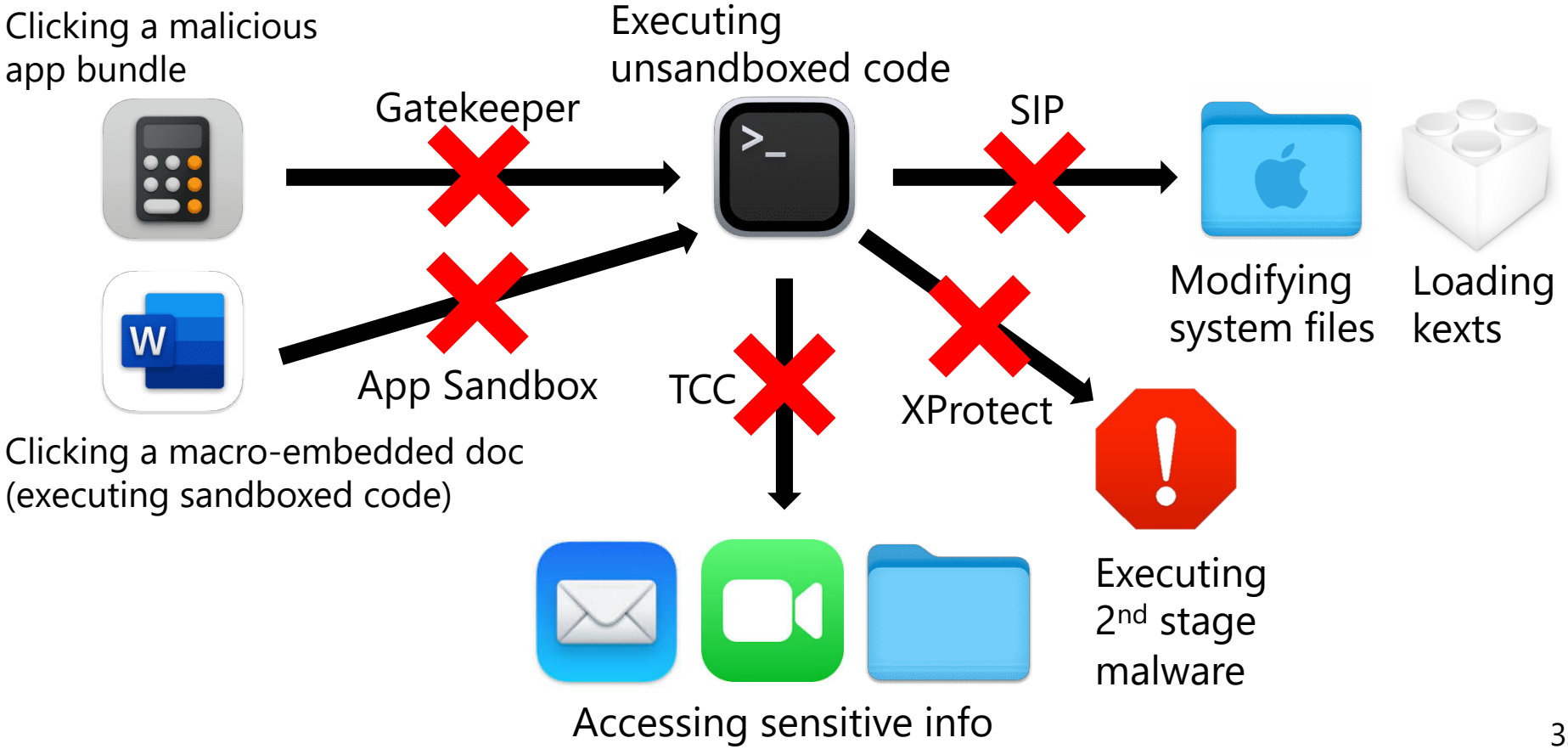
Example 1: Thunderbird CVE-2022-3155 (credited to me)

- Surprisingly, Thunderbird does not enable LSFileQuarantineEnabled for long.

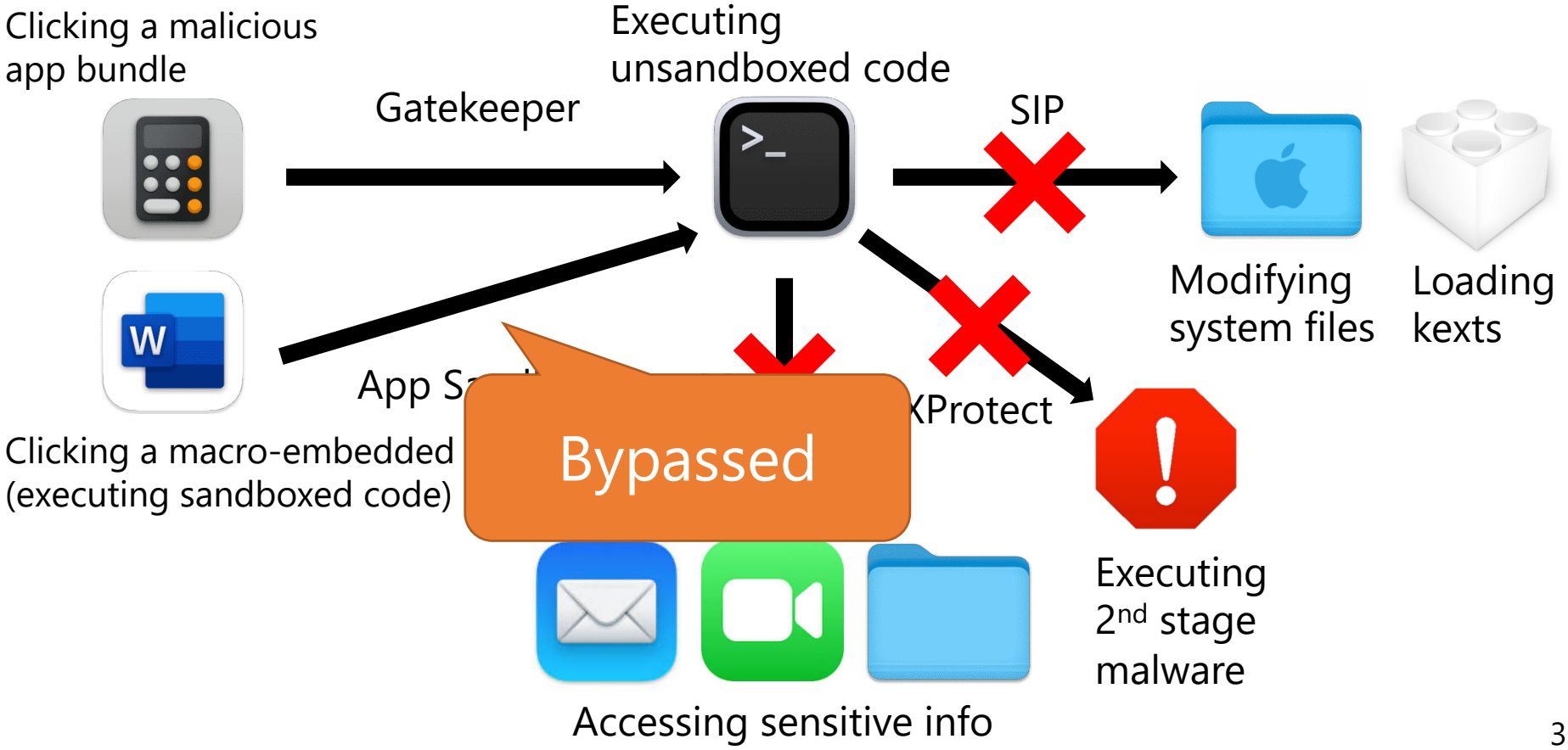
Example 2: (many) Electron-based apps

- Electron-based apps typically do not enable LSFileQuarantineEnabled
  - ✓ Because LSFileQuarantineEnabled breaks the auto update feature of Electron
- Please make sure that downloaded files are quarantined.
  - ✓ gatemaker is a possible solution.

Overview of macOS security & privacy mechanisms



Overview of macOS security & privacy mechanisms

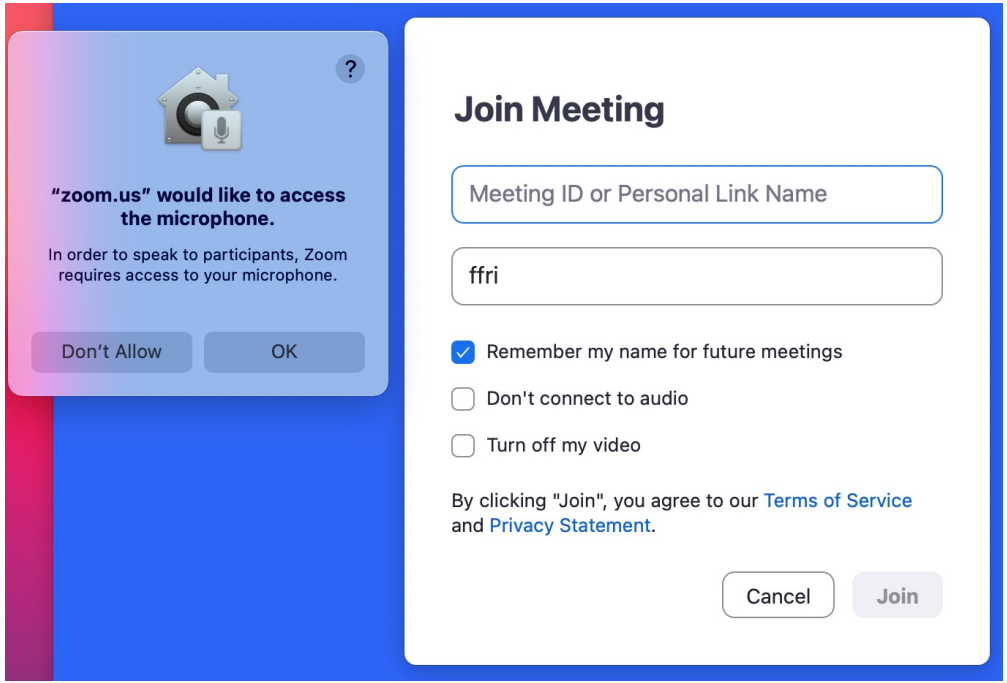


TCC bypass

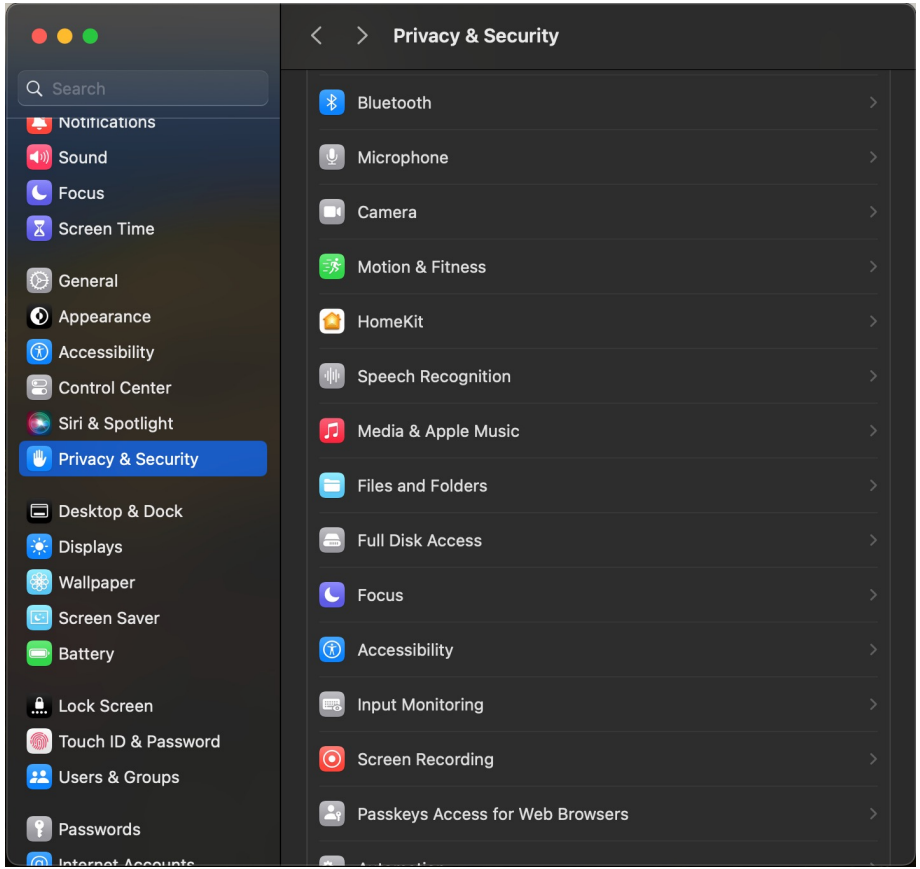
What is TCC?

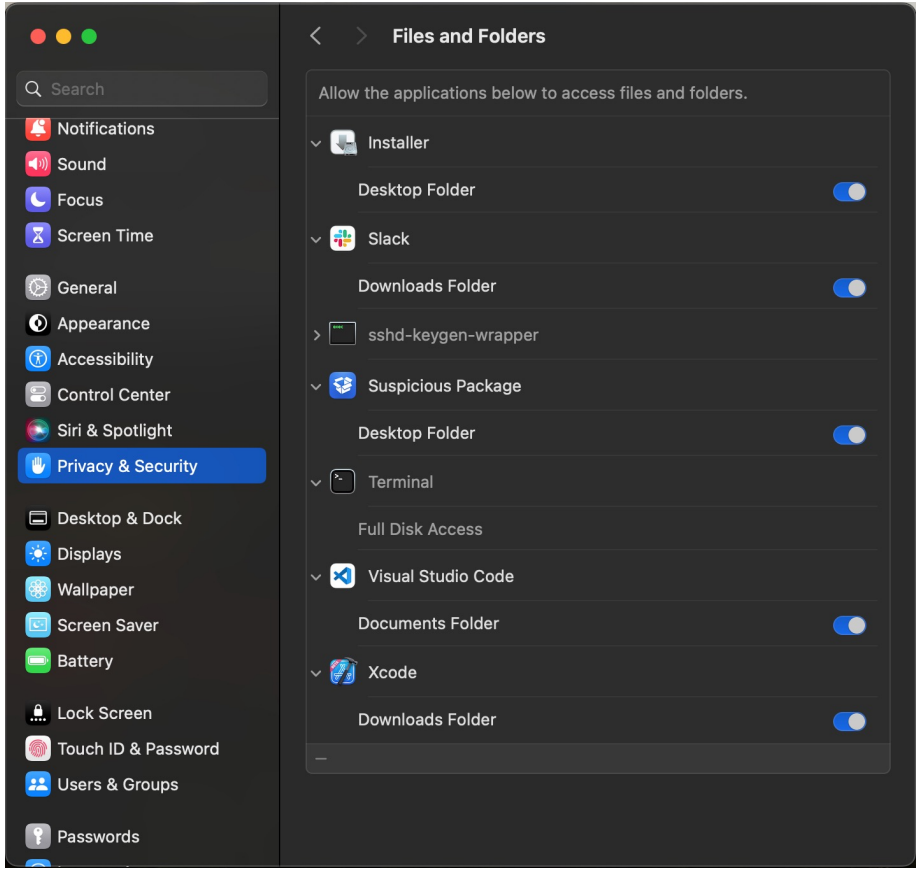
TCC is a privacy mechanism that protects a user's sensitive information.

- The sensitive information includes private folders, camera, and microphone.
- Even the root cannot access the sensitive information without the user's explicit consent.



What is TCC?


Privacy & Security
Search
Notifications
Sound
Focus
Screen Time
General
Appearance
Accessibility
Control Center
Siri & Spotlight
Privacy & Security
Desktop & Dock
Displays
Wallpaper
Screen Saver
Battery
Lock Screen
Touch ID & Password
Users & Groups
Passwords
Bluetooth
Microphone
Camera
Motion & Fitness
HomeKit
Speech Recognition
Media & Apple Music
Files and Folders
Full Disk Access
Focus
Accessibility
Input Monitoring
Screen Recording
Passkeys Access for Web Browsers


Files and Folders
Search
Notifications
Sound
Focus
Screen Time
General
Appearance
Accessibility
Control Center
Siri & Spotlight
Privacy & Security
Desktop & Dock
Displays
Wallpaper
Screen Saver
Battery
Lock Screen
Touch ID & Password
Users & Groups
Passwords
Allow the applications below to access files and folders.
Installer
Desktop Folder
Slack
Downloads Folder
sshd-keygen-wrapper
Suspicious Package
Desktop Folder
Terminal
Full Disk Access
Visual Studio Code
Documents Folder
Xcode
Downloads Folder

How does TCC work?

TCC is enforced by two tccd instances.

One runs as the root and the other runs as a logged-in user.

```
sh-3.2$ ps aux | grep tccd | grep -v grep
ffri             443   0.0  0.1  4368768   8736   ??  S     1:44PM   0:01.09 /System/Library/PrivateFrameworks/TCC.framework/Resources/tccd
root             153   0.0  0.1  4369312  11400   ??  Ss    1:44PM   0:03.76 /System/Library/PrivateFrameworks/TCC.framework/Resources/tccd system
```

There are two configuration files of TCC.

- /Library/Application Support/com.apple.TCC/TCC.db for the system (SIP-protected)
- ~/ Library/Application Support/com.apple.TCC/TCC.db for the logged-in user (TCC-protected)
  - ✓ An app with Full Disk Access can modify this TCC.db

If we can modify these database files directly, we can bypass TCC.

- However, these database files are SIP-protected or TCC-protected.

Previous research on TCC bypass techniques

TCC bypass techniques are classified into the following categories:

Running code in the context of other approved (or entitled) apps

- Example 1: Dylib injection through DYLD_INSERT_LIBRARIES (e.g., CVE-2020-24259)
- Example 2: Dylib injection through plugins (e.g., CVE-2020-27937)

Fooling tccd

- Example: Mount over the TCC directory and force tccd to use a fake TCC.db (CVE-2021-30808)

Other neat ideas

- Example 1: Abuse App Translocation (CVE-2021-30782)
- Example 2: Abuse Time Machine Snapshot (CVE-2020-9771)
- For other techniques, see the BHUSA 2021 and BHEU 2022 talks by @theevilbit and @_r3ggi

Previous research on TCC bypass techniques

TCC bypass techniques are classified into the following categories:

Running code in the context of other approved (or entitled) apps

- Example 1: Dylib inje
- Example 2: Dylib inje

Here, I will show a technique classified into this category.

Fooling tccd

- Example: Mount over the TCC directory and force tccd to use a fake TCC.db (CVE-2021-30808)

Other neat ideas

- Example 1: Abuse App Translocation (CVE-2021-30782)
- Example 2: Abuse Time Machine Snapshot (CVE-2020-9771)
- For other techniques, see the BHUSA 2021 and BHEU 2022 talks by @theevilbit and @_r3ggi

How to inject code

Code injection is strictly prohibited on macOS.

Dylib injection/hijacking

- Library validation is typically enabled for apps.
- It prevents an app from loading untrusted dylibs, and hence, dylib injection/hijacking does not work.

Thread injection

- Like CreateRemoteThread type injection on Windows
- However, this works if the app has the get-task-allow entitlement.
- Few apps have the get-task-allow entitlement.

Other code injection techniques on macOS?

- Technique using saved state (CVE-2021-30873 by @xnyhps), but this is currently fixed

How to inject code

Code injection is strictly prohibited on macOS.

Dylib injection/hijacking

- Library validation is typically enabled for apps.
- It prevents an app from loading untrusted dylibs, and hence, dylib injection/hijacking does not work.

Thread injection

- Like CreateRemoteThread type injection on Windows
- However, this works if the app has the get-task-allow entitlement.
- Few apps have the get-task-allow entitlement

Oth

- Technique using saved state (CVE-2021-30873 by @xhyrhps), but this is currently fixed

I developed a new code injection technique abusing Rosetta 2

AOT poisoning

Code injection poisoning Rosetta 2 binary translation cache

Rosetta 2 is a translation mechanism to execute x86_64 code on Apple Silicon.

- Translated artifacts (AOT files) are saved and cached.
- Rosetta 2 reuses these artifacts when a user runs the same app again.
- Rosetta 2 provides translate_tool
  - ✓ This is a CLI tool to enable the creation of an AOT file without running the executable.
  - ✓ Located at the /usr/libexec/rosetta directory

```
sh-3.2$ ls /usr/libexec/rosetta/
debugserver             oahd-helper             runtime
oahd                    oahd-root-helper        translate_tool
```

Rosetta 2 internals: AOT lookup hash

Rosetta 2 uses the dedicated hash for checking whether the app was previously translated.

- Referred to as “AOT lookup hash” in this talk

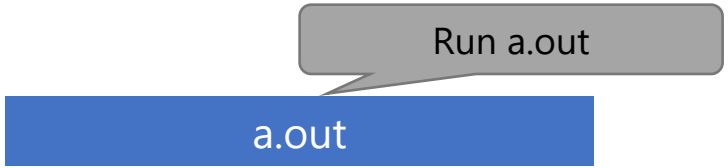
Decompile: FUN_1000058cc – (oahd)

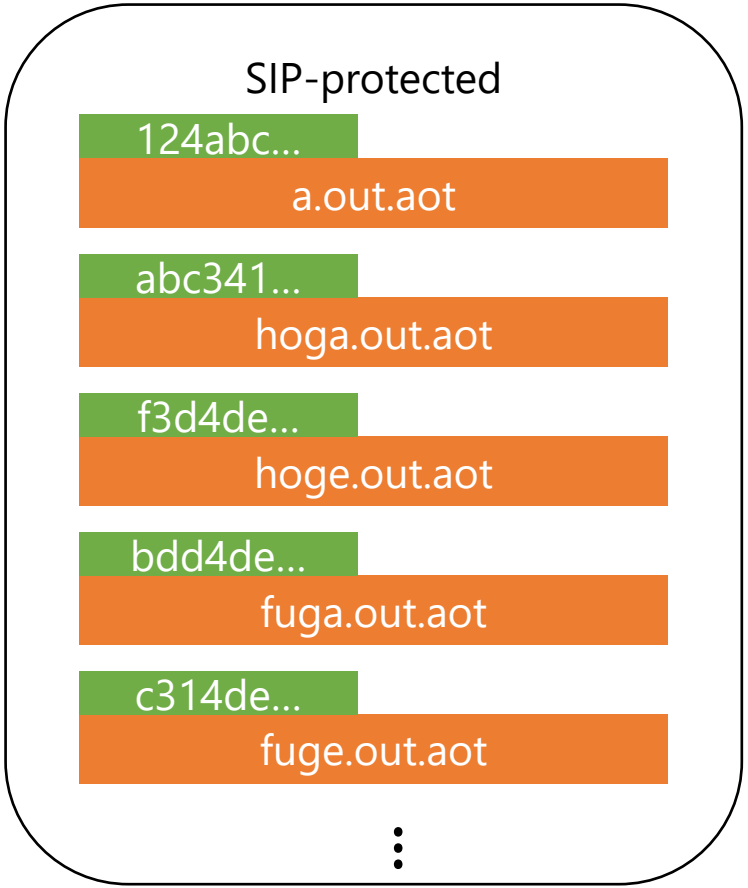
```
512             __auth_stubs::_CC_SHA256_Init((CC_SHA256_CTX *)local_8210);
513             pppppppuVar41 = local_8bf8;
514             if (-1 < local_8be8) {
515               _Stack35824._0_4_ = (CC_LONG)local_8be8._7_1_;
516               pppppppuVar41 = &local_8bf8;
517             }
518                   /* file path */
519             __auth_stubs::_CC_SHA256_Update
520                       ((CC_SHA256_CTX *)local_8210,pppppppuVar41,(CC
521                   /* mach_header + load_commands */
522             __auth_stubs::_CC_SHA256_Update
523                       ((CC_SHA256_CTX *)local_8210,&local_8080,CONC
524                   /* uid */
525             __auth_stubs::_CC_SHA256_Update((CC_SHA256_CTX *)local_
526                   /* gid */
527             __auth_stubs::_CC_SHA256_Update((CC_SHA256_CTX *)loca
528                   /* mtime */
529             __auth_stubs::_CC_SHA256_Update((CC_SHA256_CTX *)loca
530                   /* ctime */
531             __auth_stubs::_CC_SHA256_Update((CC_SHA256_CTX *)local_
532                   /* crtime */
533             __auth_stubs::_CC_SHA256_Update
534                       ((CC_SHA256_CTX *)local_8210,&stat.st_birthtimespec,0x10);
535                   /* st_size */
536             __auth_stubs::_CC_SHA256_Update((CC_SHA256_CTX *)local_8210,&stat.st_size,8);
537             __auth_stubs::_CC_SHA256_Final((uchar *)&local_8230,(CC_SHA256_CTX *)local_8210);
```

A SHA-256 hash is generated based on

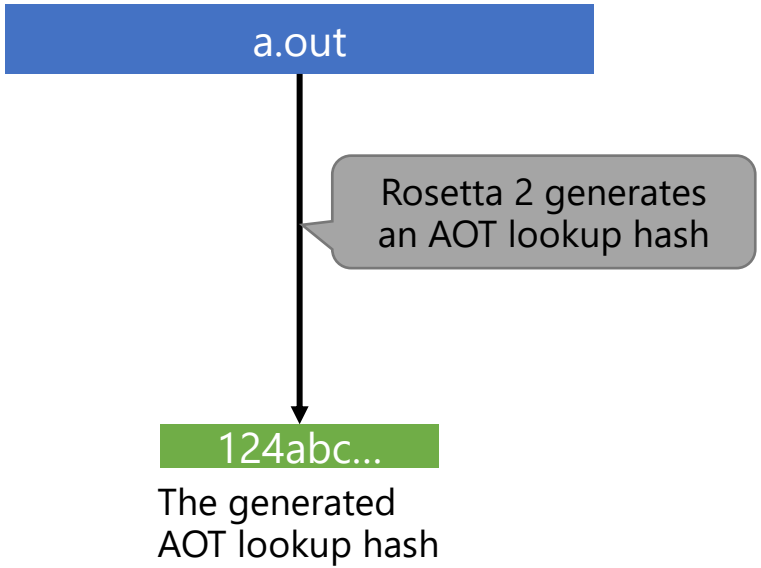
- Mach-O header and load commands
- ctime, mtime, and crtime
- file path
- uid and gid

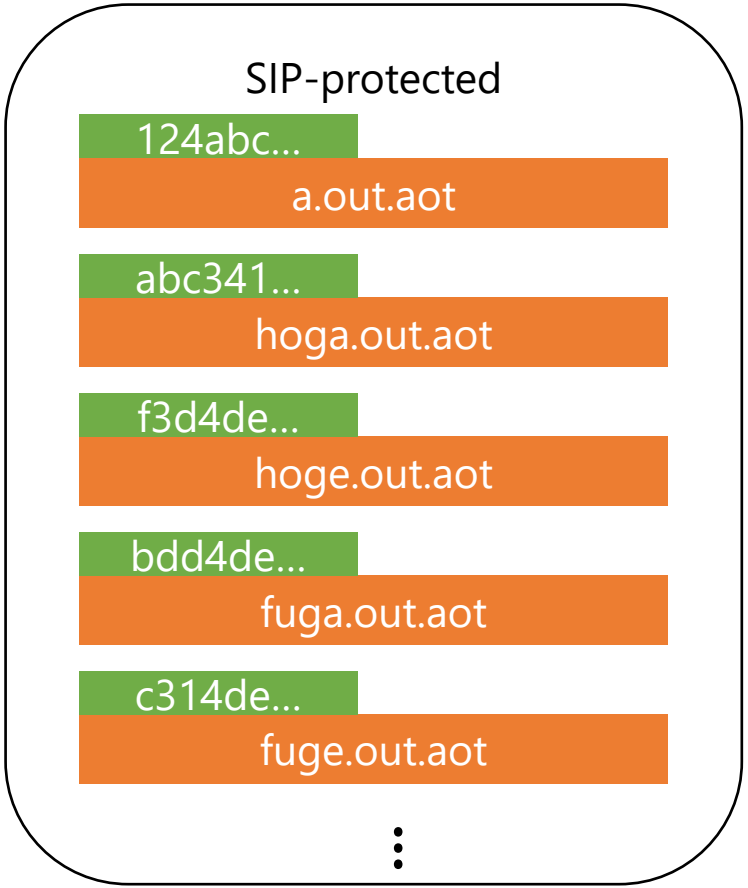
Rosetta 2 internals: AOT lookup hash



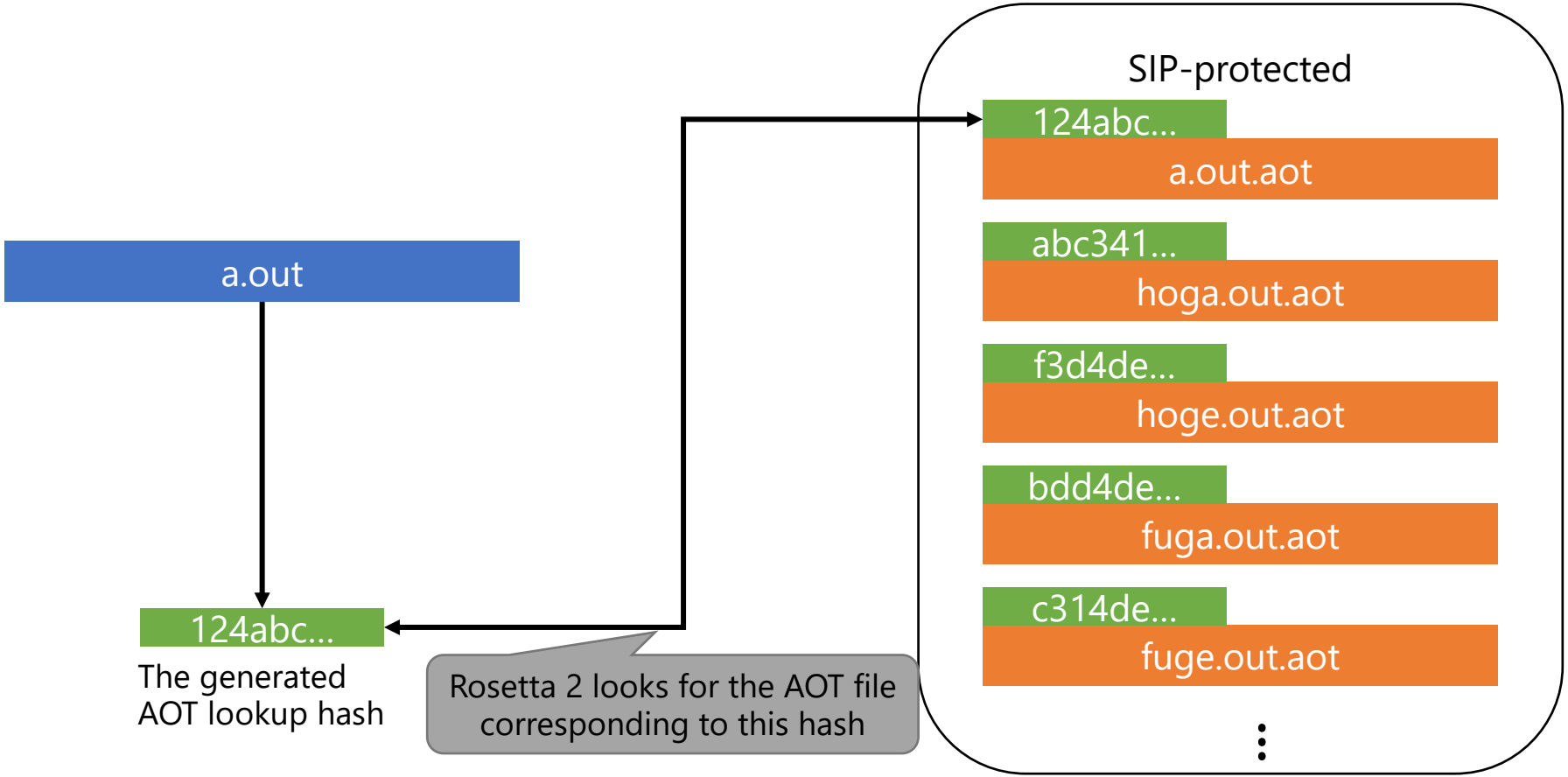


Rosetta 2 internals: AOT lookup hash

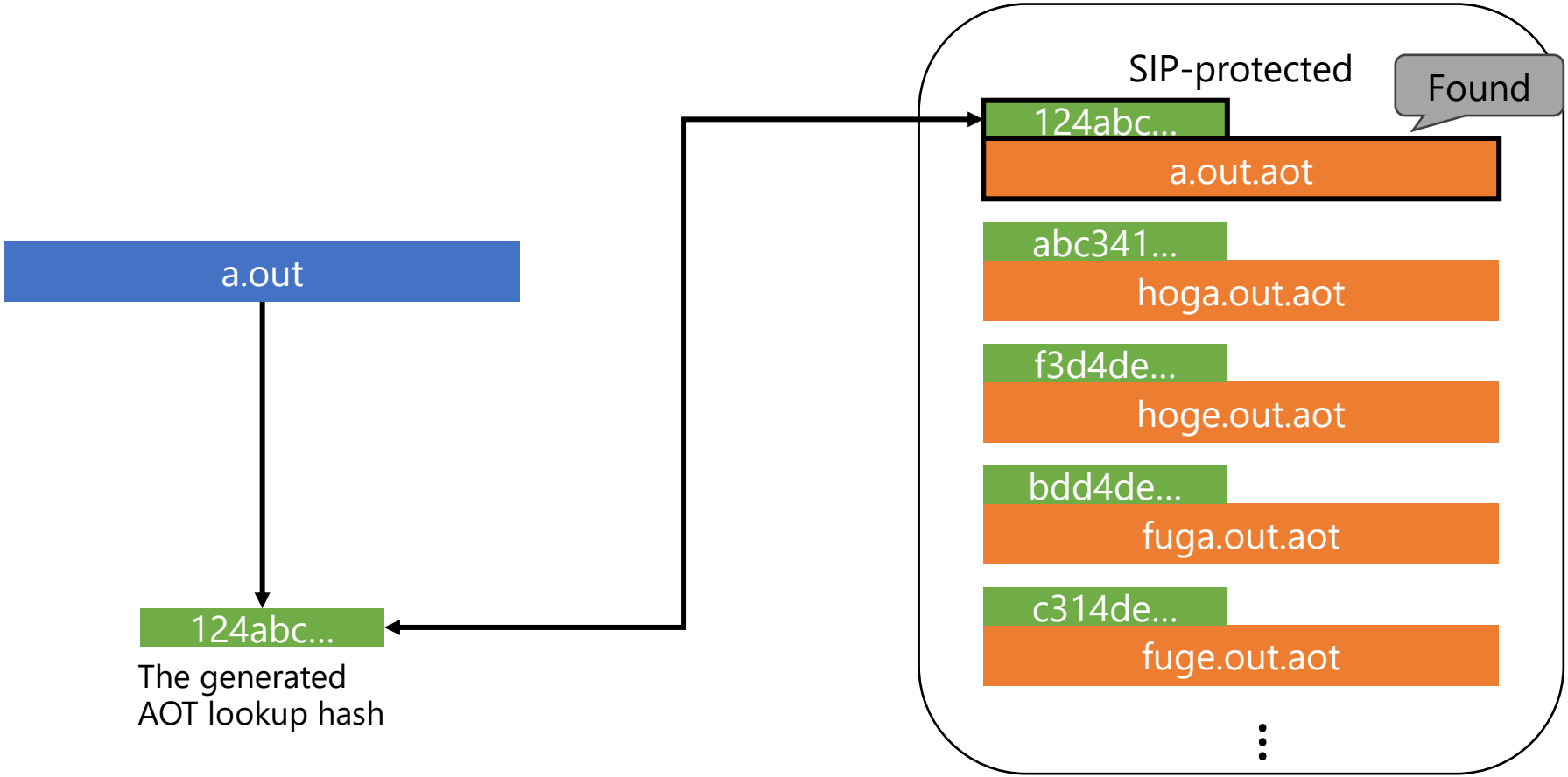




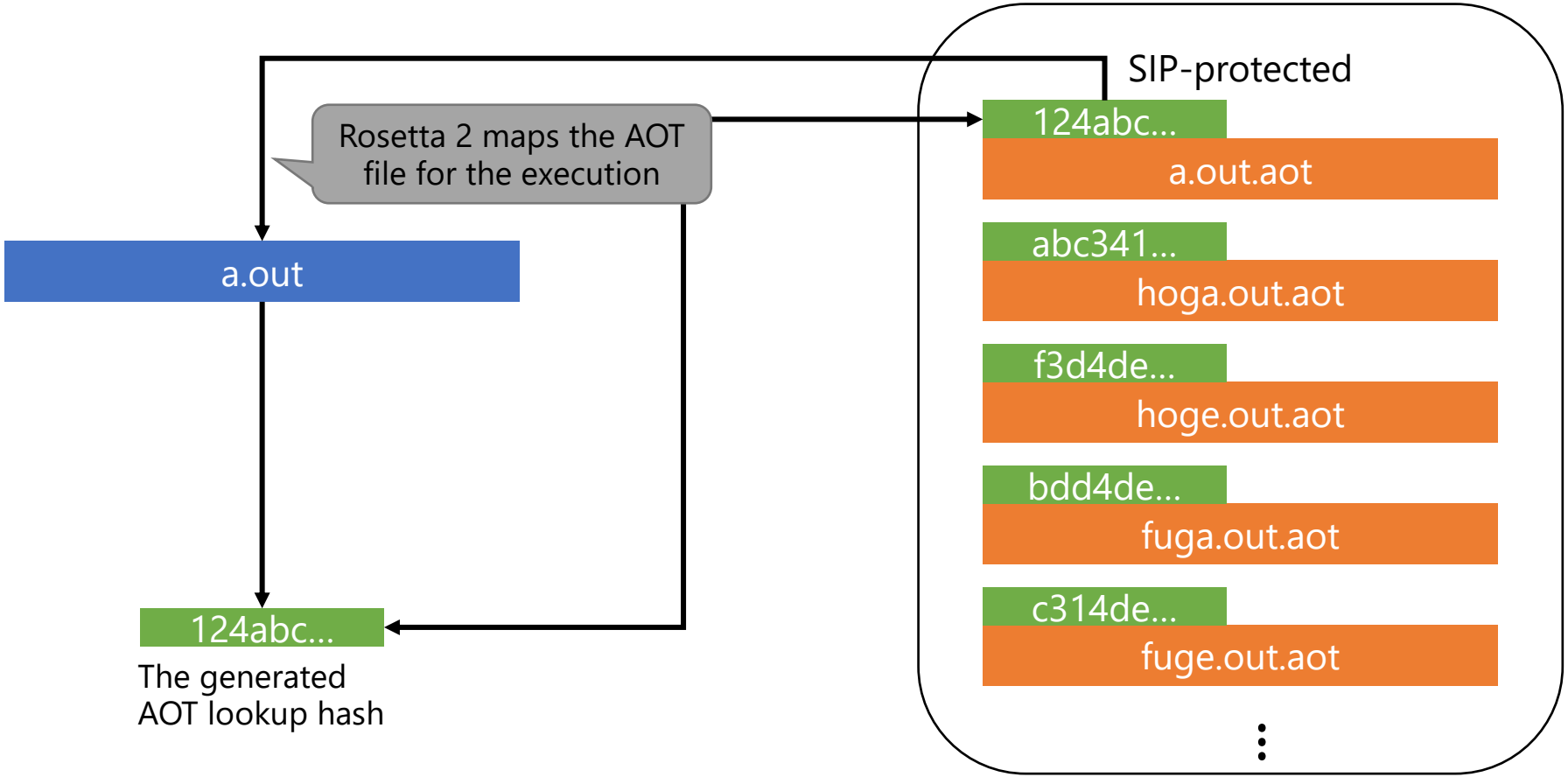
Rosetta 2 internals: AOT lookup hash



Rosetta 2 internals: AOT lookup hash



Rosetta 2 internals: AOT lookup hash



Core idea of AOT poisoning

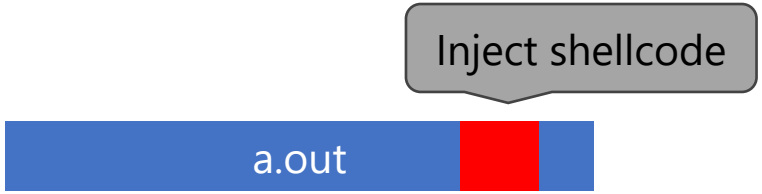
Hash collision attack on an AOT lookup hash

The AOT lookup hash is not generated based on the entire contents of an executable.

- The AOT lookup hash is a SHA-256 hash generated based on…
  - ✓ Mach-O header and load commands
  - ✓ ctime, mtime, and crtime
  - ✓ file path
  - ✓ uid and gid
- The code section **is not used** for generating this hash.

If we can modify the code in an executable while keeping its AOT lookup hash unchanged, we can force Rosetta 2 to use a different AOT file upon execution.

AOT poisoning



SIP-protected

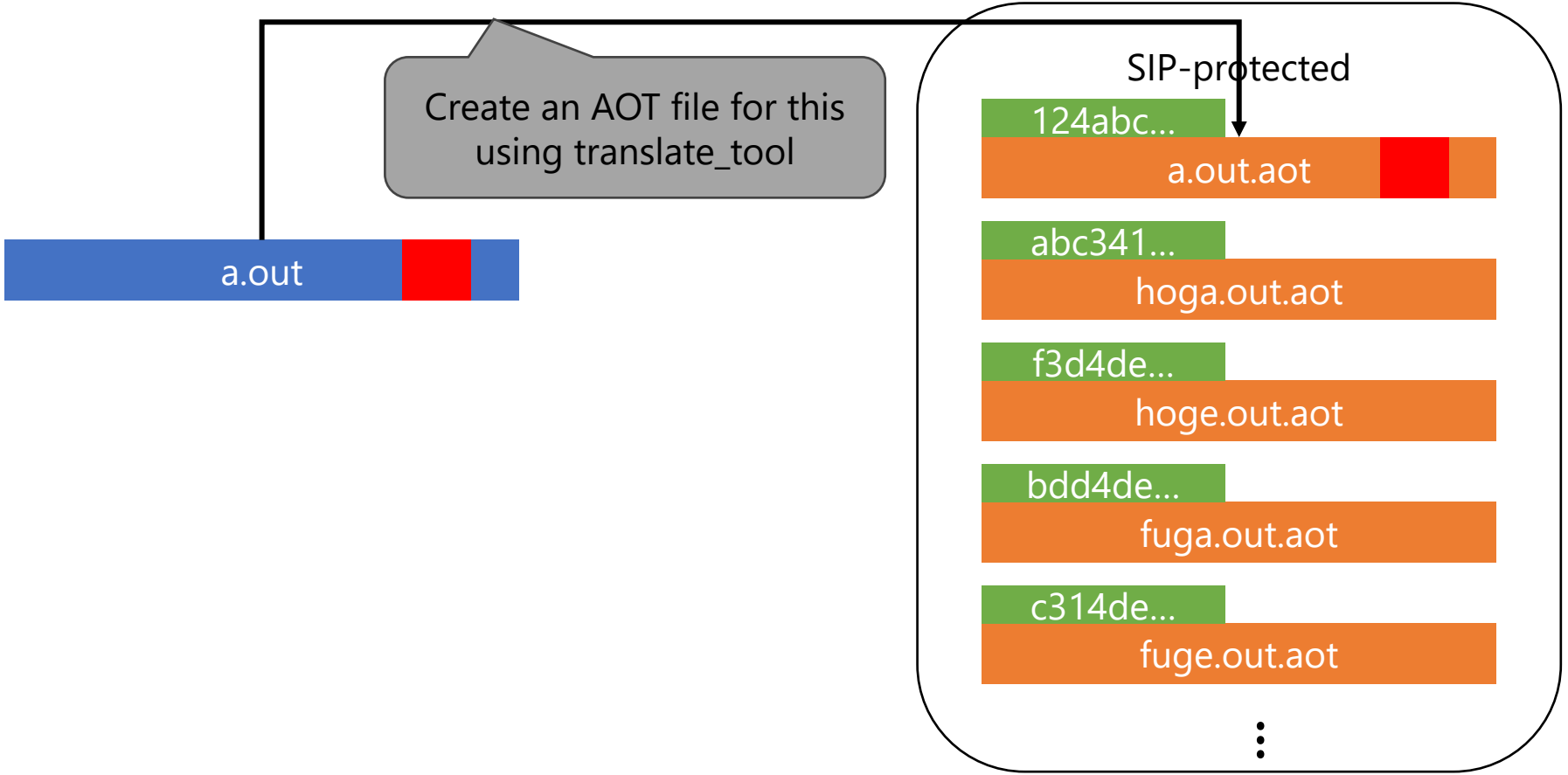
abc341...
hoga.out.aot

f3d4de...
hoge.out.aot

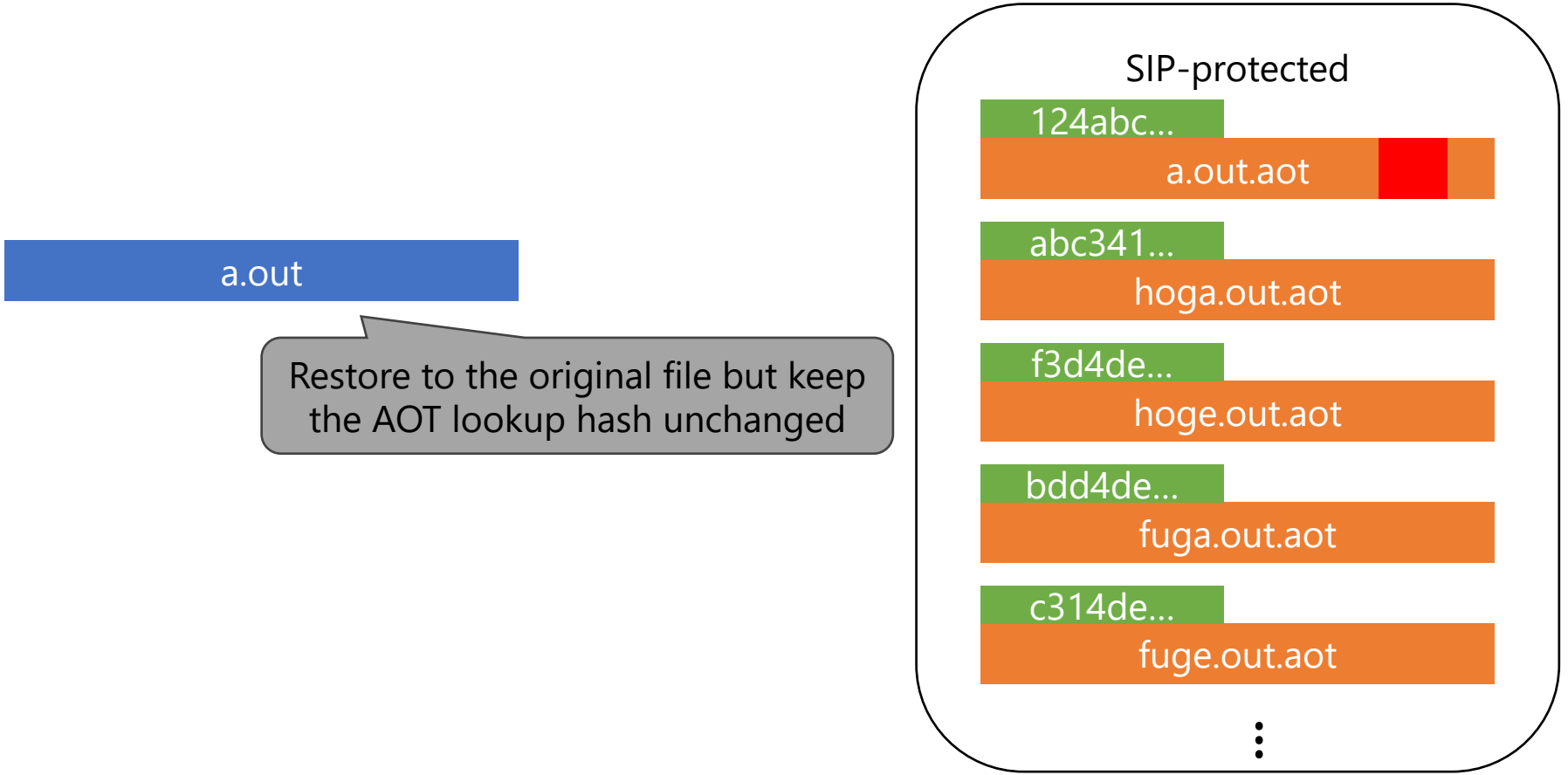
bdd4de...
fuga.out.aot

c314de...
fuge.out.aot

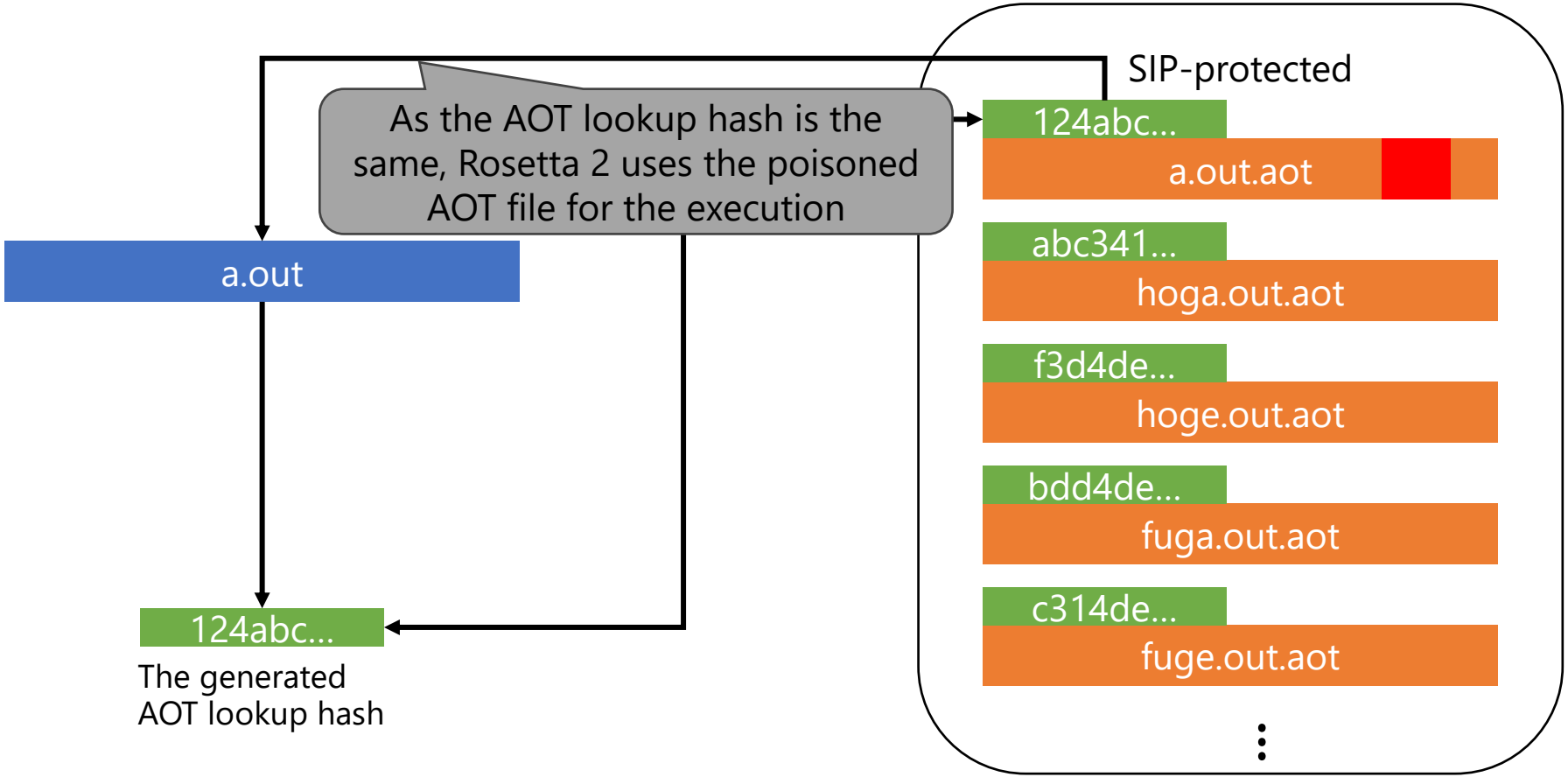
AOT poisoning



AOT poisoning



AOT poisoning



Core idea of AOT poisoning

How do we modify a code section while keeping the AOT lookup unchanged?

The AOT lookup hash is a SHA-256 hash generated based on…
- Mach-O header and load commands
- ctime, **mtime**, and crtime
- file path
- uid and gid

Of course, modifying the file updates mtime

Idea: Restore mtime after modifying the file

- However, modifying the timestamp always updates ctime (at least on the APFS filesystem).
- As the AOT lookup hash is generated based on ctime, mtime, and crtime, modifying the timestamp changes the AOT lookup hash…

Core idea of AOT poisoning

How do we modify a code section while keeping the AOT lookup unchanged?

The AOT lookup hash is a SHA-256 hash generated based on…
- Mach-O header and load commands
- ctime, **mtime**, and crtime
- file path
- uid and gid

Of course, modifying the file updates mtime

But how about other filesystems?

Idea: Restore mtime after modifying the file

- However, modifying the timestamp always updates ctime **(at least on the APFS filesystem)**.
- As the AOT lookup hash is generated based on ctime, mtime, and crtime, modifying the timestamp changes the AOT lookup hash…

Filesystem downgrade trick

Timestamps of FAT32 filesystem

mtime ctime crtime

Time Stored	Time Resolution	Date Modified	Date Accessed	Date Change	Birth
UTC	Jan 1, 1970 in local time	Updated	Updated	N/A	Creation

Table 1: FAT32 Modification times (Lee, 2015)

ctime is not defined on FAT32!
-> Modifying mtime does not update ctime

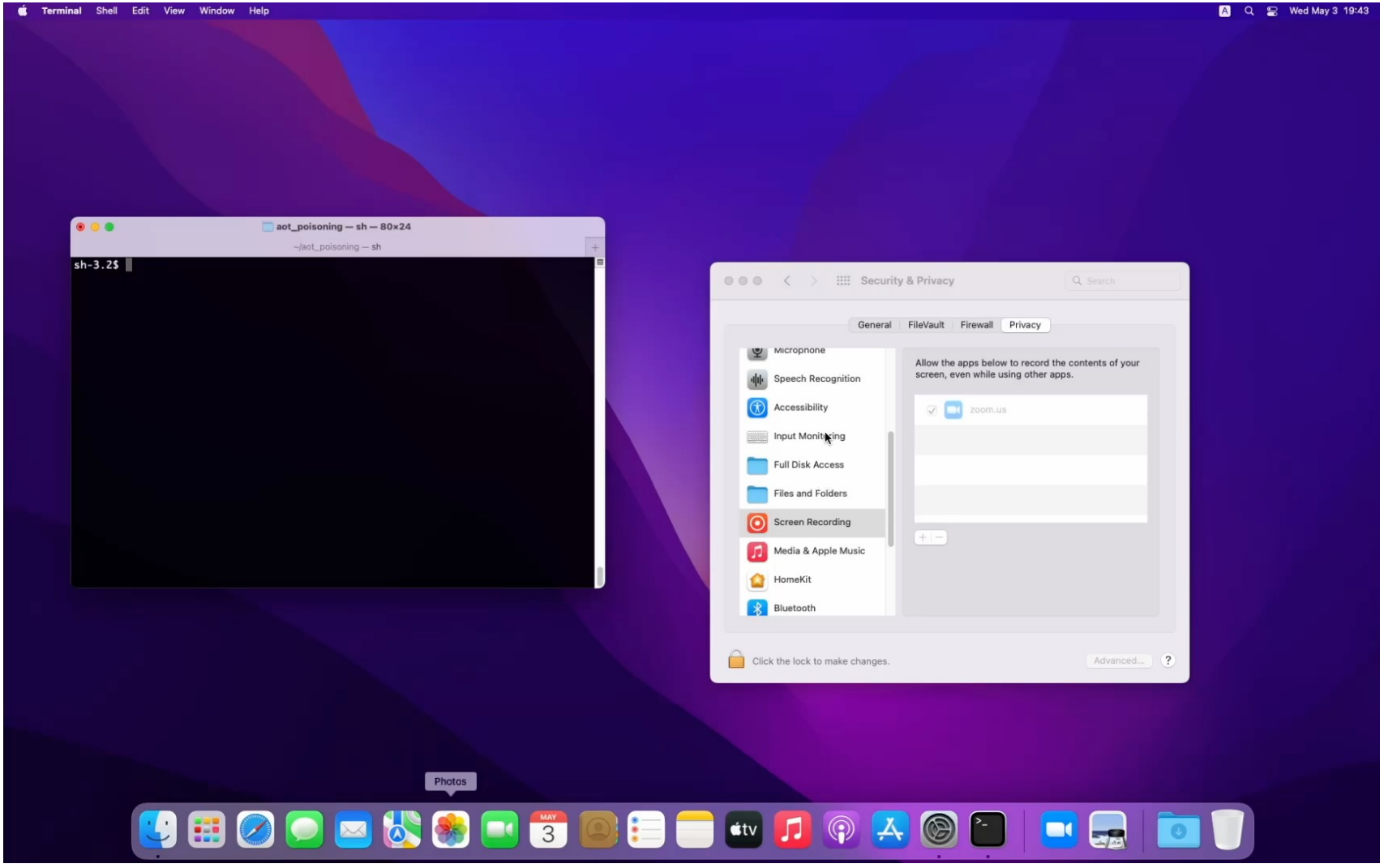
If we copy an app to the mounted FAT32 dmg, we can modify a code section while keeping the AOT lookup hash unchanged.

https://www.sans.org/white-papers/36842/

Steps to exploit

- Find a TCC-approved app.
- Create a FAT32 dmg and mount it.
- Copy the approved app to the mounted point.
- Inject shellcode into it.
- Run translate_tool to create an AOT file.
- Restore the target executable to the original executable.
- Restore the timestamps.
- Run the executable.
- We can execute code in the context of the approved app!😎

For more details, see my Black Hat Asia 2023 talk slides


Terminal Shell Edit View Window Help
Wed May 3 19:43
aot_poisoning — sh — 80×24
~/aot_poisoning — sh
sh-3.2$
Security & Privacy
Search
General
FileVault
Firewall
Privacy
Microphone
Speech Recognition
Accessibility
Input Monitoring
Full Disk Access
Files and Folders
Screen Recording
Media & Apple Music
HomeKit
Bluetooth
Allow the apps below to record the contents of your screen, even while using other apps.
zoom.us
Click the lock to make changes.
Advanced...
Photos

Apple's fix

Apple addressed this issue in macOS Ventura 13.

CVE-2022-42789 is assigned.

- We no longer use the AOT poisoning for a signed executable.
- However, we still use this technique for a non-signed executable :(

Apple gave me a generous bounty :)

AppleMobileFileIntegrity

Available for: Mac Studio (2022), Mac Pro (2019 and later), MacBook Air (2018 and later), MacBook Pro (2017 and later), Mac mini (2018 and later), iMac (2017 and later), MacBook (2017), and iMac Pro (2017)

Impact: An app may be able to access user-sensitive data

Description: An issue in code signature validation was addressed with improved checks.

CVE-2022-42789: Koh M. Nakagawa of FFRI Security, Inc.

https://support.apple.com/en-us/HT213488

Bonus: XProtect bypass

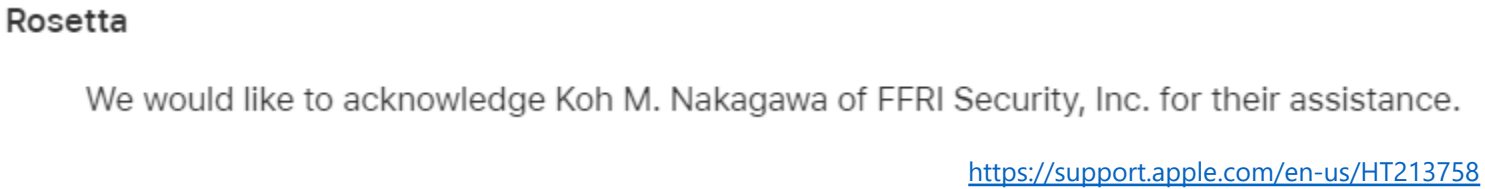
This code injection allowed an attacker to bypass XProtect.

XProtect scans an x86_64 executable **only when it is launched**.

- It does not scan an executable when an attacker generates the AOT file using translate_tool
- If an attacker injects code into a benign executable, he/she can bypass the XProtect scan.

Apple also fixed this issue in macOS Ventura 13.4.

Now, XProtect scans an x86_64 executable when its AOT file is generated.



https://support.apple.com/en-us/HT213758

About 3rd party apps

Code injection vulnerabilities of TCC bypass in 3rd party apps

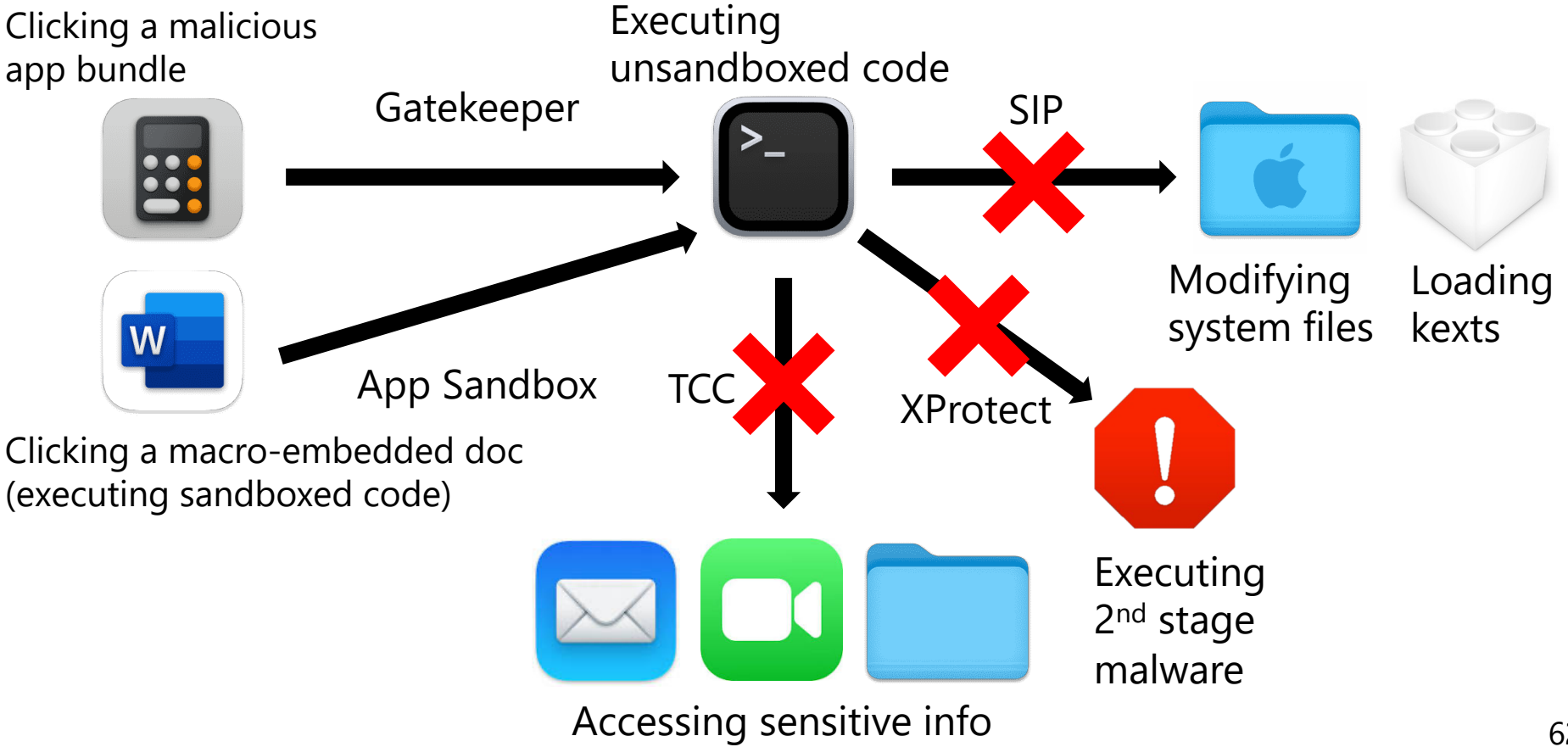
Electron-based apps that do not disable ELECTRON_RUN_AS_NODE

- Can inject any JavaScript code by specifying ELECTRON_RUN_AS_NODE
- Example: Chatwork Desktop App (CVE-2023-32546, currently fixed, credited to me)
- Please disable ELECTRON_RUN_AS_NODE by using Electron Fuse

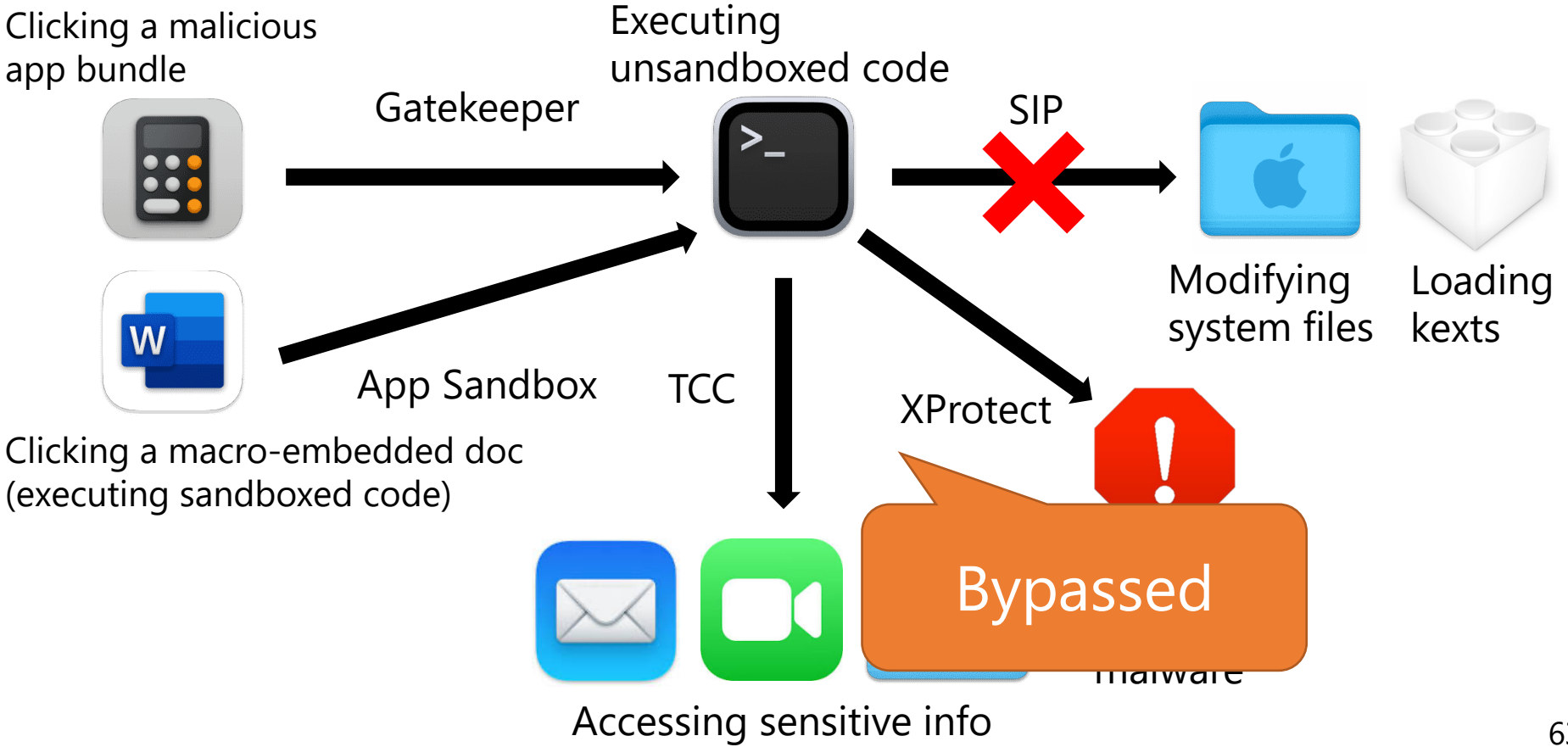
For other issues of Electron-based apps, please see the following talk and post:

- ELECTRONizing macOS privacy by @_r3ggi
- Abusing Electron apps to bypass macOS' security controls by @_r3ggi

Overview of macOS security & privacy mechanisms



Overview of macOS security & privacy mechanisms



SIP bypass

What is System Integrity Protection (SIP)?

System Integrity Protection (SIP)

Restricts some dangerous operations, such as …

- Modifying system files
- Loading untrusted kernel extensions
- Debugging system processes in user mode
- Kernel debugging

Even the root user cannot perform these dangerous operations.

- The root user is not GOD on macOS.

To disable SIP, a user needs to restart his/her Mac device in Recovery mode.

- However, it requires physical access to the Mac device.

More about SIP

SIP is configured through NVRAM variables.

NVRAM bit	Description
CSR_ALLOW_UNTRUSTED_KEXTS	Controls the loading of untrusted kernel extensions
CSR_ALLOW_UNRESTRICTED_FS	Controls write access to restricted filesystem locations
CSR_ALLOW_TASK_FOR_PID	Controls whether to allow getting a task port for Apple processes (that is, invoke the *task_for_pid* API)
CSR_ALLOW_UNRESTRICTED_NVRAM	Controls unrestricted NVRAM access
CSR_ALLOW_KERNEL_DEBUGGER	Controls whether to allow kernel debugging

https://www.microsoft.com/en-us/security/blog/2021/10/28/microsoft-finds-new-macos-vulnerability-shrootless-that-could-bypass-system-integrity-protection/
https://github.com/apple-oss-distributions/xnu/blob/main/bsd/sys/csr.h#L41-L54

More about SIP

SIP is configured through NVRAM variables.

NVRAM bit	Description
CSR_ALLOW_UNTRUSTED_KEXTS	Controls the loading o
CSR_ALLOW_UNRESTRICTED_FS	Controls write access to restricted filesystem locations
CSR_ALLOW_TASK_FOR_PID	Controls whether to allow getting a task port for Apple processes (that is, invoke the *task_for_pid* API)
CSR_ALLOW_UNRESTRICTED_NVRAM	Controls unrestricted NVRAM access
CSR_ALLOW_KERNEL_DEBUGGER	Controls whether to allow kernel debugging

My focus in this talk

https://www.microsoft.com/en-us/security/blog/2021/10/28/microsoft-finds-new-macos-vulnerability-shrootless-that-could-bypass-system-integrity-protection/
https://github.com/apple-oss-distributions/xnu/blob/main/bsd/sys/csr.h#L41-L54

SIP filesystem restrictions

Which files are protected by SIP?

Files listed in /System/Library/Sandbox/rootless.conf

- On boot, rootless-init applies file system restrictions to these files.

```
sh-3.2$ head -n 10 /System/Library/Sandbox/rootless.conf
                                /Applications/Safari.app
                                /Library/Apple
TCC                             /Library/Application Support/com.apple.TCC
CoreAnalytics                   /Library/CoreAnalytics
NetFSPlugins                    /Library/Filesystems/NetFSPlugins/Staged
NetFSPlugins                    /Library/Filesystems/NetFSPlugins/Valid
                                /Library/Frameworks/iTunesLibrary.framework
KernelExtensionManagement       /Library/GPUBundles
KernelExtensionManagement       /Library/KernelCollections
MessageTracer                   /Library/MessageTracer
```

Protected files have the restricted file flag.

- You can check this by running the ls -lO command.
- An attacker cannot attach the restricted file flag to a file manually.

```
sh-3.2$ ls -lO /bin/ls
-rwxr-xr-x  1 root  wheel  restricted,compressed 187120 Sep  6 13:50 /bin/ls
```

Why is SIP bypass so critical?

SIP bypass always means Full TCC bypass

As described in Mickey Jin's write-up

Because we can modify the SIP-protected TCC database file with this primitive

An attacker can create undeletable malware with this primitive.

Thus, an attacker gains powerful persistence with an SIP bypass vulnerability.

- Even XProtect cannot remove this malware because it does not have SIP-related entitlements.

Some SIP-related entitlements

Entitlement	Description
com.apple.rootless.install	Completely bypasses SIP filesystem checks
com.apple.rootless.install.heritable	Inherits com.apple.rootless.install to child processes

https://www.microsoft.com/en-us/security/blog/2021/10/28/microsoft-finds-new-macos-vulnerability-shrootless-that-could-bypass-system-integrity-protection/

Only Apple binaries can have these private entitlements.

Apple binary with SIP-related entitlements

system_installd

```
sh-3.2$ codesign -d --entitlements - /System/Library/PrivateFrameworks/PackageKit.framework/Versions/A/Resources/system_installd
Executable=/System/Library/PrivateFrameworks/PackageKit.framework/Versions/A/Resources/system_installd
[Dict]
        [Key] com.apple.private.apfs.create-synthetic-symlink-folder
        [Value]
                [Bool] true
        [Key] com.apple.private.launchservices.cansetapplicationstrusted
        [Value]
```

⋮

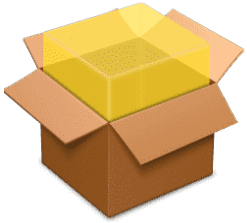
```
        [Key] com.apple.rootless.install.heritable
        [Value]
                [Bool] true
```

system_installd has com.apple.rootless.install.heritable

"system_installd is used by the system during package installation."

- However, this daemon is used for an Apple-signed macOS pkg installation.
  - ✓ For 3rd party package installation, installd is used instead.
  - ✓ Of course, installd does not have SIP-related entitlements.

Apple software package (.pkg)

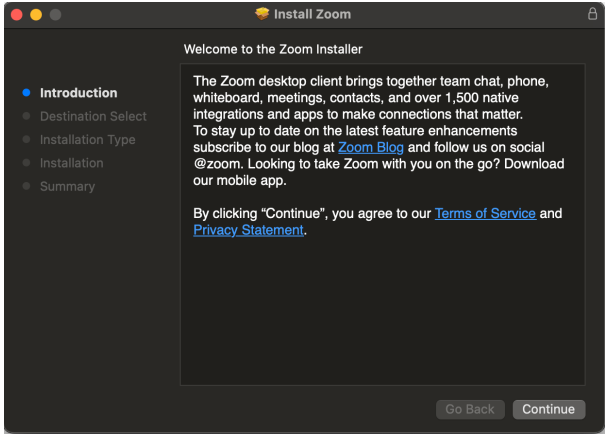


PKG is an XAR archive format.

It is commonly used for software installation on macOS.

A user can install macOS pkg file by

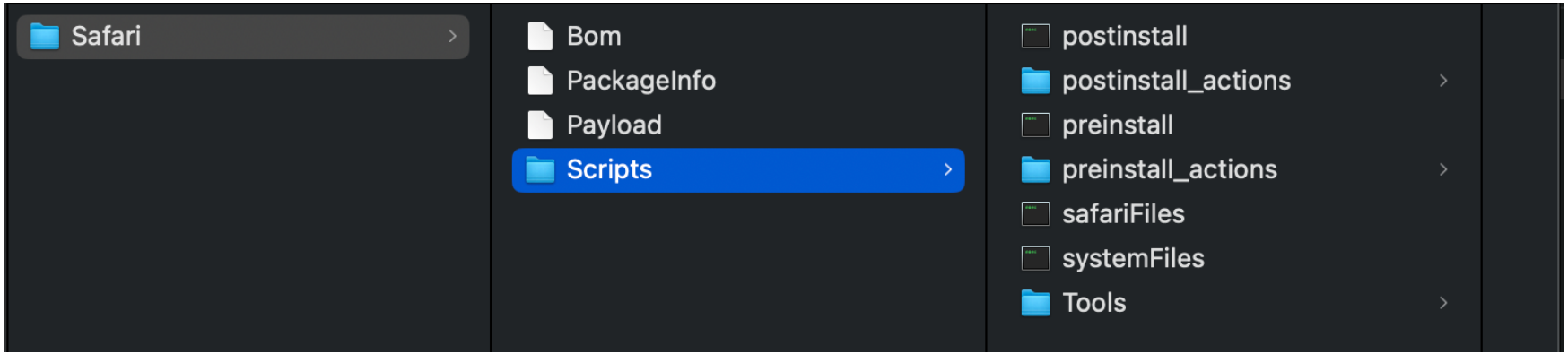
- Clicking the pkg file and following the instructions
- Running the installer command




```
INSTALLER(8)                System Manager's Manual                INSTALLER(8)

NAME
     installer – system software and package installer tool.
```

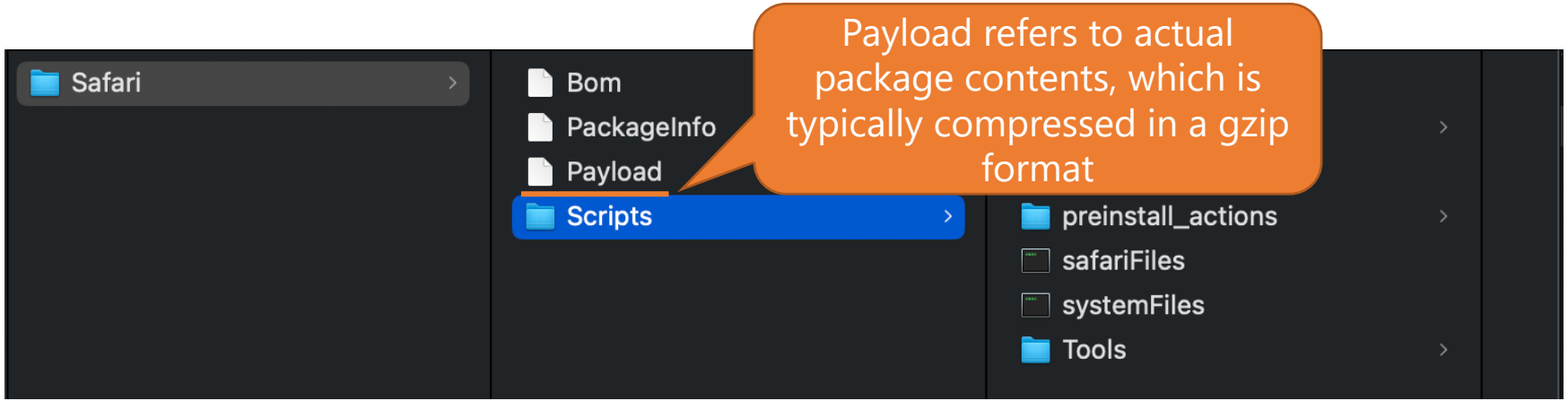
macOS PKG components

pkgutil --expand Safari16.5BigSurAuto.pkg /tmp/Safari



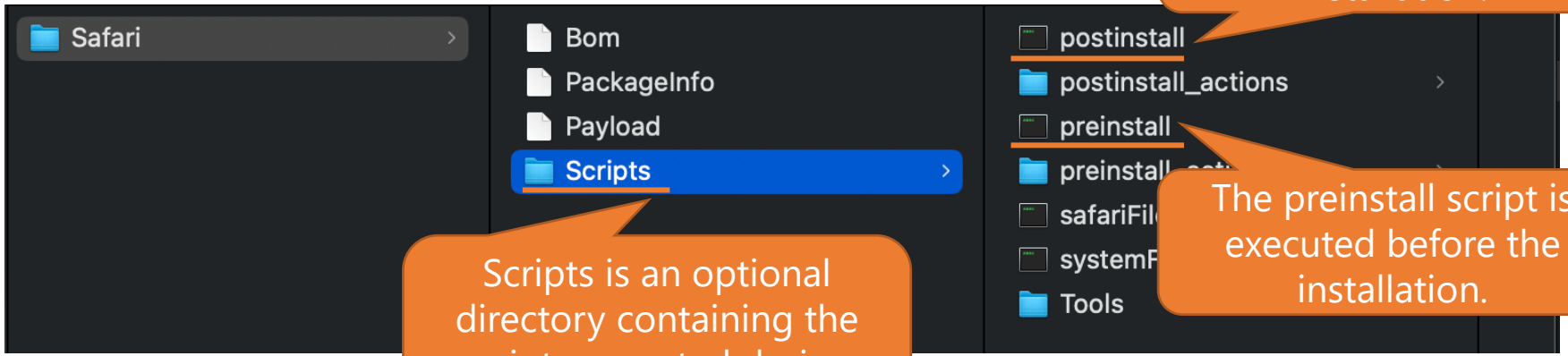
macOS PKG components

pkgutil --expand Safari16.5BigSurAuto.pkg /tmp/Safari



macOS PKG components

pkgutil --expand Safari16.5BigSurAuto.pkg /tr



The postinstall script is executed after the installation.

The preinstall script is executed before the installation.

Scripts is an optional directory containing the scripts executed during the installation.

Pre/postinstall scripts

These scripts of an Apple-signed pkg **are executed by system_installd**

Recall that system_installd has com.apple.rootless.install.heritable

- The scripts can bypass SIP filesystem checks!

Vulnerabilities in these scripts lead to SIP bypass.

Let’s hunt for bugs in the pre/postinstall scripts of Apple-signed pkg.

Case study 1: CVE-2023-23533

The postinstall script of macOS InstallAssistant.pkg

```
#!/bin/bash

SHARED_SUPPORT_PATH="${3}Applications/Install macOS Ventura beta.app/Contents/SharedSupport"
/bin/mkdir -p "${SHARED_SUPPORT_PATH}"
/bin/chmod 0755 "${SHARED_SUPPORT_PATH}"

SOURCE_DEVICE=$(/usr/bin/stat -n -f '%d' "${PACKAGE_PATH}")
TARGET_DEVICE=$(/usr/bin/stat -n -f '%d' "${SHARED_SUPPORT_PATH}")
if [ ${SOURCE_DEVICE} -eq ${TARGET_DEVICE} ]; then
    echo "Linking ${PACKAGE_PATH} into ${SHARED_SUPPORT_PATH}"
    /bin/ln -fFh "${PACKAGE_PATH}" "${SHARED_SUPPORT_PATH}/SharedSupport.dmg"
    /bin/chmod 0644 "${SHARED_SUPPORT_PATH}/SharedSupport.dmg"
    /usr/sbin/chown -R root:wheel "${SHARED_SUPPORT_PATH}/SharedSupport.dmg"
else
    echo "${PACKAGE_PATH} on different device than ${SHARED_SUPPORT_PATH} ... copying"
    /bin/cp "${PACKAGE_PATH}" "${SHARED_SUPPORT_PATH}/SharedSupport.dmg"
fi

/usr/bin/chflags -h norestricted "${SHARED_SUPPORT_PATH}/SharedSupport.dmg"
```

Case study 1: CVE-2023-23533

The postinstall script of macOS InstallAssistant.pkg

```
#!/bin/bash

SHARED_SUPPORT_PATH="${3}Applications/Install macOS Ventura beta.app/Contents/SharedSupport"
/bin/mkdir -p "${SHARED_SUPPORT_PATH}"
/bin/chmod 0755 "${SHARED_SUPPORT_PATH}"

SOURCE_DEVICE=$(/usr/bin/stat -n -f '%d' "${PACKAGE_PATH}")
TARGET_DEVICE=$(/usr/bin/stat -n -f '%d' "${SHARED_SUPPORT_PATH}")
if [ ${SOURCE_DEVICE} -eq ${TARGET_DEVICE} ]; then
    echo "Linking ${PACKAGE_PATH} into ${SHARED_SUPPORT_PATH}"
    /bin/ln -fFh "${PACKAGE_PATH}" "${SHARED_SUPPORT_PATH}/SharedSupport.dmg"
    /bin/chmod 0644 "${SHARED_SUPPORT_PATH}/SharedSupport.dmg"
    /usr/sbin/chown -R root:wheel "${SHARED_SUPPORT_PATH}/SharedSupport.dmg"
else
    echo "${PACKAGE_PATH} on different device than ${SHARED_SUPPORT_PATH} ... copying"
    /bin/cp "${PACKAGE_PATH}" "${SHARED_SUPPORT_PATH}/SharedSupport.dmg"
                                -h norestricted "${SHARED_SUPPORT_PATH}/SharedSu
```

Path of this pkg

A user in an admin group can modify this file

Case study 1: CVE-2023-23533

The postinstall script of macOS InstallAssistant.pkg

```
#!/bin/bash

SHARED_SUPPORT_PATH="${3}Applications/Install macOS Ventura beta.app/Contents/SharedSupport"
/bin/mkdir -p "${SHARED_SUPPORT_PATH}"
/bin/chmod 0755 "${SHARED_SUPPORT_PATH}"

SOURCE_DEVICE=$(/usr/bin/stat -n -f '%d' "${PACKAGE_PATH}")
TARGET_DEVICE=$(/usr/bin/stat -n -f '%d' "${SHARED_SUPPORT_PATH}")
if [ ${SOURCE_DEVICE} -eq ${TARGET_DEVICE} ]; then
    echo "Linking ${PACKAGE_PATH} into ${SHARED_SUPPORT_PATH}"
    /bin/ln -fFh "${PACKAGE_PATH}" "${SHARED_SUPPORT_PATH}/SharedSupport.dmg"
    /bin/chmod 0644 "${SHARED_SUPPORT_PATH}/SharedSupport.dmg"
    /usr/sbin/chown -R root:wheel "${SHARED_SUPPORT_PATH}/SharedSupport.dmg"
else
    echo "${PACKAGE_PATH} on different device than ${SHARED_SUPPORT_PATH} ... copying"
    /bin/cp "${PACKAGE_PATH}" "${SHARED_SUPPORT_PATH}/SharedSupport.dmg"

ted "${SHARED_SUPPORT_PATH}/Sha
```

We can control this file after the package extraction

If we change this file to a symlink, cp follows the symlink

Case study 1: CVE-2023-23533

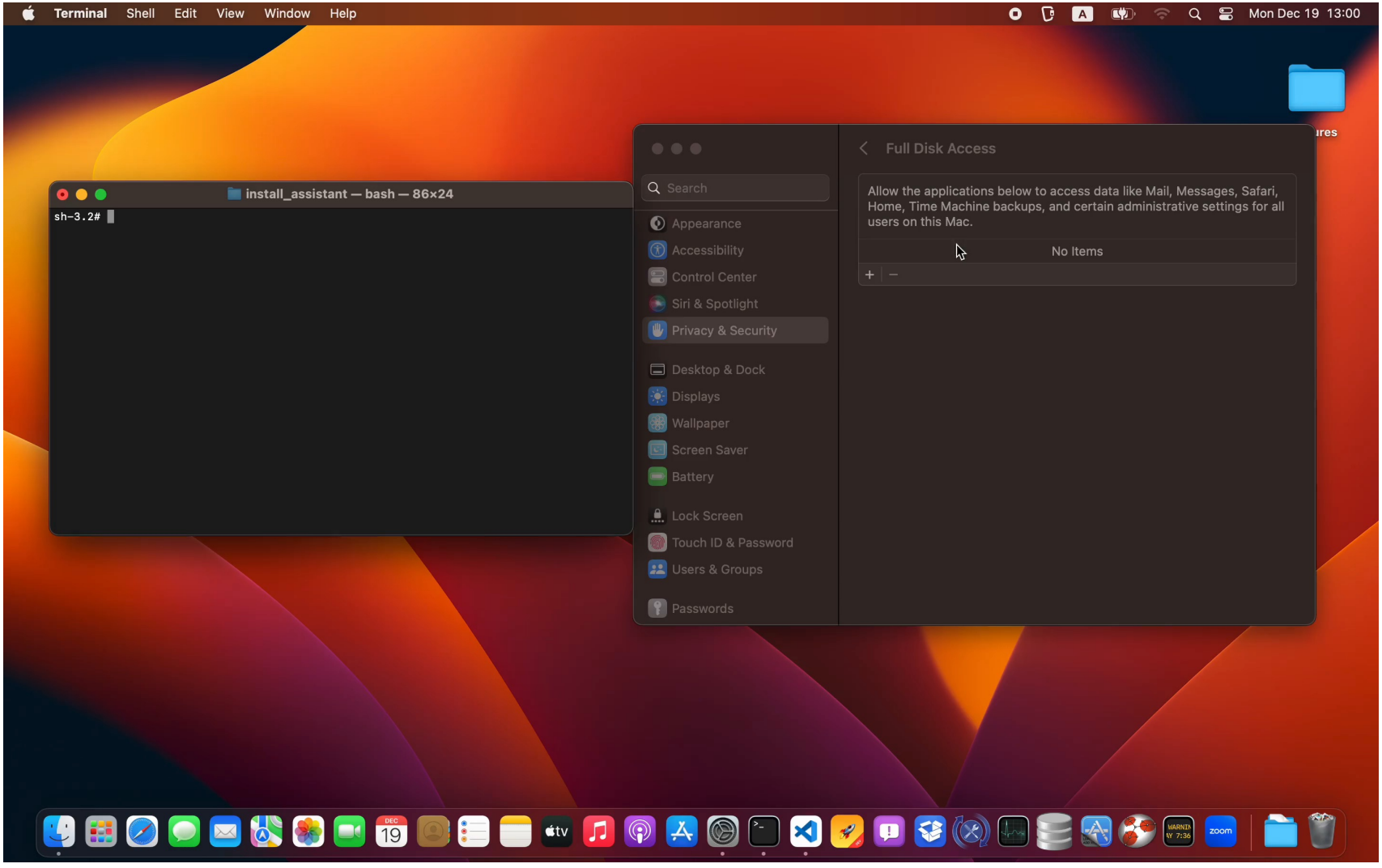
The postinstall script of macOS InstallAssistant.pkg

```
#!/bin/bash

SHARED_SUPPORT_PATH="${3}Applications/Install macOS Ventura beta.app/Contents/SharedSupport"
/bin/mkdir -p "${SHARED_SUPPORT_PATH}"
/bin/chmod 0755 "${SHARED_SUPPORT_PATH}"

SOURCE_DEVICE=$(/usr/bin/stat -n -f '%d' "${PACKAGE_PATH}")
TARGET_DEVICE=$(/usr/bin/stat -n -f '%d' "${SHARED_SUPPORT_PATH}")
if [ ${SOURCE_DEVICE} -eq ${TARGET_DEVICE} ]; then
    echo "Linking
    /bin/ln -fFh
    /bin/chmod 06
    /usr/sbin/cho
else
    echo "${PACKAGE_PATH} on different device than ${SHARED_SUPPORT_PATH} ... copying"
    /bin/cp "${PACKAGE_PATH}" "${SHARED_SUPPORT_PATH}/SharedSupport.dmg"
```

We can control the src and dst files, and thus, we can overwrite an SIP-protected file with our controllable data.

We can control this file after the package extraction

If we change this file to a symlink, cp follows the symlink


Terminal
Shell
Edit
View
Window
Help
Mon Dec 19 13:00
install_assistant — bash — 86×24
sh-3.2#
Search
Appearance
Accessibility
Control Center
Siri & Spotlight
Privacy & Security
Desktop & Dock
Displays
Wallpaper
Screen Saver
Battery
Lock Screen
Touch ID & Password
Users & Groups
Passwords
Full Disk Access
Allow the applications below to access data like Mail, Messages, Safari, Home, Time Machine backups, and certain administrative settings for all users on this Mac.
No Items

Apple's fix

Apple addressed this issue by updating Sandbox.kext

- Apple gave me a generous bounty :)

Sandbox

Available for: macOS Ventura

Impact: An app may be able to modify protected parts of the file system

Description: A logic issue was addressed with improved checks.

CVE-2023-23533: Mickey Jin (@patch1t), Koh M. Nakagawa of FFRI Security, Inc., and Csaba Fitzl (@theevilbit) of Offensive Security

https://support.apple.com/en-us/HT213670

Case study 2: CVE-2023-29166

The postinstall script of MXFPlugins.pkg

```
for TARGET_DIR in "$HOME/Library/Application Support/Compressor/Settings/MXF" \
                  "$HOME/Library/Application Support/Compressor/MXF"; do

    # Generate a unique target directory name to avoid overwriting
    # a pre-existing directory that the user may have created or customized.
    SKIP_INSTALL=0
    N=0
    UNIQUE_TARGET_DIR="$TARGET_DIR"
    while [ -e "$UNIQUE_TARGET_DIR" ]; do
    # If the target exists and is identical, no need to install anything.
    /usr/bin/diff -rq "$SOURCE_DIR" "$UNIQUE_TARGET_DIR"
    if [ $? -eq 0 ]; then
        SKIP_INSTALL=1
        break
    fi
    N=`/bin/expr $N + 1`
    UNIQUE_TARGET_DIR="$TARGET_DIR $N"
    done

    # Install to the target directory
    if [ $SKIP_INSTALL -eq 0 ]; then
    /usr/bin/sudo /bin/mkdir -p "$UNIQUE_TARGET_DIR"
    /usr/bin/sudo /bin/cp -a "$SOURCE_DIR/" "$UNIQUE_TARGET_DIR/"
    fi

done
```

Case study 2: CVE-2023-29166

The postinstall script of MXFPlugins.pkg

```
for TARGET_DIR in "$HOME/Library/Application Support/Compressor/Settings/MXF" \
                  "$HOME/Library/Application Support/Compressor/MXF"; do

    # Generate a unique target directory name to avoid overwriting
    # a pre-existing directory that the user may have created or customized.
    SKIP_INSTALL=0
    N=0
    UNIQUE_TARGET_DIR="$TARGET_DIR"
    while [ -e "$UNIQUE_TARGET_DIR" ]; do
    # If the target exists and is identical, no need to install anything.
    /usr/bin/diff -rq "$SOURCE_DIR" "$UNIQUE_TARGET_DIR"
    if [ $? -eq 0 ]; then
        SKIP_INSTALL=1
        break
    fi
    N=`/bin/expr $N + 1`
    UNIQUE_TARGET_DIR="$TARGET_DIR $N"
    done

    # Install to the target directory
    if [ $SKIP_INSTALL -eq 0 ]; then
    /usr/bin/sudo /bin/mkdir -p "$UNIQUE_TARGET_DIR"
    /usr/bin/sudo /bin/cp -a "$SOURCE_DIR/" "$UNIQUE_TARGET_DIR/"
    fi

done
```

A similar issue exists here.

We can control both the src ($SOURCE_DIR) and dst ($UNIQUE_TARGET_DIR) directories.

Case study 2: CVE-2023-29166

The postinstall script of MXFPlugins.pkg

```
for TARGET_DIR in "$HOME/Library/Application Support/Compressor/Settings/MXF" \
                  "$HOME/Library/Application Support/Compressor/MXF"; do

    # Generate a unique target directory name to avoid overwriting
    # a pre-existing directory that the user may have created or customized.
    SKIP_INSTALL=0
    N=0
    UNIQUE_TARGET_DIR="$TARGET_DIR"
    while [ -e "$UNIQUE_TARGET_DIR" ]; do
    # If the target exists and is identical, no need to install anything.
    /usr/bin/diff -rq "$SOURCE_DIR" "$UNIQUE_TARGET_DIR"
    if [ $? -eq 0 ]; then
        SKIP_INSTALL=1
        break
    fi
    N=`/bin/expr $N + 1`
    UNIQUE_TARGET_DIR="$TARGET_DIR $N"
    done

    # Install to the target directory
    if [ $SKIP_INSTALL -eq 0 ]; then
    /usr/bin/sudo /bin/mkdir -p "$UNIQUE_TARGET_DIR"
    /usr/bin/sudo /bin/cp -a "$SOURCE_DIR/" "$UNIQUE_TARGET_DIR/"
    fi

done
```

However, we cannot pre-create the dst directory before the copy.

If the dst directory exists, the postinstall script creates a new dst directory with a different name.

Case study 2: CVE-2023-29166

The postinstall script of MXFPlugins.pkg

```
for TARGET_DIR in "$HOME/Library/Application Support/Compressor/Settings/MXF" \
                  "$HOME/Library/Application Support/Compressor/MXF"; do

    # Generate a unique target directory name to avoid overwriting
    # a pre-existing directory that the user may have created or customized.
    SKIP_INSTALL=0
    N=0
    UNIQUE_TARGET_DIR="$TARGET_DIR"
    while [ -e "$UNIQUE_TARGET_DIR" ]; do
    # If the target exists and is identical, no need to install anything.
    /usr/bin/diff -rq "$SOURCE_DIR" "$UNIQUE_TARGET_DIR"
    if [ $? -eq 0 ]; then
        SKIP_INSTALL=1
        break
    fi
    N=`/bin/expr $N + 1`
    UNIQUE_TARGET_DIR="$TARGET_DIR $N"
    done

    # Install to the target directory
    if [ $SKIP_INSTALL -eq 0 ]; then
    /usr/bin/sudo /bin/mkdir -p "$UNIQUE_TARGET_DIR"
    /usr/bin/sudo /bin/cp -a "$SOURCE_DIR/" "$UNIQUE_TARGET_DIR/"
    fi

done
```

Thus, we need to add a symlink to the dst directory after this directory is created, but the time window is too narrow

Winning the race

The postinstall script of MXFPlugins.pkg

```
for TARGET_DIR in "$HOME/Library/Application Support/Compressor/Settings/MXF" \
                  "$HOME/Library/Application Support/Compressor/MXF"; do

    # Generate a unique target directory name to avoid overwriting
    # a pre-existing directory that the user may have created or customized.
    SKIP_INSTALL=0
    N=0
    UNIQUE_TARGET_DIR="$TARGET_DIR"
    while [ -e "$UNIQUE_TARGET_DIR" ]; do
    # If the target exists and is identical, no need to install anything.
    /usr/bin/diff -rq "$SOURCE_DIR" "$UNIQUE_TARGET_DIR"
    if [ $? -eq 0 ]; then
        SKIP_INSTALL=1
        break
    fi
    N=`/bin/expr $N + 1`
    UNIQUE_TARGET_DIR="$TARGET_DIR $N"
    done

    # Install to the target directory
    if [ $SKIP_INSTALL -eq 0 ]; then
    /usr/bin/sudo /bin/mkdir -p "$UNIQUE_TARGET_DIR"
    /usr/bin/sudo /bin/cp -a "$SOURCE_DIR/" "$UNIQUE_TARGET_DIR/"
    fi

done
```

A dmg is mounted on $SOURCE_DIR

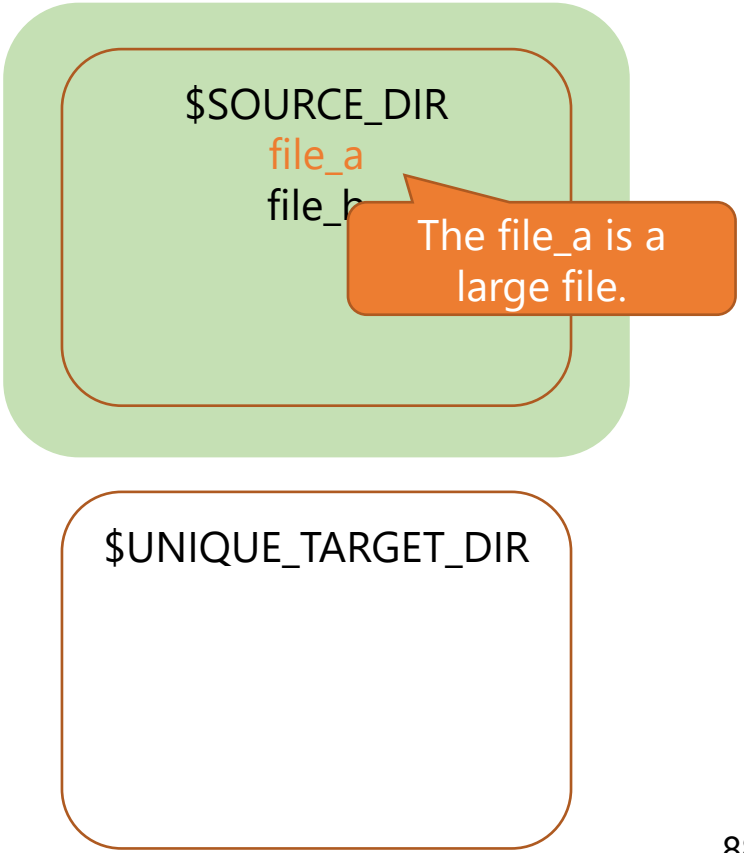
$SOURCE_DIR
file_a
file_b

$UNIQUE_TARGET_DIR

Winning the race

The postinstall script of MXFPlugins.pkg

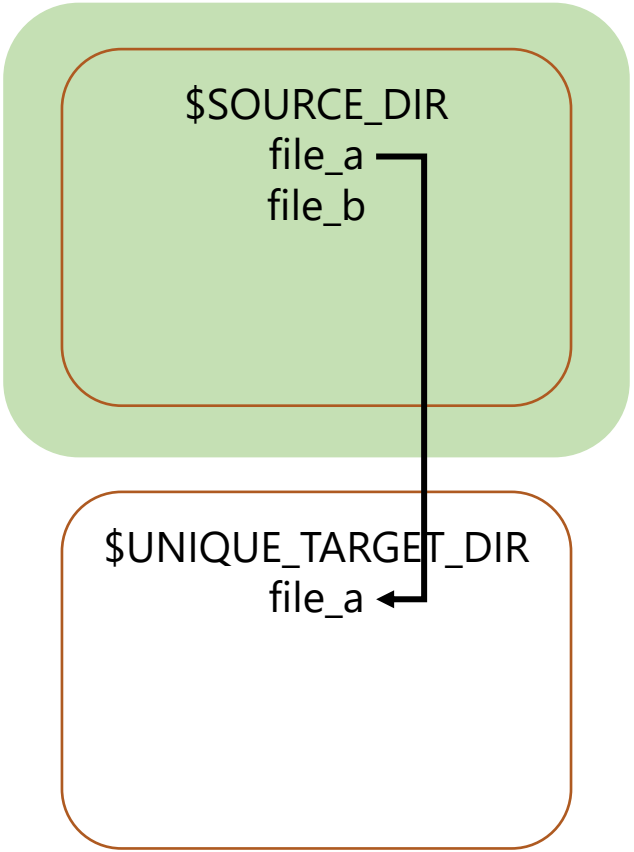
```
for TARGET_DIR in "$HOME/Library/Application Support/Compressor/Settings/MXF" \
                  "$HOME/Library/Application Support/Compressor/MXF"; do

    # Generate a unique target directory name to avoid overwriting
    # a pre-existing directory that the user may have created or customized.
    SKIP_INSTALL=0
    N=0
    UNIQUE_TARGET_DIR="$TARGET_DIR"
    while [ -e "$UNIQUE_TARGET_DIR" ]; do
    # If the target exists and is identical, no need to install anything.
    /usr/bin/diff -rq "$SOURCE_DIR" "$UNIQUE_TARGET_DIR"
    if [ $? -eq 0 ]; then
        SKIP_INSTALL=1
        break
    fi
    N=`/bin/expr $N + 1`
    UNIQUE_TARGET_DIR="$TARGET_DIR $N"
    done

    # Install to the target directory
    if [ $SKIP_INSTALL -eq 0 ]; then
    /usr/bin/sudo /bin/mkdir -p "$UNIQUE_TARGET_DIR"
    /usr/bin/sudo /bin/cp -a "$SOURCE_DIR/" "$UNIQUE_TARGET_DIR/"
    fi

done
```



Winning the race

The postinstall script of MXFPlugins.pkg

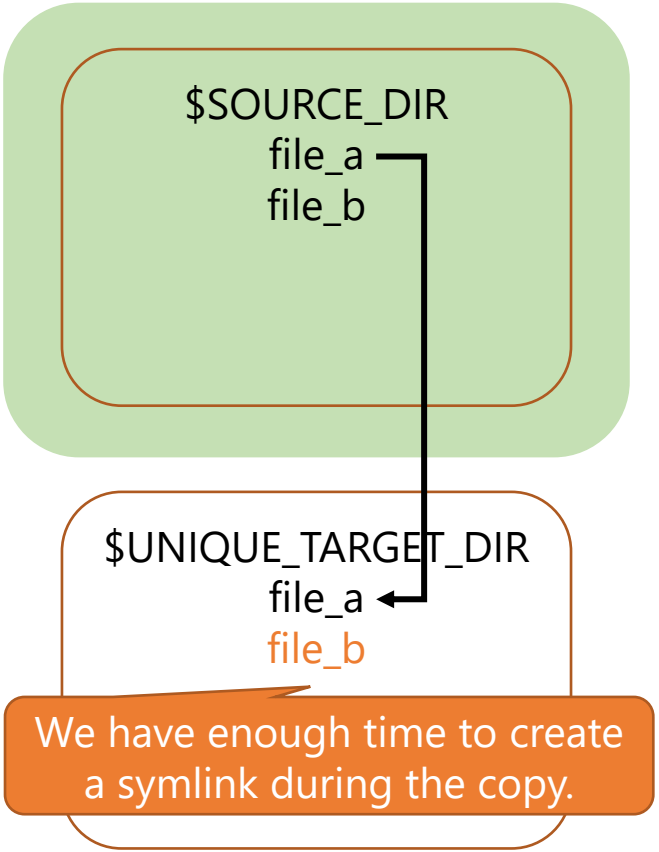
```
for TARGET_DIR in "$HOME/Library/Application Support/Compressor/Settings/MXF" \
                  "$HOME/Library/Application Support/Compressor/MXF"; do

    # Generate a unique target directory name to avoid overwriting
    # a pre-existing directory that the user may have created or customized.
    SKIP_INSTALL=0
    N=0
    UNIQUE_TARGET_DIR="$TARGET_DIR"
    while [ -e "$UNIQUE_TARGET_DIR" ]; do
    # If the target exists and is identical, no need to install anything.
    /usr/bin/diff -rq "$SOURCE_DIR" "$UNIQUE_TARGET_DIR"
    if [ $? -eq 0 ]; then
        SKIP_INSTALL=1
        break
    fi
    N=`/bin/expr $N + 1`
    UNIQUE_TARGET_DIR="$TARGET_DIR $N"
    done

    # Install to the target directory
    if [ $SKIP_INSTALL -eq 0 ]; then
    /usr/bin/sudo /bin/mkdir -p "$UNIQUE_TARGET_DIR"
    /usr/bin/sudo /bin/cp -a "$SOURCE_DIR/" "$UNIQUE_TARGET_DIR/"
    fi

done
```



Winning the race

The postinstall script of MXFPlugins.pkg

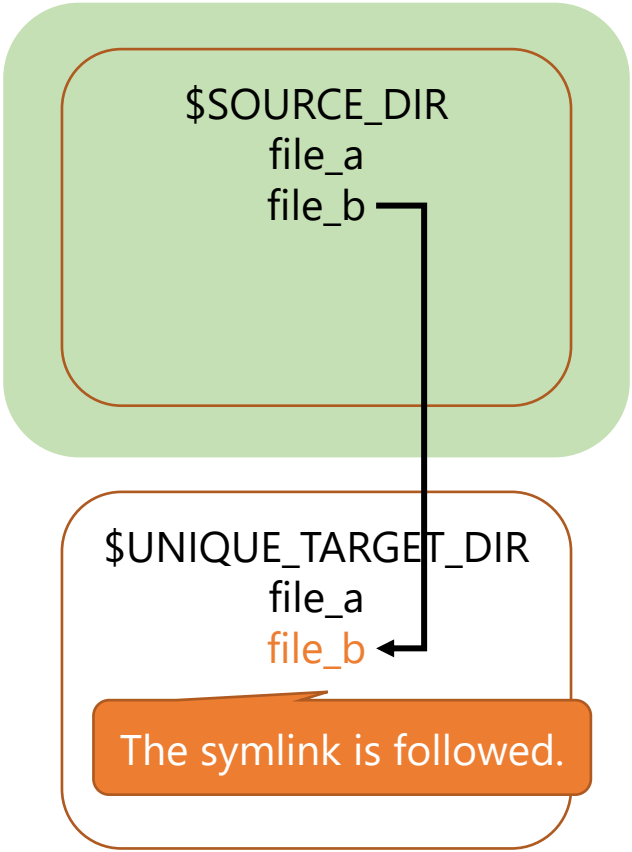
```
for TARGET_DIR in "$HOME/Library/Application Support/Compressor/Settings/MXF" \
                  "$HOME/Library/Application Support/Compressor/MXF"; do

    # Generate a unique target directory name to avoid overwriting
    # a pre-existing directory that the user may have created or customized.
    SKIP_INSTALL=0
    N=0
    UNIQUE_TARGET_DIR="$TARGET_DIR"
    while [ -e "$UNIQUE_TARGET_DIR" ]; do
    # If the target exists and is identical, no need to install anything.
    /usr/bin/diff -rq "$SOURCE_DIR" "$UNIQUE_TARGET_DIR"
    if [ $? -eq 0 ]; then
        SKIP_INSTALL=1
        break
    fi
    N=`/bin/expr $N + 1`
    UNIQUE_TARGET_DIR="$TARGET_DIR $N"
    done

    # Install to the target directory
    if [ $SKIP_INSTALL -eq 0 ]; then
    /usr/bin/sudo /bin/mkdir -p "$UNIQUE_TARGET_DIR"
    /usr/bin/sudo /bin/cp -a "$SOURCE_DIR/" "$UNIQUE_TARGET_DIR/"
    fi

done
```



Winning the race

The postinstall script of MXFPlugins.pkg

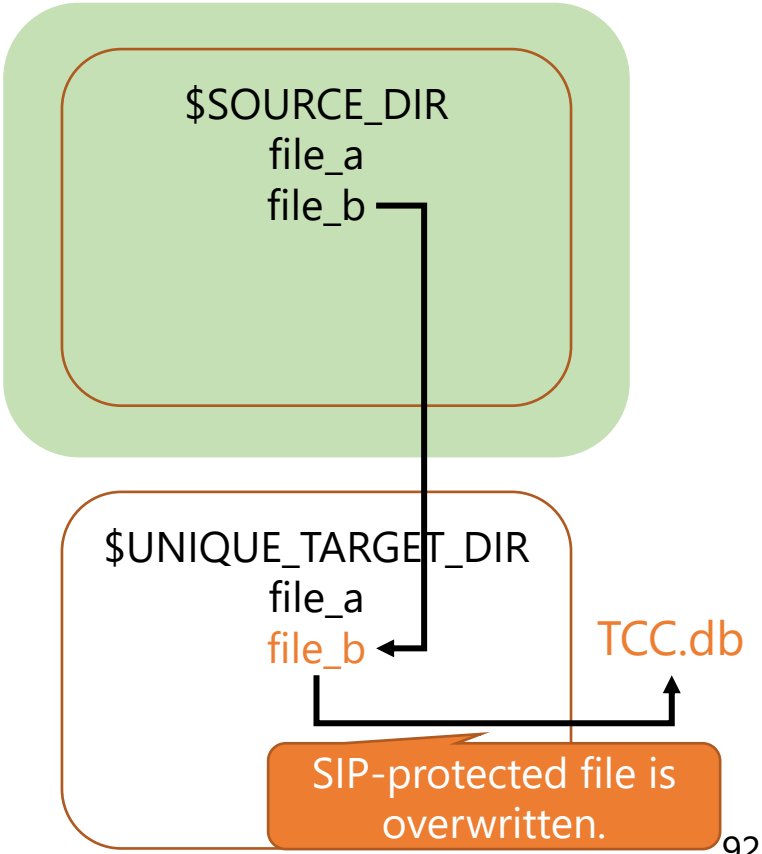
```
for TARGET_DIR in "$HOME/Library/Application Support/Compressor/Settings/MXF" \
                  "$HOME/Library/Application Support/Compressor/MXF"; do

    # Generate a unique target directory name to avoid overwriting
    # a pre-existing directory that the user may have created or customized.
    SKIP_INSTALL=0
    N=0
    UNIQUE_TARGET_DIR="$TARGET_DIR"
    while [ -e "$UNIQUE_TARGET_DIR" ]; do
    # If the target exists and is identical, no need to install anything.
    /usr/bin/diff -rq "$SOURCE_DIR" "$UNIQUE_TARGET_DIR"
    if [ $? -eq 0 ]; then
        SKIP_INSTALL=1
        break
    fi
    N=`/bin/expr $N + 1`
    UNIQUE_TARGET_DIR="$TARGET_DIR $N"
    done

    # Install to the target directory
    if [ $SKIP_INSTALL -eq 0 ]; then
    /usr/bin/sudo /bin/mkdir -p "$UNIQUE_TARGET_DIR"
    /usr/bin/sudo /bin/cp -a "$SOURCE_DIR/" "$UNIQUE_TARGET_DIR/"
    fi

done
```



Winning the race

The postinstall script of MXFPlugins.pkg

```
for TARGET_DIR in "$HOME/Library/Application Support/Compressor/Settings/MXF" \
                  "$HOME/Library/Application Support/Compressor/MXF"; do

    # Generate a unique target directory name to avoid overwriting
    # a pre-existing directory that the user may have created or customized.
    SKIP_INSTALL=0
    N=0
    UNIQUE_TARGET_DIR="$TARGET_DIR"
    while [ -e "$UNIQUE_TARGET_DIR" ]; do
    # If the target exists and is identical, no need to install anything.
    /usr/bin/diff -rq "$SOURCE_DIR" "$UNIQUE_TARGET_DIR"
    if [ $? -eq 0 ]; then
        SKIP_INSTALL=1
        break
    fi
    N=`/bin/expr $N + 1`
    UNIQUE_TARGET_DIR="$TARGET_DIR $N"
    done

    # Install to the target directory
    if [ $SKIP_INSTALL -eq 0 ]; then
    /usr/bin/sudo /bin/mkdir -p "$UNIQUE_TARGET_DIR"
    /usr/bin/sudo /bin/cp -a "$SOURCE_DIR/" "$UNIQUE_TARGET_DIR/"
    fi

done
```



Apple's fix

Apple updated the pkg and changed the postinstall script.

```
# Install to the target directory
if [ $SKIP_INSTALL -eq 0 ]; then
    /usr/bin/sudo /bin/mkdir -p "$UNIQUE_TARGET_DIR"
    /usr/bin/sudo /usr/bin/rsync "$SOURCE_DIR/*" "$UNIQUE_TARGET_DIR/"
fi
```

rsync is used for copying, and hence, the symlink is no longer followed.

Pro Video Formats

Available for: macOS 10.14.5 and later

Impact: A user may be able to elevate privileges

Description: A logic issue was addressed with improved state management.

CVE-2023-29166: Koh M. Nakagawa (@tsunek0h)

https://support.apple.com/en-us/HT213882

And more…

Some issues are still being addressed…

My thoughts

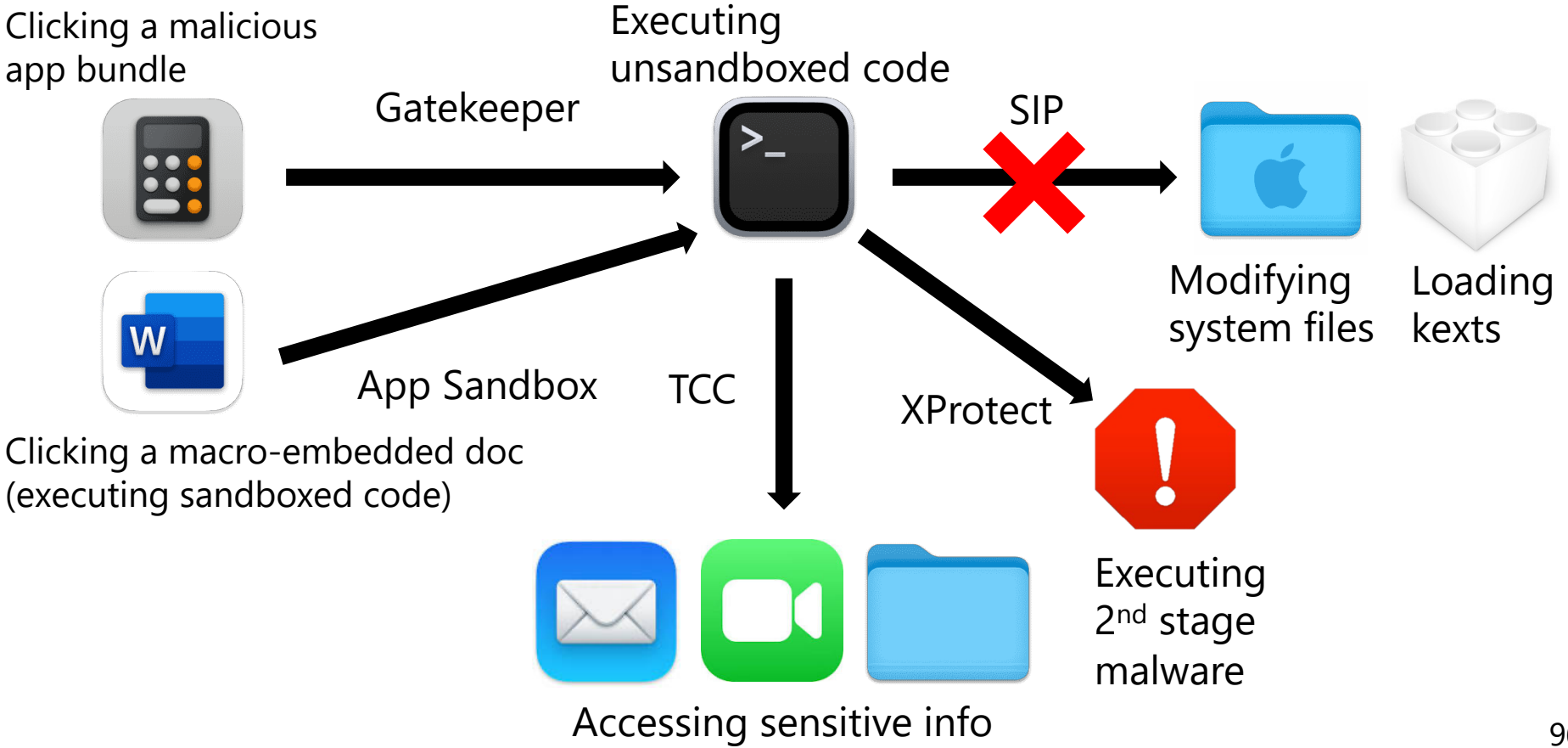
The pre/postinstall scripts of all Apple-signed pkgs do not require the SIP-bypass primitive.

- I think some pkgs should be executed by installd (not system_installd).

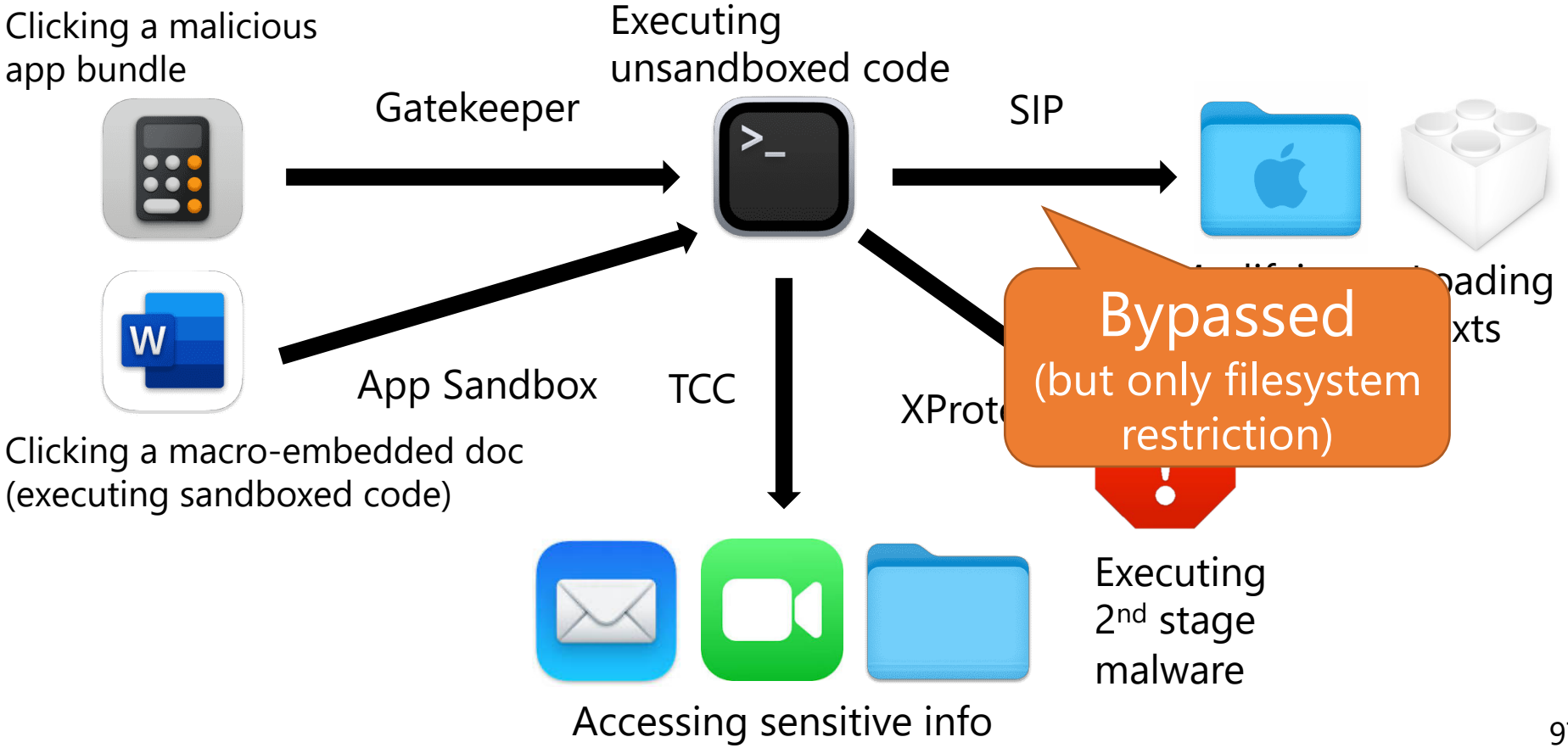
Apple should review the code of the pre/postinstall scripts of Apple-signed pkgs.

- They appear not to pay attention to these, even though vulnerabilities in these scripts are dangerous.

Overview of macOS security & privacy mechanisms



Overview of macOS security & privacy mechanisms



Summary & Takeaways

Summary

Overview of macOS security & privacy mechanisms

New bypass techniques for these mechanisms

Typical 3rd party vulns that allow an attacker to bypass these mechanisms

Takeaways

For Red Teamers

PoC code is available on GitHub (https://github.com/FFRI/PoC-public)

- This code will be helpful for future red team exercises targeting macOS.

For Security Researchers

Logic bugs are quite powerful for bypassing various security & privacy mechanisms on macOS.

- As PAC enforcement is enabled on Apple Silicon Mac, exploiting memory corruption vulnerabilities is becoming more difficult.

For macOS App Developers

Please check whether your Electron-based app is built with secure build configurations.

- An app built with default configurations has vulnerabilities that allow an attacker to bypass TCC and Gatekeeper.

Takeaways

For Red Teamers

PoC code is available on GitHub (https://github.com/FFRI/PoC-public)

- This code will be helpful for future red team exercises targeting macOS.

For Security Researchers

Logic bugs are quite powerful for bypassing various security & privacy mechanisms on macOS.

- As PAC enforcement is enabled on Apple Silicon Mac, exploiting memory corruption vulnerabilities is becoming more difficult.

For macOS App Developers

Please check v uild configuration

- An app built ker to bypass TCC and Gate

Last but not least, please keep your macOS up to date!

Q&A

𝕏 @tsunek0h