
# Dive into Apple UserFS (Userspace Filesystem)

pattern-f (@pattern_F_)
Ant Security Light-Year Lab

# About me

- pattern-f (@pattern_F_) on Twitter
- Security researcher of Ant Security Light-Year Lab
- Focus on iOS & macOS security
- speaker of Black Hat ASIA & USA 2021

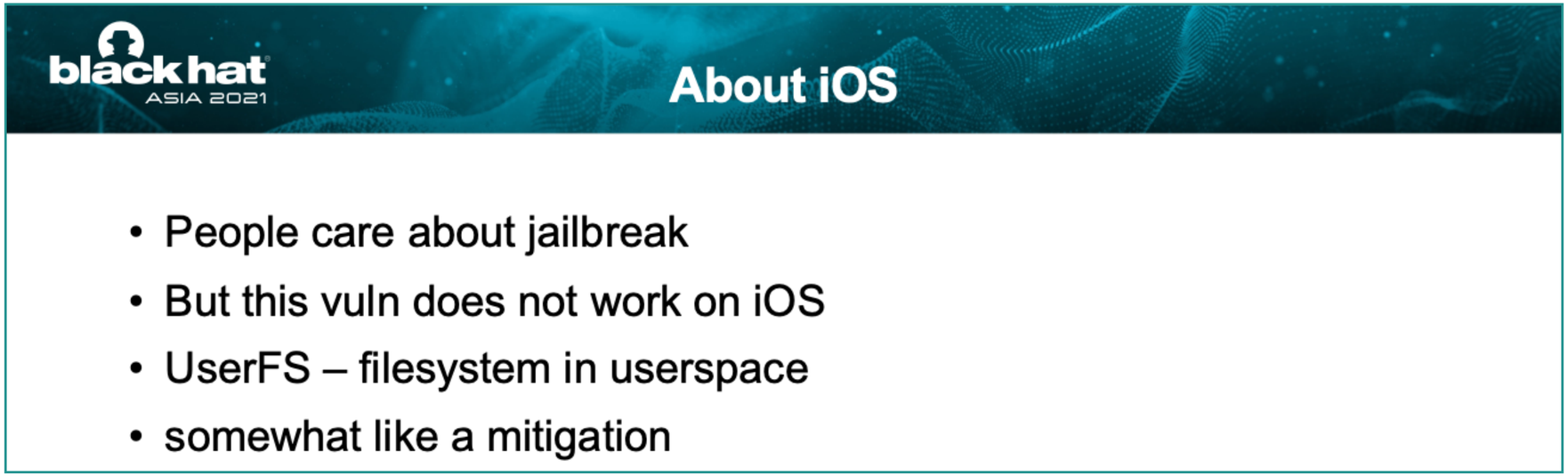
# Background

- my previous talk at Black Hat ASIA 2021


- Shared a vulnerability named xattr-oob-swap (CVE-2020-27904) [link]
- It’s a filesystem bug, can get tfp0 on macOS 10.15.x and below.
- iOS & macOS share code base, it should work on iOS too, but…

# Background

- It doesn’t crash iOS. Maybe UserFS stops it.
- I don’t know the mechanism of UserFS, so I’m not sure.



# Background

- This vulnerability doesn’t affect iOS, really?
- Add it to TODO list

# Background

- This vulnerability doesn’t affect iOS, ~~really?~~
- ~~Add it to TODO list~~

**File System**

Available for: iPhone 6s (all models), iPhone 7 (all models), iPhone SE (1st generation), iPad Pro (all models), iPad Air 2 and later, iPad 5th generation and later, iPad mini 4 and later, and iPod touch (7th generation)

Impact: An app may be able to break out of its sandbox

Description: This issue was addressed with improved checks.

CVE-2022-42861: pattern-f (@pattern_F_) of Ant Security Light-Year Lab

# What is UserFS

- We know that

## Files (Apple)

Article Talk

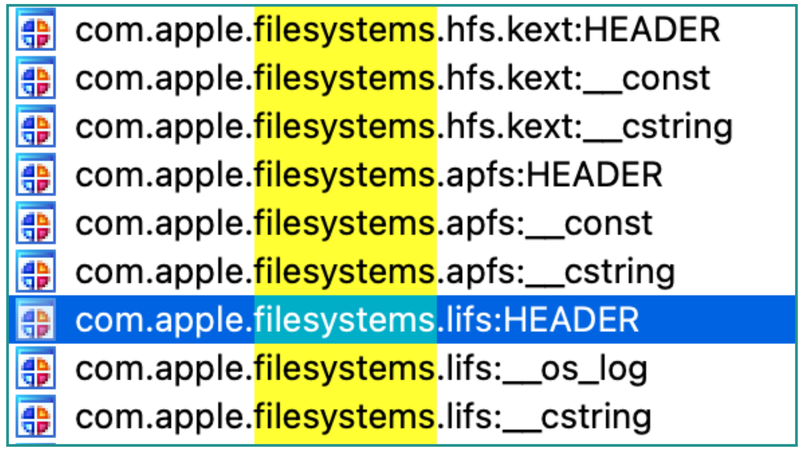
From Wikipedia, the free encyclopedia

**Files** is a file management app developed by Apple Inc. for devices that run iOS 11 and later releases of iOS and devices that run iPadOS. Discovered as a placeholder title in the App Store

- iPhone User Guide said: An external storage device must have only a single data partition, and it must be formatted as APFS, APFS (encrypted), macOS Extended (HFS+), exFAT (FAT64), FAT32, or FAT.

# What is UserFS

- iOS supports various file system
- But when analyzing the iOS kernel cache, I found only apfs & hfs support were present.
- No msdos.kext (fat32, exfat) at all



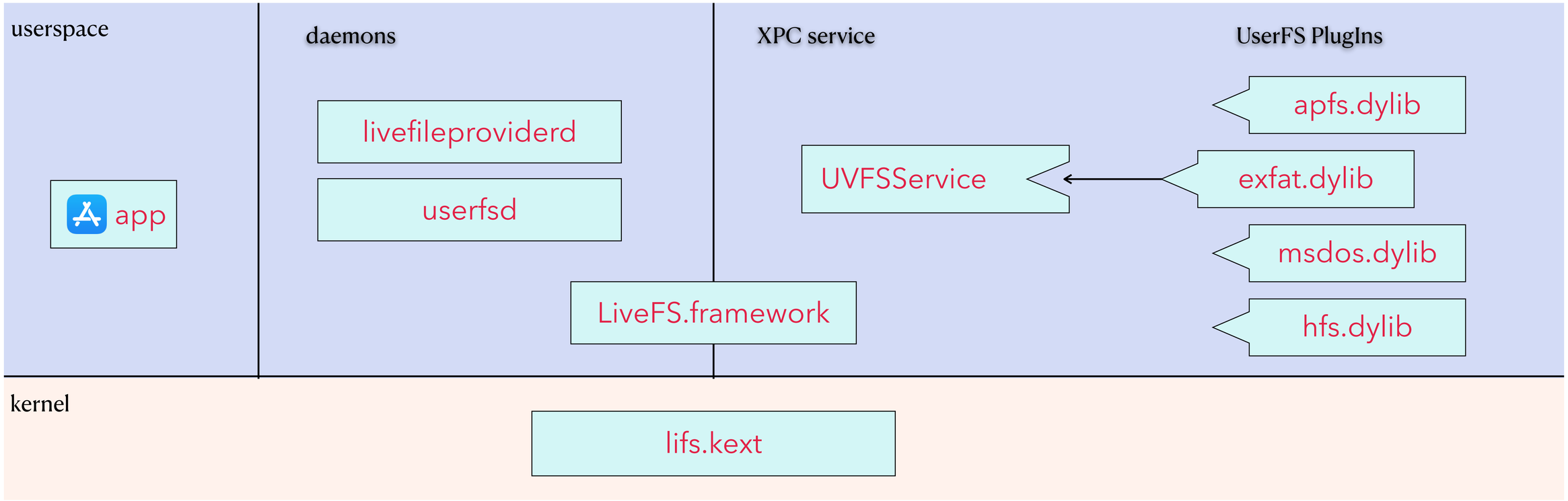
# What is UserFS

- Instead, apple implement them in user space, called UserFS plugins.
- But the plugins are not all about UserFS.

```
/System/Library/PrivateFrameworks/UserFS.framework/PlugIns/
├── liblivefiles.plugin.dummy.dylib
├── livefiles_apfs.dylib
├── livefiles_cs.dylib
├── livefiles_exfat.dylib
├── livefiles_hfs.dylib
└── livefiles_msdos.dylib
```

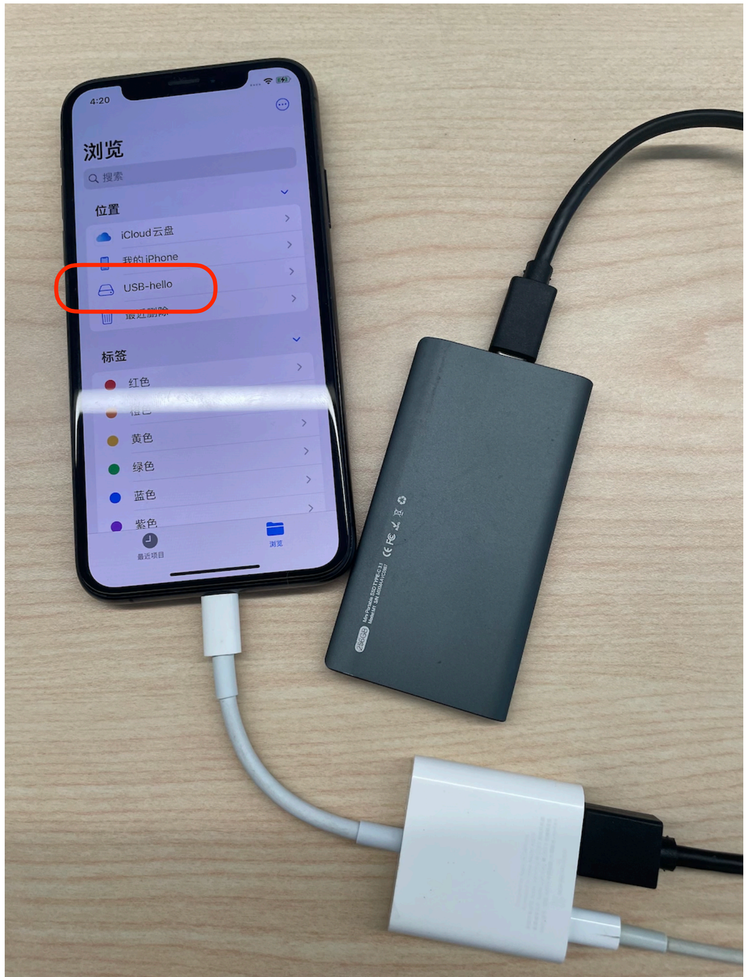
# What is UserFS

- UserFS is filesystem implemented in user space, with these components.
  - `kext, daemon, XPC service, PlugIn, and framework`



# What is UserFS

- UserFS only functions with external storage devices, so let’s started with connecting an external hard drive to iPhone.



# When connecting a USB drive to iPhone

- launchd registers a LaunchEvent “usb.device.attached” for userfsd

```
<key>/System/Library/LaunchDaemons/com.apple.filesystems.userfsd.plist</key>
<dict>
    <key>LaunchEvents</key>
    <dict>
        <key>com.apple.iokit.matching</key>
        <dict>
            <key>com.apple.userfsd.usb.device.attached</key>
            <dict>
                <key>IOProviderClass</key>
                <string>IOBlockStorageDevice</string>
                <key>IOMatchLaunchStream</key>
                <true/>
                <key>IOParentMatch</key>
                <dict>
                    <key>IOProviderClass</key>
                    <string>IOUSBHostDevice</string>
                    <key>IOPropertyMatch</key>
                    <dict>
                        <key>removable</key>
                        <true/>
```

# When connecting a USB drive to iPhone

- userfsd daemon starts and handles this IOKit notification
- Make a xpc call to UVFSService

```
void handleIOKitNotifications(void)
{
    service_info = IOServiceMatching("IOMedia");
    IOServiceAddMatchingNotification(gNotifyPort, "IOServiceFirstMatch", service_info,
        IOMediaMatchingCallback, 0, &iterator);
}

void IOMediaMatchingCallback(void *refcon, io_iterator_t iterator)
{
    diskName = CFDictionaryGetValue(properties, CFSTR("BSD Name"));
    objc_msgSend(externalVolumeLiveFSService, "LiveMountAddDisk:reply:", diskName, reply_b);
}

void -[externalVolumeLiveFSServiceDelegate LiveMountService:addDisk:reply:]
{
    -[NSXPCConnection initWithServiceName:CFSTR("com.apple.filesystems.lifs.userfsd.UVFSService")];
    objc_msgSend(remoteObjectProxy, "createVolumesForTheDevice:how:withReply:", v30, v43, &v54);
}
```

/dev/disk3

Expected identifie

# When connecting a USB drive to iPhone

- Enumerate UserFS plugins and determine filesystem type of the disk.
- xpc call again, to livefileproviderd.

```
-[UVFSService createVolumesForTheDevice:how:withReply:]
{
    -[UVFSService CreateVolumesForTheDevice:how:] {
        rawDevice = -[LiveFSRawDevice initDeviceWithName:andError:]
        volumes = [rawDevice getVolumesFromDevice] {
            _fsPlugin = -[UVFSPlugin loadFileSystemDyLibPassingParameterDict:forShimPlugin:] {
                dyLibHandle = dlopen("/PrivateFrameworks/UserFS.framework/PlugIns/livefiles_%s.dylib")
                dlsym(dyLibHandle, "livefiles_shim_plugin_init")(&self->_FSOps)
            }
            FSOps = -[UVFSPlugin FSOps](self->_fsPlugin, "FSOps");
            FSOps->fsops_taste(self->_deviceFD);
            FSOps->fsops_scanvols(self->_deviceFD)
        }
        mountClient = +[LiveFSMountClient newClientForProvider:] {
            self->conn = connectionForServiceURL("machp://com.apple.filesystems.livefileproviderd");
        }
        [mountClient mountVolume:displayName:provider:domainError:on:how:] {
            [proxyObject LiveMounterMountVolume:displayName:provider:domainError:on:how:reply:]
        }
    }
```

# When connecting a USB drive to iPhone

- familiar command “/sbin/mount”, mount_lifs this time.

```
-[lifeFilesFPNFSDMounter LiveMounterMountVolume:displayName:provider:domainError:on:how:reply:]
{
    -[lifeFilesFPNFSDMounter LiveMounterReallyMountVolume:displayName:provider:domainError:on:how:reply:]
        ...
        -[mountEntry mount:] {
            posix_spawn(&pid, "/sbin/mount_lifs", actions, attr, { "/sbin/mount_lifs", ... }, environ)
        }
    }
}

/sbin/mount_lifs
{
    mount("lifs", dir, flags, data) {
        lifs.kext`lifs_mount(...)
    }
}
```

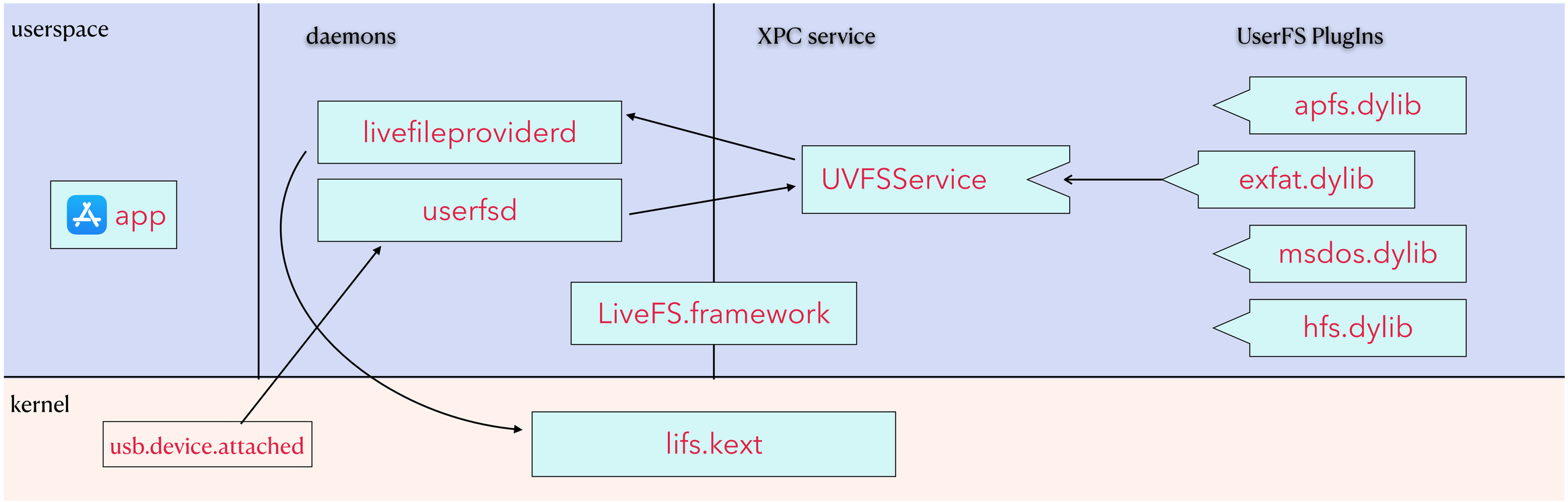
# When connecting a USB drive to iPhone

- Finally, we can access files on this exFAT formatted USB hard drive.
- /var/mobile/Library/LiveFiles/ is a dedicated directory.

```
bash.dylib-5.2# /sbin/mount
lifs://livefileproviderd@291/LiveFiles on /private/var/mobile/Library/LiveFiles (lifs, nodev, noexec, nosui
exfat://disk3s1/USB-hello on /private/var/mobile/Library/LiveFiles/com.apple.filesystems.userfsd/USB-hello
bash.dylib-5.2# ls --color /private/var/mobile/Library/LiveFiles/com.apple.filesystems.userfsd/USB-hello
 1.txt   2.txt  'System Volume Information'   autorun.inf   eaget.ico   temp   uaf.txt
```

# When connecting a USB drive to iPhone

- Summary: process of mounting a UserFS volume



# When accessing files via UserFS

- To read a file: `int fd = open(path, O_RDONLY); read(fd, …);`
- We know that SYS_open will be dispatched to vnop_open

```
(lldb) bt
* thread #2, stop reason = breakpoint 2.1
  * frame #0: 0xffffffff009e084b4 ios15-kc.out`hfs_vnop_open
    frame #1: 0xffffffff007fdf4b0 ios15-kc.out`vn_open_auth + 1884
    frame #2: 0xffffffff007fc822c ios15-kc.out`open1 + 244
    frame #3: 0xffffffff007fc8e70 ios15-kc.out`open_nocancel + 268
    frame #4: 0xffffffff008423e0c ios15-kc.out`unix_syscall + 756
```

# When accessing files via UserFS

- Accessing files via UserFS is same, except that SYS_open will be dispatched to lifs.kext`lifs_vnop_open

```
__int64 __fastcall lifs_vnop_open(vnop_open_args *a)
{
  struct vnode *a_vp; // x19
  char *fsnode; // x20
  __int64 err; // x21
  int a_mode; // w26
  int v6; // w23
  uintptr_t fsmount; // [xsp+8h] [xbp-48h] BYREF

  a_vp = a->a_vp;
  fsnode = (char *)vnode_fsnode(a_vp);
  fsmount = 0LL;
  err = get_lifs_mount_from_node((lifsnode *)fsnode, (lifsmount **)&fsmount);
  if ( !(_DWORD)err )
  {
    if ( (kdebug_enable & 0xFFFFFFF7) != 0 )
      kernel_debug_filtered(0x3140089u, fsmount, (uintptr_t)a_vp, (uintptr_t)fsnode, a
    a_mode = a->a_mode;
    v6 = a_mode & 3;
    err = lifs_open_request(fsmount, (lifsnode *)fsnode, a_mode & 3);
```

# When accessing files via UserFS

- vnop_open will build a mach message and send it to a server.

```
int lifs_open_request(lifsmount *li_mount, void *fsnode, void *a3)
{
    get_lifs_port(&svr_port);

    req.request_id = OSAddAtomic64(1LL, &lifs_request_id);
    // ... copy params to request

    lifs_add_req(&req);

    ret = lifs_open_send(svr_port, req.request_id, fsnode, a3); {
        struct { mach_msg_header_t msgHdr; } openRequest;
        openRequest.msgHdr.msgh_id = 0x2A5;
        mach_msg_send_from_kernel_proper(&openRequest.msgHdr, sizeof(openRequest));
    }
    if ( ret == 0 ) {
        lifs_wait_req_completion(&req);
        ret = req.retcode_2C;
    }

    lifs_remove_req(&req);
    return ret;
```

# When accessing files via UserFS

- Find handler for the open request.
- livefileproviderd registers a notification port. That's the server port.

```
livefileproviderd`main()
{
    mach_port_allocate(mach_task_self, 1, &svr_port);

    li_UserClient = +[LiveFSUserClient defaultClient];
    objc_msgSend(li_UserClient, "setMainMachPort:forDomain:", svr_port) {
        IOServiceOpen(IOServiceMatching("com_apple_filesystems_lifs"), mach_task_self, 0, &self->ourPort);
        IOConnectSetNotificationPort(self->ourPort, 0, svr_port, 0LL);
    }
}

AppleLIFSUserClient::registerNotificationPort(AppleLIFSUserClient *this, ipc_port_t svr_port)
{
    return lifs_set_machport(svr_port, this->clientDomain);
}
```

# When accessing files via UserFS

- livefileproviderd starts a mig server.

```
livefileproviderd`main()
{
    source = dispatch_source_create(DISPATCH_SOURCE_TYPE_MACH_RECV, svr_port, ...);
    dispatch_source_set_event_handler(source, ^{
        dispatch_mig_server(source, 2168, mig_message_dispatcher);
    });
}
                openRequest.msgHdr.msgh_id = 0x2A5;

void *mig_lifs_open_send(void *InHeadP, void *OutHeadP)
{
    lifs_open_send(request_id, fsnode, mode, token) {
        fileHandle = m_resolve_fsnode(fsnode, &a2, &mountEntry, &a4, 0);
        id fsObj = [mountEntry fsObjWithErrorHandler:];
        objc_msgSend(fsObj, "LIOpen:withMode:forPID:reply:", fileHandle, mode, pid, ^(int ret){
            int selector = 2;
            lifs_send_reply(request_id, ret, selector, (__int64)&v2, 16LL) {
                objc_msgSend(li_UserClient, "callStructMethod:inStruct:inSize:outStruct:outStructSize:");
            }
        });
    }
}
```

IOConnectCallStructMethod

# When accessing files via UserFS

- -[LiveFSUserClient callStructMethod:inStruct:inSize:outStruct:outStructSize:]

```
lifs.kext`AppleLIFSUserClient::methodGenericReply(
                                                    AppleLIFSUserClient *a1,
                                                    void *a2,
                                                    IOExternalMethodArguments *a3)
{
    lifs_request_done(*(_QWORD *)a3->structureInput, *((_DWORD *)a3->structureInput + 2), 0, 0, 0, 0);
    return 0;
}
```

# When accessing files via UserFS

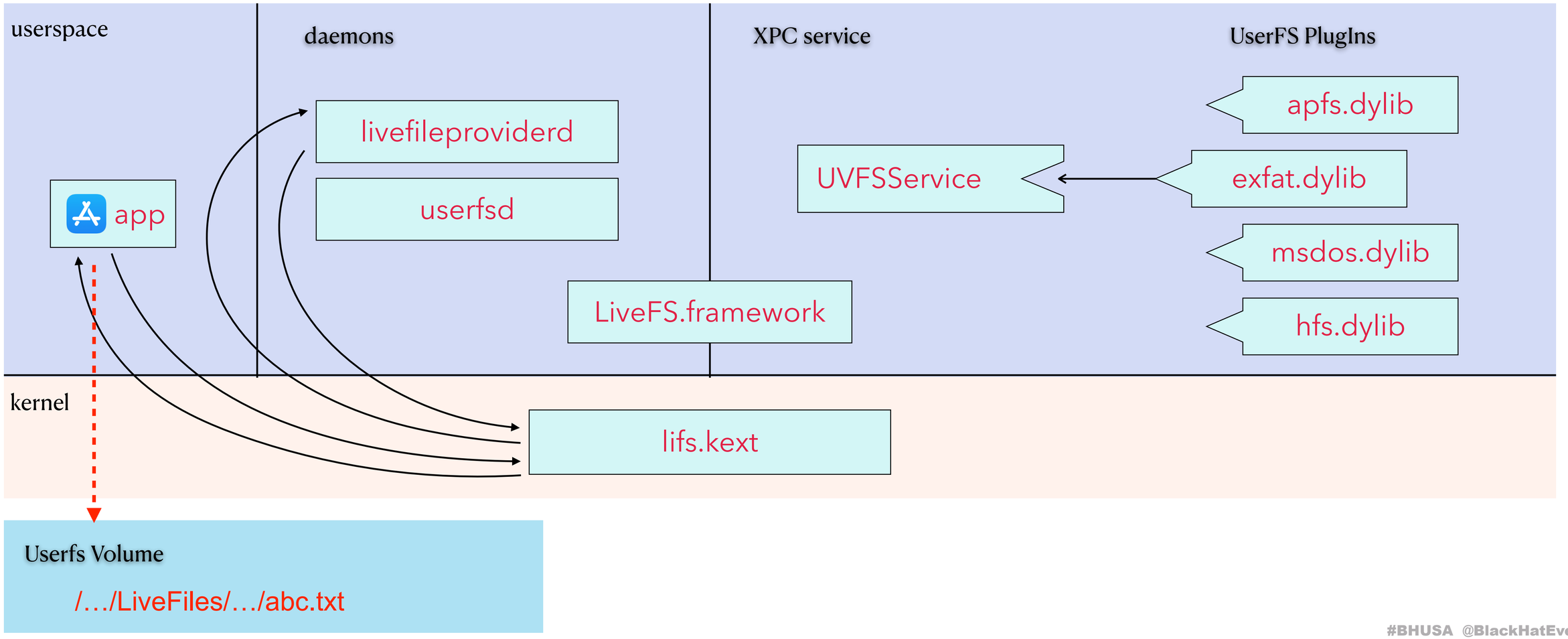
```
int lifs_open_request(lifsmount *li_mount, void *fsnode, void *a3)
{
    get_lifs_port(&svr_port);

    req.request_id = OSAddAtomic64(1LL, &lifs_request_id);
    // ...

    lifs_add_req(&req);

    ret = lifs_open_send(svr_port, req.request_id, fsnode, a3); {
        struct { mach_msg_header_t msgHdr; } openRequest;
        openRequest.msgHdr.msgh_id = 0x2A5;
        mach_msg_send_from_kernel_proper(&openRequest.msgHdr, sizeof(openRequest));
    }
    if ( ret == 0 ) {
        lifs_wait_req_completion(&req);
        ret = req.retcode_2C;
    }

    lifs_remove_req(&req);
    return ret;
```

lifs_request_done(…)

```
int fd = open(path, O_RDONLY);
```

# When accessing files via UserFS

- summarize control flow of “open” syscall



# When accessing files via UserFS

- livefileprovider d

```
objc_msgSend(v15, "LIOpen:withMode:forPID:reply:", fileHandle, mode, pid, &v24);
```

- LiveFS.framework

```
void __cdecl -[LiveFSServiceConnection LIOpen:withMode:forPID:reply:](
        LiveFSServiceConnection *self,
        SEL a2,
        id a3,
        int a4,
        int a5,
        id a6)
{
  -[NSFileProviderLiveItemImplementation LIOpen:withMode:forPID:reply:](
    self->mount,
    "LIOpen:withMode:forPID:reply:",
    a3,
    *(_QWORD *)&a4,
```

off=8; NSFileProviderLiveItemImplementation *

- UVFSService

```
void __cdecl -[liveFSVolume LIOpen:withMode:forPID:reply:](liveFSVolume *self, SEL a2, id
{
```

# When accessing files via UserFS

- UVFSService

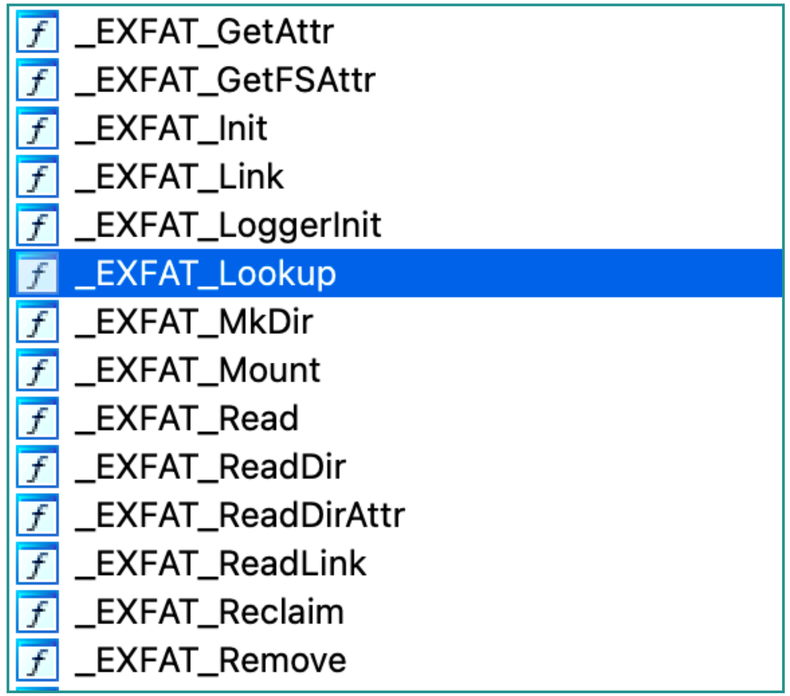
```
void -[liveFSVolume LILookup:name:forClient:reply:](
        liveFSVolume *self, SEL a2, id fileHandle, id nameStr, unsigned client, id reply_b)
{
    fileNode = -[liveFSVolume getNodeForFH:fileHandle withError:error];
    [fileNode lookup:nameStr withResultingNode:&resultNode]; {
        self->FSOps->fsops_lookup(self->_UVFSNode, [nameStr UTF8String], &uvfsNode);
        fileNode = [[liveFSNode alloc] initWithVolume:self->volume
                                            andParent:self
                                              andName:nameStr
                                          andUVFSNode:uvfsNode];
        *resultNode = fileNode;
    }
    resultFileHandle = [resultNode getFH];
    attrData = [resultNode getAttrData];
    reply_b(retcode, resultFileHandle, attrData);
}
```

```
void __cdecl -[liveFSVolume LIOpen:withMode:forPID:reply:](liveFSVolume *self, SEL a2, id
{
```

# When accessing files via UserFS

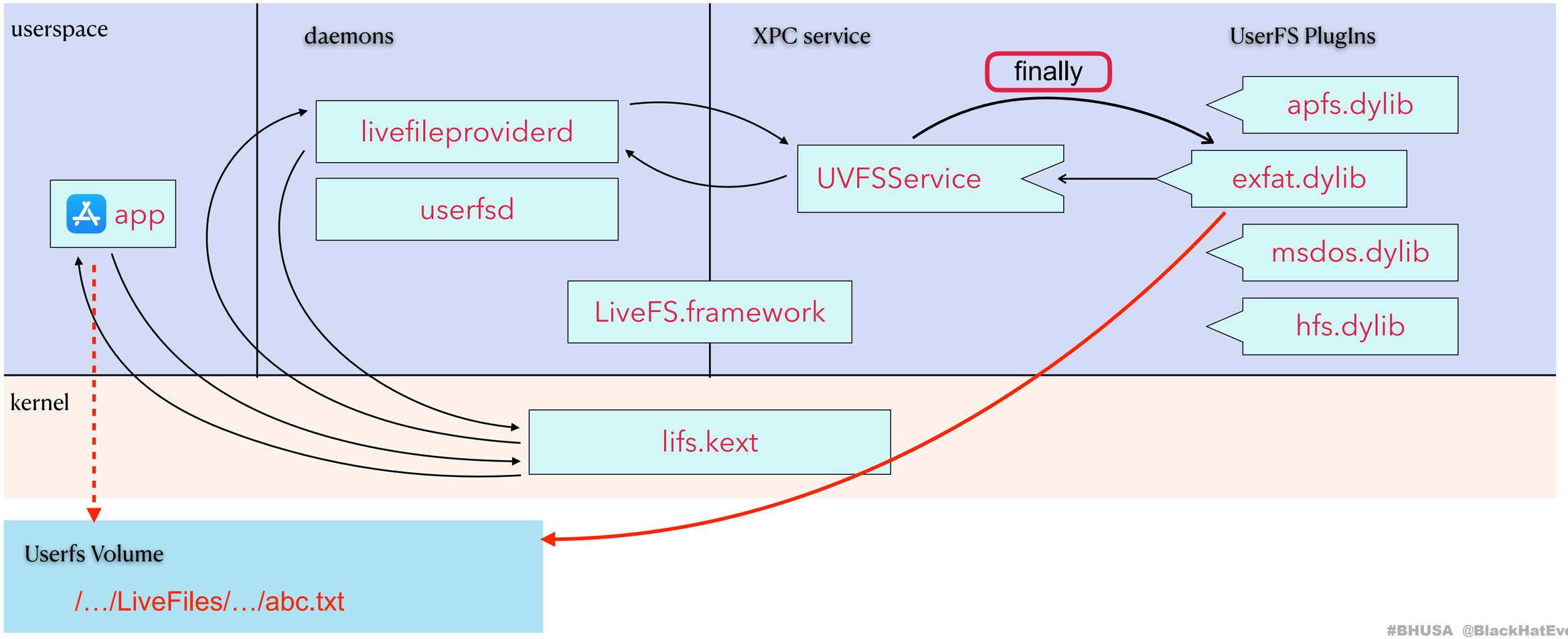
- UserFS.framework/PlugIns/livefiles_exfat.dylib will do the real job.
- Parse /dev/disk3s1 as a normal file (or disk image), execute exfat file read/write request.

- Functions exported by exfat.dylib:



# When accessing files via UserFS

- summarize full control flow of “open” syscall



# Real world vulnerabilities - 1

- Vulnerable code (CVE-2020-27904), kernel FS

```
check_and_swap_attrhdr(attr_header_t *ah, attr_info_t *ainfop)
{
    /*
     * Make sure each of the attr_entry_t's fits within total_size.
     */
    buf_end = ainfop->rawdata + ah->total_size;
    count = ah->num_attrs;
    ae = (attr_entry_t *)(&ah[1]);

    for (i = 0; i < count; i++) {
        /* Make sure the fixed-size part of this attr_entry_t fits. */
        if ((u_int8_t *) &ae[1] > buf_end) {
            return EINVAL;
        }

        /* Make sure the attribute content fits. */
        end = ae->offset + ae->length;
        if (end < ae->offset || end > ah->total_size) {
            return EINVAL;
        }

        ae = ATTR_NEXT(ae);
    }
}
```

```
*  Typical "._" AppleDouble Header File layout:
* ---------------------------------------------------------
*     .-- AD ENTRY[0]     Finder Info Entry (must be fir
* .---+-- AD ENTRY[1]     Resource Fork Entry (must be l
* |   '-> FINDER INFO
* |       /////////////  Fixed Size Data (32 bytes)
* |       EXT ATTR HDR
* |       /////////////
* |       ATTR ENTRY[0] --.
* |       ATTR ENTRY[1] --+--.
* |       ATTR ENTRY[2] --+--+--.
* |          ...          |  |  |
* |       ATTR ENTRY[N] --+--+--+--.
* |       ATTR DATA 0   <-'  |  |  |
* |       ////////////       |  |  |
* |       ATTR DATA 1   <----'  |  |
* |       /////////////         |  |
* |       ATTR DATA 2   <-------'  |
* |       /////////////            |
* |          ...                   |
* |       ATTR DATA N   <----------'
* |       /////////////
* |                        Attribute Free Space
```

MUST: ah->data_start <= attr entry offset <= ah->total_size

# Real world vulnerabilities - 1

- The vulnerability is about xattr (Extended File Attributes).
- exfat filesystem doesn’t support xattr.
- XNU provides a compatible layer, the vulnerable code exists in compatible code.
- The logic is similar in UserFS, but the code is completely rewritten.

```
void -[liveFSVolume createAppleDoubleManagerIfNeeded](liveFSVolume *self, SEL a2)
{
    if ( (self->volCapInterfaces & 0x4000) != 0 ) {
        os_log_debug(OS_LOG_DEFAULT, "LFM: volume %@ supports xattrs natively");
    }
    else {
        v3 = +[LiveFSAppleDoubleManager AppleDoubleManagerForMount:self];
        self->_appleDoubleManager = v3;
        os_log_debug(OS_LOG_DEFAULT, "LFM: created AppleDouble manager instance for volume %@");
        self->volCapInterfaces &= ~0x40000uLL;
    }
}
```

# Real world vulnerabilities - 1

- UserFS version of CVE-2020-27904, i.e., CVE-2022-42861
- The rewritten code is LiveFS`-[LiveFSAppleDouble loadAttrHeader]

```
55        while ( buf_end >= (char *)&ae_ptr[1] )
56        {
57          namelen = ae_ptr->namelen;
58          if ( &ae_ptr->name[namelen] > (u_int8_t *)buf_end )
59            break;
60          v19 = strnlen((const char *)ae_ptr->name, namelen);
61          v20 = ae_ptr->namelen;
62          if ( v19 != v20 - 1 )
63            break;
64          ae_offset = bswap32(ae_ptr->offset);
65          ae_length = bswap32(ae_ptr->length);
66          ae_ptr->offset = ae_offset;
67          ae_ptr->length = ae_length;
68          ae_ptr->flags = bswap32(ae_ptr->flags) >> 16;
69          v11 = __CFADD__(ae_length, ae_offset);
70          ae_end = ae_length + ae_offset;
71          if ( v11 || ae_end > filehdr->total_size || ae_offset < filehdr->data_start )
72            break;
73          ae_ptr = (attr_entry *)((char *)ae_ptr + (((_WORD)v20 + 14) & 0x1FC));
74          if ( (unsigned __int16)++entry_idx >= (unsigned int)num_attrs )
75            goto LABEL_17;
76        }
```

# Real world vulnerabilities - 1

- This kernel bug impacts iOS too, but only the UserFS code is affected.
- The UserFS one still exists on iOS < 16.2 & 15.7.2
- It didn't get fixed until I reported it again.
- root cause of CVE-2022-42861: the bug-fix of UserFS lost synchronization with kernel FS

# Real world vulnerabilities - 2

- While studying UserFS, I found another xattr bug in kernel FS.

```
+++ b/bsd/vfs/vfs_xattr.c
@@ -2938,9 +2938,16 @@ get_xattrinfo(vnode_t xvp, int setting, attr_info_t *ainfop, vfs_context_t conte
                                    delta, context);
                                writesize = sizeof(attr_header_t);
                        } else {
-                               /* Create a new, empty resource fork. */
+                               /* We are in case where existing resource fork of length 0, try to create a new, empty r
                                rsrcfork_header_t *rsrcforkhdr;

+                               /* Do we have enough space in the header buffer for empty resource fork */
+                               if (filehdr->entries[1].offset + delta + sizeof(rsrcfork_header_t) > ainfop->iosize) {
+                                       /* we do not have space, bail for now */
+                                       error = ENOATTR;
+                                       goto bail;
+                               }
+
                                vnode_setsize(xvp, filehdr->entries[1].offset + delta, 0, context);
```

# Real world vulnerabilities - 2

- I think it is CVE-2022-42842.
- The ability of this bug is limited.

Then init_empty_resource_fork will initialize the memory block to some fix values. A partially controlable oob-write occurs. We can write these bytes beyond the 64KB buffer.

```
00000ee0  00 00 00 00 01 00 00 00  01 00 00 00 00 00 00 00  |................|
00000ef0  00 1e 54 68 69 73 20 72  65 73 6f 75 72 63 65 20  |..This resource |
00000f00  66 6f 72 6b 20 69 6e 74  65 6e 74 69 6f 6e 61 6c  |fork intentional|
00000f10  6c 79 20 6c 65 66 74 20  62 6c 61 6e 6b 20 20 20  |ly left blank   |
00000f20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000fe0  00 00 00 00 01 00 00 00  01 00 00 00 00 00 00 00  |................|
00000ff0  00 1e 00 00 00 00 00 00  00 00 00 1c 00 1e ff ff  |................|
```

For example, we can control that

```
*(uint8_t  *)(buffer + 0x10000) = 0xff;
*(uint16_t *)(buffer + 0x10000) = 0xffff;
*(uint32_t *)(buffer + 0x10000) = 0xffff1e00;
*(uint64_t *)(buffer + 0x10000) = 0xffff1e001c000000;
```

# Real world vulnerabilities - 2

- Tales from the iOS/macOS Kernel Trenches - @jaakerblom
- With the exploit primitive (kmsg type confusion), I can exploit the kernel FS xattr bug.

## Exploitation: CVE-30937

- kmsg corruption can be done by turning a non-complex kmsg into a complex one (i.e setting bits 0x80000000 - MACH_MSGH_BITS_COMPLEX)
- In that case, the controllable inline data of a message without any real descriptors will be interpreted as if it was in fact real descriptors

# Real world vulnerabilities - 2

- But that primitive was fixed on iOS 15.2, so I can’t exploit this bug easily on macOS >=12.1,>=13.

```
[+] kapi_write64 -> 0x1234
[+] kernel base 0xfffffe0013a68000, kernel slide 0xca64000
[+] verify kernel header
_mh_execute_header
0000: feedfacf 0100000c  00000002 0000000c
[+] kernel_proc 0xfffffe0017d7c4c8, self_proc 0xfffffe1513a749f0
[+] now everything is OK
[+] spawn a root shell ->

root@secs-MacBook-Pro ~ # id
uid=0(root) gid=0(wheel) groups=0(wheel),1(daemon),2(kmem),3(sys),4(tty),5(operat
(staff),29(certusers),61(localaccounts),80(admin),701(com.apple.sharepoint.group.
tor),204(_developer),250(_analyticsusers),395(com.apple.access_ftp),398(com.apple
s_ssh),400(com.apple.access_remote_ae)
root@secs-MacBook-Pro ~ # sw_vers
ProductName:    macOS
ProductVersion: 12.0.1
BuildVersion:   21A559
root@secs-MacBook-Pro ~ #
```

# Real world vulnerabilities - 2

- Of cause, there is a corresponding UserFS version (no idea of its CVE no.)
- LiveFS`-[LiveFSAppleDouble loadADHeader]

```
306         *(_OWORD *)&block_v86.lvar3[128] = v100;
307         *(_OWORD *)&block_v86.lvar3[144] = v101;
308         *(_OWORD *)&block_v86.lvar3[48] = *(_OWORD *)buf;
309         *(_OWORD *)&block_v86.lvar3[64] = v96;
310         *(_OWORD *)&block_v86.lvar3[80] = v97;
311         *(_OWORD *)&block_v86.lvar3[96] = v98;
312         *(_QWORD *)&block_v86.lvar3[160] = v102;
313         block_v86.lvar2 = (__int64)capture_err;
314         dispatch_sync(v68, &block_v86);
315         objc_release(v68);
316         objc_release(header_end);
317         delta -= 0x11ELL;                           // sizeof(rsrcfork_header_t) = 0x11e
318         bzero(fileHeader_2->appledouble.pad, delta);
319         -[LiveFSAppleDouble initEmptyResourceFork:](
320           self,
321           "initEmptyResourceFork:",
322           (char *)self->_filehdr + self->_filehdr->entries[1].offset + delta);
323         self->_filehdr->entries[1].length = 0x11E;
324         v64 = 4096LL;
325       }
```

# Possible to pwn kernel via UserFS?

- User space oob bugs: The kernel version of them have been proved to be exploitable. So, they are exploitable, in theory. I didn’t try.
- Kernel space race condition bug: It can be converted to kernel UAF. Though it is very hard to write a workable exploit for it, it is exploitable in theory. I didn’t try.
- Sandbox: App cannot access lifs.kext directly.
- Chain the user space UserFS oob bug with the kernel space lifs UAF?
  - I think there is a chance to attack kernel via UserFS, at least, in theory (again…).

# The changes in filesystem security model

- call stack
  - `kernel FS: SYS_open -> msdos.kext`
  - `UserFS: SYS_open -> lifs.kext -> userspace daemons -> exfat.dylib`
- lifs.kext is simple, just forwards syscall to userspace daemons.
- main exploit target
  - `kernel FS: kernel extension`
  - `UserFS: userspace daemons`
- If successfully exploited
  - `kernel FS: kernel read/write`
  - `UserFS: takeover a sandboxed userspace process`
- UserFS will reduce the impact of FS vulnerabilities.

# disadvantages of UserFS

- UserFS is an extra feature. It doesn't replace (all the) kernel FS.
- Attack surface = kernel FS + UserFS
  - iOS: only apfs & hfs are kept in kernel
  - macOS: all filesystem extensions are reserved in kernel
- Maintaining two code bases with identical functionality, but it's hard to keep bug fixes in sync between them.
- XPC everywhere in UserFS, so, performance?
  - Compared to CPU, USB disk is too slow. The performance is acceptable.

# Conclusion

- More modules, more bugs.
  - UserFS is an additional feature, kernel FS is still here, thus increasing the attack surface of the filesystem.
- There is a chance to break iOS through UserFS.
  - But when accessing files stored on USB disks, only UserFS takes effect. In this case, UserFS can significantly reduce the impact of filesystem vulnerabilities.
- Overall speaking, I think UserFS is a successful effort by Apple.

# Thanks!

pattern-f (@pattern_F_)