11

PERSISTENCE MONITOR

While KnockKnock, covered in the previous chapter, provides a powerful detection capability, it doesn't protect the system in real time. To complement it, I created BlockBlock, which monitors the most important persistence locations enumerated by KnockKnock, alerts the user whenever a new item appears, and gives them the ability to block the activity.

BlockBlock's initial versions, written in 2014, were largely proofs of concept, which didn't stop employees from commercial security companies from labeling the tool "lam[e]ware" and concluding that "providing quality service for nothing can't be a one-person job."[1] Over the years, BlockBlock has matured, consistently proving its merit with a near 100 percent detection rate of persistent Mac malware, even without prior knowledge of these threats.

In this chapter, I'll discuss BlockBlock's design and show how it uses Endpoint Security to effectively detect unauthorized persistence events. You'll learn how to request and apply the required Endpoint Security client entitlement and how XPC can allow tool components to securely communicate with one another. You can find BlockBlock's source code in its entirety in the Objective-See GitHub repository at *https://github.com/objective-see/BlockBlock*.

Entitlements

Multiple BlockBlock components leverage Endpoint Security, which means the tool must receive a privileged entitlement from Apple. Without the entitlement, attempts to create an Endpoint Security client at runtime will fail unless we've disabled System Integrity Protection (SIP) and Apple Mobile File Integrity (AMFI). So, let's start by walking through the process of requesting the Endpoint Security client entitlement from Apple and, once it's granted, applying it to BlockBlock.

Applying for Endpoint Security Entitlements

You can apply for Endpoint Security entitlements at *https://developer.apple.com/contact/request/system-extension/*. The request form asks for developer information, such as your name and company, then presents a drop-down menu containing a list of entitlements you can request. Select the Endpoint Security client entitlement, **com.apple.developer.endpoint-security.client**. At the bottom of the form, describe how you intend to use the entitlement you're requesting.

Given the power of Endpoint Security, Apple is understandably cautious about granting requests for the client entitlement, even to renowned security companies. That said, you can take several measures to improve your chances of receiving one. First, register as a company, such as an LLC or equivalent. I'm aware of only one instance in which Apple granted the Endpoint Security client entitlement to an individual. Second, in your request, make sure to describe exactly what you plan to do with the entitlement. The Endpoint Security client entitlement is designed for security tools, so include details of the tool you're developing and articulate exactly why it needs the use of Endpoint Security. Finally, be prepared to wait.

Registering App IDs

Once Apple has granted you the entitlement, you must register an App ID for your tool, specifying its bundle ID and the entitlements it will use. Log in to your Apple Developer account, click **Account**, then navigate to **Certificates, Identifiers & Profiles ▸ Identifiers**. If you have any existing identifiers, they should show up here. To create a new identifier, click **+**. Select **App IDs**, then click **Continue**. Select **App** and **Continue** again.

This should bring you to the App ID registration form. Most of the fields are self-explanatory. For the Bundle ID, Apple recommends using a reverse-domain name style, generally in the form *com.company.product*. For BlockBlock, I populated the fields as shown in Figure 11-1.

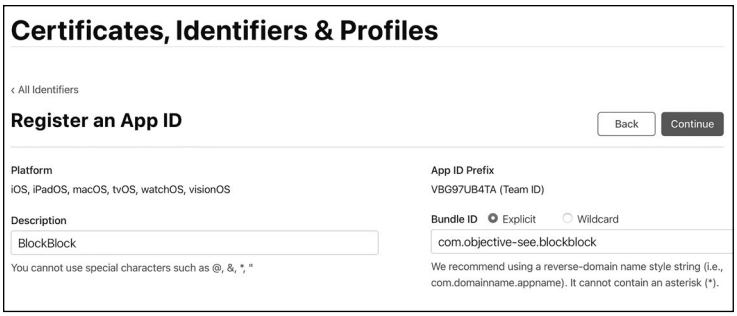

Figure 11-1: Registering the BlockBlock app ID

In the remainder of the form, you'll see options to specify either capabilities, app services, or additional capabilities for your tool. Assuming Apple has granted you the Endpoint Security client entitlement, click **Additional Capabilities**, then select the checkbox next to Endpoint Security. To register the new identifier, click **Register**.

Creating Provisioning Profiles

Now you can create the provisioning profile, which provides the mechanism that the operating system will use to authorize the use of the entitlement at runtime.[2] Clicking **Profiles** in your Developer Account should take you to a page containing all of your current profiles. You can also register a new profile by clicking +. On the first page, specify the provisioning profile's type. Unless you'll be distributing your tool via the Mac App Store, select **Developer ID** at the very bottom of the page. Click **Continue**, then select the App ID you just created.

Next, select the certificate to include in your profile. This is the same certificate you'll use to sign your application, likely your Apple Developer certificate. On the next page, you'll be given a list of available entitlements you can add to the provisioning profile. To leverage Endpoint Security, select **System Extension EndpointSecurity for macOS**. If Apple hasn't yet granted you this entitlement, it won't show up in the list.

Enabling Entitlements in Xcode

Once you've generated the provisioning profile, you can head to Xcode to add it to your project. First, tell Xcode that your project will use Endpoint Security by clicking the small + next to **Capabilities** in the Signing & Capabilities pane and then selecting **Endpoint Security** capability. Behind the scenes, this will add the entitlement to the project's entitlement file.

Now, when building the tool for deployment, you can select the provisioning profile. The first time you do this, you might have to download and import the profile into Xcode. Download the profile you generated from your Apple Developer account. Then, in Xcode's Select Certificate and Developer ID Profiles window, select the **Import Profile** option, found in the drop-down menu next to the application's name, and browse to the downloaded profile.

If all goes well, you should have a compiled, entitled tool that also contains the provisioning profile. For example, BlockBlock's provisioning profile is embedded in its app bundle at the standard location, *Contents/embedded.provisionprofile*. You can dump any embedded provisioning profile by running the macOS `security` tool, along with the command line flags `cms -D -i` and this path. The following output contains BlockBlock's App ID, information about its code signing certificate, and the entitlements it is authorized to use:

```
% security cms -D -i BlockBlock.app/Contents/embedded.provisionprofile
<?xml version="1.0" encoding="UTF-8"?>
...
<plist version="1.0">
<dict>
    <key>AppIDName</key>
    <string>BlockBlock</string>
    <key>DeveloperCertificates</key>
    <array>
        <data> ... </data>
    </array>
    <key>Entitlements</key>
    <dict>
        <key>com.apple.developer.endpoint-security.client</key>
        <true/>
        <key>com.apple.application-identifier</key>
        <string>VBG97UB4TA.com.objective-see.blockblock</string>
        ...
    </dict>
    ...
```

You can use the `codesign` utility to view any entitlements a program possesses. For BlockBlock, this list includes the Endpoint Security client entitlement:

```
% codesign -d --entitlements - BlockBlock.app
Executable=BlockBlock.app/Contents/MacOS/BlockBlock
[Dict]
    [Key] com.apple.application-identifier
    [Value]
        [String] VBG97UB4TA.com.objective-see.blockblock
    [Key] com.apple.developer.endpoint-security.client
    [Value]
        [Bool] true
    ...
```

Because macOS requires a provisioning profile to authorize the entitlement, even programs not typically developed as applications, such as daemons, must be packaged as application bundles to leverage Endpoint Security. You can read more about this design choice in Apple's documentation,[3] which also notes that if you switch from a daemon to a system extension, Xcode will automatically handle the packaging for you.

Tool Design

BlockBlock is composed of two pieces: a launch daemon and a login item. The daemon is packaged as an application bundle to accommodate the use of entitlements and provisioning profiles. It runs in the background with root privileges, monitoring for persistence events (by ingesting file input/output and other events delivered from Endpoint Security), managing rules, and blocking user-specified persistent items. Anytime it detects a persistence event, the daemon sends an XPC message to the login item. The login item, which runs in the context of the user's desktop session and thus is capable of displaying user interface (UI) elements, will then show the user an alert (Figure 11-2).

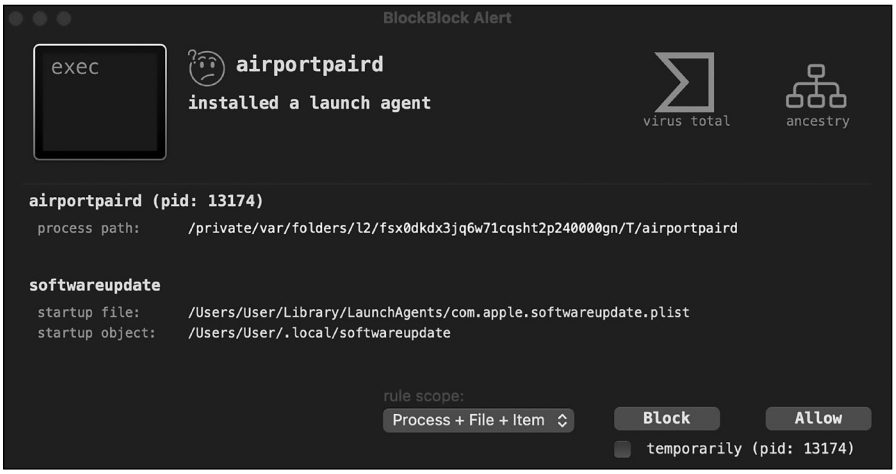

Figure 11-2: A BlockBlock alert

BlockBlock's alerts contain plenty of information about the item that installed the persistent item and the persistent item itself. This information can assist the user in deciding whether to allow or delete the item. For example, various red flags in the alert shown in Figure 11-2 indicate an infection. First, the item that installed the launch agent, *airportpaird*, is unsigned, as indicated by the perplexed frowning face. From its path, you can also see that it's running from a temporary directory.

If you turn your attention to the persistent item, you'll notice that the property list is prefixed with `com.apple`, implying that it belongs to Apple. However, it's installed in the user's Launch Agent directory, which only ever contains third-party agents. Moreover, the persistent item that this property list references is installed and runs from a hidden directory (*.local*). Finally, if you manually examined the code signing information of this binary, *softwareupdate*, you would see it is unsigned.

When I originally released BlockBlock in 2014, Apple didn't yet support System Extensions, which is why I placed the tool's core logic in a launch daemon. Today, BlockBlock continues to make use of a daemon even though

doing so isn't strictly necessary, as the approach still has benefits. For one, you might want to develop tools that maintain compatibility with older versions of macOS. It's also easy for any sufficiently privileged tool to install and manage launch daemons. On the other hand, System Extensions require additional entitlements, and to install or remove them, you'll typically need explicit user approval. This adds complexity and requires additional code. Still, there are cases where putting your code into a System Extension makes sense, as you'll see in Chapter 13.

Plug-ins

Like KnockKnock, BlockBlock uses statically compiled plug-ins to detect multiple types of persistence. Each plug-in is responsible for handling either one unique persistent event or several related ones. The tool stores metadata about each plug-in in a property list file, including the name of the plug-in class, various descriptions of it to customize alerts, and, most importantly, a regular expression describing the path or paths of file events in which the plug-in is interested. For example, Listing 11-1 shows the metadata for the plug-in that monitors file events for the additions of new launch daemons and agents.

```
<dict>
   <key>description</key>
   <string>Launch D &amp;amp; A</string>
   <key>paths</key>
   <array>
      <string>^(\/System|\/Users\/[^\/]+|)\/Library\/(LaunchDaemons|
      LaunchAgents)\/.+\.(?i)plist$</string>
   </array>
   <key>class</key>
   <string>Launchd</string>
   <key>alert</key>
   <string>installed a launch daemon or agent</string>
   ...
</dict>
```

Listing 11-1: Metadata for the launch item plug-in

The regular expression will be applied to incoming file input/output events, matching on those that were ingested due to the addition of property lists added to the launch daemons and agents directories such as */System/Library/LaunchDaemons* or *~/ Library/LaunchAgents*.

All plug-ins inherit from a custom base class named `PluginBase` that implements base methods, such as a standard initialization method and methods to check whether a file event matches an event of interest. The initialization method `initWithParams:` takes one parameter, a dictionary containing a plug-in's metadata (Listing 11-2).

```
-(id)initWithParams:(NSDictionary*)watchItemInfo {
    ...
    NSMutableArray* regexes = [NSMutableArray array];
```

```
        for(NSString* regex in watchItemInfo[@"paths"]) {
            NSRegularExpression* compiledRegex =
            [NSRegularExpression regularExpressionWithPattern:regex
            options:NSRegularExpressionCaseInsensitive error:NULL];

            [self.regexes addObject:compiledRegex];
        }

        self.alertMsg = watchItemInfo[@"alert"];
        self.description = watchItemInfo[@"description"];
        ...
        return self;
    }
```

Listing 11-2: The base class logic for plug-in object initialization

Here, you can see that the method first compiles each of the plug-in's paths of interest into regular expressions and then extracts other values from the metadata dictionary to save into instance variables.

Another important base method, `isMatch:`, accepts a file object representing an event from the *FileMonitor* library, then checks for a match against the plug-in paths of interest (Listing 11-3).

```
-(BOOL)isMatch:(File*)file {
    __block BOOL matched = NO;
    NSString* path = file.destinationPath;

  ❶ [self.regexes enumerateObjectsWithOptions:NSEnumerationConcurrent
    usingBlock:^(NSRegularExpression* _Nonnull regex, NSUInteger idx, BOOL
    * _Nonnull stop) {

      ❷ NSTextCheckingResult* match = [regex firstMatchInString:path options:0
        range:NSMakeRange(0, path.length)];
        if( (nil == match) || (NSNotFound == match.range.location) ) {
            return;
        }

      ❸ matched = YES;
        *stop = YES;
    }];

    return matched;
}
```

Listing 11-3: Filepath matching

The method runs `enumerateObjectsWithOptions:usingBlock:` on the array of the plug-in's regular expressions so it can iterate over all of them concurrently ❶. In the concurrently invoked callback block, it uses the current regular expression to check whether the destination file matches an event of interest to the plug-in ❷. For example, for the launch item plug-in, the method will check whether the file event corresponds to the creation of a

property list in a launch daemon or agent directory. If a match does occur, the method sets a flag and terminates the enumeration ❸.

Other methods in the base plug-in class are left for each plug-in to implement. For example, the `block:` method, invoked when the user clicks the Block button on the alert, will remove the persistent item. This logic must differ based on the type of item persisted. If you're interested in the specific uninstallation logic for each kind of persistent item, take a look at the code of each plug-in's `block:` method.

At its core, BlockBlock ingests events from the *FileMonitor* library, which leverages Apple's Endpoint Security. After initializing a `FileMonitor` object with the specific events of interest, it specifies a callback block and then begins file monitoring (Listing 11-4).

```
es_event_type_t events[] = {ES_EVENT_TYPE_NOTIFY_CREATE, ES_EVENT_TYPE_NOTIFY_WRITE,
ES_EVENT_TYPE_NOTIFY_RENAME, ES_EVENT_TYPE_NOTIFY_EXEC, ES_EVENT_TYPE_NOTIFY_EXIT}; ❶

FileCallbackBlock block = ^(File* file) {
    ...
    [self processEvent:file plugin:nil message:nil]; ❷
};

FileMonitor* fileMon = [[FileMonitor alloc] init];
[fileMon start:events count:sizeof(events)/sizeof(events[0]) csOption:csNone callback:block];
...
```

Listing 11-4: A helper method invoked for each file event

If you look carefully at the Endpoint Security events of interest passed to the file monitor, you'll see both file and process events ❶. It makes sense to initialize a file monitor with file events, and we need the process events to record the arguments of processes creating persistent items. Although not every process that persists an item is invoked with arguments, many are, and in those cases, we include the arguments in the alert shown to the user to help them determine whether the persistence event is benign or malicious. Before we discuss the processing of file input/output events, note that the file monitor logic is started by invoking the `start:count:csOption:callback:` method.

When the file monitor receives events, it invokes the specified callback block with a `File` object representing the event. The callback simply hands this object a helper method named `processEvent:plugin:message:` ❷. This method calls each plug-in's `isMatch:` method to see whether the file event matches any persistence locations, such as the creation of a *.plist* in the launch daemon or agent directories. If any plug-in is interested in the file event, BlockBlock creates a custom `Event` object with both the file object representing the persistence event and the relevant plug-in.

Next, the method checks whether the event matches any existing rules. Rules get created when a user interacts with an alert. They can either allow or block persistence items based on factors like the item's startup file or the process responsible for triggering the event. For example, on my developer box, where I also dabble in photography and photo editing, there are rules allowing the creation of various Adobe Creative Cloud launch agents (Figure 11-3).

Figure 11-3: BlockBlock rules can allow or block events from specified processes.

Because Adobe frequently updates these persistent items, without these rules I'd be regularly responding to BlockBlock alerts. If it finds a matching rule, BlockBlock automatically takes the action specified in the rule. Otherwise, it delivers the event to the BlockBlock login item to show an alert to the user. Shortly, we'll take a closer look at how bidirectional XPC achieves this communication. First, though, let's explore BlockBlock's use of the Endpoint Security Background Task Management events.

Background Task Management Events

One downside to using a global file monitor to detect persistence is that it's rather inefficient, as file events happen almost constantly as part of normal system behavior. While we could mitigate the influx of traffic using Endpoint Security's mute inversion capabilities covered in Chapter 9, BlockBlock needs to monitor many locations to detect multiple methods of persistence, and mute inversion may not fully alleviate the inefficiencies of a file monitor–based approach.

A better solution for our purposes would be to subscribe to persistence events rather than file events. In previous chapters, I discussed the Background Task Management subsystem, a recent addition to macOS that governs the most popular types of persistence, including login items, launch agents, and daemons. Background Task Management also added two events to Endpoint Security: `ES_EVENT_TYPE_NOTIFY_BTM_LAUNCH_ITEM_ADD` and `ES_EVENT_TYPE_NOTIFY_BTM_LAUNCH_ITEM_REMOVE`, which clients can receive whenever a login or launch item is persisted or removed.

Recent versions of BlockBlock leverage the first of these events to deprecate much of its file monitoring–based approach, providing a significant boost in efficiency and simplifying the code base. The tool still monitors persistence mechanisms such as cronjobs, however, for which Background Task Management doesn't yet generate Endpoint Security events, so it can't wholly deprecate its file monitoring.

NOTE: *Although Endpoint Security technically added these Background Task Management events in macOS 13, they didn't work correctly. For example, Endpoint Security would deliver a notification not just for a newly installed item but for every existing item as well. Worse, for login items, it delivered no event at all! After I reported these flaws, Apple fixed both issues in macOS 14.[4] When run on macOS 13 and earlier, BlockBlock falls back to the file monitoring–based approach.*

You can find the code that implements an Endpoint Security client for Background Task Management in the *Daemon/Monitors/BTMMonitor.m* folder and the plug-in to process the events in *Daemon/Plugins/Btm.m*. Let's start by considering the Background Task Management monitor. As with any code that wants to leverage Endpoint Security events, we start by defining the events of interest, creating an Endpoint Security client with a handler block, and subscribing to the specified events (Listing 11-5).

```
es_event_type_t btmESEvents[] = {ES_EVENT_TYPE_NOTIFY_BTM_LAUNCH_ITEM_ADD}; ❶

es_new_client(&_endpointClient, ^(es_client_t* client, const es_message_t* message) { ❷
    // Message handler code removed for brevity ❸
});

es_subscribe(self.endpointClient, btmESEvents, sizeof(btmESEvents)/sizeof(btmESEvents[0])); ❹
```

Listing 11-5: Subscribing to `ES_EVENT_TYPE_NOTIFY_BTM_LAUNCH_ITEM_ADD` events

The code starts by creating an array with the single event to subscribe to ❶. Then, using the `es_new_client` API, it creates a new Endpoint Security client. Because the client is an instance variable of the `BTMMonitor` class, we prepend it with an underscore (`_`) to pass it to the `es_new_client` API ❷. We must do this because the compiler automatically generates an instance variable prefixed with an underscore whenever we declare an instance variable using the Objective-C `@property` keyword.[5] We normally don't directly reference instance variables, but rather access them through an object; however, in the case of Endpoint Security's C APIs, such as `es_new_client`, which expects a pointer, we must perform a direct reference.

Recall that the `es_new_client` API accepts a handler block to invoke each time a subscribed-to event occurs ❸. Shortly, you'll see the code that BlockBlock's Background Task Management monitor executes in this callback. Of course, before Endpoint Security can deliver events, we must tell it that we're interested in subscribing, which we do via the `es_subscribe` API ❹.

Listing 11-6 shows the code in the handler block.

```
es_new_client(&_endpointClient, ^(es_client_t* client, const es_message_t* message) {
    File* file = [[File alloc] init:(es_message_t*)message csOption:csNone]; ❶

    if( (ES_BTM_ITEM_TYPE_AGENT == message->event.btm_launch_item_add->item->item_type) || ❷
        (ES_BTM_ITEM_TYPE_DAEMON == message->event.btm_launch_item_add->item->item_type) ) {
        file.destinationPath =
        convertStringToken(&message->event.btm_launch_item_add->item->item_url);
    }
    es_message_t* messageCopy = NULL;

    if(@available(macOS 11.0, *)) { ❸
        es_retain_message(message);
        messageCopy = (es_message_t*)message;
    } else {
```

```
        messageCopy = es_copy_message(message);
    }
    [monitor processEvent:file plugin:btmPlugin message:messageCopy]; ❹
});
```

Listing 11-6: The Background Task Management event monitoring logic

First, the code initializes a BlockBlock `File` object, passing in the received Endpoint Security message ❶. Then, for launch agents and daemons, it directly sets the file's destination path to the property list of the item just created. We find this property list in the `item_url` member of the `item` structure in the `btm_launch_item_add` structure, within the Endpoint Security message ❷.

Finally, the code calls BlockBlock's `processEvent:plugin:message:` method covered earlier in the chapter ❹. Here, though, the plug-in passed to the method is an instance of BlockBlock's Background Task Management plug-in, which I'll discuss next. Notice that we pass a retained instance or copy of the Endpoint Security message. This is because BlockBlock needs to retain the message for later use (for example, to process the user's asynchronous response). Note that the code will invoke the more modern `es_retain_message` API if running on a recent version of macOS, though falls back to using the `es_copy_message` if running on older versions ❸. Because it explicitly retained or copied the message, BlockBlock must free it when it's no longer needed by invoking the appropriate `es_release_message` or `es_free_message` API.

Like all other BlockBlock plug-ins, the Background Task Management plug-in implements methods to retrieve the name and path of the persisted item, to block the item if instructed by the user, and more. Of course, the logic it uses to do so is specific to Background Task Management persistence events. Let's take a look at the plug-in's `itemObject:` method, which returns the path to the persisted executable. As shown in Listing 11-7, we can extract this information from the delivered Endpoint Security message, although it differs slightly depending on whether the item persisted as a launch item or a login item.

```
-(NSString*)itemObject:(Event*)event {
    NSString* itemObject = nil;

    if( (ES_BTM_ITEM_TYPE_AGENT ==
    event.esMessage->event.btm_launch_item_add->item->item_type) || ❶
    (ES_BTM_ITEM_TYPE_DAEMON ==
    event.esMessage->event.btm_launch_item_add->item->item_type) ) {
        itemObject =
        convertStringToken(&event.esMessage->event.btm_launch_item_add->executable_path);
    } else {
        NSString* stringToken =
        convertStringToken(&event.esMessage->event.btm_launch_item_add->item->item_url); ❷
        itemObject = [[NSURL URLWithString:stringToken] path];
    }
    return itemObject;
}
```

Listing 11-7: Returning the path to the persisted item

The code first checks the type of the persisted item ❶. Conveniently, Endpoint Security indicates this information with constants such as `ES_BTM_ITEM_TYPE_AGENT` and `ES_BTM_ITEM_TYPE_DAEMON` and specifies the item type in the `item_type` member of the `item` structure. Assuming the persisted item is a launch item, the code extracts its executable path from the `executable_path` member of the `btm_launch_item_add` structure. To convert it from an `es_string_token_t` type to an Objective-C string object, we invoke the BlockBlock `convertStringToken` helper function.

For login items, we can find the path to the persisted item in the `item_url` member of the `item` structure ❷. Again, we invoke the `convertStringToken` helper function. However, the path to the item is really a URL object, so we must convert it back to a URL, then use the `path` property of the URL to get the filepath in the form of a string.

The other notable method in the Background Task Management plug-in is `block:`, which BlockBlock invokes when the user clicks Block on the alert shown for a persisted item. Because there is logic to remove both launch and login items in the older, file monitor–based plug-ins, the Background Task Management plug-in can call into the relevant plug-ins to block the item (Listing 11-8).

```
-(BOOL)block:(Event*)event {
    __block BOOL wasBlocked = NO;

    switch(event.esMessage->event.btm_launch_item_add->item->item_type) {
      ❶ case ES_BTM_ITEM_TYPE_APP:
        case ES_BTM_ITEM_TYPE_LOGIN_ITEM: {
            LoginItem* loginItem = [[LoginItem alloc] init];
            wasBlocked = [loginItem block:event];
            break;
        }
      ❷ case ES_BTM_ITEM_TYPE_AGENT:
        case ES_BTM_ITEM_TYPE_DAEMON: {
            Launchd* launchItem = [[Launchd alloc] init];
            wasBlocked = [launchItem block:event];
            break;
        }
        ...
     }
     return wasBlocked;
}
```

Listing 11-8: Blocking logic that calls into login and launch item plug-ins

To determine the type of the Background Task Management item, the code once again makes use of the `item_type` member found in the Endpoint Security Background Task Management message. For login items (which can include persisted user applications), the code instantiates an instance of BlockBlock's Login Item plug-in and then invokes its `block:` method ❶. For launch agents and daemons, it takes a similar approach, instantiating the launch item plug-in ❷.

This wraps up the discussion of BlockBlock's Background Task Management monitor and plug-in. Next, let's look at XPC communications, which BlockBlock makes extensive use of.

XPC

XPC is the de facto interprocess communication (IPC) mechanism on macOS. Anytime you write tools with multiple components, such as a privileged daemon or System Extension and an agent or app running in the user's desktop session, the components will likely need to communicate via XPC. In this section, I'll provide an overview of the topic, including the XPC APIs and specific examples. If you're interested in learning more, you can dig deeper into BlockBlock code, which makes extensive use of bidirectional XPC.

To some extent, XPC conforms to a traditional client/server model. One component (in our case, the BlockBlock daemon) sets up an XPC server, or *listener*. An authorized client (for example, BlockBlock's login item) can connect to the listener, then remotely invoke privileged methods implemented within the listener. Say a user responds to a BlockBlock alert, instructing the tool to block a persistently installed item, then creates a rule to automatically block related items in the future. Via XPC, BlockBlock's login item can invoke the daemon's privileged *block* and *create rule* methods. These methods run in the context of the privileged daemon to ensure that they have the appropriate permissions to remove even privileged persistent items. They can also create rules in a privileged context to help protect against malicious subversions.

Creating Listeners and Delegates

Let's explore how the BlockBlock daemon creates the XPC listener and, more importantly, ensures that only authorized clients can connect to it. The latter point is essential for security tools, because if we leave the XPC interface unprotected, nothing stops malware or anything else from connecting to it and invoking the daemon's privileged methods.

BlockBlock implements the XPC listener and connection logic in an interface named `XPCListener` that conforms to the `NSXPCListenerDelegate` protocol (Listing 11-9).

```
@interface XPCListener : NSObject <NSXPCListenerDelegate>
    @property(weak)NSXPCConnection* client;
    @property(nonatomic, retain)NSXPCListener* listener;
    ...
}
```

Listing 11-9: An XPC listener class

To create an XPC interface, you can use the `NSXPCListener initWithMachServiceName:` initialization method, which takes the name of the XPC service

as an argument. Listing 11-10 is the code from BlockBlock's `XPCListener` class that creates its XPC listener.

```
#define DAEMON_MACH_SERVICE @"com.objective-see.blockblock"

self.listener = [[NSXPCListener alloc] initWithMachServiceName:DAEMON_MACH_SERVICE];
```

Listing 11-10: Initializing an XPC listener

Note that Apple built XPC atop the much older Mach message passing framework. This explains why you'll run into method names such as `initWithMachServiceName:`.

Once you've created a listener, you should specify the *delegate*, which contains pertinent XPC delegate methods. The XPC system frameworks will automatically invoke these delegate methods if implemented. Once invoked, they can perform important tasks, such as verifying any clients.

Because BlockBlock's `XPCListener` class conforms to the `NSXPCListenerDelegate` protocol, it simply sets the listener delegate to itself. Then it invokes the listener's `resume` method to start processing client connections (Listing 11-11).

```
self.listener.delegate = self;
[self.listener resume];
```

Listing 11-11: Setting the delegate and resuming the listener

Now clients such as BlockBlock's login item can initiate a connection to the listener. But before we show exactly how the client can perform this action, we must ensure that only authorized clients can connect.

Extracting Audit Tokens

If you allow any client to connect to your privileged XPC interface, untrusted code could run the listener's privileged methods. This issue has plagued core macOS XPC listeners as well as many third-party tools. For a specific example, see my 2015 DEF CON talk, which details the exploitation of the unprotected and privileged macOS `writeConfig` XPC interface to elevate privileges to root.[6]

NOTE *Versions of macOS beginning with 13 simplify the authorization process, and I'll cover these steps in "Setting Client Requirements" on page 270. In this section, I'll cover authorization methods that make your tools compatible with earlier versions of the operating system.*

To authorize clients, we can turn to the `NSXPCListenerDelegate listener:shouldAcceptNewConnection:` method.[7] If a delegate provides an implementation of this method, the XPC subsystem will automatically invoke it whenever a client attempts to connect. The method should examine the candidate client and then return a Boolean value indicating whether to accept the client.

For authorized clients, this method should also configure the connection; I'll discuss how to do this shortly. Finally, because all connections start in a suspended state while they're being authorized and configured, this method should invoke the `resume` method on the passed-in `NSXPCConnection` object for authorized clients. This allows the connection to start processing any received messages, as well as to send its own (Listing 11-12).

```
-(BOOL)listener(NSXPCListener*)listener shouldAcceptNewConnection:
(NSXPCConnection*)newConnection {
    BOOL shouldAccept = NO;

    // Code to authorize the client, and ignore unauthorized ones, removed for brevity

    [newConnection resume];
    shouldAccept = YES;

bail:
    return shouldAccept;
}
```

Listing 11-12: Resuming a connection

While we could attempt to verify the client in several ways, many approaches are flawed or incomplete. For example, using the candidate client's process ID is dangerous, as an attacker can exploit the fact that the system reuses process IDs to coerce the listener into allowing an unauthorized client.

A better method is to check the client's audit token and retrieve its code signing information. Unfortunately, in older versions of macOS, Apple doesn't readily expose the client's audit token, which means we have to resort to some Objective-C trickery. The `listener:shouldAcceptNewConnection:` method's second argument is a pointer to an `NSXPCConnection` object, which contains information about the client attempting to connect to the XPC service. While it does contain the audit token in its `auditToken` property, this property is private, meaning we can't directly access it. Luckily, Objective-C is introspective, so we can access private properties via a class extension. In Listing 11-13, BlockBlock creates an extension to the `NSXPCConnection` class.

```
@interface ExtendedNSXPCConnection : NSXPCConnection {
    audit_token_t auditToken;
}
    @property audit_token_t auditToken;
@end
```

Listing 11-13: Extending the `NSXPCConnection` class to access its private audit token

Note that the extension defines a single property: the private audit token found within the `NSXPCConnection` class. Once we've declared this extension, we can access the private audit token of the connecting client, as shown in Listing 11-14.

```
-(BOOL)listener:(NSXPCListener*)listener shouldAcceptNewConnection:(NSXPCConnection*)
newConnection {
    ...
    audit_token_t auditToken = ((ExtendedNSXPCConnection*)newConnection).auditToken;
    ...
}
```

Listing 11-14: Accessing the connecting client's audit token

This code typecasts the `NSXPCConnection` object, representing the connecting client, as an `ExtendedNSXPCConnection` object. Then it can readily extract the client's audit token member. With an audit token in hand, the code can verify code signing information about the client, then securely verify the identity of the client and approve the connection if the client is authorized.

Extracting Code Signing Details

To verify the client's code signing information, BlockBlock's implementation of the `listener:shouldAcceptNewConnection:` delegate method takes the following steps. First, it uses the extracted audit token to obtain a dynamic code signing reference for the client process. It uses this reference to validate that the client's code signing information is valid, then extracts the information. Additionally, it extracts the client code signing flags to ensure that the client was compiled with the hardened runtime, guarding against runtime injection attacks. Finally, it checks that the validated code signing information contains the bundle ID of the BlockBlock helper application, the Objective-See developer code signing certificate, and supported client versions. Listing 11-15 shows the implementation of this requirement.

```
"❶ anchor apple generic and ❷ identifier \"com.objective-see.blockblock
.helper\" and ❸ certificate leaf [subject.CN] = \"Developer ID Application:
Objective-See, LLC (VBG97UB4TA)\" and ❹ info [CFBundleShortVersionString]
>= \"2.0.0\"";
```

Listing 11-15: A code signing requirement to validate connecting XPC clients

Chapter 3 covered code signing requirements, but let's break this one down. First, we require that the client be signed using a certificate issued by Apple to developers ❶. Next, we require the client identifier to match that of Objective-See's BlockBlock helper ❷. We also require that the client be signed with Objective-See's code signing certificate ❸. Finally, we require client versions of 2.0.0 or newer ❹, as older versions of BlockBlock's helper don't support the more recent hardened runtime, leaving them vulnerable to subversion.[8]

If all these validation and verification steps succeed, the BlockBlock daemon knows that the client attempting to connect to its XPC interface is indeed a recent version of the BlockBlock helper component and that an attacker or malware hasn't surreptitiously tampered with this component.

Listing 11-16 shows the code that implements the full client authorization. Note the use of various `SecTask*` code signing APIs, covered in

Chapter 3. As it's imperative to always check the return value of these APIs, this code contains basic error handling.

```
#define HELPER_ID @"com.objective-see.blockblock.helper"
#define SIGNING_AUTH @"Developer ID Application: Objective-See, LLC (VBG97UB4TA)"

-(BOOL)listener:(NSXPCListener*)listener shouldAcceptNewConnection:(NSXPCConnection*)
newConnection {
    BOOL shouldAccept = NO;
    audit_token_t auditToken = ((ExtendedNSXPCConnection*)newConnection).auditToken;

    OSStatus status = SecCodeCopyGuestWithAttributes(NULL, (__bridge CFDictionaryRef _Nullable)
    (@{(__bridge NSString*)kSecGuestAttributeAudit : [NSData dataWithBytes:&auditToken
    length:sizeof(audit_token_t)]}), kSecCSDefaultFlags, &codeRef);
    if(errSecSuccess != status) {
        goto bail;
    }

    status = SecCodeCheckValidity(codeRef, kSecCSDefaultFlags, NULL);
    if(errSecSuccess != status)  {
        goto bail;
    }

    status = SecCodeCopySigningInformation(codeRef, kSecCSDynamicInformation, &csInfo);
    if(errSecSuccess != status)  {
        goto bail;
    }

    uint32_t csFlags = [((__bridge NSDictionary*)csInfo)[(__bridge NSString*)
    kSecCodeInfoStatus] unsignedIntValue];
    if( !(CS_VALID & csFlags) && !(CS_RUNTIME & csFlags) ) {
        goto bail;
    }

    NSString* requirement = [NSString stringWithFormat:@"anchor apple generic and
    identifier \"%@\" and certificate leaf [subject.CN] = \"%@\" and info
    [CFBundleShortVersionString] >= \"2.0.0\"", HELPER_ID, SIGNING_AUTH];

    SecTaskRef taskRef = SecTaskCreateWithAuditToken(NULL, ((ExtendedNSXPCConnection*)
    newConnection).auditToken);

    status = SecTaskValidateForRequirement(taskRef, (__bridge CFStringRef)(requirement));
    if(errSecSuccess != status) {
        goto bail;
    }

    shouldAccept = YES;

    // Add code here to configure and finalize the NSXPCConnection.

bail:
    return shouldAccept;
}
```

Listing 11-16: Authorizing XPC clients

You may be surprised by how hard it is to protect privileged XPC interfaces. Apple eventually realized this too, and luckily, in macOS 13, it provided two new APIs specifically designed to simplify the process of ensuring that only authorized clients could connect. If your tools will run only on versions of macOS 13 or newer, you should make use of these APIs so you don't have to worry about accessing private audit tokens or manually extracting and verifying code signing information. The next section will detail these APIs.

Setting Client Requirements

On macOS 13 and newer, the `NSXPCListener` class's `setConnectionCodeSigningRequirement:` method[9] and the `NSXPCConnection` class's `setCodeSigningRequirement:` method[10] allow you to set code signing requirements on either the listener or the connection object. The first option applies to all connections, while the second applies to only specific ones, but you can use either to keep unauthorized clients from connecting to an XPC interface.

BlockBlock uses the listener method, which requires less granularity; it denies any and all connections that don't belong to the BlockBlock helper client. Recall that Listing 11-10 showed the code for initializing an XPC listener. Listing 11-17 builds on this foundation by adding code to run on macOS versions 13 and newer.

```
#define DAEMON_MACH_SERVICE @"com.objective-see.blockblock"
#define HELPER_ID @"com.objective-see.blockblock.helper"
#define SIGNING_AUTH @"Developer ID Application: Objective-See, LLC (VBG97UB4TA)"

self.listener = [[NSXPCListener alloc] initWithMachServiceName:DAEMON_MACH_SERVICE];

if(@available(macOS 13.0, *)) {
    NSString* requirement = [NSString stringWithFormat:@"anchor apple generic and
    identifier \"%@\" and certificate leaf [subject.CN] = \"%@\" and info
    [CFBundleShortVersionString] >= \"2.0.0\"", HELPER_ID, SIGNING_AUTH]; ❶

    [self.listener setConnectionCodeSigningRequirement:requirement]; ❷
}

self.listener.delegate = self;
[self.listener resume];
```

Listing 11-17: Authorizing clients on macOS versions 13 and newer

After allocating and initializing an `NSXPCListener` object, we use the Objective-C `@available` attribute with a value of `macOS 13.0, *` to instruct the compiler to execute the following lines on macOS 13 or newer only ❶, as the `setConnectionCodeSigningRequirement:` method isn't available on earlier versions of macOS.

We then dynamically initialize a code signing requirement string ❷ with which to validate any clients attempting to connect to the listener. The requirement is identical to the one shown previously. Finally, BlockBlock invokes the `setConnectionCodeSigningRequirement:` method to instruct the XPC

runtime to only accept connections from clients that conform to the specified code signing requirement string. Now we no longer have to manually verify clients; macOS will take care of it for us!

To confirm that the authorization works, compile and execute BlockBlock on macOS version 13 or newer, then attempt to connect to its XPC interface with an illegitimate client. The connection should fail, and the system's XPC library should print the following message to the unified log:

```
Default     0x0     56198  0    BlockBlock: (libxpc.dylib) Bogus check-in attempt. Ignoring.
```

Now that BlockBlock can authorize XPC clients, it can configure and then activate the connection.

Enabling Remote Connections

XPC communications usually occur in only one direction; a client connects to a listener and invokes its methods. BlockBlock, however, implements bidirectional communications. The daemon implements most of the XPC methods for tasks like blocking or removing persistent items and creating rules, and the client invokes these. However, the daemon also calls methods implemented in the client to, for example, display alerts to the user.

To facilitate this bidirectional IPC, we must configure the `NSXPCConnection` object. First, let's configure the listener object on the server side. This involves defining the remote methods that the client can invoke and specifying an object on the server side of the XPC interface that implements these methods. Both the server and the client must agree on what methods the client can remotely call. We can achieve this by setting the listener's `exportedInterface` property to an `NSXPCInterface` object that describes the protocol for the exported object.[11]

In this context, a *protocol* is simply a list of methods that conformant objects will implement.[12] We normally declare these protocols in header (*.h*) files, making them easy to include in both server and client code. Listing 11-18 is the BlockBlock daemon's XPC protocol.

```
@protocol XPCDaemonProtocol
    -(void)getPreferences:(void (^)(NSDictionary*))reply;
    -(void)updatePreferences:(NSDictionary*)preferences;
    -(void)getRules:(void (^)(NSData*))reply;
    -(void)deleteRule:(Rule*)rule reply:(void (^)(NSData*))reply;
    -(void)alertReply:(NSDictionary*)alert;
@end
```

Listing 11-18: The XPC daemon protocol

Once we've declared this protocol, the daemon can set the `exportedInterface` property to an `NSXPCInterface` object conformant to the `XPCDaemonProtocol` protocol. You can find the code to enable client connections in the `listener:shouldAcceptNewConnection:` delegate method (Listing 11-19).

```
-(BOOL)listener:(NSXPCListener*)listener shouldAcceptNewConnection:
(NSXPCConnection*)newConnection {
    // Code to authorize the client, and ignore unauthorized ones, removed for brevity

    newConnection.exportedInterface =
    [NSXPCInterface interfaceWithProtocol:@protocol(XPCDaemonProtocol)];
    ...
```

Listing 11-19: Setting the exported interface for the `NSXPCConnection`

Of course, you must also specify the object on the server side that implements these methods (in this case, the BlockBlock daemon). You can do this by setting the `exportedObject` property on the listener (Listing 11-20).

```
-(BOOL)listener:(NSXPCListener*)listener shouldAcceptNewConnection:
(NSXPCConnection*)newConnection {
    // Code to authorize the client, and ignore unauthorized ones, removed for brevity
    ...
    newConnection.exportedObject = [[XPCDaemon alloc] init];
    ...
```

Listing 11-20: Setting the object that implements the exported interface

BlockBlock creates a class named `XPCDaemon` to implement client-callable methods. As expected, this class conforms to the daemon protocol, `XPCDaemon Protocol` (Listing 11-21).

```
@interface XPCDaemon : NSObject <XPCDaemonProtocol>
@end
```

Listing 11-21: An interface conformant to `XPCDaemonProtocol`

Next, we'll briefly look at a few of the privileged XPC methods that the BlockBlock helper component running in the limited-privilege user session can invoke.

Exposing Methods

BlockBlock lets users define rules to automatically allow common persistence events. The privileged BlockBlock daemon manages these rules to keep unprivileged malware from tampering with them (for example, by adding an allow rule that permits the malware to persist). To display the rules to the user, the BlockBlock client will invoke the daemon's `getRules:` method via XPC (Listing 11-22).

```
-(void)getRules:(void (^)(NSData*))reply {
    NSData* archivedRules = [NSKeyedArchiver archivedDataWithRootObject:
    rules.rules requiringSecureCoding:YES error:nil];

    reply(archivedRules);
}
```

Listing 11-22: Returning serialized rules

Because XPC is asynchronous, methods that return data should do so in a block. The `getRules:` method declared in `XPCDaemonProtocol` takes such a block, which the caller can invoke with a data object containing the list of rules. Notice that the method's implementation is rather basic; it simply serializes the rules and sends them back to the client.

A more involved example of an XPC method is `alertReply:`, which the client invokes via XPC once a user has interacted with a persistence alert (for example, by clicking Block). The method takes a dictionary that encapsulates the alert. The user doesn't expect any response, so the method doesn't use any callback block. Listing 11-23 shows the method's main code implemented within the daemon.

```
-(void)alertReply:(NSDictionary*)alert {
    Event* event = nil;
    @synchronized(events.reportedEvents) {
      ❶ event = events.reportedEvents[alert[ALERT_UUID]];
    }

  ❷ event.action = [alert[ALERT_ACTION] unsignedIntValue];
    if(BLOCK_EVENT == event.action) {
      ❸ [event.plugin block:event];
    }
    ...
    if(YES != [alert[ALERT_TEMPORARY] boolValue]) {
      ❹ [rules add:event];
    }
}
```

Listing 11-23: Handling the user's response to an alert

First, we retrieve an object representing the persistent event from the alert dictionary using a UUID ❶. We wrap the object in a `@synchronized` block to ensure thread synchronization. Next, we extract the user-specified action (either block or allow) from the alert ❷. If the user has decided to block the persistent event, BlockBlock will call in the relevant plug-in's `block:` method. This will execute the plug-in–specific code to remove the persistent item ❸ and add a rule for the event, so long as the user didn't click the "temporary" checkbox on the alert ❹.

I mentioned that the BlockBlock daemon also needs to call methods implemented in the helper, for example, to display an alert to the user. It can do so over the same XPC interface once the helper has connected, although we need to specify a dedicated protocol. BlockBlock names this client protocol `XPCUserProtocol` (Listing 11-24). It contains methods the client will implement and that the daemon can remotely invoke over XPC.

```
@protocol XPCUserProtocol
    -(void)alertShow:(NSDictionary*)alert;
    ...
@end
```

Listing 11-24: The XPC user protocol

Back in the `listener:shouldAcceptNewConnection:` method, we configure the listener to allow the daemon to invoke the client's remote methods (Listing 11-25).

```
-(BOOL)listener:(NSXPCListener*)listener shouldAcceptNewConnection:
(NSXPCConnection*)newConnection {
    // Code to authorize the client, and ignore unauthorized ones, removed for brevity
    ...
    newConnection.remoteObjectInterface =
    [NSXPCInterface interfaceWithProtocol:@protocol(XPCUserProtocol)];
```

Listing 11-25: Setting the remote object interface

We set the `remoteObjectInterface` property and specify the `XPCUserProtocol` protocol.

Initiating Connections

So far, I've shown how the BlockBlock daemon sets up an XPC listener, exposes methods, and ensures that only authorized clients can connect. However, I haven't yet shown how the client initiates a connection or how it and the daemon remotely invoke the XPC methods.

Once the BlockBlock daemon is running, its XPC interface becomes available for authorized connections. To connect to the daemon, the BlockBlock helper uses the `NSXPCConnection` object's `initWithMachServiceName:options:` method, specifying the same name used by the daemon (Listing 11-26).

```
#define DAEMON_MACH_SERVICE @"com.objective-see.blockblock"

NSXPCConnection* daemon = [[NSXPCConnection alloc]
initWithMachServiceName:DAEMON_MACH_SERVICE options:0];
```

Listing 11-26: Initializing a connection to the daemon XPC service

As we did on the server side, we must set the protocol for the remote object interface. Because we're now on the client side, the "remote object interface" in this case refers to the XPC object on the daemon that exposes remotely invocable methods (Listing 11-27).

```
#define DAEMON_MACH_SERVICE @"com.objective-see.blockblock"

NSXPCConnection* daemon = [[NSXPCConnection alloc]
initWithMachServiceName:DAEMON_MACH_SERVICE options:0];

daemon.remoteObjectInterface =
[NSXPCInterface interfaceWithProtocol: @protocol(XPCDaemonProtocol)]; ❶

daemon.exportedInterface = [NSXPCInterface interfaceWithProtocol:@protocol(XPCUserProtocol)];
daemon.exportedObject = [[XPCUser alloc] init]; ❷

[daemon resume]; ❸
```

Listing 11-27: Setting up the XPC connection object on the client side

Recall that this object conforms to `XPCDaemonProtocol`, so we specify it here ❶. Also, because the daemon needs to call methods implemented in the client, the client needs to set up its own exported object. It does this via the `exportedInterface` and `exportedObject` methods ❷. The former specifies the protocol (`XPCUserProtocol`), while the latter specifies the object (`XPCUser`) in the client that implements the exported XPC methods. Finally, we resume the connection ❸, which triggers the actual connection to the daemon's XPC listener.

Invoking Remote Methods

At this point, we've finished implementing the XPC connection. I'll end this discussion of BlockBlock's XPC utilization by showing how it actually invokes remote methods, focusing on the more common case of the client side. To abstract its communications with the daemon, the BlockBlock client uses a custom class named `XPCDaemonClient`. The code in Listing 11-26 that establishes an XPC connection lives in this class, as does the code that invokes the remote XPC methods.

To connect to the daemon and invoke one of its remote privileged XPC methods (for example, to get the current rules), the client can execute the code in Listing 11-28.

```
XPCDaemonClient* xpcDaemonClient = [[XPCDaemonClient alloc] init];
NSArray* rules = [[xpcDaemonClient getRules];
```

Listing 11-28: Invoking remote XPC methods

Let's take a closer look at the `getRules` method, which invokes the daemon's remotely exposed corresponding `getRules:` method. This method provides a good example of how you can invoke XPC methods, taking into account their nuances. Note that though the method contains additional logic to deserialize the rules it receives from the daemon, here we're only focusing on the XPC logic (Listing 11-29).

```
-(NSArray*)getRules {
    __block NSDictionary* unarchivedRules = nil;
    ...
    [[self.daemon synchronousRemoteObjectProxyWithErrorHandler:^(NSError* proxyError) { ❶
        // Code to handle any errors removed for brevity ❷
    }] getRules:^(NSData* archivedRules) {
        // Code to process the serialized rules from the daemon removed for brevity ❸
    }];
    ...
    return rules;
}
```

Listing 11-29: Getting rules from the daemon

First, the code invokes the `NSXPCConnection` class's synchronous connection method ❶. While XPC is generally asynchronous, we're expecting the daemon to return data, so using a synchronous call makes the most sense in

this situation. In other places, BlockBlock uses the more common asynchronous `remoteObjectProxyWithErrorHandler:` method.

The `XPCDaemonClient` class's `init` method previously established the connection and saved it in the instance variable named `daemon`. The connection method returns the remote object, which exposes remotely invocable XPC methods. If any errors occur while retrieving this object, the code invokes an error block ❷.

With a remote object in hand, we can then invoke its methods, such as its `getRules:` method. To return data, this XPC call takes a reply block; Listing 11-22 showed the implementation of this method, found within the daemon. When the call completes, the block executes, taking as a parameter a data object containing the serialized rules from the daemon ❸.

Conclusion

BlockBlock's approach is simple: detect persistent items, alert the user, and allow them to remove unwanted items. While straightforward, this design has proved incredibly effective against even the most sophisticated of persistent Mac malware.

In this chapter, you saw how to request an Endpoint Security entitlement from Apple. You also looked at BlockBlock's design, its use of Endpoint Security events, and its bidirectional XPC communications. If you're building your own security tools, I encourage you to draw from the system frameworks, APIs, and mechanisms that BlockBlock employs.

The next chapter explores a tool designed to heuristically detect some of the most insidious malware specimens: those that surreptitiously spy on victims through their mics and webcams.

Notes

1. "Writing Bad @$$ Lamware for OS X," *reverse.put.as*, August 7, 2015, *https://reverse.put.as/2015/08/07/writing-bad-lamware-for-os-x/*.
2. "TN3125: Inside Code Signing: Provisioning Profiles," Apple Developer Documentation, *https://developer.apple.com/documentation/technotes/tn3125-inside-code-signing-provisioning-profiles*.
3. "Signing a Daemon with a Restricted Entitlement," Apple Developer Documentation, *https://developer.apple.com/documentation/xcode/signing-a-daemon-with-a-restricted-entitlement*.
4. asfdadsfasdfasdfsasdafads, "Endpoint Security Event: ES_EVENT_TYPE_NOTIFY_BTM_LAUNCH_ITEM_ADD is . . . broken?," Apple Developer Forums, November 15, 2024, *https://developer.apple.com/forums/thread/720468*.
5. Keith Harrison, "Automatic Property Synthesis with Xcode 4.4," *Use Your Loaf*, August 1, 2012, *https://useyourloaf.com/blog/property-synthesis-with-xcode-4-dot-4/*.

6. Patrick Wardle, "Stick That in Your (Root) Pipe and Smoke It," Speaker Deck, August 9, 2015, *https://speakerdeck.com/patrickwardle/stick-that-in-your-root-pipe-and-smoke-it*.

7. "listener:shouldAcceptNewConnection:," Apple Developer Documentation, accessed May 25, 2024, *https://developer.apple.com/documentation/foundation/nsxpclistenerdelegate/1410381-listener?language=objc*.

8. You can read about such subversive attacks in "The Story Behind CVE-2019-13013," *Objective Development*, August 26, 2019, *https://blog.obdev.at/what-we-have-learned-from-a-vulnerability*, which details the exploitation of a popular commercial macOS firewall product.

9. "setConnectionCodeSigningRequirement:," Apple Developer Documentation, *https://developer.apple.com/documentation/foundation/nsxpclistener/3943310-setconnectioncodesigningrequirem?language=objc*.

10. "setCodeSigningRequirement:," Apple Developer Documentation, *https://developer.apple.com/documentation/foundation/nsxpcconnection/3943309-setcodesigningrequirement?language=objc*.

11. "exportedInterface," Apple Developer Documentation, *https://developer.apple.com/documentation/foundation/nsxpcconnection/1408106-exportedinterface*.

12. "Working with Protocols," Apple Developer Documentation, *https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithProtocols/WorkingwithProtocols.html*.