5

PERSISTENCE

Arguably one of the best ways to detect malicious threats on macOS is to focus on persistence. Here, *persistence* refers to the means by which software, including malware, installs itself on a system to ensure it will automatically re-execute upon startup, user login, or some other deterministic event. Otherwise, it might never run again if the user logs out or the system reboots. In this chapter, I focus solely on enumerating persistent items. In Part II, where I cover approaches that allow events to be observed as they occur, I'll discuss how to leverage Apple's Endpoint Security to monitor for persistence events.

As a shared characteristic of most malware, persistence serves as a robust detection mechanism capable of uncovering most infections. On

macOS, malware generally persists in one of two ways: as launch items (daemons or agents) or as login items. In this chapter, I'll show you exactly how to enumerate such items to reveal almost any Mac malware specimen.

Of course, not all macOS malware persists. For example, ransomware that encrypts user files or stealers that grab and exfiltrate sensitive user data often have no need to run multiple times, and thus rarely install themselves persistently.

On the other hand, legitimate programs designed to run continuously, such as auto-updaters, security tools, or even simple helper utilities, also tend to persist. Thus, the fact that something is persistently installed doesn't mean our code should flag it as malicious.

Examples of Persistent Malware

Because this chapter focuses on uncovering malware that persists as either a login item or a launch item, let's start with a brief example of each. Initially disclosed by the researcher Taha Karim, the WindTail malware targeted employees working in government and critical infrastructure in the Middle East.[1] In a detailed research paper,[2] I noted that the malware, which often masquerades as a PowerPoint presentation named *Final_Presentation*, persists itself as a login item to ensure that it automatically re-executes each time the user logs in. In the malware's application bundle, we find its main binary, a file named *usrnode*. Decompiling this file uncovers the persistence logic at the start of its `main` function:

```
int main(int argc, const char* argv[])
    r12 = [NSURL fileURLWithPath:NSBundle.mainBundle.bundlePath];

    rbx = LSSharedFileListCreate(0x0, _kLSSharedFileListSessionLoginItems, 0x0);
    LSSharedFileListInsertItemURL(rbx, _kLSSharedFileListItemLast, 0x0, 0x0, r12, 0x0, 0x0);
    ...
}
```

Once the malware determines where on the host it's running from, it invokes the `LSSharedFileListCreate` and `LSSharedFileListInsertItemURL` functions to install itself as a persistent login item. This login item makes the malware visible in the Login Items pane of the System Preferences application (Figure 5-1). Apparently, the malware authors considered this an acceptable trade-off for persistence.

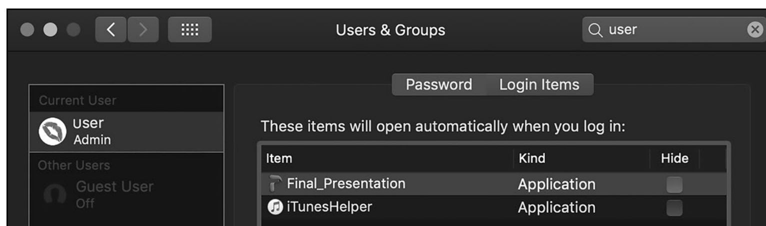

Figure 5-1: WindTail persists itself as a login item named Final_Presentation.

Let's take a look at another persistent macOS malware specimen. Named DazzleSpy, this sophisticated nation-state malware leveraged zero-day vulnerabilities to remotely infect macOS users.[3] While DazzleSpy's infection vector posed detection challenges, the malware's approach to persistence was rather obvious, giving defenders a straightforward way to detect it.

After gaining initial code execution and escaping the browser sandbox, DazzleSpy would persist itself as a launch agent that masqueraded as an Apple software updater. To persist as a launch agent, an item usually creates a property list in one of the *LaunchAgents* directories. DazzleSpy creates a property list within the current user's *Library/LaunchAgents* directory and names its property list *com.apple.softwareupdate.plist*. The malware's binary hardcodes references to the launch agent directory, as well as to the name of the plist, making them readily visible in the output of the `strings` command:

```
% strings - DazzleSpy/softwareupdate
...
%@/Library/LaunchAgents
/com.apple.softwareupdate.plist
```

If we load the malware in a decompiler, we find a class method named `installDaemon` that makes use of these strings. As its name implies, the method will persistently install the malware (albeit not as a launch daemon, but rather as an agent):

```
+(void)installDaemon {
    rax = NSHomeDirectory();
    ...
    var_78 = [NSString stringWithFormat:@"%@/Library/LaunchAgents", rax];
    var_80 = [var_78 stringByAppendingFormat:@"/com.apple.softwareupdate.plist"];
    ...
    var_90 = [[NSMutableDictionary alloc] init];
    var_98 = [[NSMutableArray alloc] init];
    ...
    rax = @(YES);
    [var_90 setObject:rax forKey:@"RunAtLoad"];
```

```
    [var_90 setObject:@"com.apple.softwareupdate" forKey:@"Label"];
    [var_90 setObject:var_98 forKey:@"ProgramArguments"];
    ...
    [var_90 writeToFile:var_80 atomically:0x0];
...
}
```

From this decompilation, we can see that the malware first dynamically builds a path to the current user's *Library/LaunchAgents* directory and then appends the string *com.apple.softwareupdate.plist* to it. It then builds a dictionary with keys such as `RunAtLoad`, `Label`, and `ProgramArguments`, whose values describe how to restart the persisted item, how to identify it, and its path. To complete the persistence, the malware writes this dictionary to the property list file in the launch agent directory.

By executing the malware on an isolated analysis machine under the watchful eye of a file monitor, we can confirm DazzleSpy's persistence. As expected, the file monitor shows the binary (*softwareupdate*) creating its property list file in the current user's *LaunchAgents* directory:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
...
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/Users/User/Library/LaunchAgents/com.apple.softwareupdate.plist",
    "process" : {
      "pid" : 1469,
      "name" : "softwareupdate",
      "path" : "/Users/User/Desktop/softwareupdate"
    }
  }
}
```

Then, by examining the contents of this newly created file, we can find the path to which the malware has persistently installed itself, */Users/User/.local/softwareupdate*:

```
<?xml version=”1.0” encoding=”UTF-8”?>
...
<plist version="1.0">
<dict>
    <key>KeepAlive</key>
    <true/>
    <key>Label</key>
    <string>com.apple.softwareupdate</string>
    <key>ProgramArguments</key>
    <array>
        <string>/Users/User/.local/softwareupdate</string>
        <string>1</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
```

```
    <key>SuccessfulExit</key>
    <true/>
</dict>
</plist>
```

The malware set the RunAtLoad key to true, so macOS will automatically restart the specified binary each time the user logs in. In other words, DazzleSpy has attained persistence.

At the start of this chapter, I mentioned that legitimate software also persists. How can you determine whether a persisted item is malicious? Arguably the best way involves examining the item's code signing information using the approaches described in Chapter 3. Legitimate items should be signed by readily recognizable companies and notarized by Apple.

Malicious persisted items often have common characteristics too. Consider DazzleSpy, which runs from the hidden *.local* directory and isn't signed or notarized. The name of the malware's property list, *com.apple.softwareupdate*, suggests that this persistent item belongs to Apple. However, Apple never installs persistent components to users' *LaunchAgents* directories, and all of its launch items reference binaries signed solely by Apple proper. In these respects, DazzleSpy isn't an outlier; most malicious persistent items are equally easy to classify as suspicious due to such anomalies.

Background Task Management

How can we determine whether an item has persisted? A naive approach is to simply enumerate all *.plist* files found in the launch item directories, which include the system and user *LaunchDaemon* and *LaunchAgent* directories. However, as of macOS 13, Apple encourages developers to move their launch items directly into their application bundles.[4] These changes essentially deprecate persistence via a user's launch item directories, meaning that manually enumerating persistent items requires scanning every application bundle, which is inefficient. Moreover, software can persist as login items, which don't leverage property lists or dedicated directories.

Luckily, starting with macOS 13, Apple has consolidated the management of the most common persistence mechanisms (including launch agents, launch daemons, and login items) into a proprietary subsystem named *Background Task Management*. This subsystem provides the list of login and launch items that populate the Login Items pane in the System Preferences application (Figure 5-2).

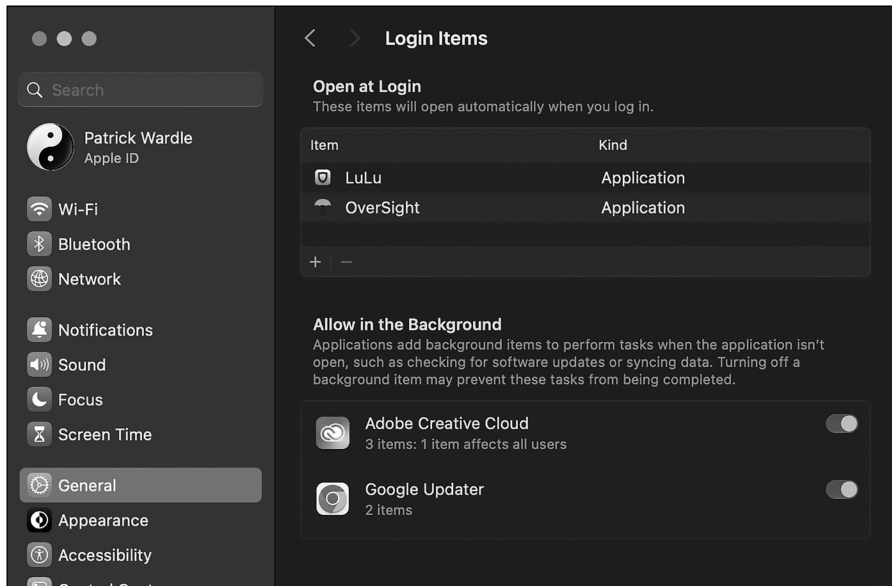

Figure 5-2: Login and launch items shown in the System Preferences app

On my computer, several of my Objective-See tools install themselves as login items, while Adobe's cloud-syncing app and Google Chrome's updater install persistent launch items.

Of course, we want the ability to obtain this list of persistent items programmatically, as any persistent malware will likely show up here as well. Although the components of the Background Task Management subsystem are proprietary and closed source, dynamic analysis reveals that the subsystem stores detailed metadata about the persistent items it tracks in a single database file. For our purposes, the presence of this centralized database is a godsend. Unfortunately, as its format is proprietary and undocumented, we have a bit of work in front of us if we'd like to use it.

Examining the Subsystem

Let's walk through the Background Task Management subsystem's interactions with this database. Understanding these operations will help us create a tool capable of programmatically extracting its contents. Using a file monitor, we can see that when an item is persisted, the Background Task Management daemon, *backgroundtaskmanagementd*, updates a file in the */private/var/db/com.apple.backgroundtaskmanagement/* directory. To perform this operation atomically, it first creates a temporary file, then moves it into the *com.apple.backgroundtaskmanagement* directory via a `rename` operation:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
```

```
    "destination" :
    "/private/var/folders/zz/.../TemporaryItems/.../BackgroundItems-vx.btm",
    "process" : {
       "pid" : 612,
       "name" : "backgroundtaskmanagementd",
       ...
     }
  }
  ...
}

{
  "event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
  "file" : {
    "destination" :
    "/private/var/folders/zz/.../TemporaryItems/.../BackgroundItems-vx.btm",
    "process" : {
      "pid" : 612,
      "name" : "backgroundtaskmanagementd",
      ...
    }
  }
  ...
}

{
  "event" : "ES_EVENT_TYPE_NOTIFY_RENAME",
  "file" : {
    "source" :
    "/private/var/folders/zz/.../TemporaryItems/.../BackgroundItems-vx.btm",
    "destination" :
    "/private/var/db/com.apple.backgroundtaskmanagement/BackgroundItems-vx.btm",
    "process" : {
      "pid" : 612,
      "name" : "backgroundtaskmanagementd",
      ...
    }
  }
  ...
}
```

If we disassemble the daemon's binary, located in the */System/Library/PrivateFrameworks/BackgroundTaskManagement.framework/Versions/A/Resources/* directory, we find references to a format string, `BackgroundItems-v%ld.btm`, in `storeNameForDatabaseVersion:`, a method of the BTMStore class:

```
+[BTMStore storeNameForDatabaseVersion:]
    pacibsp
    sub    sp, sp, #0x20
    stp    fp, lr, [sp, #0x10]
    add    fp, sp, #0x10
    nop
    ldr    x0, =_OBJC_CLASS_$_NSString
```

```
str    x2, [sp, #0x10 + var_10]
adr    x2, #0x100031f10           ; @"BackgroundItems-v%ld.btm"
...
```

Further reverse engineering reveals that the name of the database contains a version number, which increases as newer versions of macOS are released. In the examples shown here, we've abstracted this version number with an *x*, but on your system, it's likely to be 8 or higher. Using the `file` command, we can see that the contents of the *BackgroundItems-vx.btm* file are stored as a binary property list. To view these details yourself, be sure to supply the correct version number for your system when running the command:

```
% file /private/var/db/com.apple.backgroundtaskmanagement/BackgroundItems-vx.btm
/private/var/db/com.apple.backgroundtaskmanagement/BackgroundItems-vx.btm:
Apple binary property list
```

We can convert the contents of a binary property into XML using `plutil`. Unfortunately, the resulting XML contains not only spelling mistakes but also serialized objects that aren't readily human readable:

```
% plutil -p /private/var/db/com.apple.backgroundtaskmanagement/BackgroundItems-vx.btm
{
  "$archiver" => "NSKeyedArchiver"
  "$objects" => [
    0 => "$null"
    1 => {
      "$class" =>
      <CFKeyedArchiverUID 0x600002854240 [0x1e3bcf9a0]>{value = 265}

      "itemsByUserIdentifier" =>
      <CFKeyedArchiverUID 0x600002854260 [0x1e3bcf9a0]>{value = 2}

      "mdmPaloadsByIdentifier" =>
      <CFKeyedArchiverUID 0x600002854280 [0x1e3bcf9a0]>{value = 263}

      "userSettingsByUserIdentifier" =>
      <CFKeyedArchiverUID 0x6000028542a0 [0x1e3bcf9a0]>{value = 257}
    }
    ...

    265 => {
      "$classes" => [
         0 => "Storage"
         1 => "NSObject"
      ]
      "$classname" => "Storage"
    }
    ...
```

Serialization is the process of taking an initialized, in-memory object and converting it to a format in which it can be saved (for example, to a

file). While serialization is an efficient way for programs to interact with objects, serialized objects aren't generally human readable. Moreover, if the objects are of an undocumented class, we must first understand the internal details of the class before we can write code that makes sense of them.

As part of the Background Task Management subsystem, Apple ships a command line utility named `sfltool` that can interact with *BackgroundItems-vx .btm* files. If executed with the `dumpbtm` flag, the tool will deserialize and print out the file's contents:

```
# sfltool dumpbtm

#1:
              UUID: 8C271A5F-928F-456C-B177-8D9162293BA7
              Name: softwareupdate
    Developer Name: (null)
              Type: legacy daemon (0x10010)
       Disposition: [enabled, allowed, visible, notified] (11)
        Identifier: com.apple.softwareupdate
               URL: file:///Library/LaunchDaemons/com.apple.softwareupdate.plist
   Executable Path: /Users/User/.local/softwareupdate
        Generation: 1
 Parent Identifier: Unknown Developer

#2:
     UUID: 9B6C3670-2946-4F0F-B58C-5D163BE627C0
              Name: ChmodBPF
    Developer Name: Wireshark
   Team Identifier: 7Z6EMTD2C6
              Type: curated legacy daemon (0x90010)
       Disposition: [enabled, allowed, visible, notified] (11)
        Identifier: org.wireshark.ChmodBPF
               URL: file:///Library/LaunchDaemons/org.wireshark.ChmodBPF.plist
   Executable Path: /Library/Application Support/Wireshark/ChmodBPF/ChmodBPF
        Generation: 1
 Assoc. Bundle IDs: [org.wireshark.Wireshark ]
 Parent Identifier: Wireshark
```

In this example, the deserialized objects include DazzleSpy (*software update*) and Wireshark's *ChmodBPF* daemon. As `sfltool` can produce deserialized output from the proprietary database, reverse engineering it should help us understand its deserialization and parsing logic. This, in turn, should enable us to write our own parser capable of enumerating all persistent items managed by the Background Task Management subsystem, including any malware.

Dissecting sfltool

While the focus of this book is not on reverse engineering, I'll briefly discuss how to dissect `sfltool` so you can understand its interactions with other Background Task Management components and the ever-so-important *.btm*

file. In a terminal, let's begin by streaming messages from the system log while running `sfltool` with the `dumpbtm` flag:

```
% log stream
...
backgroundtaskmanagementd: -[BTMService listener:shouldAcceptNewConnection:]:
connection=<NSXPCConnection: 0x152307aa0> connection from pid 52886 on mach service named
com.apple.backgroundtaskmanagement

backgroundtaskmanagementd dumpDatabaseWithAuthorization: error=Error
Domain=NSOSStatusErrorDomain Code=0 "noErr: Call succeeded with no error"
```

As you can see in the log output (which I've slightly modified for brevity), the Background Task Management daemon has received a message from a process with an ID of 52886 corresponding to the running instance of `sfltool`. You can see that the tool has made an XPC connection to the daemon. If the connection succeeds, `sfltool` can then invoke remote methods found within the daemon. For example, from the log messages, you see that it invoked the daemon's `dumpDatabaseWithAuthorization:` method to get the contents of the Background Task Management database.

In Listing 5-1, we try to implement this same approach. We leverage the private `BackgroundTaskManagement` framework, which implements necessary classes, such as `BTMManager`, and methods including the client-side `dumpDatabaseWithAuthorization:error:`.

```
#import <dlfcn.h>
#import <Foundation/Foundation.h>
#import <SecurityFoundation/SFAuthorization.h>

#define BTM_DAEMON "/System/Library/PrivateFrameworks/\
BackgroundTaskManagement.framework/Resources/backgroundtaskmanagementd"

@interface BTMManager : NSObject
    +(id)shared;
    -(id)dumpDatabaseWithAuthorization:(SFAuthorization*)arg1 error:(id*)arg2;
@end

int main(int argc, const char* argv[]) {
    void* btmd = dlopen(BTM_DAEMON, RTLD_LAZY);

    Class BTMManager = NSClassFromString(@"BTMManager");
    id sharedInstance = [BTMManager shared];

    SFAuthorization* authorization = [SFAuthorization authorization];
    [authorization obtainWithRight:"system.privilege.admin"
    flags:kAuthorizationFlagExtendRights error:NULL];

    id dbContents = [sharedInstance dumpDatabaseWithAuthorization:authorization error:NULL];
    ...
}
```

Listing 5-1: Attempting to dump the Background Task Management database

Unfortunately, this approach fails. As shown in the following log messages, the failure appears to be due to the fact that our binary (which, in this instance, has a process ID of 20987) doesn't possess a private Apple entitlement needed to connect to the Background Task Management daemon:

```
% log stream
...
backgroundtaskmanagementd: -[BTMService listener:shouldAcceptNewConnection:]:
process with pid=20987 lacks entitlement 'com.apple.private.backgroundtaskmanagement.manage'
or deprecated entitlement 'com.apple.private.coreservices.canmanagebackgroundtasks'
```

We can confirm that this is why we can't connect to the daemon by reverse engineering the code in the daemon responsible for handling new XPC connections from clients:

```
/* @class BTMService */
-(BOOL)listener:(NSXPCListener*)listener
shouldAcceptNewConnection:(NSXPCConnection*)newConnection {
    ...
    x24 = [x0 valueForEntitlement:@"com.apple.private.coreservices.canmanagebackgroundtasks"];
    ...
    if(objc_opt_isKindOfClass(x24, objc_opt_class(@class(NSNumber))) == 0x0 ||
    [x24 boolValue] == 0x0) {
        // Reject the client that is attempting to connect.
    }
```

In this disassembly, you can see the check for the private entitlement *com.apple.private.coreservices.canmanagebackgroundtasks*, which matches the one we saw in the logs. If the client doesn't hold it (or the newer *com.apple.private.backgroundtaskmanagement.manage* entitlement), the system will deny the connection.

Using the `codesign` utility, you can see that `sfltool` indeed contains the necessary entitlement:

```
% codesign -d --entitlements - /usr/bin/sfltool
Executable=/usr/bin/sfltool
[Dict]
    [Key] com.apple.private.coreservices.canmanagebackgroundtasks
    [Value]
        [Bool] true
    [Key] com.apple.private.sharedfilelist.export
    [Value]
        [Bool] true
```

Since we can't obtain the private Apple entitlement needed to connect to the Background Task Management daemon for our own program, we're left having to access and parse the database directly from disk.

When given full disk access, it's easy to access the database's contents. However, parsing its contents requires a bit more work, as it contains undocumented serialized objects. Luckily, continued reverse engineering reveals

that once the daemon has read the contents of the database, its deserialization logic starts in a method named `_decodeRootData:error:`

```
-(void*)_decodeRootData:(NSData*)data error:(void**)arg3 {
    ...
    x0 = [NSKeyedUnarchiver alloc];
    x21 = [x0 initForReadingFromData:data error:&error];
    ...
    x0 = [x21 decodeObjectOfClass:objc_opt_class(@class(Storage)) forKey:@"store"];
```

When the Background Task Management daemon reads the contents of the database, it performs deserialization by following these standard steps:

1. Reading the contents of the database into memory as an `NSData` object
2. Initializing an `NSKeyedUnarchiver` object with this data
3. Deserializing the objects in the unarchiver via a call to the `NSKeyedUnarchiver decodeObjectOfClass:forKey:` method

Take note of the serialized class name, `Storage`, and its key in the archiver, `store`, as these will come into play shortly. Also note that when the `decodeObjectOfClass:forKey:` method is invoked, the `initWithCoder:` method of any embedded object is also automatically invoked behind the scenes. This allows objects to perform their own deserialization.

Writing a Background Task Management Database Parser

We're now ready to write our own parser. Let's take what we've learned through reverse engineering and write a tool capable of deserializing the metadata of all persistent items found in the Background Task Management database. I'll walk through the relevant code snippets here, but you can find the entire code for this parser, dubbed *DumpBTM*, in Objective-See's GitHub repository at *https://github.com/objective-see/DumpBTM*. At the end of this discussion, I'll show how you can make use of this library in your own code to programmatically obtain a list of items persisting on any macOS system.

Finding the Database Path

Let's begin by writing some code that dynamically finds the path of the database. Although it's located in the */private/var/db/com.apple.backgroundtaskmanagement/* directory, Apple occasionally bumps up the version number in the name across releases of macOS. Even with these name changes, though, finding the database is easy enough through its unique extension, *.btm*. The code in Listing 5-2 uses a simple predicate to find all *.btm* files in the *com.apple.backgroundtaskmanagement* directory. There should only be one, but to be safe, the code grabs the one with the highest version.

```
#define BTM_DIRECTORY @"/private/var/db/com.apple.backgroundtaskmanagement/"

NSURL* getPath(void) {
```

```
❶ NSArray* files = [NSFileManager.defaultManager contentsOfDirectoryAtURL:
  [NSURL fileURLWithPath:BTM_DIRECTORY] includingPropertiesForKeys:nil options:0 error:nil];

❷ NSArray* btmFiles = [files filteredArrayUsingPredicate:[NSPredicate
  predicateWithFormat:@"self.absoluteString ENDSWITH '.btm'"]];

❸ return btmFiles.lastObject;
}
```

Listing 5-2: Finding the most recent Background Task Management database

First, the code creates a list of all files in the directory ❶. Then, via the predicate `self.absoluteString ENDSWITH '.btm'` and the method `filteredArrayUsingPredicate:`, it creates a second list containing solely *.btm* files ❷. It then returns the last file in this list, which should be the one with the highest version ❸.

Deserializing Background Task Management Files

I noted that the serialized objects in the Background Task Management file are instances of undocumented classes specific to the subsystem. To deserialize them, we must, at a minimum, provide a class declaration. We found these classes embedded in the daemon, including the top-level object in the serialized database that belongs to an undocumented class named `Storage`. Recall that we also saw this class name in the `plutil` output.

This class contains various instance variables that describe its properties, including a dictionary called `itemsByUserIdentifier`. To deserialize the `Storage` object, we create the declaration shown in Listing 5-3.

```
@interface Storage : NSObject <NSSecureCoding>
    @property(nonatomic, retain)NSDictionary* itemsByUserIdentifier;
@end
```

Listing 5-3: The `Storage` class interface

Further reverse engineering reveals more details about the `Storage` class's `itemsByUserIdentifier` dictionary. For example, it contains key-value pairs whose values are of another undocumented Background Task Management class named `ItemRecord`. The `ItemRecord` class contains metadata about each persistent item managed by the subsystem, such as its path, its code signing information, and its state (for example, enabled or disabled).

Again, as `ItemRecord` is an undocumented class, making use of it in our code requires providing a declaration extracted from the daemon. Listing 5-4 shows such a declaration.

```
@interface ItemRecord : NSObject <NSSecureCoding>
    @property NSInteger type;
    @property NSInteger generation;
    @property NSInteger disposition;
    @property(nonatomic, retain)NSURL* url;
    ...
```

```
    @property(nonatomic, retain)NSString* identifier;
    @property(nonatomic, retain)NSString* developerName;
    @property(nonatomic, retain)NSString* executablePath;
    @property(nonatomic, retain)NSString* teamIdentifier;
    @property(nonatomic, retain)NSString* bundleIdentifier;
@end
```

Listing 5-4: The `ItemRecord` class interface

With the relevant classes declared, we're almost ready to trigger the serialization of all objects in the Background Task Management file. However, as the deserialization process invokes each object's `initWithCoder:` method, and each object conforms to the `NSSecureCoding` protocol, we should provide an implementation of this method to keep the linker happy and ensure that deserialization succeeds. To reimplement the `initWithCoder:` methods for the undocumented objects, we can use a disassembler to find their implementations. For example, here is the decompilation of the `ItemRecord` object's `initWithCoder:` method:

```
-(void*)initWithCoder:(NSCoder*)decoder {
  x0 = objc_opt_class(@class(NSUUID));
  x0 = [decoder decodeObjectOfClass:x0 forKey:@"uuid"];
  self.uuid = x0;

  x0 = objc_opt_class(@class(NSString));
  x0 = [decoder decodeObjectOfClass:x0 forKey:@"executablePath"];
  self.executablePath = x0;

  x0 = objc_opt_class(@class(NSString));
  x0 = [decoder decodeObjectOfClass: x0 forKey:@"teamIdentifier"];
  self.teamIdentifier = x0;
  ...
}
```

We can easily mimic the method in our own code (Listing 5-5).

```
-(id)initWithCoder:(NSCoder *)decoder {
    self = [super init];
    if(nil != self) {
        self.uuid = [decoder decodeObjectOfClass:[NSUUID class] forKey:@"uuid"];

        self.executablePath =
        [decoder decodeObjectOfClass:[NSString class] forKey:@"executablePath"];

        self.teamIdentifier =
        [decoder decodeObjectOfClass:[NSString class] forKey:@"teamIdentifier"];
        ...
    return self;
}
```

Listing 5-5: A reimplementation of the `ItemRecord` `initWithCoder:` method

In our reimplementation of the `ItemRecord` object's `initWithCoder:` method, we deserialize the properties of the object, including its UUID, executable path, team identifier, and more. This is as easy as invoking the `decodeObjectOfClass:forKey:` method for each property on the serialized object that is passed in as an `NSCoder`.

However, there is a simpler way to access these methods. As you saw in the disassembly, the Background Task Management daemon contains class implementations of serialized `Storage` and `ItemRecord` objects, including their `initWithCoder:` methods. Thus, if we load and link the daemon binary into our process's address space, we'll have access to those methods without needing to reimplement them ourselves. As all executables are now compiled in a position-independent manner, we can link to anything we'd like in our own program, including the daemon. Listing 5-6 contains the code to load and link the daemon, then makes use of its objects when triggering the full deserialization of the objects stored in the database.

```
#define BTM_DAEMON "/System/Library/PrivateFrameworks/\
BackgroundTaskManagement.framework/Resources/backgroundtaskmanagementd"

❶ void* btmd = dlopen(BTM_DAEMON, RTLD_LAZY);

❷ NSURL* path = getPath();
❸ NSData* data = [NSData dataWithContentsOfURL:path options:0 error:NULL];

❹ NSKeyedUnarchiver* keyedUnarchiver =
[[NSKeyedUnarchiver alloc] initForReadingFromData:data error:NULL];

❺ Storage* storage = [keyedUnarchiver decodeObjectOfClass:
[NSClassFromString(@"Storage") class] forKey:@"store"];
```

Listing 5-6: Deserializing Background Task Management objects

After invoking the `dlopen` function ❶, which loads and links the Background Task Management daemon into a process's memory space, the code invokes a helper function we've written to get the path of the system's Background Task Management database file ❷. Once it has found and loaded the contents of the database into memory ❸, the code initializes a keyed unarchiver object with the database data ❹.

Now the code is ready to trigger the deserialization of the objects in the database via the keyed archiver's `decodeObjectOfClass:forKey:` method. Previously, I noted that the class of the database's top-level object is named `Storage`. As it's undocumented, we dynamically resolve it via `NSClassFromString(@"Storage")`. This resolution succeeds because we've loaded the daemon that implements this class into our process space. For the key required to begin the deserialization, we mimic the daemon by specifying the string `"store"` ❺.

Behind the scenes, this code will trigger an invocation of the `Storage` class's `initWithCoder:` method, giving it a chance to deserialize the top-level `Storage` object in the database. Recall that this object includes a dictionary

containing an ItemRecord object describing each persisted item. An invocation to the ItemRecord class's initWithCoder: method will automatically deserialize these embedded objects.

Accessing Metadata

Once we've completed the deserialization, we can access the metadata about each item persisted on the system and managed by Background Task Management (Listing 5-7).

```
int itemNumber = 0;

❶ for(NSString* key in storage.itemsByUserIdentifier) {
    ❷ NSArray* items = storage.itemsByUserIdentifier[key];
    for(ItemRecord* item in items) {
        printf(" #%d\n", ++itemNumber);
      ❸ printf(" %s\n", [[item performSelector:NSSelectorFromString
        (@"dumpVerboseDescription")] UTF8String]);
    }
}
```

Listing 5-7: Printing deserialized items

Accessing the metadata is as simple as iterating over the deserialized Storage object's itemsByUserIdentifier dictionary ❶, which organizes the persistent items by user UUID ❷. For all ItemRecord objects, we can invoke the class's dumpVerboseDescription method ❸ to print out each object in a nicely formatted manner. Because we didn't declare this method in the class interface, we instead use the Objective-C performSelector: method to invoke it by name.

Compiling and running the code produces output that provides the same information as Apple's closed source sfltool:

```
% ./dumpBTM
Opened /private/var/db/com.apple.backgroundtaskmanagement/BackgroundItems-vx.btm
...
#1
                 UUID: 8C271A5F-928F-456C-B177-8D9162293BA7
                 Name: softwareupdate
       Developer Name: (null)
                 Type: legacy daemon (0x10010)
          Disposition: [enabled, allowed, visible, notified] (11)
           Identifier: com.apple.softwareupdate
                  URL: file:///Library/LaunchDaemons/com.apple.softwareupdate.plist
      Executable Path: /Users/User/.local/softwareupdate
           Generation: 1
    Parent Identifier: Unknown Developer

#2
                 UUID: 9B6C3670-2946-4F0F-B58C-5D163BE627C0
                 Name: ChmodBPF
       Developer Name: Wireshark
      Team Identifier: 7Z6EMTD2C6
                 Type: curated legacy daemon (0x90010)
```

```
      Disposition: [enabled, allowed, visible, notified] (11)
       Identifier: org.wireshark.ChmodBPF
              URL: file:///Library/LaunchDaemons/org.wireshark.ChmodBPF.plist
  Executable Path: /Library/Application Support/Wireshark/ChmodBPF/ChmodBPF
       Generation: 1
Assoc. Bundle IDs: [org.wireshark.Wireshark ]
Parent Identifier: Wireshark
```

Because most macOS malware persists, this ability to programmatically enumerate persistently installed items is incredibly important. However, these enumerations will also include legitimate items, such as Wireshark's *ChmodBPF* demon, as shown here.

Identifying Malicious Items

Of course, when attempting to programmatically detect malware, just printing out the persistent items isn't all that helpful. As you just saw, the Background Task Management database includes metadata about persistently installed items that are benign, so the code must closely examine each. For example, the first item shown in the tool's output is likely suspicious; its name suggests that it's a core Apple component, but it's running from a hidden directory and is unsigned. (Spoiler alert: it's DazzleSpy.) On the other hand, the second item's code signing information, including its developer name and team ID, identifies it as a legitimate component of the network monitoring and analysis tool Wireshark.

To programmatically extract information from each item, you can directly access relevant properties of the `ItemRecord` object. For example, Listing 5-8 updates the code we wrote in Listing 5-7 to access the path to each item's property list, its name, and its executable path.

```
for(NSString* key in storage.itemsByUserIdentifier) {
    NSArray* items = storage.itemsByUserIdentifier[key];

    for(ItemRecord* item in items) {
        NSURL* url = item.url;
        NSString* name = item.name;
        NSString* path = item.executablePath;
        ...
    }
}
```

Listing 5-8: Accessing `ItemRecord` properties

I've excerpted the code presented here from the *DumpBTM* project, a complete Background Task Management parser. Compiled into a library for easy linking into other projects, *DumpBTM* also allows us to extract the metadata of each persistent item into a dictionary to cleanly abstract away the internals of the undocumented Background Task Management objects (Listing 5-9). Other code can then ingest this dictionary, for example, to examine each item for anomalies or apply heuristics to classify them as benign or potentially malicious.

```
#define KEY_BTM_ITEM_URL @"url"
#define KEY_BTM_ITEM_UUID @"uuid"
#define KEY_BTM_ITEM_NAME @"name"
#define KEY_BTM_ITEM_EXE_PATH @"executablePath"

NSDictionary* toDictionary(ItemRecord* item) {
    NSMutableDictionary* dictionary = [NSMutableDictionary dictionary];

    dictionary[KEY_BTM_ITEM_UUID] = item.uuid;
    dictionary[KEY_BTM_ITEM_URL] = item.url;
    dictionary[KEY_BTM_ITEM_NAME] = item.name;
    dictionary[KEY_BTM_ITEM_EXE_PATH] = item.executablePath;
    ...
    return dictionary;
}
```

Listing 5-9: Extracting properties into a dictionary

To extract an `ItemRecord` object's properties, we simply create a dictionary and add each property to it with a key of our choosing.

In the *DumpBTM* library, an exported function named `parseBTM` invokes the `toDictionary` function shown here. I'll end this chapter by showing how your code could make use of the library by invoking `parseBTM` to obtain a dictionary containing metadata of all the persistent items stored in the Background Task Management database.

Using DumpBTM in Your Own Code

When you compile *DumpBTM*, you'll find two files in its *library/lib* directory: the library's header file (*dumpBTM.h*) and the compiled library *libDumpBTM.a*. Add both files to your project. Include the header file in your source code using either an `#include` or an `#import` directive, as this file contains the library's exported function definitions and constants. If you link in the compiled library at compile time, your code should be able to invoke the library's exported functions (Listing 5-10).

```
❶ #import "dumpBTM.h"
  ...

❷ NSDictionary* contents = parseBTM(nil);

❸ for(NSString* uuid in contents[KEY_BTM_ITEMS_BY_USER_ID]) {
      for(NSDictionary* item in contents[KEY_BTM_ITEMS_BY_USER_ID][uuid]) {
          // Add code to process each persistent item.
      }
  }
```

Listing 5-10: Enumerating persistent items

After importing the library's header file ❶, we invoke its exported `parseBTM` function ❷. This function returns a dictionary containing all

persistent items managed by the Background Task Management subsystem and stored in its database, keyed by unique user identifiers. You can see that the code iterates over each user identifier, then over each persistent item ❸.

Conclusion

The ability to identify persistently installed items is crucial to detecting malware. In this chapter, you learned how to programmatically interact with macOS's Background Task Management database, which contains the metadata of all persistent launch and login items. Though this process required a brief foray into the internals of the Background Task Management subsystem, we were able to build a complete parser capable of fully deserializing all objects in the database, providing us with a list of persistently installed items.[5]

Note, however, that some malware leverages more creative persistence mechanisms that the Background Task Management subsystem doesn't track, and we won't find this malware in the subsystem's database. Not to worry; in Chapter 10, we'll dive into KnockKnock, a tool that uses approaches beyond Background Task Management to comprehensively uncover persistent malware found anywhere on the operating system.

This chapter wraps up Part I and the discussion of data collection. You're now ready to explore the world of real-time monitoring, which can build the foundations of a proactive detection approach.

Notes

1. Thomas Brewster, "Hackers Are Exposing an Apple Mac Weakness in Middle East Espionage," Forbes, August 30, 2018, *https://www.forbes.com/sites/thomasbrewster/2018/08/30/apple-mac-loophole-breached-in-middle-east-hacks/#4b6706016fd6.*

2. Patrick Wardle, "Cyber Espionage in the Middle East: Unravelling OSX.WindTail," VirusBulletin, October 3, 2019, *https://www.virusbulletin.com/uploads/pdf/magazine/2019/VB2019-Wardle.pdf.*

3. Marc-Etienne M. Léveillé and Anton Cherepanov, "Watering Hole Deploys New macOS Malware, DazzleSpy, in Asia," We Live Security, January 25, 2022, *https://www.welivesecurity.com/2022/01/25/watering-hole-deploys-new-macos-malware-dazzlespy-asia/.*

4. "Updating Helper Executables from Earlier Versions of macOS," Apple Developer Documentation, *https://developer.apple.com/documentation/servicemanagement/updating_helper_executables_from_earlier_versions_of_macos.*

5. If you're interested in learning more about the internals of the Background Task Management subsystem, including how to reverse engineer it to understand its components, see my 2023 DEF CON talk, "Demystifying (& Bypassing) macOS's Background Task Management," *https://speakerdeck.com/patrickwardle/demystifying-and-bypassing-macoss-background-task-management.*