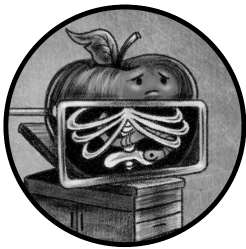
# 8

# DEBUGGING



While the passive dynamic analysis tools covered in the last chapter can often provide insight into a malicious sample, they allow you to observe the sample's actions only indirectly and may not fully reveal its internal workings. In certain cases, you'll need something more comprehensive.

The ultimate dynamic analysis tool is the debugger. A *debugger* is a program that allows you to execute another program instruction by instruction. At any time, you can examine or modify its registers and memory contents, manipulate control flow, and much more. In this chapter, I'll introduce various debugging concepts by means of the de facto debugger for macOS: LLDB. Then we'll walk through a case study, applying these concepts to uncover surreptitious cryptocurrency mining logic in an application that was found in Apple's official App Store.

## Why You Need a Debugger

The following example should clearly illustrate the power of the debugger. Take a look at this snippet of disassembled code from malware known as Mami (and named by yours truly). In this snippet, we find a large chunk of embedded, encrypted data that is passed to a method named `setDefaultConfiguration` (Listing 8-1):

```
[SBConfigManager setDefaultConfiguration:
@"uZmgulcipekSbayTO9ByamTUu_zVtsflazc2NsuqgqOdXkoOzKMJMNTULoLpd-QV9qQy6VRluzRXqWOGscgheRvikLkPR
zs1pJbey2QdaUSXUZCX-UNERrosul22NsW2vYpS7HQO4VG5l8qic3rSH_fAhxsBXpEe557eHIr245LUYcEIpemnvSPTZ_lN
p2XwyOJjzcJWirKbKwtc3Q61pD..."];
```

*Listing 8-1: Encrypted data (Mami)*

If a malicious sample includes encrypted data, the malware author is generally attempting to conceal something, either from detection tools or a malware analyst. Therefore, when we encounter such data, we should be motivated to decrypt it in order to uncover its secrets. Based on the Mami method's name, we can reasonably assume that this embedded data determines some initial configuration. It may contain information valuable to malware analysts, such as the addresses of command and control servers, insights into the malware's capabilities, and more.

So how do we decrypt it? Static analysis approaches are generally inefficient, as they require us to both understand the cryptographic algorithm used and recover the decryption key. File or process monitors are also of little use in this case, because Mami's encrypted configuration information is not written to disk, nor passed to any other processes. In other words, it exists decrypted solely in the Mami process memory space.

Using a debugger, we can easily extract this information. First, we can instruct the malware to execute until it reaches the `setDefaultConfiguration:` method. Then, by *stepping through*, or executing each instruction one at a time, we can allow the malware to continue execution in a controlled manner, pausing when it has completed the decryption of its configuration information. As a debugger can directly inspect the memory of the process it is debugging, we can then *dump*, or print, the now-decrypted configuration information (Listing 8-2):

```
{
 "dnsChanger" =  {
   "affiliate" = "";
   "blacklist_dns" = ();
   "encrypt" = true;
   "external_id" = 0;
   "product_name" = dnsChanger;
   "publisher_id" = 0;
    ...
   "setup_dns" =        (
      "82.163.143.135",
```

```
      "82.163.142.137"
    );
    "shared_storage" = "/Users/%USER_NAME%/Library/Application Support";
    "storage_timeout" = 120;
   };
 "installer_id" = 1359747970602718687;
 ...
}
```

*Listing 8-2: Decrypted configuration data (Mami)*

Various decrypted key/value pairs, such as `"product_name" = dnsChanger` and the `setup_dns` array, provide insight into the malware's goal: hijacking infected systems' DNS settings and then forcing domain name resolutions to be routed through attacker-controlled servers. Incidentally, from the decrypted configuration we now know that these servers are found at `82.163.143.135` and `82.163.142.137`. Perhaps the most noteworthy aspect of this analysis is that we barely lifted a finger. Nor did we have to spend any time understanding how exactly this data was encrypted!

This is but one example of a debugger's power. In general, you should use a debugger to fully understand a code sample, as well as to dynamically modify it on the fly, such as to bypass anti-analysis logic (discussed in Chapter 9). Of course, some challenges temper these benefits. A debugger is a complex tool requiring specific, low-level knowledge; thus, completing an analysis can require a significant amount of time. However, once you understand debugger concepts and the techniques for debugging efficiently, a debugger will become your best malware analysis friend. Often it proves to be both the most efficient and comprehensive way to analyze any sample.

However, one word of caution that is worth reiterating. Dynamic analysis of a sample (which includes analysis within a debugger) involves executing the (potentially) malicious code, so it should always be performed on an isolated analysis system or virtual machine. The latter affords the benefit of snapshots, which allow you to easily revert if a debugging session of a malicious sample goes awry.

## The LLDB Debugger

In this chapter we'll focus on using *LLDB*, the de facto tool for debugging programs, including malware, on macOS. Although other applications, such as Hopper, have built user-friendly interfaces on top of it, you'll probably discover that directly interacting with LLDB's command line interface is the most efficient approach. If you already have Apple's Xcode installed, you'll find LLDB installed alongside at */usr/bin/lldb*. If not, you can install LLDB as a standalone program by entering `lldb` in the terminal and agreeing to the installation prompt.

In this section we'll look at various debugging concepts such as breakpoints and manipulating control flow, and I'll illustrate how these can be applied via LLDB to facilitate the analysis of malicious software. It should be noted that the LLDB website provides a wealth of detailed knowledge, such as an in-depth tutorial.[1] Moreover, while debugging, you can always consult the LLDB `help` command for inline information about any command.

At a high level, a debugging session generally flows in the following manner:

1. You initialize a debugger session by loading an item, such as a malicious sample, into the debugger.
2. You set breakpoints at various locations in the sample's code, such as at its main entry point or at method calls of interest. The sample is started and runs uninhibited until a breakpoint is encountered, at which point execution is halted.
3. Once the debugger has halted execution, you are free to poke around, examining memory and register values, manipulating control flow, setting other breakpoints, and more.
4. You can either resume execution until another breakpoint is hit or execute individual instructions one at a time.

Remember that when a malicious sample is debugged, it is being allowed to execute. As such, always perform debugging in a virtual machine or a standalone analysis system. This ensures that no persistent damage occurs, and if you are debugging in a virtual machine, you can always revert it to a previous state. This is often quite useful during debugging sessions. For example, you might accidentally miss a breakpoint and run the malware in its entirety.

### Starting a Debugger Session

There are several ways to start a debugging session in LLDB. The simplest is to execute LLDB from the terminal, passing it the path of a binary to analyze, followed by any additional arguments (Listing 8-3):

```
% lldb ~/Downloads/malware <arg0 arg1 arg2>

(lldb) target create "malware"
Current executable set to 'malware' (x86_64).
```

*Listing 8-3: Starting a debugging session*

As you can see, the debugger will display a target creation message, make note of the executable set to be debugged, and identify its architecture. Although LLDB has created the debugging session, it has not yet executed any of the program's instructions.

**NOTE** *If you're attempting to debug core operating system processes, you'll likely fail due to macOS's System Integrity Protection (SIP). To debug such processes, turn off SIP by executing* `csrutil disable` *from a terminal in macOS's Recovery Mode.*[2]

You can also attach LLDB to an instance of a running process as follows:

```
% lldb -pid <target pid>
```

Once the debugger has attached to the process, a debugging session can commence. However, we rarely use this approach to analyze malware, because once the malware is already running, its core logic, which we are generally seeking to understand, may have already executed. Moreover, this logic could include anti-debugger code that prevents the debugger from attaching.

A third way of starting a debugging session is to run the `process attach` command with a process name and the `--waitfor` flag from the LLDB shell, as shown in Listing 8-4. This instructs the debugger to wait for a process that matches this name and then attach as the process is starting.

```
% lldb
(lldb) process attach --name malware --waitfor
```

*Listing 8-4: Waiting to attach to a process (named malware)*

After attaching to the process, the debugger will pause execution. The output will look similar to Listing 8-5:

```
Process 14980 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = signal SIGSTOP
...

Executable module set to "~/Downloads/malware".
Architecture set to: x86_64h-apple-macosx-.
```

*Listing 8-5: Process attachment, triggered by the `--waitfor` flag*

The `--waitfor` flag is particularly useful when malware spawns other malicious processes that you'd like to debug as well.

## Controlling Execution

One of the most powerful aspects of a debugger is its ability to precisely control the execution of the process it is debugging. For example, you could instruct a process to execute a single instruction and then halt. Table 8-1 describes several LLDB commands related to execution control.

**Table 8-1:** LLDB Commands for Controlling Execution

| LLDB command | Description |
|---|---|
| `run` (`r`) | Run the debugged process. Starts the execution, which will continue unabated until a breakpoint is hit, an exception is encountered, or the process terminates. |
| `continue` (`c`) | Continue execution of the debugged process. Similar to the `run` command, it will continue execution until it reaches a breakpoint, an exception, or process termination. |
| `nexti` (`n`) | Execute the next instruction, as pointed to by the program counter register, and then halt. This command will skip over function calls and repeated instructions. |
| `stepi` (`s`) | Execute the next instruction, as pointed to by the program counter register, and halt. Unlike the `nexti` command, this command will step into function calls, allowing analysis of the called function. |
| `finish` (`f`) | Execute the rest of the instructions in the current function (called a *frame*), return, and halt. |
| CTRL-C | Pause execution. If the process has been run (`r`) or continued (`c`), this will cause the process to halt wherever it is currently executing. |

Notice that you can shorten the majority of LLDB commands to single or double letters. For example, you can enter `s` for the `stepi` command. Also note that LLDB includes several names for its commands in order to maintain backward compatibility with the GNU Project Debugger (GDB), a well-known predecessor to LLDB.[3] For example, to perform a single step, LLDB supports both `thread step-inst` and `step`, which matches GDB. For the sake of simplicity, this chapter describes the LLDB command names that are compatible with GDB.

While you could step through each of the binary's executable instructions one at a time, doing so is tedious. On the other hand, instructing the debugger to run the malware uninhibited defeats the purpose of debugging in the first place. The solution is to use breakpoints.

## Using Breakpoints

A *breakpoint* is a command that instructs the debugger to halt execution at a specified location. You'll often set breakpoints at the entry point of the binary, at method or function calls, or on the addresses of instructions of interest. You may have to first triage a binary via static analysis tools such as a disassembler in order to know exactly where to set such breakpoints. Once a breakpoint has been hit and the debugger has halted execution, you'll be able to inspect the current state of the process, including its memory, the CPU register contents, call stacks, and more.

You can use the `breakpoint` command (or `b` for short) to set a breakpoint at a named location, such as a function or method name, or at an address. Behind the scenes, the debugger will transparently modify the process memory space to overwrite the byte at the specified location with

a breakpoint instruction. On Intel x86_64 systems, this is the interrupt 3 instruction, whose value is `0xCC`. Once set, whenever the memory address containing the breakpoint is executed, the interrupt 3 will cause the CPU to return control to the debugger, which halts execution. Of course, if execution is continued, the debugger will first execute the original instruction (which was transparently overridden to set the breakpoint), such that normal program functionality is maintained.

Suppose we wanted to debug a malicious sample called *malware* and halt execution at its `main` function (Listing 8-6). If the malware's symbols were not stripped (that is, compiled with debugging symbols), we could start a debugging session and then enter the following to set a breakpoint by name.

```
(lldb) b main
Breakpoint 1: where = malware`main,
              address = 0x100004bd9
```

*Listing 8-6: Setting a breakpoint on a program's `main` function*

With this breakpoint set, we can use the `run` command to instruct the debugger to run the debugged process. Execution will commence and then halt when it reaches the instruction at the start of the `main` function (Listing 8-7):

```
(lldb) run

(lldb) Process 1953 stopped
stop reason = breakpoint 1.1
->  0x100004bd9 <+0>: pushq  %rbp
```

*Listing 8-7: Breakpoint hit; execution halted*

Often, though, the names of functions are not available in a compiled binary, so we must set breakpoints by specifying an address. You might also want to set a breakpoint at some address, say, within a function of interest. To set a breakpoint on an address, specify the hex address preceded by `0x`.

In the previous example, if the `main` function (found at `0x100004bd9`) had not been named, we could still set a breakpoint at its start as follows (Listing 8-8):

```
(lldb) b 0x100004bd9
Breakpoint 1: where = malware`__lldb_unnamed_symbol1$$malware,
              address = 0x100004bd9
```

*Listing 8-8: Setting a breakpoint by address*

Luckily, a large percentage of Mac malware is written in Objective-C, meaning that, even in its compiled form, it will contain both class and method names. As such, we can also set breakpoints on these method names, or any Apple API it invokes, by passing the class and full method name to the `breakpoint` (`b`) command.

### Setting Breakpoints on Method Names

Recall that in Chapter 5 we leveraged the `class-dump` tool to extract Objective-C class and method names. If you spot methods of interest, you can then set breakpoints upon them to take a closer look. For example, by running `class-dump` on the installer for malware known as FinFisher, we'll find a method named `installPayload` in a class named `appAppDelegate`. Specifying the class and method name will allow us to set a breakpoint so that we can dynamically analyze how the malware persistently installs itself (Listing 8-9):

```
Target 0: (installer) stopped.
(lldb) b -[appAppDelegate installPayload]

Breakpoint 1: where = installer`-[appAppDelegate installPayload],
address = 0x000000010000336c
```

*Listing 8-9: Setting a breakpoint on an `installPayload` method (FinFisher)*

Note that setting breakpoints on Apple Objective-C methods can be somewhat nuanced due to various opaque compiler optimizations and abstractions. For example, imagine that, in a disassembler, you notice a malicious sample is invoking the Apple class `NSTask`'s launch method. You'd like to set a debugger breakpoint on this method so that the malware is halted when it attempts to launch an external command or program. However, at runtime, the launch method call will actually be handled not by the `NSTask` class but rather its subclass, `NSConcreteTask`. Thus, you actually have to set the breakpoint in the following manner:

```
b -[NSConcreteTask launch]
```

This might raise the following valid question: How do you know what class or subclass will actually handle a method? One approach is to track invocations of the `objc_msgSend` function (and its variants). As Objective-C calls are routed through this function at runtime, it is possible to uncover all classes and the methods they invoke. Shortly I'll illustrate exactly how to do this via an LLDB debugger script. For an in-depth discussion of debugging Objective-C code, including more information on setting breakpoints, see Ari Grant's excellent write-up "Dancing in the Debugger—A Waltz with LLDB."[4]

### Conditionally Triggering a Breakpoint

Often you'll want a breakpoint to always trigger. Other times, it may be more efficient for them to trigger and halt the process only under certain conditions. Luckily, LLDB supports the notion of applying conditions to breakpoints. These conditions must evaluate to true for the breakpoint to trigger and halt the process. To add a condition to a breakpoint, use the `-c` flag and then specify the condition. For example, imagine that a malicious sample is sending encrypted data to a remote command and control server.

In a debugger, we could set a breakpoint on the function responsible for encrypting the data prior to transmission in order to view its plaintext contents. Unfortunately, if the malware also sends small "heartbeat" messages at regular intervals, this will continually trigger our breakpoint. We most likely want to ignore such messages, as they contain no meaningful data and will slow down our analysis.

The solution? Adding a condition to the breakpoint! Specifically, we'll instruct the breakpoint to only trigger if the size of the data being encrypted and exfiltrated is larger than the heartbeat message. For the sake of the example, let's assume the message-encryption function takes, as its second argument, the size of the message (which can be found in the `$rsi` register) and that heartbeat messages are at most 128 bytes. To add this condition to breakpoint number 1, we would execute the commands in Listing 8-10:

```
(lldb) br modify -c '$rsi > 128' 1
(lldb) br list
Current breakpoints:
1: address = 0x100003d28, locations = 1, resolved = 1, hit count = 0
Condition: $rsi  > 128
```

*Listing 8-10: Setting a conditional breakpoint*

With such a conditional added to the breakpoint, the debugger will only halt when messages with data larger than 128 bytes are passed into the encryption and exfiltration function. Perfect!

### Adding Commands to Breakpoints

Usually we set a breakpoint and perform a deterministic action once it is hit. In the previous example, we'll likely always want to print out unencrypted data to see what the malware is about to exfiltrate. While we could perform this action manually each time the breakpoint is hit, it may be more efficient to add what is known as a *command* to the breakpoint. This command, which consists of one or more debugger commands, will be automatically executed each time the breakpoint is hit. To add one to a breakpoint, use `breakpoint command add` and specify the breakpoint by number. Following this, specify the commands to be executed, and then enter `DONE`. Keeping with the previous example, let's assume the message-encryption function takes as its first argument the plaintext contents of the message (which can be found in the `RDI` register). To add a breakpoint action to print this out, we'll use the `print object` (`po`) command (discussed later in this chapter). We'll also tell the debugger to then simply continue (Listing 8-11):

```
(lldb) breakpoint command add 1
Enter your debugger command(s).  Type 'DONE' to end.
> po $rdi
> continue
> DONE
```

*Listing 8-11: Adding breakpoint commands*

Now, whenever this breakpoint is hit, the debugger will print out the plaintext message passed to the function and then merrily continue on its way. We can simply sit back and watch.

#### Managing Breakpoints

The LLDB debugger also supports various commands to manage breakpoints. Breakpoints can be set, modified, deleted, enabled, disabled, or listed using the commands described in Table 8-2.

**Table 8-2:** LLDB Commands for Managing Breakpoints

| LLDB command | Description |
|---|---|
| `breakpoint` (b) *<function/method name>* | Set a breakpoint on a specified function or method name. |
| `breakpoint` (b) 0x*<address>* | Set a breakpoint on an instruction at a specified memory address. |
| `breakpoint list` (`br l`) | Display all current breakpoints, including their numbers. |
| `breakpoint enable/disable` *<number>* (`br e/dis`) | Enable or disable a breakpoint (specified by number). |
| `breakpoint modify` *<modifications>* *<number>* (`br mod`) | Modify the options on a breakpoint (specified by number). |
| `breakpoint delete` *<number>* (`br del`) | Delete a breakpoint (specified by number). |

Running the `help` command with the `breakpoint` parameter provides a comprehensive list of breakpoint-related commands, including those mentioned in Table 8-2.

```
(lldb) help breakpoint
Syntax: breakpoint <subcommand> [<command-options>]
```

For more information on the breakpoint commands supported by LLDB, see the tool's documentation on the topic.[5]

### Examining All the Things

Once you've halted execution, you can instruct the debugger to display many things, including the values of CPU registers, the contents of the process memory, or other process state information such as the current call stack. This powerful capability allows you to examine runtime information that often isn't directly available during static analysis. For example, in the case study at the beginning of this chapter, we were able to view the malware's decrypted in-memory configuration information.

To dump the contents of the CPU registers, use the `register read` command (or the shortened `reg r`). To view the value of a specific register, pass in the register name as the final parameter:

```
(lldb) reg read rax
rax = 0x0000000000000000
```

Often we're also interested in what the registers point to. That is to say, we'd like to examine the contents of actual memory addresses. The `memory read` or GDB-compatible `x` command can be used to read the contents of memory. Note that these instructions both require register names to be prefixed with `$`; for example, `$rax`.

But unless we explicitly specify a format for the data, LLDB will print out the raw hex bytes. Table 8-3 lists a variety of format specifiers that instruct LLDB to treat the memory address as a string, instructions, or byte.

**Table 8-3:** LLDB Commands for Displaying Memory Contents

| LLDB command | Description |
|---|---|
| `x/s` *<register or memory address>* | Display the memory as a null-terminated string. |
| `x/i` *<register or memory address>* | Display the memory as an assembly instruction. |
| `x/b` *<register or memory address>* | Display the memory as a byte. |

You can also specify the number of items to display by adding a numerical value after the `/`. For example, to disassemble 10 instructions, starting at the current location of the instruction pointer (`RIP`), enter `x/10i $rip`.

The LLDB debugger also supports the `print` command. When executed with a register or memory address, it will display the contents at the specified location. You can also specify a typecast to instruct the `print` command to format the data. For example, if the `RSI` register points to a null-terminated string, you can display this by typing `print (char*)$rsi`.

The `print` command can also be executed with the `object` specifier. This can be used to print out the contents (or *description*, in Objective-C parlance) of any Objective-C object. For instance, consider the example presented at the start of the chapter. Within the `setDefaultConfiguration` method, the Mami malware decrypts its configuration information into an Objective-C object referenced by the `RAX` register. Thus, using the `print object` command, we can print the verbose description of the object, including all of its key/value pairs (Listing 8-12):

```
(lldb) print object $rax
{
 "dnsChanger" =  {
   "affiliate" = "";
   "blacklist_dns" = ();
   "encrypt" = true;
   "external_id" = 0;
   "product_name" = dnsChanger;
   "publisher_id" = 0;

    ...
   "setup_dns" =        (
      "82.163.143.135",
      "82.163.142.137"
    );
    "shared_storage" = "/Users/%USER_NAME%/Library/Application Support";
    "storage_timeout" = 120;
```

```
   };
 "installer_id" = 1359747970602718687;
 ...
}
```

*Listing 8-12: Printing a dictionary object (Mami)*

You might be wondering how, given an arbitrary value or address, you can decide which display command to use. That is to say, how do you know if the address is a pointer to an Objective-C object, a string, or a sequence of instructions? If the value to display is a parameter or return value from a documented API, its type will be noted in its documentation. For example, most of Apple's Objective-C APIs or methods return objects, which should be displayed using the `print object` command. However, if no context is available, the disassembly of the binary may provide some insight, or trial and error could suffice. For example, if the `print object` command doesn't produce meaningful output, perhaps try `x/b` to dump the contents of the specified data as raw hex bytes.

The `backtrace` (or `bt`) debugger command, which prints a sequence of stack frames, is another useful debugging command for examining the process. When a breakpoint is hit, we're often interested in determining the program flow up to that point. For example, imagine we've set a breakpoint on a malware's string-decryption function, which may have been invoked in multiple places in the malicious code to decrypt embedded strings. When the breakpoint triggers, we'd like to know the location of the caller, that is, the address of the code responsible for invoking the function. This can be accomplished via `backtrace`. Whenever a function is called, a stack frame will be created on the call stack—this contains the address that the process will return to once the function is done, among other things. As the return address is the address of the instruction immediately following the call, we can check it to accurately determine the address of the caller. Moreover, as the backtrace contains previous stack frames as well, the entire function call hierarchy can be reconstructed. If you're interested in learning more about backtraces and call stacks, see Apple's write-up "Examining the Call Stack."[6]

### Modifying Process State

Normally, a debugging session is rather passive once you've set your breakpoints to halt execution. However, you can interact with a process by directly modifying its state or even its control flow. This is especially useful when analyzing a malicious specimen that implements anti-debugging logic, a topic discussed in the next chapter.

Once you've located anti-analysis logic, one option is to instruct the debugger to simply skip over the code by modifying the instruction pointer. In some cases, you can also overcome such anti-analysis code by simply changing the value of a register. For example, modifying the `RAX` register can subvert the value returned by a function.

The most common way to modify the state of the binary is to change either CPU register values or the contents of memory. The `register write`

command can be used to change values of the former, while the `memory write` command modifies the latter.

The `register write` (or `reg write`) command takes two parameters: the target register and its new value. Let's see exactly how we can leverage this to wholly bypass the anti-analysis logic found in a widespread adware installer. In Listing 8-13, we first use the `x` command with the `2i` and the program counter register (`RIP`) to display the next two instructions to be executed. The call instruction at `0x100035cbe` will trigger anti-debugging logic. (The details of this logic are not pertinent for this example.)

```
(lldb) x/2i $rip
0x100035cbe: ff d0 callq *%rax
0x100035cc0: 48 83 c4 10 addq $0x10, %rsp

(lldb) register write $rip 0x100035CC0

(lldb) x/i $rip
0x100035cc0: 48 83 c4 10 addq $0x10, %rsp
```

*Listing 8-13: Modifying the instruction pointer*

In order to bypass the call to the anti-debugging logic, we use LLDB's `register write` command to modify the instruction pointer (`RIP`) to point to the next instruction (at `0x100035cc0`). Redisplaying the value of the instruction pointer confirms it has been successfully updated. After this modification, the problematic call at address `0x100035cbe` is never invoked; thus, the malware's anti-debugger logic is never executed, and our debugging session can continue unimpeded. Moreover, the malware is generally none the wiser.

There are other reasons to modify CPU register values to influence the debugged process. For example, imagine a piece of malware that attempts to connect to a remote command and control server before persistently installing itself. If the server is offline but we want the malware to continue to execute so we can observe how it installs itself, we may have to modify a register that contains the result of this connection check. As the return value from a function call is stored in the `RAX` register, this may involve setting the value of `RAX` to `1` (true), causing the malware to believe the connection check succeeded (Listing 8-14):

```
(lldb) reg write $rax 1
```

*Listing 8-14: Modifying a register*

Easy peasy!

We can change the contents of any writable memory with the `memory write` command. During malware analysis, this command could be useful to change the default values of an encrypted configuration file that are only decrypted in memory. Such a configuration may include a trigger date, which instructs the malware to remain dormant until the date is encountered. To coerce immediate activity so you can observe the malware's full behavior, you could directly modify the trigger date in memory to the current time.

As another example, the memory write command could be used to modify the memory that holds the address of a malicious sample's remote command and control server. This provides a simple and non-destructive way for an analyst to specify an alternate server, such as one under their control. Being able to modify the address of a malware's command and control server or specify an alternate server has its perks. In a research paper titled "Offensive Malware Analysis: Dissecting OSX/FruitFly.b Via a Custom C&C Server," I illustrated how malware connecting to an alternate server under an analyst's control could be tasked to reveal its capabilities.[7]

The format of the memory write command is described by LLDB's help command. The simplest way to leverage memory write is with:

- The memory address to modify
- The -s flag and optionally a number (to specify the number of bytes to modify if the default of 1 byte does not suffice)
- The value of the bytes to write to memory

For example, to change the memory at address 0x100100000 to 0x41414141, you would run the following:

```
(lldb) memory write 0x100100000 -s 4 0x41414141
```

The modification can then be confirmed with the memory read command:

```
(lldb) memory read 0x100100000
0x100100000: 41 41 41 41 00 00 00 00 00 00 00 00 00 00 00 00  AAAA...
```

## LLDB Scripting

One of the more powerful features of LLDB is its support for debugging scripts, which allow you to extend the capabilities of the debugger or simply automate repetitive tasks. Let's walk through an example of building a simple debugger script to illustrate important concepts and show how such a script can improve your dynamic malware analysis.

Earlier in this chapter, I mentioned how tracking invocations of the objc_msgSend function can reveal the majority of the Objective-C calls made by the process. When analyzing malware, this can provide valuable insight into the functionality of a specimen, as well as drive subsequent analysis. One naive approach to monitoring calls to the objc_msgSend function is simply setting a breakpoint on the function. Yes, this will halt the process and allow you to examine the function's arguments, which include both class and method names. However, as you'll quickly see, this approach is very inefficient, and the many, many calls to the objc_msgSend function will become overwhelming.

A more efficient approach is to create a debugger script that will automatically set a breakpoint, attach a command to print out the Objective-C

class and method names, and then allow the process to continue. Debugger scripts for LLDB are written in Python and loaded via the debugger command `command script import <path to script>`. These scripts should import the LLDB module so that the LLDB API can be accessed by the rest of the Python code. For more information on this API, see the official LLDB documentation: "Python Reference."[8]

More often than not, you'll want your script to automatically perform an action once it's loaded (such as setting a breakpoint). To facilitate this, LLDB provides the `__lldb_init_module` convenience function, which if it's implemented in your debugger script will be automatically invoked whenever the script is loaded. In our debugger script, we'll use this function to set a breakpoint and breakpoint callback (Listing 8-15):

```
import lldb

def __lldb_init_module(debugger, internal_dict):
    target = debugger.GetSelectedTarget()
    breakpoint = target.BreakpointCreateByName("objc_msgSend")
    breakpoint.SetScriptCallbackFunction('objc.msgSendCallback')
```

*Listing 8-15: Setting a breakpoint via a debugger script*

First, our code gets a reference to the process that is running within the debugger. With this reference, we can then invoke the `BreakpointCreateByName` function to set a breakpoint on the `objc_msgSend` function. Finally, we attach our callback function with a call to the `SetScriptCallbackFunction` function. Note that the parameter to this function is your module or script's name, followed by a period and the name of the callback (for example, `objc.msgSendCallback`).

Now, whenever the `objc_msgSend` function is invoked, our callback, `msgSendCallback`, will be invoked. In this callback, we simply want to print out the Objective-C class and method name that is being invoked, before allowing the debugged process to continue. Recall that, in previous discussions of the `objc_msgSend` function, we noted that its first parameter is the Objective-C class name, while the second is the method name. We also know that on Intel x86_64 platforms, the first two parameters will be passed in the `RDI` and `RSI` registers, respectively. This means we can implement our callback in the following manner (Listing 8-16):

```
def msgSendCallback(frame, bp_loc, dict):
    lldb.debugger.HandleCommand('po [$rdi class]')
    lldb.debugger.HandleCommand('x/s $rsi')

    frame.thread.process.Continue()
```

*Listing 8-16: Implementing a breakpoint action via a debugger script*

In order to execute built-in debugger commands, we can use the `HandleCommand` API. First, we print out the name of the Objective-C class that

can be found within the RDI register. We make use of the po (print object) command, because the class name we want to display is an Objective-C string object. Following this, we print out the method's name stored in the RSI register. As it is a null-terminated C string, the x/s command suffices for this purpose. Then we instruct the debugger to continue, so the debugged process can resume.

We can save the code in Listings 8-15 and 8-16 (for example, to *~/objc.py*), load it into a debugger, and then execute a malicious sample we're interested in further analyzing (Listing 8-17):

```
(lldb) command script import ~/objc.py

(lldb) NSTask
0x1d8dcd07c: "alloc"

(lldb) NSConcreteTask
0x1d8dccbdd: "init"

(lldb) NSConcreteTask
0x1d8e1b67a: "setLaunchPath:"

(lldb) NSConcreteTask
0x1d8e1b771: "launch"
```

*Listing 8-17: Our debugger script in action*

From the output of our script, we see that the malware is leveraging the NSTask class. Behind the scenes, we see that a NSConcreteTask is initialized, a launch path is set, and then the task is launched. To investigate further, we can now manually set a breakpoint on the NSConcreteTask's launch method to see exactly what the malware is executing.

LLDB debugger scripts are a powerful way to extend the debugger and provide an invaluable capability, especially when analyzing more sophisticated malware samples. Here we've only scratched the surface of what they can do through a trivial, albeit useful, example. To learn more, consult online examples, such as Taha Karim's script to automatically dump the Bundlore malware's payload.[9] These examples highlight more advanced use cases while also providing valuable insight into LLDB's scripting API.

## A Sample Debugging Session: Uncovering Hidden Cryptocurrency Mining Logic in an App Store Application

In early 2018, a popular application called Calendar 2, found in Apple's official Mac App Store, was discovered to contain logic that surreptitiously mined cryptocurrency on users' computers (Figure 8-1). Though it isn't exactly malware per se, this application provides an illustrative case study of how a debugger can help us understand a binary's hidden or subversive capabilities. Moreover, due to the rise of malicious crytocurrency miners targeting macOS, this example is particularly relevant.

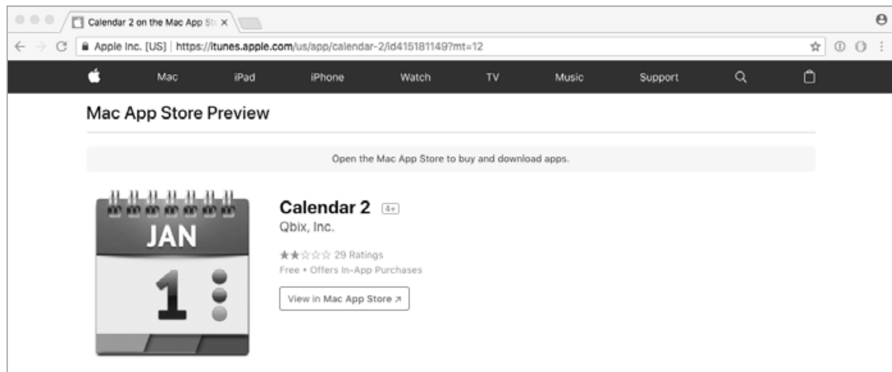



*Figure 8-1: A surreptitious cryptocurrency miner in Apple's official Mac App Store*

During my initial static analysis triage, I uncovered various methods whose names referenced cryptocurrency mining (Listing 8-18). This was odd, as the application claimed to simply be a calendar application.

```
/* @class MinerManager */
-(void)runMining {
    rdx = self->_coreLimit;
    r14 = [self calculateWorkingCores:rdx];
    [Coinstash_XMRSTAK9Coinstash setCPULimit:self->_cpuLimit];
    r15 = [self getPort];
    r12 = [self algorythm];
    [self getSlotMemoryMode];

    [Coinstash_XMRSTAK9Coinstash startMiningWithPort:r15
                                 password:self->_token
                                 coreCount:r14
                                 slowMemory:self->_slowMemoryMode
                                 currency:r12];
    ...

    return;
}
```

*Listing 8-18: Cryptocurrency mining logic within an App Store application?*

In this listing, we can see a method named `runMining` that contains code that invokes methods in a framework named `Coinstash_XMRSTAK`. As the framework is written in Swift, the method names are slightly mangled, though still mostly readable.

One of the goals of the subsequent dynamic analysis was to uncover information about the cryptocurrency account, where any mined coins were to be sent. Based on the method names (such as `startMiningWithPort`, `:password:` and so on), I reasoned that, in a debugging session, setting a breakpoint on either of the methods would reveal this information.

After firing up LLDB and loading the application, we can set a breakpoint on the runMining method by name, as shown in Listing 8-19:

```
% lldb CalendarFree.app
(lldb) target create "CalendarFree.app"
Current executable set to 'CalendarFree.app' (x86_64).

(lldb) b -[MinerManager runMining]
Breakpoint 1: where = CalendarFree`-[MinerManager runMining],
              address = 0x0000000100077fc0
```

*Listing 8-19: Initializing a debugging session and setting an initial breakpoint*

Once the breakpoint is set, we instruct the debugger to run the application. As expected, it halts at the breakpoint we set (Listing 8-20):

```
(lldb) r
Process 782 launched: 'CalendarFree.app/Contents/MacOS/CalendarFree' (x86_64)

CalendarFree[782:7349] Miner: Stopped
Process 782 stopped
 stop reason = breakpoint 1.1

CalendarFree`-[MinerManager runMining]:
->  0x100077fc0 <+0>: pushq  %rbp
    0x100077fc1 <+1>: movq   %rsp, %rbp
    0x100077fc4 <+4>: pushq  %r15
    0x100077fc6 <+6>: pushq  %r14
```

*Listing 8-20: Breakpoint hit; execution halted*

Let's step through the instructions until we reach the call to the Coinstash startMiningWithPort:... method. As its name suggests, it begins the actual mining. Because we want to step over the other method calls prior to reaching it, we use the nexti (or n) command (Listing 8-21). This allows the calls to execute but avoids us having to step through them, instruction by instruction.

```
(lldb) n

Process 782 stopped
 stop reason = instruction step over

CalendarFree`-[MinerManager runMining] + 35:
->  0x100077fe3 <+35>: movq 0xaa3d6(%rip), %r13   ;0x00007fff58acba00: objc_msgSend
```

*Listing 8-21: Stepping through instructions and over method calls*

Eventually we approach the invocation of the method of interest. Recall that, in assembly, Objective-C calls are routed through the objc_msgSend function. In the debugger, we first see this function's address being moved into the R13 register. Though we could just set a breakpoint on the call to the objc_msgSend function (at address 0x100078067) that invokes the startMiningWithPort:...

method, we'll take a more exhaustive approach and continue stepping, instruction by instruction, until the call has been reached (Listing 8-22):

```
(lldb) n

Process 782 stopped
 stop reason = instruction step over

CalendarFree`-[MinerManager runMining] + 167:
->  0x100078067 <+167>: callq  *%r13

(lldb) reg read $r13
r13 = 0x00007fff58acba00  libobjc.A.dylib`objc_msgSend
```

*Listing 8-22: Stepping through instructions until the call of interest is reached*

Note that, via the `reg read` command, we confirmed that the `R13` register indeed contains the `objc_msgSend` function.

Recall from Chapter 6 that, at the time of a call to the `objc_msgSend` function, certain registers hold specific argument values by convention. For example, the function's first argument (held in the `RDI` register) is the class or object upon which the method is being invoked. During the static analysis triage, this was identified as a class named `Coinstash_XMRSTAK.Coinstash`. Using the `print object` (`po`) command, we can dynamically see that this is indeed correct:

```
(lldb) po $rdi
Coinstash_XMRSTAK.Coinstash
```

The second argument (held in the `RSI` register) will be a null-terminated string that names the method to be invoked. Let's confirm this is the case, and that its value is the `startMiningWithPort:...` method. To print out a null-terminated string, we use the `x` command with the `s` format specifier:

```
(lldb) x/s $rsi
0x1000f1576: "startMiningWithPort:password:coreCount:slowMemory:currency:"
```

Following the class and method name are the method's arguments. From the method's name, we can gather it takes five arguments that include a port, password, and currency. We couldn't easily figure out the values of these arguments using static analysis methods, such as a disassembler, because they didn't readily appear. With the debugger, it's a breeze.

We know that the next arguments are stored in the `RDX`, `RCX`, `R8`, and `R9` registers, as specified in the application binary interface. As this method takes more than four arguments, the last argument will be found on the stack (`RSP`). Let's have a peek (Listing 8-23):

```
(lldb) po $rdx
7777

(lldb) po $rcx
qbix:greg@qbix.com
```

```
(lldb) reg read $r8
r8 = 0x0000000000000001

(lldb) po $r9
always

(lldb) x/s $rsp
0x7ffeefbfe0d0: "graft"
```

*Listing 8-23: Displaying the `startMiningWithPort:...` method's parameters*

Note that for the arguments that are objects, we use the `po` command to display their contents. For those that aren't, we use the other appropriate display commands, such as `reg read $r8` to view the contents of a register and `x/s` to display a NULL-terminated string.

By examining the arguments, we've uncovered the port (`7777`), the account password (`qbix:greg@qbix.com`), cryptocurrency (`graft`), and more! Moreover, if we continue our debugging session, we'll encounter additional data, for example, within a `NSURLRequest` object (which in this debugging session is found in memory at 0x1018f04e0). In the debugger, in conjunction with the `po` command, we can invoke the `NSURLRequest`'s `HTTPBody` method on the object ❶ to display the contents (specifically the body), of this network request. This reveals detailed account information and cryptomining statistics (Listing 8-24):

```
❶ (lldb) po [0x1018f04e0 HTTPBody]
{
  "mining": {
    "statistic": {
      "ZeroCounter": 0,
      "AverageHashRate": 0.92911845445632935,
      "CounterTime": 30,
    },
    "params": {
      "Token": "qbix:greg@qbix.com",
      "Algorithm": "graft",
      "CPULimit": 25,
      "EnableMiningMode": true,
      "CPUBatteryLimit": 10,
      "CoreLimit": 25,
      "Ports": {
        "7777": 1000000,
        "5555": 160,
        "3333": 40
      }
    }
  },
  ...
}
```

*Listing 8-24: Displaying a network object containing cryptocurrency miner account information and statistics*

It is also worth noting that, as this information is securely transmitted over the network (encrypted), it would have been rather involved to recover

it via a simple network monitor. Via the debugger, it was relatively straightforward. If you're interested in the full analysis of this application, including more details on the use of a debugger to uncover and understand its cryptomining logic, see my write-up "A Surreptitious Cryptocurrency Miner in the Mac App Store?"[10]

## Up Next

In this chapter I introduced the debugger, the most thorough tool for analyzing even complex malware threats. Specifically, I showed how to debug a binary via breakpoints, instruction by instruction, while examining or modifying registers and memory contents, skipping functions you don't want to execute, and much more. Now that you're armed with this analysis capability, malware doesn't stand a chance.

Of course, malware authors are less than stoked that their malicious creations can be deconstructed so easily. In the next chapter, we'll dive into the kinds of anti-analysis logic employed by malware authors to thwart (or at least complicate) both static and dynamic analysis efforts.

## Endnotes

1. "LLDB Tutorial," *LLDB Debugger*, *https://lldb.llvm.org/use/tutorial.html*.
2. "Disabling and Enabling System Integrity Protection," *Apple Developer Documentation*, *https://developer.apple.com/documentation/security/disabling_and_enabling_system_integrity_protection/*.
3. "GDB to LLDB command map," *LLDB Debugger*, *https://lldb.llvm.org/use/map.html*.
4. Ari Grant, "Dancing in the Debugger—A Waltz with LLDB," *Objc*, *https://www.objc.io/issues/19-debugging/lldb-debugging/*.
5. "LLDB Tutorial: Setting Breakpoints," *LLDB Debugger*, *https://lldb.llvm.org/use/tutorial.html#setting-breakpoints/*.
6. "Examining the Call Stack," *Apple Developer Documentation Archive*, *https://developer.apple.com/library/archive/documentation/General/Conceptual/lldb-guide/chapters/C5-Examining-The-Call-Stack.html*.
7. Patrick Wardle, "Offensive Malware Analysis: Dissecting OSX/FruitFly.b Via A Custom C&C Server," *Virus Bulletin Conference*, October 2017, *https://www.virusbulletin.com/uploads/pdf/magazine/2017/VB2017-Wardle.pdf*.
8. "Python Reference," *LLDB Debugger*, December 2014, *https://lldb.llvm.org/use/python-reference.html*.
9. OSX/Bundlore Payload Dumper (bundlore_python_dump2.py), *https://gist.github.com/tahaconfiant/36bd7594f094e4d1b2afc14264f923dc/*.
10. Patrick Wardle, "A Surreptitious Cryptocurrency Miner in the Mac App Store?" *Objective-See*, March 11, 2018, *https://objective-see.com/blog/blog_0x2B.html*.