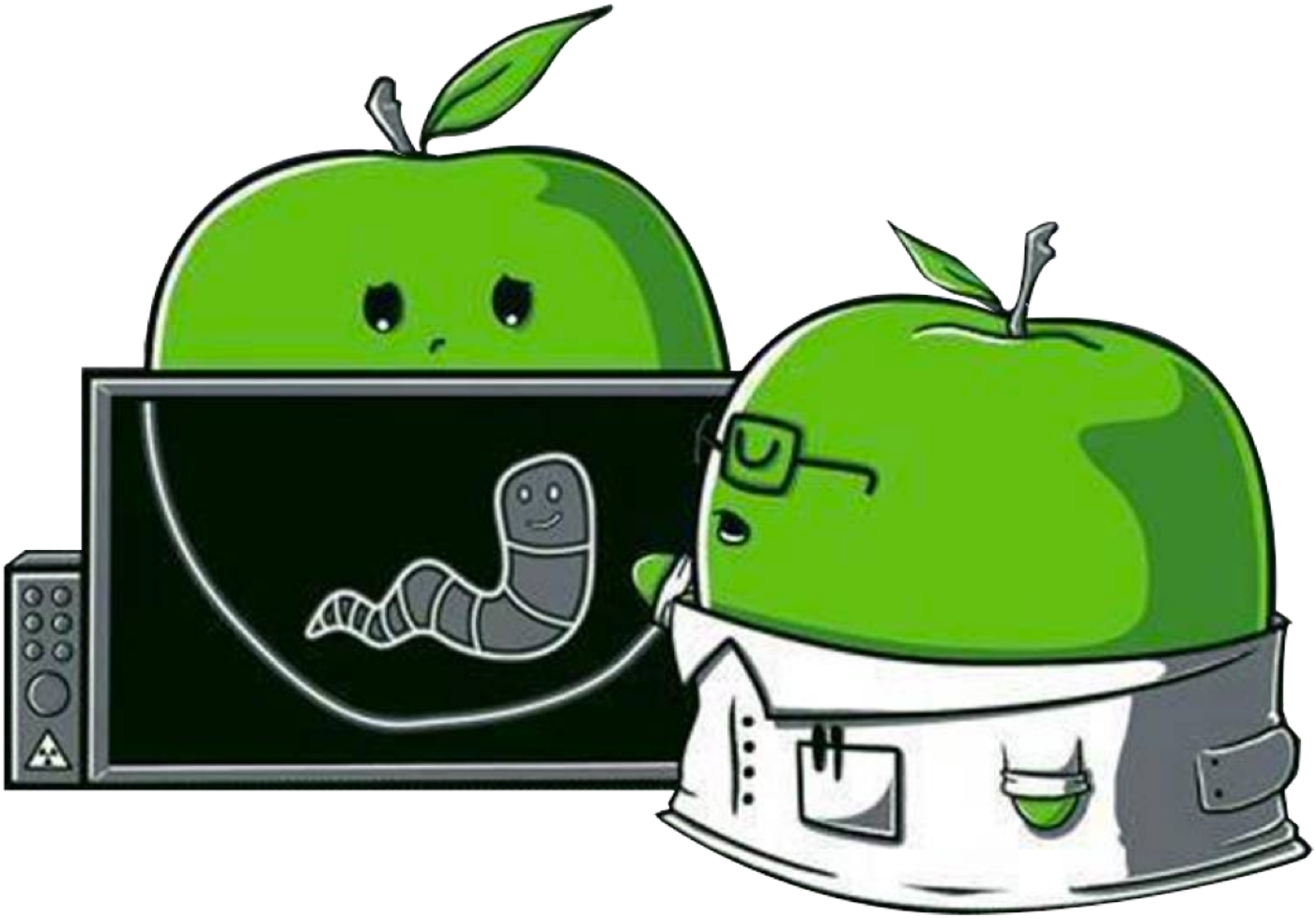


Chapter 0x0A: Debugging

📝 Note:

This book is a work in progress.

You are encouraged to directly comment on these pages ...suggesting edits, corrections, and/or additional content!

To comment, simply highlight any content, then click the icon which appears (to the right on the document's border).

In the previous chapter, we covered passive dynamic analysis tools including process, file, and network monitors. While these tools can often (quickly!) provide invaluable insights into a malicious sample, other times they cannot. Moreover, while they may allow one to observe the actions of the sample under scrutiny, such observations are indirect, and thus may not provide true insight into the internal workings of the sample. Something more powerful is needed!

The ultimate dynamic analysis tool is the debugger. Simply put, a debugger allows one to execute a binary, instruction by instruction. At any time one can examine (or modify) registers and memory contents, skip (bypass) entire functions, and much more.

Before diving into debugging concepts, a quick example is in order ...an example that clearly illustrates the power of the debugger. `OSX.Mami` [1] contains a large chunk of embedded, encrypted data that it passes to a method named `setDefaultConfiguration`:

```
01 [SBConfigManager setDefaultConfiguration:
02 @"uZmgulcipekSbayTO9ByamTUu_zVtsflazc2Nsuqgq0dXkoOzKMJMNTULoLpd-QV9qQy6VRluzRXqWOG
03 scgheRvikLkPRzs1pJbey2QdaUSXUZCX-UNERrosul22NsW2vYpS7HQO4VG5l8qic3rSH_fAhxsBXpEe55
04 7eHIr245LUYcEIpemnvSPTZ_lNp2XwyOJjzcJWirKbKwtc3Q61pD..."];
```

Generally speaking, encrypted data within a malicious sample is data the malware author is attempting to hide ...either from detection tools, or from a malware analyst. As the latter, we're of course quite motivated to decrypt this data to uncover the secrets it hides.

In the case of `OSX.Mami`, based on context (i.e the invocation of the method named `setDefaultConfiguration:`), it seems reasonable to assume that this embedded data is the malware's (initial) configuration, which may contain valuable information such as address of command and control servers, insights into the malware's capabilities, and more.

So how to decrypt? Well, static analysis approaches would be rather slow and inefficient, while file or process monitors would be of little use, as the encrypted configuration information is not written to disk nor passed to any (other) processes. In other words, it exists decrypted, solely in memory.

Via a debugger (and more on this shortly), we can instruct the malware to execute, stopping at the `SBConfigManager`'s `setDefaultConfiguration:` method. Then, "stepping" (executing) instruction by instruction, we allow the malware to continue execution in a controlled manner, pausing again 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 then can simply "dump" (print) the now decrypted configuration information (pointed to by the `rax` register):

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

...

(lldb) po $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;
 ...
}
```

Various decrypted key/value pairs (such as `"product_name" = dnsChanger`) provide insight into the malware's ultimate goal; hijacking infected systems DNS settings, forcing domain name resolutions to be routed through attacker controlled servers (found, as specified in the decrypted configuration, at `82.163.143.135` and `82.163.142.137`).

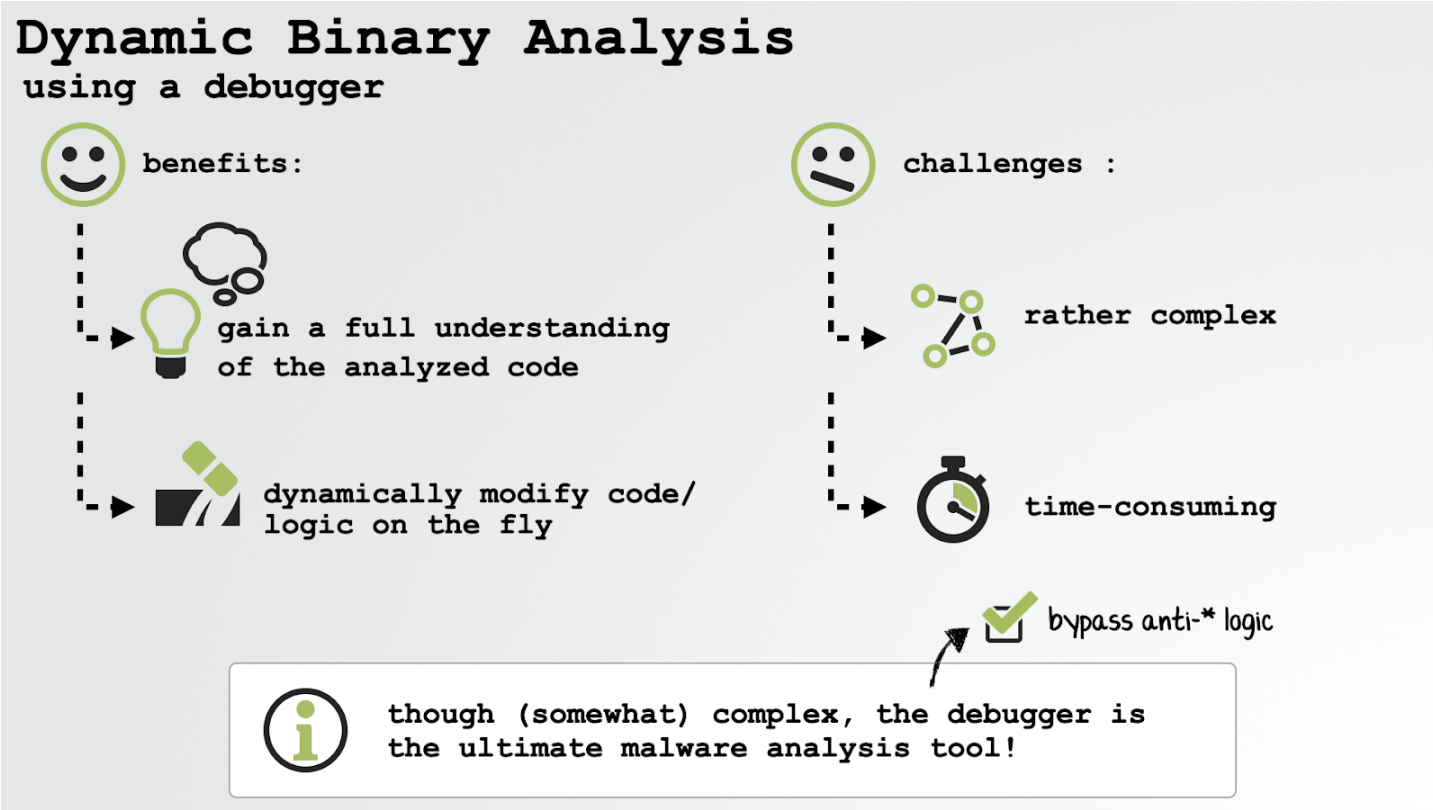
Perhaps the most noteworthy aspect of this analysis approach and decryption of the embedded configuration information, was barely having to lift a finger! Instead, via a debugger, we simply allowed the malware to happily execute ...and then, (unbeknownst to the malware), extracted the decrypted data from memory.

This is but one example that illustrates the power of a debugger! More comprehensively, the benefits of a debugger include the ability to:

- gain a comprehensive understanding of the analyzed code
- dynamically modify code on the fly, for example to bypass anti-analysis logic

Of course there are some challenges that somewhat temper these benefits, including the fact that a debugger:

- is a rather complex tool (requiring specific, low level knowledge)
- can require a significant amount of time to complete the analysis



However, once you have gained an understanding of debugger concepts and techniques to debug efficiently, a debugger will become your best (malware analysis) friend.

Debugging Concepts

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

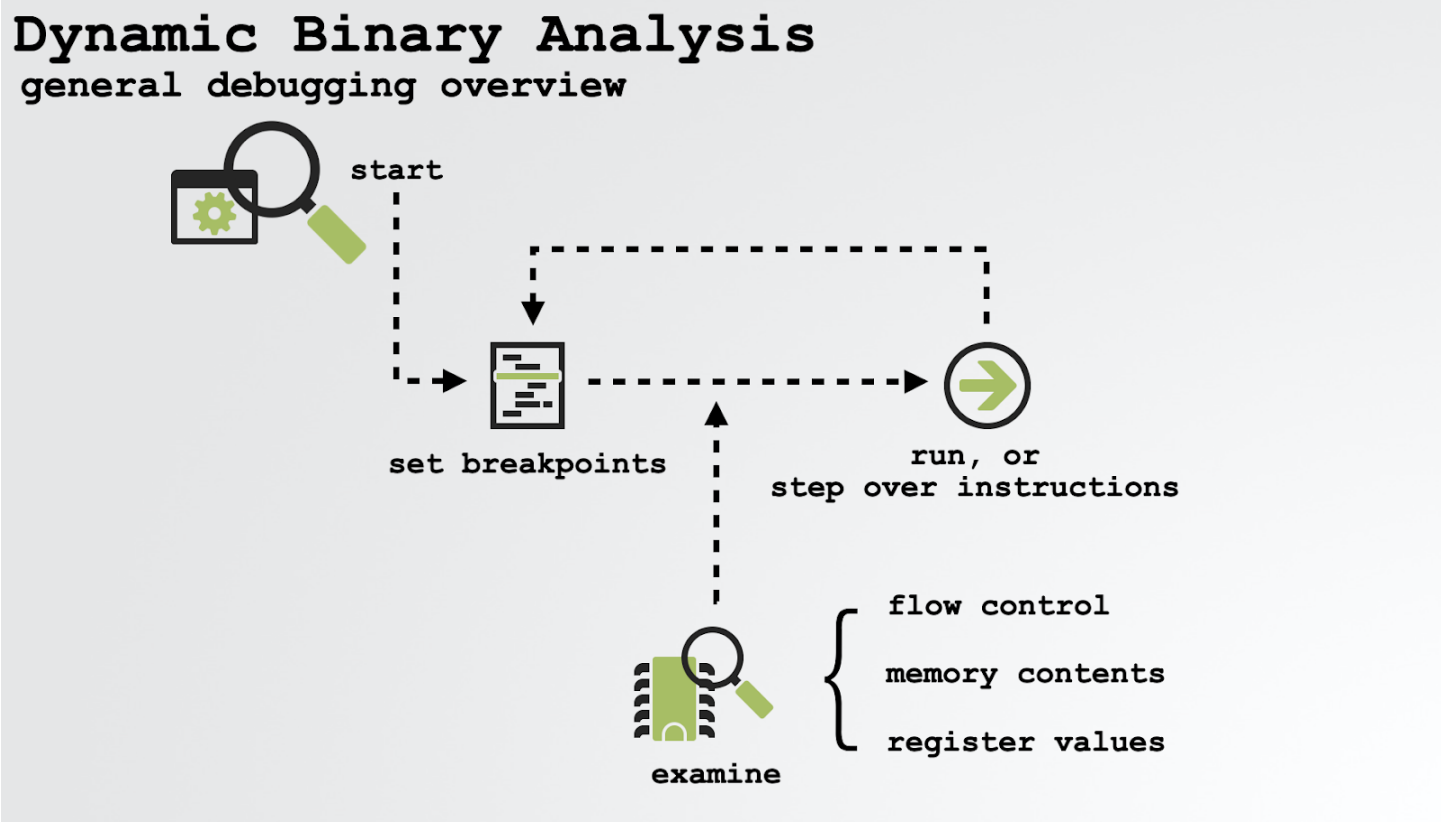
1. The target process (i.e the malware specimen to analyze) is "loaded" into the debugger.

2. Breakpoints are set at various locations within the code, for example at the malware's main entrypoint or at method calls of interest.

3. The sample is started and runs until a breakpoint is encountered, at which point execution is halted.

4. Once halted, one is free to poke around, examining memory and register values (for example to retrieve unencrypted data), control flow (i.e. call stacks), and more.

5. Execution is then either resumed (and runs until another breakpoint is hit), or individual instructions can be executed one at a time.



📝 Note:

When a malicious sample is debugged, it is being allowed to execute (albeit in an instrumented environment). As such, **always** perform debugging in a virtual machine.

Besides ensuring that no persistence damage occurs, a virtual machine can be reverted to a previous state. This is oftentimes quite useful during debugging sessions (for example when a breakpoint was missed and the malware executed in its entirety).

To start a debugging session, simply load the sample into the debugger. The other option is to attach it to an already running process. However, when analyzing a malicious specimen, we generally want to begin debugging at the start of the malware's execution.

📝 Note:

Here we focus on using lldb, the de facto tool for macOS binary debugging. Though applications (such as Hopper) have built user-friendly interfaces on top of it, directly interacting with lldb via its command line interface is arguably the most powerful and efficient approach to debugging.

lldb is installed alongside Xcode (/usr/bin/lldb). However, if you prefer to only install lldb from the terminal: type lldb, hit enter, and agree to the installation prompt.

The lldb website [2] provides a wealth of detailed knowledge, such as an in depth tutorial [3] of the tool.

Moreover, the lldb help command can be consulted for any command in order to provide inline information. For example, the following describes the commands for operating on breakpoints:

(lldb) help breakpoints

The official lldb tutorial, also notes that there is an 'apropos' command that *"will search the help text for all commands for a particular word and dump a summary help string for each matching command."* [3]

There are several ways to start a debugging session in `lldb`. The simplest is executing `lldb` from the terminal, along with the path to a binary to analyze (plus any additional arguments for said binary):

```
$ lldb ~/Downloads/malware.bin any args

(lldb) target create "malware.bin"
```

```
Current executable set to 'malware.bin' (x86_64).
```

As shown above, `lldb` will display a target creation message, make note of the executable set to be debugged, and identify its architecture. Although the debugging session has been created, none of the instructions of the sample (e.g. `malware.bin`) have yet been executed.

`lldb` can also attach to an instance of a running process via `-pid <target pid>`. However, in the context of analyzing malware, this approach is not commonly used (as the malware is already off and running and thus can actually prevent such attachment).

Finally, from the lldb shell, one can utilize the `--waitfor` command, *"which waits for the next process that has that name to show up, and attaches to it"* [3].

```
$ lldb

(lldb) process attach --name malware.bin --waitfor
```

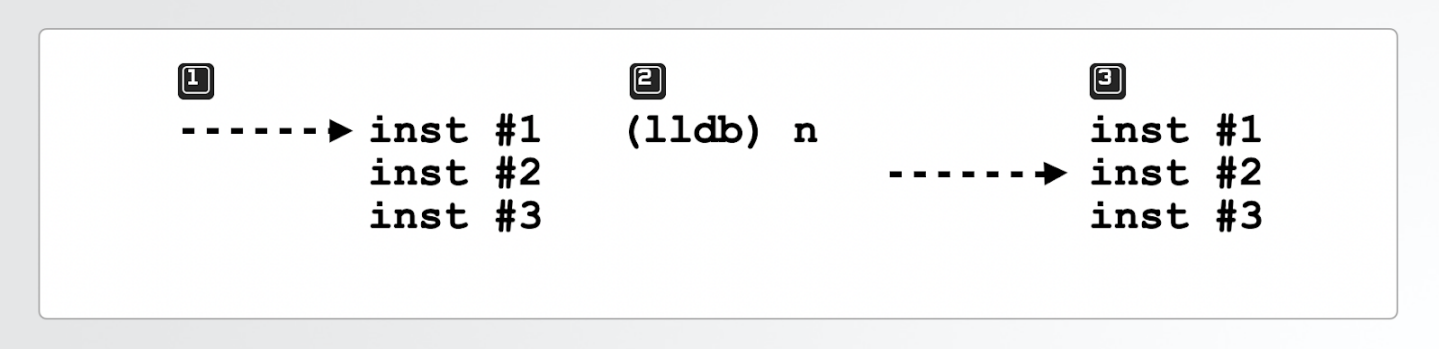
Now, whenever a process named `malware.bin` is started, the debugger will automatically attach:

```
(lldb) process attach --name malware.bin --waitfor
...

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

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

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

Flow Control

Before we discuss breakpoints (which instruct the debugger to halt at specified locations), let's briefly talk about execution control. 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, instructing a process to execute a single instruction (then halt).



The following table describes several lldb commands related to execution control:

(lldb) Command	Description
run (r)	Run the debugged process. Starting execution, which will continue unabated until a breakpoint is hit or the process terminates.
continue (c)	Continue execution of the debugged process. Similar to the `run` command, executing will continue until a breakpoint or process termination.
nexti (n)	Execute the next instruction, as pointed to by the program counter (`rip`) register and halt. This command will skip over function calls.
stepi (s)	Execute the next instruction, as pointed to by the program counter (`rip`) register and halt. Unlike the `nexti` command, this command will step into function calls.
finish (f)	Execute the rest of the instructions in the current function ("frame") return and halt.
control + 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.

> 📝 Note:
>
> Generally speaking, lldb maintains backward compatibility with gdb commands (gdb being the GNU Project debugger, commonly used before lldb). For example, to single step, lldb supports both "thread step-inst" or, to match gdb, simply "step".
>
> For the sake of simplicity (and due to the author's familiarity with gdb), generally we describe the lldb command name that's compatible with gdb.
>
> Finally, note that the majority of commands can be shortened to single or double letters. For example "s" will be interpreted as the "step" command.
>
> For a detailed mapping of gdb to lldb commands, see:
>
> "GDB to LLDB command map" [4]

Though we could single step through each of the binary's executable instructions, this is rather tedious. On the other hand, simply instructing the debugger to allow the debugee to run (uninhibited) rather defeats the purpose of debugging in the first place. The "solution"? Breakpoints!

Breakpoints

A breakpoint is a "command" for the debugger that instructs the debugger to halt execution at a specified location. Often one sets breakpoints at the entry point of the binary (or one its constructors), at method calls, or on the addresses of instructions of interest. As noted, once the debugger has halted execution, one is able inspect the current state of the debugee, including its memory and the CPU register contents, call stack(s), and much more.

Using the `breakpoint` command (or `b` for short), one can set a breakpoint at a named location (such as function or method name) or at an address.

Supposed we want to debug a malicious sample (malware.bin), and want to halt execution at its `main` function. After staring an lldb debugging session, we can simply type `b main`:

```
(lldb) b main
Breakpoint 1: where = malware.bin`main,
              address = 0x0000000100004bd9
```

With this breakpoint set, if we then instruct the debugger run the debugged process (via the `run` command), execution will commence, but halt it when it reaches the instruction at (the start of) the main function:

```
(lldb) run

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

A large percentage of Mac malware is written in Objective-C, meaning (even in its compiled form) it will contain both class and method names. As such, we can also set breakpoints on these method names. How? Simply pass the full method name to the `breakpoint` (b) command. For example, to break on the `NSDictionary` class' `objectForKey:` method, type: `b -[NSDictionary objectForKey:]`.

> 📝 Note:
>
> For an in depth discussion of debugging Objective-C code, see the excellent writeup:
>
> "Dancing in the Debugger – A Waltz with LLDB" [5]

As noted, breakpoints can also be set by address (useful, for example, to set a breakpoint within a function). To set a breakpoint on an address, specify the (hex) address preceded with 0x. For example, we could also have set a breakpoint on the main function of the malicious sample (found at address `0x0000000100004bd9`) via: `b 0x0000000100004bd9`.

Breakpoints can be listed or deleted via commands described below:

(lldb) Command	Description
breakpoint (b) <function/method name>	Set a breakpoint on a specified function or method name.
breakpoint (b) <address>	Set a breakpoint on an instruction at a specified memory address.
breakpoint list (br l)	Display (list) all current breakpoints.
breakpoint enable/disable <#> (br e/dis)	Enable or disable a breakpoint (specified by number).
breakpoint delete <#>	Delete a breakpoint (specified by number).

The `help` command (with a parameter of `breakpoints`) provides a comprehensive list of

breakpoint related commands:

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

The following subcommands are supported:

clear --   Delete or disable breakpoints matching the specified source file and
           line.
command -- Commands for adding, removing and listing LLDB commands executed when a
           breakpoint is hit.

delete  -- Delete the specified breakpoint(s).
           If no breakpoints are specified, delete them all.
disable -- Disable the specified breakpoint(s) without deleting them.
           If none are specified, disable all breakpoints.
enable  -- Enable the specified disabled breakpoint(s).
           If no breakpoints are specified, enable all of them.
list    -- List some or all breakpoints at configurable levels of detail.
modify  -- Modify the options on a breakpoint or set of breakpoints in the
           executable.  If no breakpoint is specified, acts on the last created
           breakpoint.
name    -- Commands to manage name tags for breakpoints
read    -- Read and set the breakpoints previously saved to a file with
           "breakpoint write".
set     -- Sets a breakpoint or set of breakpoints in the executable.
write   -- Write the breakpoints listed to a file that can be read in
           with "breakpoint read".  If given no arguments, writes all breakpoints.
```

> 📝 Note:
>
> For more information on the breakpoint commands supported by lldb, see
>
> “Breakpoint Commands” [6]

Examining All The Things

At the beginning of this chapter, we looked at an example to highlight the power of the debugger. Specifically, we illustrated dumping a malware’s configuration information that had been decrypted and stored (only) in memory.

So far, we've discussed flow control (i.e. single stepping through instructions), and setting breakpoints. However, we've yet to talk about instructing the debugger to display the state of CPU registers or of the debuggee's memory. This powerful capability allows one to examine runtime information or "state", that often is not (directly) available during static analysis. For example, (as we saw), viewing malwares' decrypted in-memory configuration information.

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

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

Generally, we're more interested in what the registers point to ...that is to say, examining (the contents of) actual memory addresses. The `memory read` or (gdb compatible) `x` command can be used to read the contents of memory. However, unless we explicitly specify the (expected) format of the data, `lldb` will simply print out the raw (hex) bytes.

There are a variety of format specifiers (for reading memory) that instruct lldb to treat the specified memory address as a string, instructions, and more:

(lldb) Command	Description
x/s <reg/memory address>	Display the memory as a null-terminated string.
x/i <reg/memory address>	Display the memory as assembly instruction.
x/b <reg/memory address>	Display the memory as byte.

One can also specify the number of items that should be displayed. For example, to disassemble the instructions at a specified address, type: `x/10i <address>`.

The most powerful display command is the "`print object`" (`po`) command. This command can be used to print out the contents (the "description" in Objective-C parlance) of any Objective-C object. For instance, in the example presented at the start of the chapter, within the `[SBConfigManager setDefaultConfiguration:]` method, the malware decrypts its configuration information into an Objective-C object referenced by the `RAX` register.

Thus, using the `print object` (`po`) command we can print the verbose description of the object, including all key/value pairs:

```
(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"
    );

   ...
```

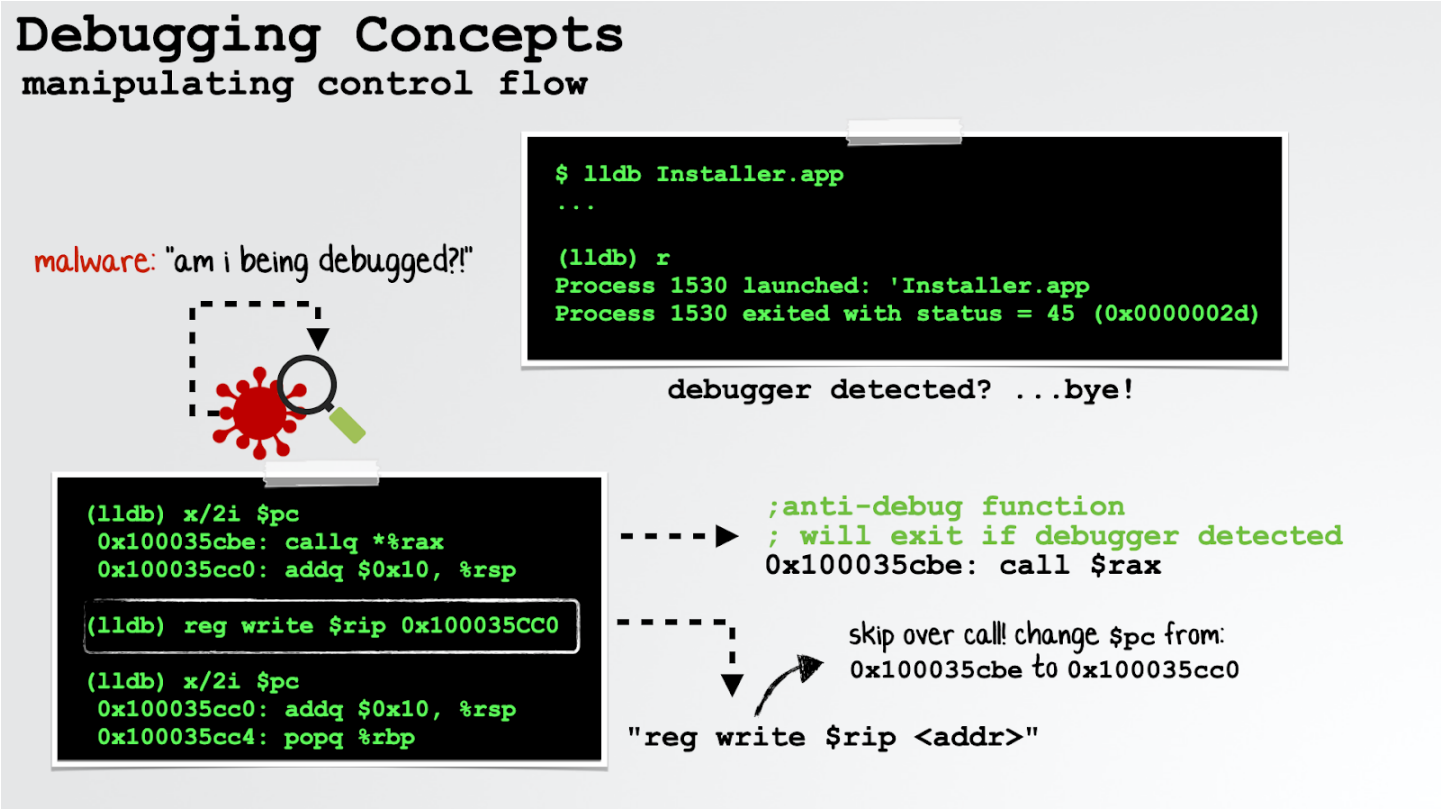
> 📝 Note:
>
> Given an arbitrary value or address, how does one know which display command to use? That is to say, is it a pointer to an Objective-C object? Or string? Or a sequence of instructions?
>
> If the value to display is a parameter or return value from a documented API, its type will be noted in its documentation. For example, most of Apple's Objective-C APIs or methods return objects, and thus should be displayed via the "print object" (po) command.
>
> However, if no context is available, it really comes down to trial and error. The "print object" lldb command doesn't produce meaningful output? Perhaps try x/b to dump it as raw (hex) bytes.

Modifying Process State

The final debugging topic we'll cover in this chapter involves modifying the state and manipulating the control flow of a debugged process. Normally during a debugging session

analysis is rather passive (other than setting breakpoints to halt execution at various locations). However, one can more "invasively" 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 (discussed in the next chapter).

As shown in the image below, once such anti-analysis logic has been located, one can instruct the debugger to skip over the code by modifying the instruction pointer:



The most common way to modify the state of the debuggee is by changing 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 shortened `reg write`) command takes two parameters: the target register and its new value. For example, in the image above we skipped over a call to a function implementing anti-debug logic (`0x100035cbe: call $rax`) by first setting a breakpoint on the call, then modifying the instruction pointer (`rip`) to point to the next instruction (at `0x100035cc0`).

```
(lldb) reg write $rip 0x100035cc0
```

This ensures the call (at address `0x100035cbe`) is never invoked, and thus the malware's anti-debugger logic is never executed ...meaning our debugging session can continue unimpeded!

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 (check-in) to a remote command and control server before persistently installing itself. If the server is (now) 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` (after the function) call to 1 (true), causing the malware to believe the connection check succeeded:

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

Easy peasy!

As noted, we can "invasively" manipulate the debuggee by changing the contents of any (writable) memory, via the `memory write` command. Apple's "Getting Started with LLDB" [7] guide states that this command allows one to "*write to the memory of the process being debugged*".

An example of where this command would be useful during malware analysis is changing 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 said date is encountered. To coerce immediate activity (to observe the malware's full behavior), one could directly modify the trigger date in memory to the current time.

Another example is modifying the memory that holds the address of a remote command and control server. This provides a simple (and non-destructive) way for an analyst to specify an alternate server ...likely one under their control.

> 📝 Note:
>
> Being able to modify the address of a malware's command and control server (or specify an alternate server), has it perks!
>
> In a research paper titled, "Offensive Malware Analysis: Dissecting OSX/FruitFly.b Via A Custom C&C Server" [8], I illustrated how malware connecting an alternate server

(under an analyst's control) could be tasked in order to reveal its capabilities.

"Malware analysis is a time-consuming and often strenuous process. And while traditional analysis techniques such as static analysis and debugging can reveal the full functionality of a malware specimen, there may be a better way. In this research paper, we fully analysed an interesting piece of macOS malware by creating our own custom command-and-control (C&C) server. In conjunction with various monitoring utilities, via this server we were able simply to task the malware in order to coerce it into revealing its entire capabilities." [8]

The format of the `memory write` command is described via `lldb`'s help command:

```
(lldb)  help memory write

Write to the memory of the current target process.

Syntax: memory write <cmd-options> <address> <value> [<value> [...]]

Command Options Usage:
  memory write [-f <format>] [-s <byte-size>] <address> <value> [<value> [...]]
  memory write -i <filename> [-s <byte-size>] [-o <offset>] <address> <value>
[<value> [...]]

       -f <format> ( --format <format> )
            Specify a format to be used for display.

       -i <filename> ( --infile <filename> )
            Write memory using the contents of a file.

       -o <offset> ( --offset <offset> )
            Start writing bytes from an offset within the input file.

       -s <byte-size> ( --size <byte-size> )
            The size in bytes to use when displaying with the selected format.
```

For example, to change the memory at address `0x100100000` to `0x41414141`, one would write: `memory write 0x100600000 -s 4 0x41414141`. The modification can then be confirmed with the `memory read` command:

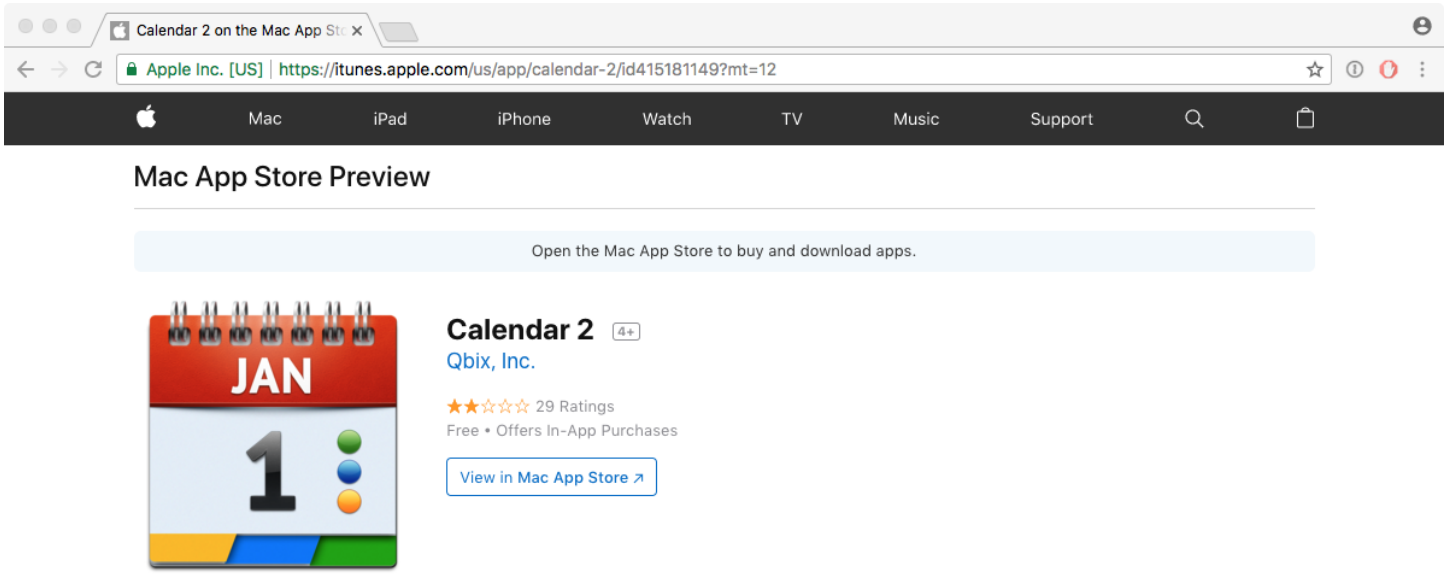
```
(lldb) memory write 0x100600000 -s 4 0x41414141

(lldb) memory read 0x100600000
```

```
0x100600000: 41 41 41 41 00 00 00 00 00 00 00 00 00 00 00 00  AAAA............
```

Sample Debugging Session

Let's end this chapter by briefly looking at another real-life example illustrating many of the debugging concepts we've just discussed.

In early 2018, a popular application called "Calendar 2" (or CalendarFree), found in the official Mac App Store, was discovered to contain logic to (rather surreptitiously) mining crypto currency on users' computers [9]. Though not malware per se, this application provides an illustrative case study, showing how a debugger can provide a comprehensive understanding of a binary, even revealing hidden or subversive capabilities.



During static analysis triage, some interesting method calls were uncovered, such as a call to `-[MinerManager runMining]`, which in turn invoked `+[Coinstash_XMRSTAK.Coinstash startMiningWithPort:password:coreCount:slowMemory:currency:]`:

```
01  /* @class MinerManager */
02  -(void)runMining {
03      rdx = self->_coreLimit;
```

```
04      r14 = [self calculateWorkingCores:rdx];
05      [Coinstash_XMRSTAK9Coinstash setCPULimit:self->_cpuLimit];
06      r15 = [self getPort];
07      r12 = [self algorythm];
08      [self getSlotMemoryMode];
09
10      [Coinstash_XMRSTAK9Coinstash startMiningWithPort:r15
11                                  password:self->_token
12                                  coreCount:r14
13                                  slowMemory:self->_slowMemoryMode
14                                  currency:r12];
15      ...
16
17      return;
18  }
```

One of the goals of the analysis was to uncover the crypto currency account. It was reasoned that in a debugging session, setting a breakpoint on the `runMining` method (or `startMiningWithPort:password:coreCount:slowMemory:currency:` method) would reveal this information.

Firing up `lldb`, we can first set a breakpoint on the `-[MinerManager runMining]` method:

```
$ 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
```

Once the breakpoint is set, we instruct the debugger to run the application (via the `r` command). As expected, it halts at the breakpoint we set:

```
(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
```

Let's (single) step through the instructions until we reach the call to the `+[Coinstash_XMRSTAK.Coinstash startMiningWithPort:password:coreCount:slowMemory:currency:]` method.

As we want to step over the (other) method calls prior to the `startMiningWithPort: ...` method, we use the `nexti` (or `n`) command.

Eventually we get to invocation of the `startMiningWithPort: ...` method. Recall that Objective-C (and Swift) calls are made via the `objc_msgSend` function.

This fact can be observed in the debugger: the address of the `objc_msgSend` function is moved into the `r13` register (at the instruction found at `0x100077fe3`):

```
(lldb) n

Process 782 stopped
 stop reason = instruction step over

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

The actual call to the `startMiningWithPort: ...` method (via the `objc_msgSend` function, who's address, recall, is stored in the `r13` register) happens at address `0x100078067`:

```
(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
```

Note that via the `reg read` command, we confirmed that the target of the call (found in the `r13` register) is indeed the `objc_msgSend` function.

As we discussed in chapter 0x7 on static analysis, at the time of a call to the `objc_msgSend` function, the registers and their values will be the following:

Argument	**Register**	**(for) `objc_msgSend`**
1st argument	`rdi`	`self`: object that the method is being invoked upon
2nd argument	`rsi`	`op`: name of the method
3rd argument	`rdx`	1st argument to the method
4th argument	`rcx`	2nd argument to the method
5th argument	`r8`	3rd argument to the method
6th argument	`r9`	4th argument to the method
7th+ argument	`rsp+` (on the stack)	5th+ argument to the method

Via a debugger, it's easy to confirm this!

The first argument (held in the `rdi` register) is the class (or instance) the method is being invoked upon. 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 yes, this is indeed correct:

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

The second argument will be a (null-terminated) string that is the name of the method to be invoked. Via our static analysis, we know that this should be "`startMiningWithPort: ...`". 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:"
```

📝 Note:

When calling the objc_sendMsg function, the rsi register holds the name of the method as a null-terminated ("C") string.

If we attempt to print out this string, but don't instruct lldb as to its format (a null-terminated string), lldb will simply display it in its default format - an unsigned long:

```
(lldb) print $rsi
(unsigned long) $1 = 4295955830
```

There are a few options to get the actual method name to appear as a string. These include:

- x/s $rsi
  ```
  (lldb) x/s $rsi
  0x1000f1576: "startMiningWithPort:password:coreCount:slowMemory:currency:"
  ```
- print (char*)$rsi
  ```
  (lldb) print (char*)$rsi
  (char *) $1 = 0x00000001000f1576
  "startMiningWithPort:password:coreCount:slowMemory:currency:"
  ```
- reg read $rsi
  ```
  (lldb) reg read $rsi
  rsi = 0x00000001000f1576
  "startMiningWithPort:password:coreCount:slowMemory:currency:"
  ```

The final option (the reg read command) will automatically figure out that the value of $rsi is a pointer to a null-terminated string.

Following the class (`rdi`) and method name (`rsi`) are the arguments for the method, such as port, password, and currency. Via static analysis methods, the values of these arguments were not easily ascertained. However, via the debugger, it's a breeze.

Consulting the above table, we see that the arguments will be in the `rdx`, `rcx`, `r8`, `r9` registers, and then (as this method takes more than 4 arguments), the last argument will be found on the stack (`rsp`). Let's have a peek:

```
(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: "Ugraft"
```

Using various display commands (`po`, `reg read`, etc), we are able to see the values passed to the cryptocurrency framework method, providing the insight (and some possibly attributable account information) our analysis sought!

Up Next...

In this chapter we detailed the debugger ...arguably the most thorough and comprehensive way to analyze even the most complex malware threats.

Specifically, we showed how to debug a binary, instruction by instruction, while examining (or modifying) registers and memory contents, skipping (bypassing) functions, and much more. Armed with this analysis capability malware doesn't stand a chance!

Of course, if you're a malware author you're less than stoked that your malicious creations can be trivially deconstructed. So what do to? In the next chapter, we'll dive into various anti-analysis logic employed by malware authors that aims to thwart (or at least complicate) analysis efforts.

References

1. OSX.Mami
   https://objective-see.com/blog/blog_0x26.html

2. lldb
   https://lldb.llvm.org/

3. lldb tutorial
   https://lldb.llvm.org/use/tutorial.html

4. "GDB to LLDB command map"
   https://lldb.llvm.org/use/map.html

5. "Dancing in the Debugger – A Waltz with LLDB"
   https://www.objc.io/issues/19-debugging/lldb-debugging/

6. "Breakpoint Commands"
   https://lldb.llvm.org/use/map.html#breakpoint-commands

7. "Getting Started with LLDB"
   https://developer.apple.com/library/archive/documentation/IDEs/Conceptual/gdb_to_lldb_transition_guide/document/lldb-basics.html

8. "Offensive Malware Analysis: Dissecting OSX/FruitFly.b Via A Custom C&C Server"
   https://www.virusbulletin.com/uploads/pdf/magazine/2017/VB2017-Wardle.pdf

9. "A Surreptitious Cryptocurrency Miner in the Mac App Store?"
   https://objective-see.com/blog/blog_0x2B.html