Università
Ca'Foscari
Venezia

Master's Degree
in Computer Science

Final Thesis

LLDBagility

Practical macOS kernel debugging

Supervisor
Dr. Stefano Calzavara

Assistant Supervisor
Nicolas Couffin

Graduand
Francesco Cagnin
840157

Academic Year
2018/2019

Acknowledgements

This work concludes a long and at times difficult journey.

I express my deep gratitude to my supervisors, Nicolas Couffin, for his guidance and knowledge, and Stefano Calzavara, for his patience and valuable comments. My thanks extend to Quarkslab for a challenging internship in an office full of brilliant and pleasant colleagues.

To the old and new friends I met at Ca' Foscari, I am truly grateful for the many 'intelligent' and meaningful conversations, countless moments of plain fun, and unforgettable days and nights of hacking (and pizza) with *c00kies@venice*.

For their love, understanding, and financial support, I give my heartfelt thanks to my parents, Michele and Emilia.

Abstract

The effectiveness of debugging software issues largely depends on the capabilities of the tools available to aid in such task. At present, to debug the macOS kernel there are no alternatives other than the basic debugger integrated in the kernel itself or the GDB stub implemented in VMware Fusion. However, due to design constraints and implementation choices, both approaches have several drawbacks, such as the lack of hardware breakpoints and the capability of pausing the execution of the kernel from the debugger, or the inadequate performance of the GDB stub for some debugging tasks.

The aim of this work is to improve the overall debugging experience of the macOS kernel, and to this end LLDBagility has been developed. This tool enables kernel debugging via virtual machine introspection, allowing to connect the LLDB debugger to any unmodified macOS virtual machine running on a patched version of the VirtualBox hypervisor. This solution overcomes all the limitations of the other debugging methods, and also implements new useful features, such as saving and restoring the state of the virtual machine directly from the debugger. In addition, a technique for using the lldbmacros debug scripts while debugging kernel builds that lack debug information is provided. As a case study, the proposed solution is evaluated on a typical kernel debugging session.

Contents

Chapter 1

Introduction

No matter how much effort is put into development, software will always contain programming errors, or ‘bugs’, that may cause it to misbehave, sometimes with serious consequences if the software is running on critical systems or if the bug opens a security hole that allows unauthorised accesses to resources.

The process of searching and removing bugs is called debugging. Depending on the complexity of the software to fix and many other factors, this procedure can be challenging and time consuming. Its effectiveness is often determined by the techniques and tools available to assist the analysis, such as debuggers, computer programs that allow to inspect and modify the run time state of other programs.

To allow for debugging, operating system (OS) kernels typically implement an internal stub that, when enabled, permits a debugger running on a different machine to control the execution of the entire system by sending commands over some communication channel, like Ethernet or the serial interface. In the case of macOS, its kernel implements remote debugging with the Kernel Debugging Protocol (KDP), a debug interface that offers most of the basic debugging capabilities, such as reading and writing CPU registers and memory and setting software breakpoints. This mechanism and its current implementation are however not without limitations, such as the lack of hardware breakpoints and the capability of pausing the execution of the kernel from the debugger, which make macOS debugging not very practical and less effective than it could be. The only real alternative to KDP is the GDB stub implemented in VMware Fusion, which allows debugging a macOS virtual machine (VM) at the hypervisor level, completely bypassing KDP and its drawbacks; but depending on the use case this solution may also be inadequate, being VMware Fusion not free and its stub not so performant.

This work aims to improve the situation with LLDBagility, a new tool for macOS kernel debugging based on virtual machine introspection (VMI). LLDBagility allows to connect the LLDB debugger to any macOS virtual machine running on a patched version of the VirtualBox hypervisor that implements the Fast Debugging Protocol (FDP), a third-party interface for introspection and debugging.

This work is then structured as follows. First, chapter 2 provides some technical context by briefly introducing the reader to interactive software debugging, hardware virtualisation, and relevant macOS terminology. Chapter 3 then discusses in detail how to debug recent versions of macOS, with a particular focus on KDP, and the current limitations of the available methods. Chapter 4 presents LLDBagility, the newly proposed solution for macOS kernel debugging, and is followed by chapter 5 which illustrates a typical use case of the tool. Lastly, chapter 6 provides a summary of this work together with considerations for future improvements.

Chapter 2

Background

This chapter introduces the reader to the background topics of this work: section 2.1 illustrates what interactive software debugging is and how the core of an operating system is debugged; section 2.2 briefly discusses hardware virtualisation together with the possibilities offered by virtual machine introspection; and section 2.3 presents relevant macOS terminology and features.

2.1 On interactive software debugging

Debugging is the process of searching and correcting hardware or software bugs[1] that may cause electronic systems or computer programs to behave incorrectly. As every programmer knows, typical consequences of uncaught bugs include the computation of wrong results, often followed by a forced and abrupt termination of the faulty program; and if the error occurred in a critical part of the OS, the entire system may hang or 'panic', halting all activity as a safety measure. Frequently, bugs also lead to security vulnerabilities: according to the list of Common Vulnerabilities and Exposures (CVE) published by Mitre[2], as many as 21 578 vulnerabilities of varying severity were reported publicly throughout 2018, many of which could allow an attacker to compromise the machine and access confidential information.

Depending on the type and complexity of the bugs to investigate, the debugging process may involve different tools and techniques. Hardware is probed with oscilloscopes and logic analysers, while software is scrutinised through procedures such as:

[1]Although the term 'bug' has been coined by Edison in the 19th century to indicate any 'fault or trouble in the connections or working of electric apparatus', its popular usage as a synonym for design flaws or execution errors in computer systems started in the late 1940s when Grace Hopper and her colleagues found the malfunctioning of the calculator Mark II was caused by a moth trapped in a relay. See Alexander B. Magoun and Paul Israel. *Did You Know? Edison Coined the Term "Bug"*. URL: https://spectrum.ieee.org/the-institute/ieee-history/did-you-know-edison-coined-the-term-bug.

[2]The MITRE Corporation. *CVE 2018 entries*. URL: https://cve.mitre.org/data/downloads/allitems-cvrf-year-2018.xml.

- Interactive debugging, described later.
- Print debugging, the simple but often effective practice of monitoring the execution of a computer program by inserting print commands in the code to output information about its run time state.
- Static code analysis, the analysis of a computer program performed on its source or object code without executing it.
- Profiling, the collection of statistics about memory usage or the execution time of a computer program.
- Analysis of log files.

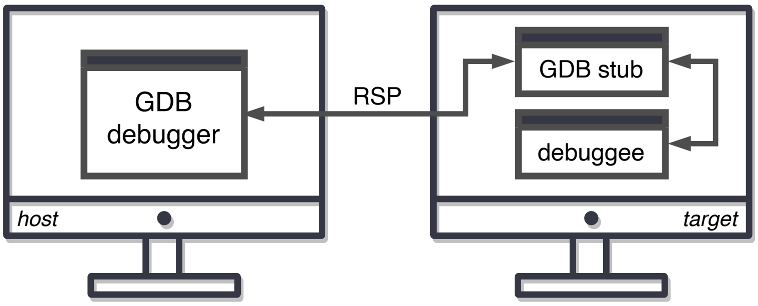
The viability of these approaches is determined by several elements, including the programming language used to develop the program to debug, the environment in which this is executed, and most importantly the tools available to aid the specific debugging procedure.

Interactive software debugging (colloquially 'debugging') is generally understood as the possibility of monitoring and manipulating the execution of a computer program, referred to as the 'debuggee', through a second control program, the 'debugger'. Depending on multiple factors, for instance the amount of symbolic and debugging information at disposal, debugging may happen either at source or machine code level. In both cases, debuggers are expected nowadays to offer the possibility to:

- Inspect and alter the internal state of the debuggee, allowing access and modification of CPU registers (where appropriate) and memory content, in relation to the type of software being debugged and the level at which debugging takes place (discussed in section 2.1.2).
- Set and unset breakpoints[3] to interrupt temporarily the execution of the debuggee upon the occurrence of certain events, so that the program can be examined while it's paused in the desired state. Typically, breakpoints can be set either on instructions, so to interrupt the program just before their execution, or on data (then called 'watchpoints'), to pause the program when it accesses specific memory locations.
- Pause and resume the execution of the debuggee at will, usually also allowing to single-step through the program one instruction at a time; for example, the x86 architecture provides the `TF` trap flag, part of the `FLAGS` register, to run the processor in single-step mode, generating an internal type-1 interrupt after executing each machine instruction.

[3]The term 'breakpoint' was invented for ENIAC, one of the earliest digital computers, by Betty Holberton: 'Well you know, the thing is we did develop the, the one word that's in the language today, which is "breakpoint," at that time. Because we actually did pull the wire to stop the programs so we could read the accumulators off.' See National Museum of American History. *Computer Oral History Collection, 1969-1973, 1977. Jean J. Bartik and Frances E. (Betty) Snyder Holberton Interview*. URL: https://amhistory.si.edu/archives/AC0196_bart730427.pdf.

Figure 2.1: Debugging a user space program remotely with GDB. The debuggee runs in the target machine together with the GDB remote stub, which is allowed to inspect and manipulate its state. The debugger runs instead in the host machine, managing the debugging session remotely by sending commands to the stub using the GDB remote serial protocol (RSP)[5].



2.1.1 Local and remote debugging

Most of the times debugger and debuggee run locally on a single machine as two processes of the same operating system, but this may not always be possible, for example because of limited computational resources, or the two programs being built for different CPU architectures, or when debugging the OS kernel (as explained in section 2.1.2). For such cases, an often viable solution is debugging remotely, i.e. debugging a process running on a different system than the debugger's. This procedure employs a 'host' machine, running the full debugger that manages the debugging session, and a 'target' machine, running both the program to debug and a debugging stub that controls the debuggee as instructed by the remote debugger over some communication channel, such as the serial line or Ethernet. Naturally, the stub must reimplement all the debugging features that were originally implemented in the full debugger, such as reading and writing CPU registers and memory. An example of remote debugging involving the GDB debugger[4] is represented in fig. 2.1.

2.1.2 User- and kernel-mode debugging

In a computer system, programs run either in user-mode, with a very limited access to CPU registers and system memory, or in kernel-mode, with unrestricted access to hardware resources. These two modes of operation are commonly enforced at hardware level through CPU states with different privileges.

As the name suggests, user-mode debugging is the debugging of computer programs that run in user space. In this case, the debugger is typically just another user space application, allowed by the operating system kernel to monitor and modify the state of the debuggee by means of mechanisms like:

- ptrace, abbreviation of 'process trace', a system call implemented by Unix-like operating systems which provides a basic interface for user space processes to inspect and manipulate the execution of other user programs. This

[4] *GDB: The GNU Project Debugger*. URL: https://www.gnu.org/software/gdb/.

powerful capability is restricted to child processes or to the superuser.

- Mach exception ports, primitives for inter-process communication in Darwin (see section 2.3), conceptually similar to unidirectional pipelines in Unix-like operating systems. Exception ports are used to inform a task about exceptions happened during its execution, so that this can either handle its own exceptions or let another task do so; for example, a debugger can register one of its own ports as the debugged task's exception port, so to handle all of the program's exceptions[6].

In user-mode debugging, the debugger may access only the resources (e.g. virtual memory) that are also accessible to the debuggee. Although user-mode debuggers typically run on the same system as the program being debugged, remote debugging is also possible, as shown in fig. 2.1.

On the other hand, kernel-mode debugging is the debugging of computer programs that run in kernel space, such as device drivers or the kernel itself. Unlike in user-mode, kernel-mode debuggers must be allowed to and capable of accessing any part of the system without limitations, including having direct access to physical memory and privileged CPU registers and the ability to alter the execution of the kernel. Clearly, in-depth interactive debugging cannot be performed from within the same machine, since, for instance, a breakpoint hit would naturally imply a freeze of the entire operating system; thus, excluding limited forms of local debugging for inspecting memory, kernels must be debugged remotely. As described in section 2.1.1, this implies implementing a debugging stub, either internally to the kernel or as a kernel extension, which, when enabled, allows the entire system to be controlled remotely from an external kernel-mode debugger running on a second machine.

2.1.3 Hardware and software breakpoints

Breakpoints can be implemented in hardware or in software. In the hardware case, both instruction and data breakpoints use dedicated CPU registers, e.g. `DR0` to `DR7` in the x86 architecture, to define both the type of breakpoint and which addresses to stop the execution at. Hardware breakpoints are very fast and their presence doesn't slow down the execution of the debuggee, but the number of concurrent stopping points is usually quite limited, e.g. at most four concurrent breakpoints in x86, or subject to microarchitecture limitations (for example, hardware breakpoints may be problematic on instructions located in branch delay slots[7]).

In the software case, instruction breakpoints are implemented by replacing the instruction at the desired stopping location with another opcode that causes the interruption of the program and a call to the debug exception handler, such as

[6] Apple. *Kernel Programming Guide. Mach Overview*. URL: https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/Mach/Mach.html; Landon Fuller. *Mach Exception Handlers*. URL: https://www.mikeash.com/pyblog/friday-qa-2013-01-11-mach-exception-handlers.html; Amit Singh. *Mac OS X internals: a systems approach*. Addison-Wesley Professional, 2006.

[7] Etnus. *MIPS Delay Slot Instructions*. URL: http://www.jaist.ac.jp/iscenter-new/mpc/old-machines/altix3700/opt/toolworks/totalview.6.3.0-1/doc/html/ref_guide/MIPSDelaySlotInstructions.html.

INT 3 in x86. This particular instruction is assigned to the opcode 0xCC which, being only one byte long, can replace the first byte of any instruction without overwriting other code. Software watchpoints are implemented by single-stepping the program and examining the desired locations after each step (a very slow process). The only advantage of software breakpoints is their virtually unlimited number, while disadvantages include being slower than their hardware counterpart, and being impossible to use if the program's code resides in protected memory that cannot be modified or if the program modifies its own instructions during execution.

2.1.4 LLDB

LLDB[8] is a free and open-source debugger developed as part of the LLVM project. Starting with Apple's adoption of LLVM with Xcode 3.2[9], LLDB eventually replaced GDB as the debugger of choice for macOS and its kernel[10], and nowadays 'is the default debugger in Xcode on macOS and supports debugging C, Objective-C and C++ on the desktop and iOS devices and simulator.' On macOS, LLDB is shipped as part of the Command Line Tools[11], a small self-contained package consisting of the macOS SDK and tools for command line development. The debugger exposes its full application programming interface (API) both as a shared library and also through Python bindings. LLDB is released under the Apache License version 2.0 with LLVM exceptions[12] to ensure the software is very permissively licensed.

Listing 2.1: An example debugging session with LLDB and the /bin/date command-line utility which displays on screen the current date and time. A breakpoint is set on the puts() function so to stop the execution of the program before the date and time are printed. When the breakpoint is hit, the output string is modified by replacing the token 'Tue' with 'ABC'. The execution of the program is then resumed and the patched date and time is printed on screen.

```
(lldb) file /bin/date
Current executable set to '/bin/date' (x86_64).
(lldb) breakpoint set -b puts
Breakpoint 1: where = libsystem_c.dylib`puts, address = 0x000000000003f810
(lldb) run
Process 1501 launched: '/bin/date' (x86_64)
Process 1501 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00007fff6c45f810 libsystem_c.dylib`puts
libsystem_c.dylib`puts:
->  0x7fff6c45f810 <+0>: pushq  %rbp
    0x7fff6c45f811 <+1>: movq   %rsp, %rbp
```

[8] *The LLDB Debugger*. URL: https://lldb.llvm.org.

[9] Apple. *LLVM Compiler Overview*. URL: https://developer.apple.com/library/archive/documentation/CompilerTools/Conceptual/LLVMCompilerOverview/.

[10] Apple. *LLDB Quick Start Guide. About LLDB and Xcode*. URL: https://developer.apple.com/library/archive/documentation/IDEs/Conceptual/gdb_to_lldb_transition_guide/document/Introduction.html; eskimo. *Re: debugging kernel drivers*. URL: https://forums.developer.apple.com/message/28317#27581.

[11] Apple. *Technical Note TN2339: Building from the Command Line with Xcode FAQ*. URL: https://developer.apple.com/library/archive/technotes/tn2339/_index.html.

[12] *New LLVM Project License Framework*. URL: https://llvm.org/docs/DeveloperPolicy.html%5C#new-llvm-project-license-framework.

```
    0x7fff6c45f814 <+4>: pushq  %r14
    0x7fff6c45f816 <+6>: pushq  %rbx
Target 0: (date) stopped.
(lldb) p/x $rdi
(unsigned long) $0 = 0x00007ffeefbff2a0
(lldb) x/s $rdi
0x7ffeefbff2a0: "Tue Nov 26 07:01:07 CET 2019"
(lldb) memory write $rdi 41 42 43
(lldb) x/s $rdi
0x7ffeefbff2a0: "ABC Nov 26 07:01:07 CET 2019"
(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()' or Ctrl-D.
>>> lldb.frame.register["rdi"].value
'0x00007ffeefbff2a0'
>>> lldb.process.ReadMemory(0x00007ffeefbff2a0, 10, lldb.SBError())
b'ABC Nov 26'
>>> ^D
now exiting InteractiveConsole...
(lldb) continue
Process 1501 resuming
ABC Nov 26 07:01:07 CET 2019
Process 1501 exited with status = 0 (0x00000000)
```

2.2 On hardware virtualisation

In computing, hardware virtualisation refers to the process of creating and running a virtual representation of a real computer system, called system-level[13] VM, which requires emulating in software all the hardware resources required by the system, such as CPUs, RAM, hard disk drives and peripherals. A virtual machine can be seen as a set of parameters that describe both the configuration of the virtual hardware in which it should be run (e.g. the number of virtual CPUs or the amount of RAM), and information related to the run time state of the VM. The most important benefit of system-level virtualisation is running multiple unmodified operating systems simultaneously, each inside its own VM, all sharing the resources of a single physical machine, with the only practical constraint of memory and hard disk size.

The creation and execution of virtual machines is managed by the virtual machine monitor (VMM), also called hypervisor, which enables the VMs to execute their software as if this was running directly on the physical hardware, isolated from all the other virtual environments and the underlying host. By the definition of Popek and Goldberg[14], any generic hypervisor:

- Provides an environment where programs exhibit identical effects to those produced if the same programs had been run directly on the original physical machine, but without considering ‘differences caused by the availability of system resources and differences caused by timing dependencies.’
- Maintains full control on system resources, prohibiting accesses to the ones

[13]As opposed to process-level virtual machines, like the Java VM or .NET Framework, which allow to execute computer programs in the same way across different hardware and OSes.

[14]Gerald J Popek and Robert P Goldberg. ‘Formal requirements for virtualizable third generation architectures’. In: *Communications of the ACM* 17.7 (1974), pp. 412–421.

not explicitly allocated and possibly reclaiming back their control at any moment.

- Executes most of the virtual system's instructions directly on the real CPU for maximum efficiency.

The term 'host' is used to indicate the physical hardware and the software that runs on it (e.g. the hypervisor), while 'guest' designates the virtual machines and the OS and processes that run inside them. Hypervisors are commonly classified as:

- Type-1, also called native or bare-metal hypervisors, which run directly on the host's hardware by providing a minimal OS for executing virtual machines.

- Type-2, also called hosted hypervisors, which run as (privileged) user applications on the host's OS.

In modern CPUs, virtualisation is often hardware-assisted, so to offload some workload of the hypervisor to the hardware and increase performances; examples of virtualisation-enabled hardware are AMD-V and Intel VT-x, both for x86. Lastly, virtualisation is not emulation: in this second case, the guest code cannot run directly on the host because the CPU architecture is different, and thus every guest machine instruction must be translated to the host CPU's instruction set before it can be executed.

2.2.1 Virtual machine introspection

VMI is a technique for monitoring the run time state of system-level virtual machines. Most of the times, no additional software needs to be installed on the VM: inspection and manipulation of its internal state occur at hypervisor level through a dedicated API, which usually allows at minimum to access the VM processor's registers and memory and to track hardware events such as interrupts or memory writes. Additional VMI tools may help transforming this low-level view of the VM state into a more meaningful representation. When the API offers support for breakpoints, then it becomes possible to debug a VM without the guest being aware of the process; examples of hypervisor debuggers are xendbg[15], pyvmidbg[16] and the GDB stub in VMware Fusion (described later in section 3.6.3). If the hypervisor doesn't implement an interface for VMI, introspection may still be carried out with the help of an agent installed in the guest OS, but in this case examination is possible only after the agent has been started. Virtual machine introspection has been a popular solution in computer forensics, debugging and computer security, since it provides a complete view of the (virtual) system to analyse.

2.2.2 VirtualBox

Oracle VM VirtualBox[17] is a type-2 hypervisor developed by Oracle Corporation which 'runs on Windows, Linux, Macintosh, and Solaris hosts and supports a large

[15]Spencer Michaels. *xendbg*. URL: https://github.com/nccgroup/xendbg.
[16]Mathieu Tarral. *pyvmidbg*. URL: https://github.com/Wenzel/pyvmidbg.
[17]*Oracle VM VirtualBox*. URL: https://www.virtualbox.org.

number of guest operating systems', including macOS. Unlike all the other major professional solutions, VirtualBox is free and open-source: its core package (containing the full VirtualBox source code and platform binaries) is licensed under the GNU General Public License version 2[18], which allows changes and distribution provided that the modified software is open-sourced under equivalent license terms.

2.3 On macOS, Darwin and XNU

Developed by Apple, macOS (previously Mac OS X and OS X) is a series of operating systems for the Macintosh family of personal computers (nowadays commonly branded as Mac). macOS is built on top of Darwin, a Unix-based[19] OS also developed by Apple; the foundational layer of both operating systems is the XNU kernel. In Apple's words[20]:

> The kernel, along with other core parts of OS X are collectively referred to as Darwin. Darwin is a complete operating system based on many of the same technologies that underlie OS X. However, Darwin does not include Apple's proprietary graphics or applications layers, such as Quartz, QuickTime, Cocoa, Carbon, or OpenGL.

Both Darwin and XNU are released free and open-source[21] mostly under the Apple Public Source License, although sources are not published all the times or are distributed only months after the corresponding macOS release. Today, Darwin and XNU constitute the basis also for other operating systems, like iOS and iPadOS for the iPhone and iPad families of smartphones and tablets, albeit with some important differences; for instance, the XNU version for iOS is stripped of all kernel debugging functionalities, which are instead left available for macOS. In this work, the terms 'macOS', 'macOS kernel', 'Darwin kernel' and 'XNU' are used interchangeably, with the last one referring exclusively to the specific build for macOS.

2.3.1 System Integrity Protection

System Integrity Protection (SIP), also referred to as 'rootless', is a modern security feature of macOS whose aim is to limit the power of the superuser. Restrictions, enforced directly by the kernel, include[22]:

[18] *GNU General Public License, version 2*. URL: https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html.

[19] Since macOS 10.5 Leopard, every version of macOS has been certified as compatible with the UNIX 03 certification. See The Open Group. *Mac OS X Version 10.5 Leopard on Intel-based Macintosh computers*. URL: https://www.opengroup.org/openbrand/register/brand3555.htm; The Open Group. *macOS version 10.15 Catalina on Intel-based Mac computers*. URL: https://www.opengroup.org/openbrand/register/brand3653.htm.

[20] Apple. *Kernel Programming Guide. Kernel Architecture Overview*. URL: https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/Architecture/Architecture.html.

[21] Apple. *Apple Open Source*. URL: https://opensource.apple.com.

[22] Apple. *About System Integrity Protection on your Mac*. URL: https://support.apple.com/en-us/HT204899; Apple. *System Integrity Protection Guide*. URL: https://developer.apple.com/library/archive/documentation/Security/Conceptual/System_Integrity_Protection_Guide/.

- Preventing modifications to critical system files and directories.
- Forbidding run time attachment to system processes (e.g. with DTrace).
- Disallowing the execution of kernel extensions that are not signed with a 'Developer ID for Signing Kexts' certificate.
- Disallowing write access to some variables stored in the Mac's non-volatile random-access memory (NVRAM), used to store settings like time zone information, screen resolution, and sound volume[23].

The actual SIP configuration is also stored in NVRAM, and as such it applies to all installations of macOS running on the machine; modifying such configuration is too an operation restricted by SIP itself, since otherwise disabling this security measure would be trivial. SIP is enabled by default, but can be disabled with the `csrutil` command-line utility, either from macOS Recovery[24] (a minimal operating system for system recovery residing on a hidden partition of the Mac's hard disk drive) or a bootable macOS installation disk. Currently, this utility allows disabling SIP either in full or selectively; for example, executing `csrutil enable ↪ --without nvram` will disable only the lock on NVRAM variables modification[25]. However, `csrutil` also informs the user that disabling SIP in part generates an unsupported configuration that is likely to break in the future.

[23] Apple. *Reset NVRAM or PRAM on your Mac*. URL: https://support.apple.com/en-us/HT204063.
[24] Apple. *About macOS Recovery*. URL: https://support.apple.com/en-us/HT201314.
[25] Max108. *Enabling parts of System Integrity Protection while disabling specific parts?* URL: https://forums.developer.apple.com/thread/17452#thread-message-52814.

Chapter 3

Debugging the macOS kernel

This chapter describes how to debug recent version of the macOS kernel with a major focus on the the Kernel Debugging Protocol, XNU's mechanism for remote kernel debugging. Several sections are dedicated to explain how KDP is implemented in the kernel, how to set up recent versions of macOS for remote debugging, and the notable limitations of this approach. The only valid alternative to KDP is the GDB stub in VMware Fusion, presented at the end of the chapter, which brings improvements in many aspects but is also affected by a couple of different drawbacks. Unless otherwise stated, references to source code files are provided for XNU 4903.221.2[1] from macOS 10.14.1 Mojave, the most up-to-date source release available at the time of the study; at the time of writing, the sources of XNU 4903.241.1[2] from macOS 10.14.3 Mojave and XNU 6153.11.26[3] from macOS 10.15 Catalina have been published. Many of the outputs presented were edited or truncated for clarity of reading. As already noted in section 2.3, the terms 'macOS', 'macOS kernel', 'Darwin kernel' and 'XNU' (this last referring exclusively to the specific build for macOS) will be used interchangeably, sacrificing a little accuracy for better readability.

3.1 The Kernel Debugging Protocol

Like every other major operating system[4], macOS supports remote kernel debugging to allow, under certain circumstances, a kernel-mode debugger running on a

---

[1] Apple. *XNU 4903.221.2 Source*. URL: https://opensource.apple.com/source/xnu/xnu-4903.221.2/.

[2] Apple. *XNU 4903.241.1 Source*. URL: https://opensource.apple.com/source/xnu/xnu-4903.241.1/.

[3] Apple. *XNU 6153.11.26 Source*. URL: https://opensource.apple.com/tarballs/xnu/xnu-6153.11.26.tar.gz.

[4] Windows supports remote kernel debugging with KDNET and the WinDbg debugger, and the Linux kernel with KGDB and the GDB debugger. See *Getting Started with WinDbg (Kernel-Mode)*. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/getting-started-with-windbg--kernel-mode-; Jason Wessel. *Using kgdb, kdb and the kernel debugger internals*. URL: https://www.kernel.org/doc/html/v4.17/dev-tools/kgdb.html.

second machine to inspect and manipulate the state of the entire system. As mentioned in the kernel's README[5], for such purpose XNU implements the Kernel Debugging Protocol, a debugging interface to be interacted via a custom client–server protocol over UDP. As a typical kernel debugging mechanism, the KDP solution consists of two parts:

- A debug server running internally to the macOS kernel, listening for connections on port 41139, capable to alter the normal execution of the operating system in order to execute debugging commands sent by a client. Throughout this work, this component is also referred to as the KDP stub or agent.
- An external kernel-mode debugger running on a different machine, typically LLDB (see section 2.1.4), which manages the debugging session by sending requests to the KDP server and eventually receiving back results and notifications of CPU exceptions.

KDP can be used either via Ethernet, FireWire or the serial interface, with the possibility of using Thunderbolt adapters in case such ports are not available; but since network interfaces can be used for debugging only when their driver explicitly supports KDP, debugging over Wi-Fi is not supported[6]. When the serial interface is used, 'KDP still encapsulates every message inside a fake Ethernet and UDP packet.'[7] Since debugging has to be available as early as possible in the boot process, KDP 'does not use the kernel's networking stack but has its own minimal UDP/IP implementation'[8].

The behaviour of the KDP stub can be configured through boot-arg variables. The most important one is debug, used to specify if and when the agent should activate, among other purposes; see listing 3.1 for a list of supported bitmasks. A summary scan of XNU sources reveals further options: `kdp_crashdump_pkt_size`, to set the size of the crash dump packet; `kdp_ip_addr`, to set a static IP address for the KDP server; `kdp_match_name`, to select which port to use (e.g. `en1`) for Ethernet, Thunderbolt or serial debugging. Additionally, the IONetworkingFamily kernel extension[9] parses the variable `kdp_match_mac` to match against a specific MAC address; this indicates that likely more KDP-related options exist for configuring other kernel extensions.

Listing 3.1: Bitmasks for the debug boot-arg from `osfmk/kern/debug.h` [XNU]

```
/* Debug boot-args */
#define DB_HALT         0x1
//#define DB_PRT          0x2 -- obsolete
#define DB_NMI          0x4
#define DB_KPRT         0x8
#define DB_KDB          0x10
#define DB_ARP          0x40
#define DB_KDP_BP_DIS   0x80
//#define DB_LOG_PI_SCRN  0x100 -- obsolete
#define DB_KDP_GETC_ENA 0x200
```

[5] `README.md` [XNU]
[6] Amit Singh. *Mac OS X internals: a systems approach*. Addison-Wesley Professional, 2006.
[7] Charlie Miller et al. *iOS Hacker's Handbook*. John Wiley & Sons, 2012.
[8] Singh, *Mac OS X internals: a systems approach*.
[9] Apple. *IONetworkingFamily 129.200.1 Source*. URL: https://opensource.apple.com/source/IONetworkingFamily/IONetworkingFamily-129.200.1/IOKernelDebugger.cpp.auto.html.

```

#define DB_KERN_DUMP_ON_PANIC           0x400 /* Trigger core dump on panic*/
#define DB_KERN_DUMP_ON_NMI             0x800 /* Trigger core dump on NMI */
#define DB_DBG_POST_CORE                0x1000 /*Wait in debugger after NMI core */
#define DB_PANICLOG_DUMP                0x2000 /* Send paniclog on panic,not core*/
#define DB_REBOOT_POST_CORE             0x4000 /* Attempt to reboot after
                                                * post-panic crashdump/paniclog
                                                * dump.
                                                */
#define DB_NMI_BTN_ENA          0x8000  /* Enable button to directly trigger NMI */
#define DB_PRT_KDEBUG           0x10000 /* kprintf KDEBUG traces */
#define DB_DISABLE_LOCAL_CORE   0x20000 /* ignore local kernel core dump support */
#define DB_DISABLE_GZIP_CORE    0x40000 /* don't gzip kernel core dumps */
#define DB_DISABLE_CROSS_PANIC  0x80000 /* x86 only - don't trigger cross panics. Only
                                         * necessary to enable x86 kernel debugging on
                                         * configs with a dev-fused co-processor running
                                         * release bridgeOS.
                                         */
#define DB_REBOOT_ALWAYS        0x100000 /* Don't wait for debugger connection */
```

The current revision of the communication protocol used by KDP is the 12th[10], around since XNU 1456.12.6[11] from macOS 10.6 Snow Leopard. As in many other networking protocols, KDP packets are composed of a common header and specialised bodies. The header, shown in listing 3.2, contains, among other fields:

- The type of KDP request, such as KDP_READMEM64 or KDP_BREAKPOINT_SET; the full set of possible requests is shown in listing B.1.
- A flag for distinguishing between requests and replies. With the exclusion of KDP_EXCEPTION which is a notification[12], KDP requests are only sent by the debugger to the debuggee (and not vice versa)[13].
- A sequence number to discard duplicate or out-of-order messages and retransmit replies[14].

Listing 3.2: The KDP packet header from osfmk/kdp/kdp_protocol.h [XNU]

```
typedef struct {
        kdp_req_t       request:7;      /* kdp_req_t, request type */
        unsigned        is_reply:1;     /* 0 => request, 1 => reply */
        unsigned        seq:8;          /* sequence number within session */
        unsigned        len:16;         /* length of entire pkt including hdr */
        unsigned        key;            /* session key */
} KDP_PACKED kdp_hdr_t;
```

[10] osfmk/kdp/kdp.c#L109 [XNU]

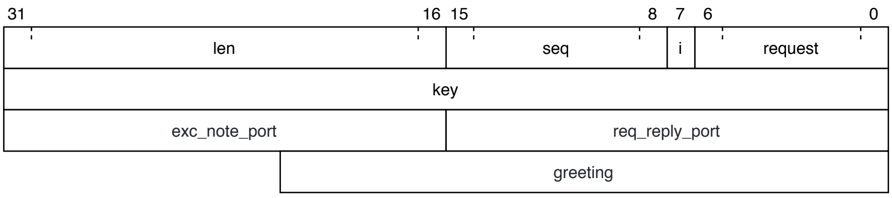
[11] Apple. *XNU 1456.1.26 Source*. URL: https://opensource.apple.com/source/xnu/xnu-1456.1.26/.

[12] osfmk/kdp/kdp_protocol.h#L104 [XNU]

[13] osfmk/kdp/kdp_udp.c#L1087 [XNU]

[14] osfmk/kdp/kdp_udp.c#L1095 [XNU]

Figure 3.1: Representation of a `KDP_CONNECT` request packet. The KDP header occupies the first 8 bytes, in which `is_reply` is the eighth least significant bit. The `req_reply_port` and `exc_note_port` fields are the client UDP ports where to send replies and exception notifications. The `greeting` field is an arbitrary null-terminated string of variable length.



Instead, bodies contain each different fields that define all the possible KDP requests and replies, as shown in listing 3.3 as an example for `KDP_READMEM64` packets.

Listing 3.3: The `KDP_READMEM64` request and reply packets, as defined in `osfmk/kdp/kdp_protocol.h` [XNU]

```
typedef struct {                        /* KDP_READMEM64 request */
        kdp_hdr_t       hdr;
        uint64_t        address;
        uint32_t        nbytes;
} KDP_PACKED kdp_readmem64_req_t;

typedef struct {                        /* KDP_READMEM64 reply */
        kdp_hdr_t       hdr;
        kdp_error_t     error;
        char            data[0];
} KDP_PACKED kdp_readmem64_reply_t;
```

As might be expected, XNU assumes at most one KDP client is attached to it at any given time. With the initial `KDP_CONNECT` request, the debugger informs the kernel on which UDP port should notifications be sent back when exceptions occur. The interactions between the KDP stub and the LLDB debugger during the attach phase are explored in detail in Appendix A.

3.1.1 Triggering the debugging stub

Naturally, the macOS kernel is not open to debugging by default. From a thorough search of XNU sources and some testing, it seems the KDP stub is allowed to take over the normal execution of the kernel only in three specific situations:

- The kernel is a DEVELOPMENT or DEBUG build (see section 3.2) and the `DB_HALT` bit has been set for the debug boot-arg. If these conditions are met during kernel boot, then the debugging stub pauses the startup process waiting for a debugger.

Listing 3.4: Checking for `DB_HALT` in `osfmk/kern/debug.c` [XNU]

```
        /*
         * Initialize the value of the debug boot-arg
         */
```

```
        debug_boot_arg = 0;
#if ((CONFIG_EMBEDDED && MACH_KDP) || defined(__x86_64__))
        if (PE_parse_boot_argn("debug", &debug_boot_arg, sizeof (debug_boot_arg))) {
#if DEVELOPMENT || DEBUG
                if (debug_boot_arg & DB_HALT) {
                        halt_in_debugger=1;
                }
#endif
```

- The kernel is being run on a hypervisor (according to the CPU feature flags outputted by the `CPUID` instruction[15]), the `DB_NMI` bit has been set for the debug boot-arg and a non-maskable interrupt (NMI) is triggered at any time during the OS execution.

Listing 3.5: Starting KDP on NMIs in `osfmk/kdp/kdp_protocol.h` [XNU]

```
        } else if (!mp_kdp_trap &&
                   !mp_kdp_is_NMI &&
                   virtualized && (debug_boot_arg & DB_NMI)) {
                /*
                 * Under a VMM with the debug boot-arg set, drop into kdp.
                 * Since an NMI is involved, there's a risk of contending with
                 * a panic. And side-effects of NMIs may result in entry into,
                 * and continuing from, the debugger being unreliable.
                 */
                if (__sync_bool_compare_and_swap(&mp_kdp_is_NMI, FALSE, TRUE)) {
                        kprintf_break_lock();
                        kprintf("Debugger␣entry␣requested␣by␣NMI\n");
                        kdp_i386_trap(T_DEBUG, saved_state64(regs), 0, 0);
                        printf("Debugger␣entry␣requested␣by␣NMI\n");
                        mp_kdp_is_NMI = FALSE;
                } else {
                        mp_kdp_wait(FALSE, FALSE);
                }
```

- The debug boot-arg has been set to any nonempty value (even invalid ones) and a panic occurs[16], in which case the machine is not automatically restarted. Panics can be triggered programmatically with DTrace from the command-line by executing `dtrace -w -n "BEGIN{ panic(); }"` (assuming SIP is disabled, see section 2.3.1).

Once the KDP stub is triggered with any of these methods, the kernel simply loops waiting for an external debugger to attach. Notably, all three cases require changing the kernel boot-args to set the value of debug.

3.2 The Kernel Debug Kit

For some macOS releases and XNU builds, Apple publishes the corresponding Kernel Debug Kit (KDK), an accessory package for kernel debugging containing:

[15] `osfmk/i386/machine_routines.c#L711` [XNU]
[16] `osfmk/kern/debug.c#L290` [XNU]

- The DEVELOPMENT and DEBUG builds of the kernel, compiled with additional assertions and error checking with respect to the RELEASE version distributed with macOS; occasionally, also a KASAN build compiled with address sanitisation is included. Unlike RELEASE, these debug builds also contain full symbolic information.
- DWARF companion files generated at compile time containing full debugging information, such as symbols and data type definitions, for each of the kernel builds included in the debug kit and also for some kernel extensions shipped with macOS. If XNU sources are also available, then source-level kernel debugging becomes possible (e.g. with LLDB and the command `settings set target.source-map`[17]).
- lldbmacros (discussed in section 3.2.1), a set of debug scripts to assist the debugging of Darwin kernels.

All available KDKs can be downloaded from the Apple Developer website[18] after authenticating with a free Apple ID account. Being distributed as .pkg packages, KDKs are usually installed through the macOS GUI, procedure that simply copies the package content into the local file system at `/Library/Developer/KDKs/`.

Listing 3.6: Kernels builds from the KDK for macOS 10.14.5 Mojave build 18F132

```
$ ls -l /Library/Developer/KDKs/KDK_10.14.5_18F132.kdk/System/Library/Kernels/
total 193192
-rwxr-xr-x  1 root  wheel  15869792 Apr 26  2019 kernel
drwxr-xr-x  3 root  wheel        96 Apr 26  2019 kernel.dSYM
-rwxr-xr-x  1 root  wheel  21428616 Apr 26  2019 kernel.debug
drwxr-xr-x  3 root  wheel        96 Apr 26  2019 kernel.debug.dSYM
-rwxr-xr-x  1 root  wheel  17018112 Apr 26  2019 kernel.development
drwxr-xr-x  3 root  wheel        96 Apr 26  2019 kernel.development.dSYM
-rwxr-xr-x  1 root  wheel  44591632 Apr 26  2019 kernel.kasan
```

Kernel Debug Kits are incredibly valuable for kernel debugging: information about data types makes it easy to explore kernel data structures through the debugger, and lldbmacros provide deep introspection capabilities. Unfortunately, for unknown reasons Apple does not distribute KDKs for all macOS releases and updates, and when it does these packages are often published with weeks or months of delay. By searching the Apple Developer portal for the non-beta builds of macOS 10.14 Mojave as an example, at the time of this study in late May 2019 the KDKs published on the same day as the respective macOS release were only three (18A391, 18C54 and 18E226) out of a total ten; one KDK was released two weeks late (18B75); and no KDK was provided for the other six kernel builds (18B2107, 18B3094, 18D42, 18D43, 18D109, 18E227). As of September 2019 four more macOS updates have been distributed, for which two KDKs (18F132, 18G84) were promptly released and the other two (18G87 and 18G95) are missing. From a post on the Apple Developer Forums it appears that nowadays 'the correct way to request a new KDK is to file a bug asking for it.'[19]

[17] Zach Cutlip. *Source Level Debugging the XNU Kernel*. URL: https://shadowfile.inode.link/blog/2018/10/source-level-debugging-the-xnu-kernel/.

[18] Apple. *More Software Downloads - Apple Developer*. URL: https://developer.apple.com/download/more/?=Kernel%5C%20Debug%5C%20Kit.

[19] eskimo. *Re: Where can I find Kernel Debug Kit for 10.11.6 (15G22010)?* URL: https://forums.

3.2.1 lldbmacros

As a replacement for the now abandoned kgmacros for GDB, since XNU 2050.7.9[20] from macOS 10.8 Mountain Lion Apple has been releasing lldbmacros, a set of Python scripts for extending LLDB's capabilities with helpful commands and macros for debugging Darwin kernels. Examples are `allproc`[21] to display information about processes, `pmap_walk`[22] to perform virtual to physical address translation, and `showallkmods`[23] for a summary of all loaded kexts.

Listing 3.7: Example output of the `allproc` macro, executed during the startup process of macOS 10.14.5 Mojave build 18F132

```
(lldb) allproc
Process 0xffffff800c8577f0
    name kextcache
    pid:11     task:0xffffff800c023bf8   p_stat:2      parent pid: 1
Cred: euid 0 ruid 0 svuid 0
Flags: 0x4004
    0x00000004 - process is 64 bit
    0x00004000 - process has called exec
State: Run
Process 0xffffff800c857c60
    name launchd
    pid:1      task:0xffffff800c022a70   p_stat:2      parent pid: 0
Cred: euid 0 ruid 0 svuid 0
Flags: 0x4004
    0x00000004 - process is 64 bit
    0x00004000 - process has called exec
State: Run
Process 0xffffff8006076b58
    name kernel_task
    pid:0      task:0xffffff800c023048   p_stat:2      parent pid: 0
Cred: euid 0 ruid 0 svuid 0
Flags: 0x204
    0x00000004 - process is 64 bit
    0x00000200 - system process: no signals, stats, or swap
State: Run
```

If the KDK for the kernel being debugged is installed in the host machine, just after attaching LLDB will detect the availability of lldbmacros and suggest loading them with the command command `script import`. In addition to be distributed as part of any KDKs, lldbmacros are also released together with XNU sources; however, to operate properly (or at all) most macros require debugging information, which is released only within the debug kits in the form of DWARF companion files.

developer.apple.com/thread/108732#351881.

[20] Apple. *XNU 2050.7.9 Source*. URL: https://opensource.apple.com/source/xnu/xnu-2050.7.9/.

[21] tools/lldbmacros/process.py#L109 [XNU]

[22] tools/lldbmacros/pmap.py#L895 [XNU]

[23] tools/lldbmacros/memory.py#L1388 [XNU]

3.3 Setting macOS up for remote debugging

Apple's documentation on kernel debugging[24] is outdated (ca. 6 years old as of 2019) and no longer being updated, but fortunately the procedure described there has not changed much to this day. In addition, many third-party guides on how to set up recent versions of macOS for debugging are available on the Internet[25]. According to all these resources, the suggested steps for enabling remote kernel debugging via KDP on a modern macOS target machine involve:

1. Finding the exact build version of the macOS kernel to debug, for example by executing the `sw_vers` command-line utility and reading its output at the line starting with `BuildVersion`.

   Listing 3.8: Example output of the `/usr/bin/sw_vers` utility

   ```
   $ sw_vers
   ProductName:    Mac OS X
   ProductVersion: 10.15.2
   BuildVersion:   19C57
   ```

2. Downloading and installing the Kernel Debug Kit for the specific build version, so to obtain a copy of the debug builds of the kernel. The same KDK should be installed also in the host machine (supposing it's running macOS), so to give LLDB or any other debugger access to a copy of the kernel executables that is going to be debugged.

3. Disabling System Integrity Protection from macOS Recovery, to remove the restrictions on replacing the default kernel binary and modifying boot-args in NVRAM.

4. Installing at least one of the debug builds of the kernel by copying the desired executable (e.g. `kernel.development`) from the directory of the installed KDK to `/System/Library/Kernels/`.

5. Setting the `kcsuffix` and debug boot-args to appropriate values, the first to select which of the installed kernel builds to use for the next macOS boot and the second to actually enable the debugging features of the kernel. Boot-args can be set using the `nvram` command-line utility, as shown in listing 3.9. Proper values for `kcsuffix` are typically 'development', 'debug' or 'kasan'; valid bitmasks for debug were listed in listing 3.1. Apple's docs and other resources[26] also mention setting `pmuflags=1` to avoid watchdog timer problems, but this parameter doesn't seem to be parsed anywhere in

[24] Apple. *Kernel Programming Guide. Building and Debugging Kernels*. URL: https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/build/build.html.

[25] Scott Knight. *macOS Kernel Debugging*. URL: https://knight.sc/debugging/2018/08/15/macos-kernel-debugging.html; GeoSn0w. *Debugging macOS Kernel For Fun*. URL: https://geosn0w.github.io/Debugging-macOS-Kernel-For-Fun/; RedNaga Security. *Remote Kext Debugging*. URL: https://rednaga.io/2017/04/09/remote_kext_debugging/; LightBulbOne. *Introduction to macOS Kernel Debugging*. URL: https://lightbulbone.com/posts/2016/10/intro-to-macos-kernel-debugging/.

[26] Apple, *Kernel Programming Guide*; Xiang Lei. *XNU kernel debugging via VMWare Fusion*. URL: http://trineo.net/p/17/06_debug_xnu.html.

recent XNU sources (although it could still be used by some kernel extension); possibly related, the READMEs of some recent KDKs suggest instead to set `watchdog=0` to 'prevent the macOS watchdog from firing when returning from the kernel debugger.'

Listing 3.9: Example usage of the `/usr/sbin/nvram` utility

```
$ sudo nvram boot-args="-v␣kcsuffix=development␣debug=0x4"
$ nvram -p | grep boot-args
boot-args       -v kcsuffix=development debug=0x4
```

6. Recreating the 'kextcache', to allow macOS to boot with a different kernel build than the last one used. Caches can apparently be rebuilt either by executing the command `touch` on the `/System/Library/Extensions/` directory of the installation target volume, as recommended by the `kextcache` manual page[27], or by executing the command `kextcache -invalidate ↪ /Volumes/<Target Volume>`, as suggested by several other resources including the READMEs of some KDKs. Both methods appear to work, even though it's not clear which of the two is to be preferred on recent versions of macOS.

7. Lastly, rebooting the machine into macOS and triggering the activation of the debugging stub in the kernel, in accordance with how it has been set up via the debug boot-arg. In all cases, either during boot or in response to panics or NMIs, the kernel will deviate from its normal execution to wait for a remote debugger to connect; at that point, any debugger supporting KDP (such as LLDB with the `kdp-remote` command) can attach to the kernel and start debugging.

Although these instructions allow to correctly set up a Mac for kernel debugging, they are in some parts neither exhaustive nor completely accurate. Some observations:

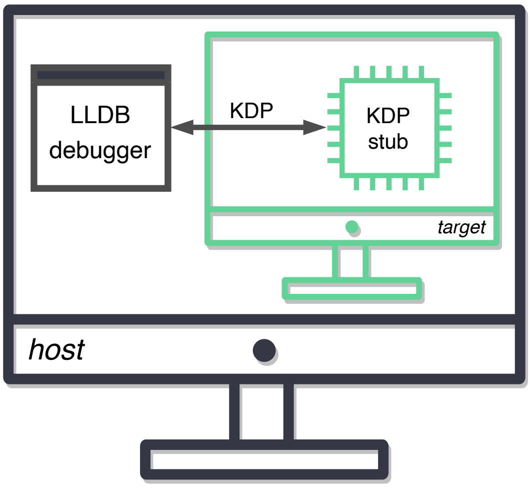
- All resources cited above suggest to disable SIP, but this is not required at all: installing the debug binaries and setting boot-args can be done directly from macOS Recovery without disabling SIP for macOS (operation that requires a reboot into macOS Recovery anyway).
- Similarly, no resource points out that to work properly KDP doesn't actually require SIP to be turned off, and so that it's always possible to re-enable it prior to debugging: depending on the parts of the kernel that will be analysed, it may be desirable to leave this security mechanism on.
- Mentioned only once and superficially, installing a debug build of the kernel is not strictly required since it is also possible to debug the RELEASE kernel (see section 3.1.1), although it's generally preferable as debug executables aid the debugging process in multiple ways (e.g. with address sanitisation).

To eventually restore macOS to the RELEASE kernel, it is at minimum required to:

- Remove the `kcsuffix` argument from the boot-args.

[27] *man page kextcache section 8*. URL: http://www.manpagez.com/man/8/kextcache/.

Figure 3.2: Debugging macOS running on a virtual machine with KDP



- Remove all kernel caches at `/System/Library/Caches/com.apple.kext.caches/Startup/kernelcache.*.` and all prelinked kernels at `/System/Library/PrelinkedKernels/prelinkedkernel.*.`

Instead of using a physical Mac, as shown in some other online tutorials[28] it is also possible to debug macOS when this is running on a virtual machine, without changes in the configuring procedure described above; this is very convenient since VMs are much easier to set up, debug, reuse and dispose than physical machines. The process is illustrated in fig. 3.2. As additional benefit, when the target machine is a VM rebooting into macOS Recovery to modify NVRAM and boot-args is not required, since this operation is usually done by the hypervisor bypassing the guest OS (e.g. with the VirtualBox command `VBoxManage setextradata "<Vm ↪ Name>" "VBoxInternal2/EfiBootArgs" "<Var>=<Value>"`). Regardless of how the virtual machine running macOS is used, to comply with the Apple end-user license agreement (EULA) the general consensus[29] is that it must run on top of another copy of macOS installed directly on Apple hardware, i.e. a real Mac.

3.4 An example debugging session

This section demonstrates a simple KDP debugging session, conducted on a host machine running macOS 10.15 Catalina build 19A602 with LLDB 1100.0.28.19 and a target VirtualBox VM running macOS 10.14.5 Mojave build 18F132. Both machines installed the Kernel Debug Kit build 18F132. The target machine was

[28] Kedy Liu. *Debugging macOS Kernel using VirtualBox*. URL: https://klue.github.io/blog/2017/04/macos_kernel_debugging_vbox/; Damien DeVille. *Kernel debugging with LLDB and VMware Fusion*. URL: http://ddeville.me/2015/08/kernel-debugging-with-lldb-and-vmware-fusion; snare. *Debugging the Mac OS X kernel with VMware and GDB*. URL: http://ho.ax/posts/2012/02/debugging-the-mac-os-x-kernel-with-vmware-and-gdb/; Lei, *XNU kernel debugging via VMWare Fusion*.

[29] John Lockwood. *OSX installing on virtual machine (legal issues)*. URL: https://discussions.apple.com/thread/7312791?answerId=29225237022#29225237022.

configured as explained in section 3.3: first, SIP was fully disabled from macOS Recovery by executing the command `csrutil disable` in the Terminal application; then, the DEBUG kernel was copied from `/Library/Developer/KDKs/KDK_10.14.5_18F132.kdk/System/Library/Kernels/kernel.debug` to `/System/Library/Kernels/`; next, the boot-arg variable was set to `"kcsuffix=debug debug=0x1"`; lastly, the kext cache was rebuilt in consequence of executing the command `touch ↪ /System/Library/Extensions/`.

Since `DB_HALT` was set, shortly after initiating booting on the target machine the KDP stub assumed control and eventually the string 'Waiting for remote debugger connection.' was printed on screen; LLDB, running on the host machine, could then attach with the `kdp-remote` command:

```
(lldb) kdp-remote 10.0.2.15
Version: Darwin Kernel Version 18.6.0: Thu Apr 25 23:15:12 PDT 2019;
     ↪ root:xnu_debug-4903.261.4~1/DEBUG_X86_64;
     ↪ UUID=4578745F-1A1F-37CA-B786-C01C033D4C22; stext=0xffffff800f000000
Kernel UUID: 4578745F-1A1F-37CA-B786-C01C033D4C22
Load Address: 0xffffff800f000000
warning: 'kernel' contains a debug script. To run this script in this debug
     ↪ session:

    command script import "/System/. . ./KDKs/KDK_10.14.5_18F132.kdk/. .
     ↪ ./kernel.py"

To run all discovered debug scripts in this session:

    settings set target.load-script-from-symbol-file true

Kernel slid 0xee00000 in memory.
Loaded kernel file /System/. .
     ↪ ./KDKs/KDK_10.14.5_18F132.kdk/System/Library/Kernels/kernel.debug
Loading 68 kext modules
     ↪ ----.--.------.---....--------------.---------------......-------.-- done.
Failed to load 53 of 68 kexts:
    com.apple.AppleFSCompression.AppleFSCompressionTypeDataless
     ↪ 38BD7794-FDCB-3372-8727-B1209351EF47
    com.apple.AppleFSCompression.AppleFSCompressionTypeZlib
     ↪ 8C036AB1-8BF0-32BE-9B7F-75AD4C571D34
    . . .
    com.apple.security.quarantine
     ↪ 11DE02EC-241D-35AA-BBBB-A2E7969F20A2
    com.apple.security.sandbox
     ↪ ECE8D480-5444-3317-9844-559B22736E5A
Process 1 stopped
* thread #1, stop reason = signal SIGSTOP
    frame #0: 0xffffff800f2e3bf5 kernel.debug`kdp_call at kdp_machdep.c:331:1
Target 0: (kernel.debug) stopped.
```

The debugger had now control of the target machine. Active stack frames were retrieved with the `bt` command:

```
(lldb) bt
* thread #1, stop reason = signal SIGSTOP
  * frame #0: 0xffffff800f2e3bf5 kernel.debug`kdp_call . . .
    frame #1: 0xffffff800f05e28f kernel.debug`kdp_register_send_receive . . .
    frame #2: 0xffffff7f901b82df
     ↪ IONetworkingFamily`IOKernelDebugger::registerHandler . . .
```

```
    frame #3: 0xffffff7f901b8429 IONetworkingFamily`IOKernelDebugger::handleOpen
    ↪ . . .
    frame #4: 0xffffff800fb94be5 kernel.debug`IOService::open . . .
    frame #5: 0xffffff7f901b7910 IONetworkingFamily`IOKDP::start . . .
    frame #6: 0xffffff800fb96ac2 kernel.debug`IOService::startCandidate . . .
    frame #7: 0xffffff800fb960eb kernel.debug`IOService::probeCandidates . . .
    frame #8: 0xffffff800fb8e351 kernel.debug`IOService::doServiceMatch . . .
    frame #9: 0xffffff800fb97626 kernel.debug`_IOConfigThread::main . . .
    frame #10: 0xffffff800f2c278e kernel.debug`call_continuation + 46
```

CPU registers could be read and modified:

```
(lldb) register read
General Purpose Registers:
        rax = 0xffffff800fec5930  kernel.debug`halt_in_debugger
        rbx = 0xffffff801c8830c0
        rcx = 0x0000000000000001
        rdx = 0xffffff801241104c
        rdi = 0x0000000000000000
        rsi = 0xffffff800fc726cd  "_panicd_corename"
        rbp = 0xffffff81952e5b00
        rsp = 0xffffff81952e5b00
        r8 = 0xffffff81952e5ad0
        r9 = 0x0000000000000000
        r10 = 0x837bdd93184000bc
        r11 = 0x0000000000000079
        r12 = 0xffffff7f901bbd78
    ↪ IONetworkingFamily`IONetworkController::debugRxHandler(IOService*, void*,
    ↪ unsigned int*, unsigned int) at IONetworkController.cpp:1594
        r13 = 0xffffff7f901bbd98
    ↪ IONetworkingFamily`IONetworkController::debugSetModeHandler(IOService*,
    ↪ bool) at IONetworkController.cpp:1635
        r14 = 0xffffff801bcde530
        r15 = 0xffffff7f901d06d8
    ↪ IONetworkingFamily`IONetworkInterface::gMetaClass
        rip = 0xffffff800f2e3bf5  kernel.debug`kdp_call + 5 at kdp_machdep.c:331:1
    rflags = 0x0000000000000202
        cs = 0x0000000000000008
        fs = 0x0000000000000000
        gs = 0x0000000000000000
(lldb) register write $rax 0xffffffff
(lldb) register read $rax
        rax = 0x00000000ffffffff
```

Memory could also be read and modified:

```
(lldb) memory write -s4 $rsi 41414141
(lldb) memory read $rsi
0xffffff800fc726cd: 41 41 41 41 69 63 64 5f 63 6f 72 65 6e 61 6d 65
    ↪ AAAAicd_corename
0xffffff800fc726dd: 00 6b 64 70 5f 69 70 5f 61 64 64 72 00 5f 6b 64
    ↪ .kdp_ip_addr._kd
```

A breakpoint was installed on the `unix_syscall64()` routine:

```
(lldb) breakpoint set -a unix_syscall64
Breakpoint 1: where = kernel.debug`unix_syscall64 at systemcalls.c:275, address =
    ↪ 0xffffff800fae9210
```

Then, execution was resumed:

```
(lldb) continue
Process 1 resuming
```

And shortly thereafter the breakpoint fired, causing the kernel to stop its regular execution and return the control to LLDB:

```
Process 1 stopped
* thread #1, stop reason = breakpoint 1.1
    frame #0: 0xffffff800fae9210 kernel.debug`unix_syscall64(state=<unavailable>)
     ↪ at systemcalls.c:275
Target 0: (kernel.debug) stopped.
```

The debugging session was then terminated.

3.5 Limitations

As discussed and demonstrated in the previous sections, the Kernel Debugging Protocol offers most of the basic kernel debugging capabilities, such as reading and modifying CPU registers and memory and pausing the execution of the system with breakpoints. However, due to design and implementation choices this solution also presents several limitations and inconveniences, listed below in no particular order:

- To enable KDP and the debugging capabilities of the kernel it is required at minimum to set the debug boot-arg in NVRAM (see section 3.1.1), but this modification requires in turn to at least reboot into macOS Recovery, either to update the boot-args directly or to disable SIP. As already mentioned, this isn't necessary when the machine is a VM.

- Debugging the initial part of the kernel boot process is not possible since debugging can start only after the initialisation of the KDP stub, which happens relatively late in the startup phase.

- The debugging process alters in possibly unknown ways the default behaviour of the kernel and of some kernel extensions included in macOS; debugging a system not operating on default settings may not be desirable, especially in some security research contexts.

- Debugging has various side effects on the whole system, including: the modification of the value of global variables (e.g. `kdp_flag`[30]); the mapping of the 'low global vector' page at a fixed memory location[31]; and the altering of kernel code due to the temporary replacement of instructions with the `0xCC` opcode for software breakpoints. All these and likely others may impede debugging in adversarial situations, in which malware or exploits actively try to detect if the system is being debugged in order to conceal their behaviour; for these programs it is then sufficient to examine NVRAM or other global variables, or to detect breakpoints by searching code sections for `0xCC` bytes.

[30] `osfmk/kdp/kdp_udp.c#L252/#L433` [XNU]
[31] `osfmk/x86_64/pmap.c#L1171` [XNU]

- Hardware breakpoints and watchpoints are not supported. LLDB sets breakpoints by issuing `KDP_BREAKPOINT_SET` or `KDP_BREAKPOINT_SET64` requests which trigger the execution of `kdp_set_breakpoint_internal()`[32], whose implementation makes clear that only software breakpoints are used. The unavailability of hardware breakpoints is corroborated by the facts that there is no KDP request in kernel sources to do so and that the `KDP_WRITEREGS` request doesn't allow to modify x86 debug registers[33]. Moreover, the lack of watchpoints is also explicitly stated in LLDB sources[34].

Listing 3.10: Trying to set watchpoints in a KDP debugging session

```
(lldb) watchpoint set expression &((boot_args
    ↪ *)PE_state.bootArgs)->CommandLine
error: Watchpoint creation failed (addr=0xffffff8012411008, size=8).
error: watchpoints are not supported in kdp remote debugging
```

- Rather strangely, LLDB cannot pause the execution of the kernel once this has been resumed[35] (e.g. with the `continue` command). Inspection of XNU sources suggests that this feature seems to be supported by KDP with the `KDP_SUSPEND` request, although this has not been tested. At present, the only known way to pause a running macOS and return the control to the debugger is to trigger a breakpoint trap manually, for example with DTrace from the command-line of the debuggee by executing `dtrace -w -n "BEGIN{ ↪ breakpoint(); }"`, or by generating an NMI, either with specific hardware keys combinations if the target machine is a real Mac (e.g. by holding down both the left and right command keys while pressing the power button[36]) or through hypervisor commands in the case of virtual machines (e.g. the VirtualBox command `VBoxManage debugvm "<Vm Name>" injectnmi`).

Listing 3.11: Trying to interrupt a KDP debugging session

```
(lldb) process interrupt
error: Failed to halt process: Halt timed out. State = running
```

- After disconnecting from the remote kernel for any reason, it's apparently not always possible to reattach: 'Do not detach from the remote kernel!'[37]

- Multiple users report the whole debugging process via KDP to be frail, especially when carried out over UDP: 'LLDB frequently gets out of sync or loses contact with the debug server, and the kernel is left in a permanently halted state.'[38] This phenomenon seems to be acknowledged even in XNU sources[39].

[32] `osfmk/kdp/kdp.c#L900` [XNU]
[33] `osfmk/kdp/ml/x86_64/kdp_machdep.c#L240` [XNU]
[34] `source/Plugins/Process/MacOSX-Kernel/ProcessKDP.cpp#L699` [LLDB]
[35] DeVille, *Kernel debugging with LLDB and VMware Fusion*.
[36] Apple. *Technical Q&A QA1264: Generating a Non-Maskable Interrupt (NMI)*. URL: https://developer.apple.com/library/archive/qa/qa1264/_index.html.
[37] Apple, *Kernel Programming Guide*.
[38] Cutlip, *Source Level Debugging the XNU Kernel*.
[39] `osfmk/kdp/kdp_udp.c#L1346` [XNU]

Lastly, a significant obstacle to the efficacy of the debugging process is the absence of lldbmacros for most macOS releases, being part of the KDK which are released sporadically, as mentioned in section 3.2.

3.6 Other debugging options

Mentioned for completeness, at least two other methods for kernel debugging have been supported at some point in several XNU releases: the DDB debugger and the kmem device file. Unfortunately, these do not constitute neither an alternative nor a supplement to KDP debugging, since DDB has been removed from kernel sources since a few releases and kmem only offers access to kernel memory (in addition to be somewhat deprecated). A third debugging option is the GDB stub implemented in VMware Fusion, which completely bypasses KDP by moving the debugging process to the hypervisor level; this approach is explored further in the next chapter.

3.6.1 DDB

The archived Apple's documentation[40] suggests to use the DDB debugger (or its predecessor, KDB), built entirely into the kernel and to be interacted with locally through a hardware serial line, when debugging remotely via KDP is not possible or problematic, e.g. when analysing hardware interrupt handlers or before the network hardware is initialised. DDB first appeared as a facility of the Mach kernel (of which XNU is a derivative[41]) developed at Carnegie Mellon University in the nineties, and apparently can still be found in most descendants of the BSD operating system[42]. On macOS, enabling DDB required 'building a custom kernel using the DEBUG configuration.'[43] Support for this debugger seems however to have been dropped after XNU 1699.26.8[44], given that the directory `osfmk/ddb/` containing all related files was removed in the next release; nevertheless, some references to DDB and KDB are still present in XNU sources, such as the bitmask `DB_KDB` for the debug boot-arg[45].

3.6.2 kmem

The README of the Kernel Debug Kit for macOS 10.7.3 Lion build 11D50, among others, alludes to the possibility of using the device file `/dev/kmem` for limited self-debugging:

[40]Apple, *Kernel Programming Guide*.
[41]Singh, *Mac OS X internals: a systems approach*.
[42]*On-Line Kernel Debugging Using DDB*. URL: https://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/kerneldebug-online-ddb.html; *ddb(4) - OpenBSD manual pages*. URL: https://man.openbsd.org/ddb; *ddb(4) - NetBSD Manual Pages*. URL: https://netbsd.gw.com/cgi-bin/man-cgi?ddb+4+NetBSD-current.
[43]Apple, *Kernel Programming Guide*.
[44]Apple. *XNU 1699.26.8 Source*. URL: https://opensource.apple.com/source/xnu/xnu-1699.26.8/.
[45]osfmk/kern/debug.h#L424 [XNU]

> Live (single-machine) kernel debugging was introduced in Mac OS X Leopard. This allows limited introspection of the kernel on a currently-running system. This works using the normal kernel and the symbols in this Kernel Debug Kit by specifying `kmem=1` in your boot-args; the DEBUG kernel is not required.

This method still works in recent macOS releases provided that System Integrity Protection is disabled, but newer KDKs do not mention it anymore, and a note from Apple's docs[46] says that support for kmem will be removed entirely in a unspecified future.

3.6.3 GDB stub in VMware Fusion

VMware Fusion is a type-2 hypervisor for Mac and macOS developed by VMware[47]. Among other features, this software implements and exposes a GDB remote stub, allowing any external debugger implementing the GDB remote serial protocol (e.g. GDB itself or LLDB with the `gdb-remote` command) to debug running virtual machines through virtual machine introspection (see section 2.2.1), no matter the guest OS. The process is represented in fig. 3.3. In the case of macOS, VMware Fusion makes then possible debugging XNU without relying on KDP, eliminating many of the restrictions that it comports; for instance, the GDB stub has no problems with interrupting the execution of the kernel at any time. While being a very solid alternative to KDP, this solution is not without its drawbacks:

- VMware Fusion is not free.
- Using the GDB protocol, notoriously slow[48] because of the high amount of data exchanged between GDB and its stub, makes debugging difficult when trying to analyse race conditions or when breakpoints are hit very frequently, in which case the machine is often slowed down to the point that debugging is impossible.

Multiple guides exist on the Internet explaining how to set up VMware Fusion for macOS debugging[49].

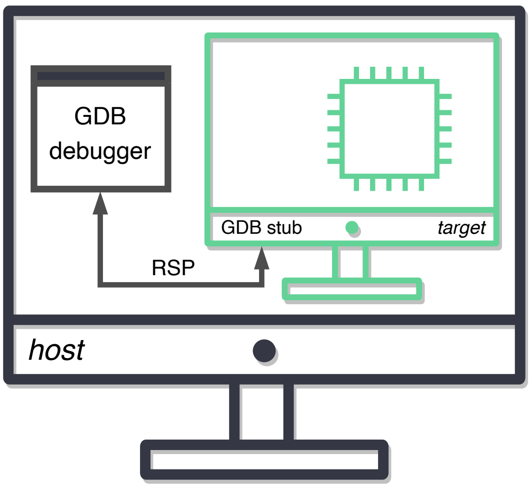
[46]Apple. *Kernel Programming Guide. Security Considerations*. URL: https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/security/security.html.

[47]*VMware*. URL: https://www.vmware.com.

[48]Nicolas Couffin. *Winbagility: Débogage furtif et introspection de machine virtuelle*. 2016; Gene Sally. *Pro Linux embedded systems*. 2010; *Poor performance in visual mode during remote debugging via gdbserver*. URL: https://github.com/radareorg/radare2/issues/3808; *Why is debugging of native shared libraries used by Android apps slow?* URL: https://stackoverflow.com/questions/8051458/why-is-debugging-of-native-shared-libraries-used-by-android-apps-slow.

[49]Cutlip, *Source Level Debugging the XNU Kernel*; snare. *VMware debugging II: "Hardware" debugging*. URL: http://ho.ax/posts/2012/02/vmware-hardware-debugging/; Damien DeVille. *Using the VMware Fusion GDB stub for kernel debugging with LLDB*. URL: http://ddeville.me/2015/08/using-the-vmware-fusion-gdb-stub-for-kernel-debugging-with-lldb.

Figure 3.3: Debugging macOS running on a virtual machine with the GDB stub in VMware Fusion. The KDP stub is not running, since debugging occurs at the hypervisor level.



Chapter 4

LLDBagility: practical macOS kernel debugging

As explained in the previous chapter, macOS allows remote kernel debugging with the Kernel Debugging Protocol, whose current implementation suffers from notable drawbacks that make debugging less effective and not very practical. The most valid alternative to KDP is the GDB stub in VMware Fusion, which, while certainly offering many improvements over the former, it's also not without limitations. This chapter presents LLDBagility, a new tool for debugging the macOS kernel based on virtual machine introspection. First, LLDBagility is introduced with an overview of its features; then, the third-party Fast Debugging Protocol API is briefly described; next, the general architecture of LLDBagility is explained, and in particular how it connects transparently LLDB to macOS virtual machines; next, a few implementation details are illustrated, together with some of the challenges that have been solved while developing the tool; lastly, a solution for using lldbmacros with kernels lacking debug information is proposed.

4.1 Overview

In one sentence, LLDBagility is a software tool that allows LLDB to debug any macOS VirtualBox virtual machine, by replacing the Kernel Debugging Protocol from XNU with analogous VMI capabilities offered at hypervisor level by the Fast Debugging Protocol.

LLDBagility has been developed as part of an internship at Quarkslab[1]. LLDBagility 1.0.0[2] was released open-source in June 2019, alongside two accompanying posts[3]

[1] *Quarkslab*. URL: https://www.quarkslab.com.

[2] Francesco Cagnin. *LLDBagility 1.0.0 Source*. URL: https://github.com/quarkslab/LLDBagility/tree/v1.0.0.

[3] Francesco Cagnin. *An overview of macOS kernel debugging*. URL: https://blog.quarkslab.com/an-overview-of-macos-kernel-debugging.html; Francesco Cagnin. *LLDBagility: practical macOS kernel debugging*. URL: https://blog.quarkslab.com/lldbagility-practical-macos-kernel-debugging.html.

in Quarkslab's blog which constituted the basis for this work; since then the tool has been updated with minor fixes, to be released as version 1.1.0 in the next future.

4.1.1 Motivation

LLDBagility has been developed to improve the current state of kernel debugging on macOS, given that all the other available options for the task, specifically KDP and the GDB stub in VMware Fusion, have minor and major downsides that make the debugging process less practical and effective than it could be. The initial incentives for developing LLDBagility have been the longing for hardware breakpoints (not available in KDP) and for reducing the long reset time after each crash of the debugged macOS.

4.1.2 Features

To the end user, LLDBagility is a set of new LLDB commands for attaching to and debugging an instance of macOS running in a VirtualBox virtual machine:

- `fdp-attach`, to attach LLDB to the VM and start debugging the kernel.
- `fdp-interrupt`, to pause the execution of the VM and let the debugger take control.
- `fdp-hbreakpoint`, to set hardware breakpoints, either on instructions or on data.
- `fdp-save` and `fdp-restore`, to save and restore the state of the VM directly from the debugger.

These commands are intended to be used alongside the ones already available in LLDB, like `register read`, `memory write`, `breakpoint set` (for software breakpoints), `step` and all the others. Furthermore, in case the Kernel Debug Kit of the debugged kernel is available for use (and possibly even when it isn't, as discussed in section 4.5), the vast majority of lldbmacros also work as expected when loaded in the debugger.

4.1.3 Requisites

The major requisites for debugging with LLDBagility are:

- A recent version of macOS as host operating system, accompanied by any recent release[4] of the LLDB debugger (installed for example as part of the Command Line Tools).
- A VirtualBox build with FDP support, compiled using the provided patches for VirtualBox 5.2.14 or 6.0.8 sources for macOS hosts.
- A VirtualBox VM to debug, running any non-ancient version of macOS.

[4]The latest LLDB releases require Python 3, which will be supported in LLDBagility 1.1.0.

4.1.4 License

LLDBagility has been licensed under the Apache License version 2.0[5], a permissive license which allows using, modifying, and distributing the software for any purpose while requiring the preservation of the license notice and the documentation of changes to the original code.

4.2 The Fast Debugging Protocol

The Fast Debugging Protocol is a third-party API for virtual machine introspection and debugging, developed as part of the Winbagility project[6], LLDBagility's older counterpart for the WinDbg debugger and non-DEBUG Windows x86-64 systems. Currently released only as a patch for VirtualBox, FDP equips the hypervisor with a debugging interface that enables external programs to introspect the state of any running virtual machine, allowing to:

- Read and modify its CPU registers.
- Read and modify its virtual and physical memory.
- Set and unset hardware and software breakpoints.
- Pause, resume and single-step its execution.
- Save a snapshot of its state and restore it at a later time.

The two major strengths of FDP are:

- Stealthiness, result of the manipulation of the Extended Page Tables (EPT) for implementing 'hyper' breakpoints. This novel class of breakpoints was invented by FDP itself to get round PatchGuard[7], a Windows feature of non-DEBUG systems that protects certain kernel data structures from modifications, for example causing the system to bug check when trying to patch kernel code. The following is a simplified discussion of software hyper breakpoints in FDP. When a breakpoint has to be installed at address A on the guest physical page G, first the corresponding host physical page H is copied to H2. Second, the instruction at address A in H2 is replaced with `HLT`. Third, page H is marked as readable and writable, while H2 as executable-only. Lastly, the EPT entry for G is updated to point to H2. The execution of the VM is then resumed, and two situations may occur:
  - If the guest executes code in G, then H2 will be used, and when eventually `HLT` is reached, FDP will handle the generated exception by pausing the execution of the machine and updating H2 with the original instruction that was overwritten by the breakpoint.
  - If instead the guest tries to read or write on G, an EPT violation exception will be raised (since H2 is executable-only); in this case, FDP will handle the exception by updating the EPT entry to point to the original

[5] *Apache License, Version 2.0*. URL: https://www.apache.org/licenses/LICENSE-2.0.
[6] Nicolas Couffin. *Winbagility: Débogage furtif et introspection de machine virtuelle*. 2016.
[7] *Driver x64 Restrictions*. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/driver-x64-restrictions.

page H and then resuming execution, without the guest noticing. The EPT will be updated again to H2 on the next EPT violation exception, this time caused by the guest trying to execute code from H which is not marked as executable.

More details and discussion of hard hyper breakpoints and page hyper breakpoints can be found in the Winbagility article[8].

- Efficiency, thanks to the use of shared memory for communications and data exchange between the FDP server and client, thus avoiding system calls for simple reading or writing; as examples, in some tests the API was able to single-step the execution of a VM around 105 000 times per second, and to complete ca. 600 000 reads of virtual memory pages of 4096 bytes per second (ca. 800 000 reads for physical memory pages)[9].

As disadvantages, the current FDP implementation is limited in debugging only virtual machines running on a single CPU core, due to unsolved possible race conditions in accessing and modifying the EPT in the multi-core context; but this constraint should not represent a problem for the vast majority of users. Moreover, at present FDP has been released only as a patch for VirtualBox, which has been preferred to other hypervisors because both open-source (unlike VMware) and cross-platform (unlike KVM[10] and Xen[11]); the API can anyway be ported to other hypervisors by implementing in their sources the necessary FDP subroutines, similarly to writing a GDB remote stub.

Listing 4.1: The FDP API, as defined in FDPutils/FDP/include/FDP.h [LLDBagility]

```
    // FDP API
FDP_EXPORTED    FDP_SHM*    FDP_CreateSHM(char *shmName);
FDP_EXPORTED    FDP_SHM*    FDP_OpenSHM(const char *pShmName);
FDP_EXPORTED    bool        FDP_Init(FDP_SHM *pShm);
FDP_EXPORTED    bool        FDP_Pause(FDP_SHM *pShm);
FDP_EXPORTED    bool        FDP_Resume(FDP_SHM *pShm);
FDP_EXPORTED    bool        FDP_ReadPhysicalMemory(FDP_SHM *pShm, uint8_t
     ↪ *pDstBuffer, uint32_t ReadSize, uint64_t PhysicalAddress);
FDP_EXPORTED    bool        FDP_WritePhysicalMemory(FDP_SHM *pShm, uint8_t
     ↪ *pSrcBuffer, uint32_t WriteSize, uint64_t PhysicalAddress);
FDP_EXPORTED    bool        FDP_ReadVirtualMemory(FDP_SHM *pShm, uint32_t CpuId,
     ↪ uint8_t *pDstBuffer, uint32_t ReadSize, uint64_t VirtualAddress);
FDP_EXPORTED    bool        FDP_WriteVirtualMemory(FDP_SHM *pShm, uint32_t CpuId,
     ↪ uint8_t *pSrcBuffer, uint32_t WriteSize, uint64_t VirtualAddress);
FDP_EXPORTED    uint64_t    FDP_SearchPhysicalMemory(FDP_SHM *pShm, const void
     ↪ *pPatternData, uint32_t PatternSize, uint64_t StartOffset);
FDP_EXPORTED    bool        FDP_SearchVirtualMemory(FDP_SHM *pFDP, uint32_t
     ↪ CpuId, const void *pPatternData, uint32_t PatternSize, uint64_t
     ↪ StartOffset);
FDP_EXPORTED    bool        FDP_ReadRegister(FDP_SHM *pShm, uint32_t CpuId,
     ↪ FDP_Register RegisterId, uint64_t *pRegisterValue);
FDP_EXPORTED    bool        FDP_WriteRegister(FDP_SHM *pShm, uint32_t CpuId,
     ↪ FDP_Register RegisterId, uint64_t RegisterValue);
FDP_EXPORTED    bool        FDP_ReadMsr(FDP_SHM *pShm, uint32_t CpuId, uint64_t
     ↪ MsrId, uint64_t *pMsrValue);
```

[8] Couffin, *Winbagility: Débogage furtif et introspection de machine virtuelle*.
[9] Couffin, *Winbagility: Débogage furtif et introspection de machine virtuelle*.
[10] *KVM*. URL: https://www.linux-kvm.org/page/Main_Page.
[11] *Xen Project*. URL: https://xenproject.org.

```
FDP_EXPORTED    bool        FDP_WriteMsr(FDP_SHM *pShm, uint32_t CpuId, uint64_t MsrId, uint64_t MsrValue);
FDP_EXPORTED    int         FDP_SetBreakpoint(FDP_SHM *pShm, uint32_t CpuId, FDP_BreakpointType BreakpointType, uint8_t BreakpointId, FDP_Access BreakpointAccessType, FDP_AddressType BreakpointAddressType, uint64_t BreakpointAddress, uint64_t BreakpointLength, uint64_t BreakpointCr3);
FDP_EXPORTED    bool        FDP_UnsetBreakpoint(FDP_SHM *pShm, uint8_t BreakpointId);
FDP_EXPORTED    bool        FDP_VirtualToPhysical(FDP_SHM *pShm, uint32_t CpuId, uint64_t VirtualAddress, uint64_t *pPhysicalAddress);
FDP_EXPORTED    bool        FDP_GetState(FDP_SHM *pShm, FDP_State *pState);
FDP_EXPORTED    bool        FDP_GetFxState64(FDP_SHM *pShm, uint32_t CpuId, FDP_XSAVE_FORMAT64_T *pFxState64);
FDP_EXPORTED    bool        FDP_SetFxState64(FDP_SHM *pFDP, uint32_t CpuId, FDP_XSAVE_FORMAT64_T* pFxState64);
FDP_EXPORTED    bool        FDP_SingleStep(FDP_SHM *pShm, uint32_t CpuId);
FDP_EXPORTED    bool        FDP_GetPhysicalMemorySize(FDP_SHM *pShm, uint64_t *pPhysicalMemorySize);
FDP_EXPORTED    bool        FDP_GetCpuCount(FDP_SHM *pShm, uint32_t *pCPUCount);
FDP_EXPORTED    bool        FDP_GetCpuState(FDP_SHM *pShm, uint32_t CpuId, FDP_State *pState);
FDP_EXPORTED    bool        FDP_Reboot(FDP_SHM *pShm);
FDP_EXPORTED    bool        FDP_Save(FDP_SHM *pShm);
FDP_EXPORTED    bool        FDP_Restore(FDP_SHM *pShm);
FDP_EXPORTED    bool        FDP_GetStateChanged(FDP_SHM *pShm);
FDP_EXPORTED    void        FDP_SetStateChanged(FDP_SHM *pShm);
FDP_EXPORTED    bool        FDP_InjectInterrupt(FDP_SHM *pShm, uint32_t CpuId, uint32_t uInterruptionCode, uint32_t uErrorCode, uint64_t Cr2Value);

FDP_EXPORTED    bool        FDP_SetFDPServer(FDP_SHM* pFDP, FDP_SERVER_INTERFACE_T* pFDPServer);
FDP_EXPORTED    bool        FDP_ServerLoop(FDP_SHM* pFDP);
```

4.2.1 PyFDP

In addition to the low-level C interface, FDP also provides Python bindings, useful especially for quick prototyping and proof of concepts (PoCs). Listing 4.2 shows an example PyFDP script in which a hardware breakpoint is installed to pause the selected VM every time a system call is executed.

Listing 4.2: An example PyFDP script

```
#!/usr/bin/env python3
from PyFDP.FDP import FDP

# Connect to the VM named "macOS-18E226"
fdp = FDP("macOS-18E226")

# Pause the VM execution to set a breakpoint
fdp.Pause()
# Set a hardware breakpoint manually by modifying x86 debug registers
UnixSyscall64 = 0xffffff80115bae84
fdp.dr0 = UnixSyscall64
fdp.dr7 = 0x403
# Resume the VM execution
fdp.Resume()

while True:
```

```
    # Wait for a breakpoint hit
    fdp.WaitForStateChanged()
    # Handle the interruption as desired
    print("0x{:016x}".format(fdp.rax))
    # Jump over the breakpoint
    fdp.SingleStep()
    # Continue the VM execution
    fdp.Resume()
```

4.3 Architecture

As presented at length in chapter 3, in KDP debugging LLDB interacts with the macOS kernel by sending commands to its internal KDP stub, which, being itself part of the kernel, is then able to inspect and alter the state of the machine as requested. Comparing KDP and FDP, all the important features of the former are also provided by the latter, since both protocols allow for virtual and physical memory access, CPU registers access, breakpoints installation and removal, CPU suspension and resuming; the kernel debugging stub can be replaced entirely by an equivalent alternative external to the debugged machine. LLDBagility is then a bridge between LLDB and FDP, translating requests from the debugger into equivalent FDP calls. By maintaining compatibility with the KDP protocol, LLDBagility can also take advantage of LLDB's existing support for the macOS kernel without modifying the debugger in any aspect.

LLDBagility is conceptually organised in three layers:

- The FDP layer, third-party code for hypervisor-level debugging of generic VMs in VirtualBox.
- The core layer, implementing the logic that translates the KDP requests generated by LLDB into corresponding FDP requests for the lower layer.
- The LLDB layer, implementing the set of new LLDB commands, which interact directly with the core layer.

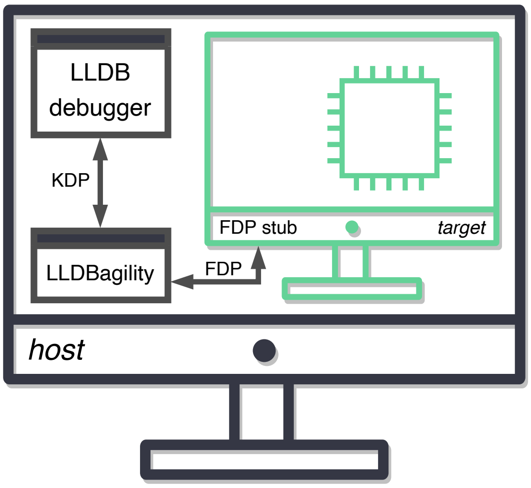
4.4 Implementation

LLDBagility has been developed almost entirely using the Python programming language; the choice was dictated for the language versatility and ease of use and also because FDP and LLDB can be scripted using this language.

The core part of the tool comprises:

- `KDPutils/` [LLDBagility], a Python package that reimplements the communication protocol used by KDP.
- `LLDBagility/kdpserver.py` [LLDBagility], which implements the KDPServer class as a replacement for the KDP server found in XNU.
- `LLDBagility/stubvm.py` [LLDBagility], implementing the STUBVM class for performing simple and complex introspection operations on the debugged VM.

Figure 4.1: Debugging macOS running on a virtual machine with LLDBagility. The debugger, led to believe it's communicating with XNU, sends instead the KDP requests to LLDBagility's KDP server. The requests are then translated into corresponding FDP calls to inspect or modify the state of the target machine accordingly. Results are encoded as KDP replies and sent back to the debugger.



- `LLDBagility/lldbagility.py` [LLDBagility], a Python script for LLDB implementing the new fdp- commands.

The next sections discuss each of these components in more detail.

4.4.1 The KDPutils package

As discussed in section 3.1, communications between LLDB and XNU in KDP debugging are carried out using a custom UDP protocol, and for compatibility reasons LLDBagility interfaces with the debugger using the same protocol. The KDPutils package implements all the necessary helper routines to craft valid KDP request and reply messages and send them over the network, abstracting away all the low-level details related to the correct assembly and disassembly of network packets.

Listing 4.3: `kdp_connect()` from `KDPutils/kdputils/requests.py` [LLDBagility]

```
def kdp_connect(req_reply_port, exc_note_port, greeting):
    return dict(
        is_reply=0x0,
        request=protocol.KDPRequest.KDP_CONNECT,
        seq=-1,
        len=-1,
        key=-1,
        req_reply_port=req_reply_port,
        exc_note_port=exc_note_port,
        greeting=greeting,
    )
```

As a usage example, the package also implements a KDPClient class (see listing D.1) that connects to the specified KDP server (e.g. XNU's or LLDBagility's) and retrieves the kernel version string by sending `KDP_REATTACH`, `KDP_CONNECT` and `KDP_KERNELVERSION` requests.

4.4.2 The KDPServer class

As the name suggests, the KDPServer class (see listing D.2) reimplements the functionalities of XNU's KDP server, to which LLDB connects for debugging the kernel. The main task of KDPServer is to wait for messages from clients, parse and execute the requests, and if needed send back proper responses; this behaviour is implemented in the debug() method (shown in listing 4.4), which takes as argument a STUBVM object representing the VM being debugged.

Listing 4.4: The debug loop of the KDPServer class, as defined in `LLDBagility/kdpserver.py` [LLDBagility]

```
    def debug(self, vm):
        # it is implicitly assumed the first two KDP requests received are
        # KDP_REATTACH and KDP_CONNECT (this is always true when LLDB connects)
        while self._continue_debug_loop:
            time.sleep(0.003)

            try:
                # receive a request
                reqpkt, cl_addr = kdputils.protocol.recv(self.sv_sock)
            except socket.error:
                pass
            else:
                # process the request
                replypkt = self._process(vm, reqpkt, cl_addr)
                if replypkt:
                    # send the response
                    cl_addr = (self._cl_host, self._cl_reply_port)
                    kdputils.protocol.send(
                        self.sv_sock, cl_addr, replypkt, reqpkt["seq"],
    ↪ reqpkt["key"]
                    )

            if self._cl_connected and vm.is_state_changed():
                _, exception = vm.state()
                if exception:
                    (exception, code, subcode) = exception
                    reqpkt = kdputils.requests.kdp_exception(
                        n_exc_info=0x1,
                        cpu=0x0,
                        exception=exception,
                        code=code,
                        subcode=subcode,
                    )
                    cl_addr = (self._cl_host, self._cl_exception_port)
                    kdputils.protocol.send(
                        self.sv_sock,
                        cl_addr,
                        reqpkt,
                        next(self._cl_exception_seq),
                        self._cl_session_key,
                    )
```

Every KDP request received is processed in the `_process()` method (partly shown in listing 4.5), where the state of the virtual machine is inspected or modified according to the request; for example, upon receiving a KDP_CONNECT, the VM is paused and all of its existing breakpoints are removed. For all requests, an appropriate KDP reply is generated and sent back to requester, using the helper routines of the KDPutils package.

Listing 4.5: Excerpt of _process() from LLDBagility/kdpserver.py [LLDBagility]

```
    def _process(self, vm, reqpkt, cl_addr):
        if reqpkt["request"] == KDPRequest.KDP_CONNECT:
            assert self._cl_host and self._cl_reply_port
            self._cl_exception_port = reqpkt["exc_note_port"]
            self._cl_exception_seq = itertools.cycle(range(256))
            self._cl_connected = True
            vm.halt()
            vm.unset_all_breakpoints()
            replypkt = kdputils.replies.kdp_connect(KDPError.KDPERR_NO_ERROR)

        elif reqpkt["request"] == KDPRequest.KDP_DISCONNECT:
```

In addition to processing incoming requests, KDPServer also notifies the connected KDP client when exceptions occur in the debugged VM by sending KDP_EXCEPTION messages.

4.4.3 The STUBVM class

The STUBVM class implements all the methods for interacting with debugged VM, such as:

- get_kernel_version()
- read_register() and write_register()
- read_msr64() and write_msr64()
- read_virtual_memory() and write_virtual_memory()
- set_soft_exec_breakpoint() and set_hard_breakpoint()
- halt(), interrupt(), single_step() and resume()
- interrupt_and_take_snapshot() and interrupt_and_restore_last_snapshot()

A simple example of the additional logic that had to be implemented on top of the basic FDP calls is given by the write_register() method (shown in listing 4.6), which must forbid writes to the FLAGS register (explained later in section 4.4.5). A more complex example is given by the read_virtual_memory() method, which addresses two special cases: attempts to read kernel virtual memory from user space and attempts to read the kdp struct (both also explained in section 4.4.5).

Listing 4.6: write_register() from LLDBagility/stubvm.py [LLDBagility]

```
    @lldbagilityutils.indented(logger)
    @lldbagilityutils.synchronized
    def write_register(self, reg, val):
        logger.debug("write_register(reg='{}', val=0x{:x})".format(reg, val))
        if reg == "rflags":
```

```
            if val & EFL_TF:
                logger.debug(">  singlestep_at_next_resume")
                self._singlestep_at_next_resume = True
            # disallow changes to RFLAGS
            return
        setattr(self.stub, reg, val)
```

Originally, STUBVM was developed to be used only in combination with FDP, but later the implementation evolved to allow for different APIs; for example, the provided `LLDBagility/VMSN.py` [LLDBagility] PoC enables LLDBagility to work with a snapshot from a VMware virtual machine, and similarly `LLDBagility/PMEM.py` [LLDBagility] with MacPmem[12]. However, discussion of this feature is omitted since is still a work in progress.

4.4.4 The fdp- commands

By interfacing with the lldb Python module, LLDBagility implements five new LLDB commands for debugging a macOS VM, namely `fdp-attach`, `fdp-save`, `fdp-restore`, `fdp-interrupt`, and `fdp-hbreakpoint`. All commands interact with the debugged VM using a global STUBVM object recreated at each execution of `fdp-attach`. The process of attaching to a VM via the FDP stub, shown in listing 4.7, involves:

- Creating a new STUBVM object, which is associated to a PyFDP client for interacting with the VM.
- Continuing the execution of the VM until it reaches kernel code (as explained later in section 4.4.5).
- Detaching LLDB from any other debugged process, if any, and deleting all existing breakpoints.
- Starting a new KDPServer instance in a background thread.
- Using the LLDB command `kdp-remote`, connecting the debugger to the KDP server just started, thus kicking off the debugging process.

Listing 4.7: `_attach()` from `LLDBagility/lldbagility.py` [LLDBagility]

```
def _attach(debugger, exe_ctx, vm_stub, vm_name):
    global vm
    print(lldbagilityutils.LLDBAGILITY)

    print("* Attaching to the VM")
    try:
        vm = stubvm.STUBVM(vm_stub, vm_name)
    except Exception as exc:
        print("* Could not attach! {}".format(str(exc)))
        return

    print("* Resuming the VM execution until reaching kernel code")
    vm.complete_attach()
    print("* Kernel load address: 0x{:016x}".format(vm.kernel_load_vaddr))
    print("* Kernel slide:        0x{:x}".format(vm.kernel_slide))
```

[12] *MacPmem - OS X Physical Memory Access*. URL: https://github.com/google/rekall/tree/master/tools/osx/MacPmem.

```python
    print("* Kernel cr3:          0x{:x}".format(vm.kernel_cr3))
    print("* Kernel version:      {}".format(vm.kernel_version))
    print("* VM breakpoints deleted")

    # detach the previous process (if any)
    exe_ctx.process.Detach()

    # remove all LLDB breakpoints
    exe_ctx.target.DeleteAllBreakpoints()
    print("* LLDB breakpoints deleted")

    # start the fake KDP server
    kdpsv = kdpserver.KDPServer()
    th = threading.Thread(target=kdpsv.debug, args=(vm,))
    th.daemon = True
    th.start()

    # connect LLDB to the fake KDP server
    kdpsv_addr, kdpsv_port = kdpsv.sv_sock.getsockname()
    _exec_cmd(debugger, "kdp-remote '{}:{}'".format(kdpsv_addr, kdpsv_port))

    # trigger a memory write to find out the address of the kdp struct
    vm.store_kdp_at_next_write_virtual_memory()
    if _exec_cmd(debugger, "memory write &kdp 41",
    ↪ capture_output=True).GetError():
        print("* Unable to find the 'kdp' symbol. Did you specify the target to
    ↪ debug?")
        vm.abort_store_kdp_at_next_write_virtual_memory()
```

A few observations about the other fdp- commands:

- The implementation of `fdp-interrupt` simply pauses the VM through FDP; LLDB is informed of the interruption only with the exception notification sent by the KDP server.
- When FDP saves the state of the machine, all installed breakpoints are removed. To improve on this limitation, when `fdp-save` is executed the currently installed hardware and software breakpoints are temporarily backed up and restored just after the FDP snapshot is taken.
- When `fdp-restore` is executed, after restoring the VM state with FDP the machine is then reattached again with `fdp-attach`, so to trigger a new execution of `kdp-remote`. This is required because the kernel load address in the new state may differ from the previous, and without reconnecting to the target LLDB wouldn't be aware of the change.

4.4.5 Some technical challenges

This section discusses the solutions adopted to some of the technical challenges encountered during LLDBagility's development.

Finding the kernel load virtual address

The first step for any meaningful introspection activity on kernel memory is finding where the kernel executable has been loaded. This memory address is not fixed, since for security reasons modern operating systems, macOS included, choose a random base address for the kernel at every boot (address space layout randomisation). To find the kernel load address, LLDBagility distinguishes two cases:

- When debug is included in the boot-args, as mentioned in the previous chapter the 'kernel's load address will be noted in the `lowglo` page at a fixed address.'[13] In this situation, the value of the base address of the kernel is written at the fixed virtual address of `lgStext`, i.e. 0xFFFFFF8000002010[14]. Note that this approach fails if LLDBagility attaches to the kernel before `lgStext` is initialised, in which case the second strategy is used.
- If instead the boot-args do not include debug, then a simple memory scan is performed, by reading the first 4 bytes of every memory page starting from `VM_MIN_KERNEL_ADDRESS`, i.e. 0xFFFFFF8000000000[15]; the search stops at the first address containing the value 0xFEEDFACF, which corresponds to `magic` value of the kernel Mach-O[16].

Listing 4.8: `_search_kernel_load_vaddr()` as defined in `LLDBagility/stubvm.py` [LLDBagility]

```python
    @lldbagilityutils.indented(logger)
    def _search_kernel_load_vaddr(start_vaddr):
        logger.debug(
            "_search_kernel_load_vaddr(start_vaddr=0x{:016x})".format(start_vaddr)
        )
        # try to find the load address manually
        assert _in_kernel_space(start_vaddr)
        vaddr = start_vaddr & ~(I386_PGBYTES - 1)
        while vaddr >= VM_MIN_KERNEL_ADDRESS:
            if _is_kernel_load_vaddr(vaddr):
                return vaddr
            vaddr -= I386_PGBYTES
        else:
            raise AssertionError
```

Listing 4.9: `_is_kernel_load_vaddr()` as defined in `LLDBagility/stubvm.py` [LLDBagility]

```python
    @lldbagilityutils.indented(logger)
    def _is_kernel_load_vaddr(vaddr):
        logger.debug("_is_kernel_load_vaddr()")
        if not _in_kernel_space(vaddr):
            return False
        data = vm.read_virtual_memory(vaddr, 0x4)
        return data and lldbagilityutils.u32(data) == MH_MAGIC_64
```

Pausing the VM execution in kernel space

Since LLDBagility permits to attach to a running virtual machine at any arbitrary moment, it's possible and likely that this operation will occur when the machine is not in kernel space. Being interested in debugging the kernel, when attaching to the VM LLDBagility lets its execution continuing until reaching kernel code.

[13] `osfmk/x86_64/pmap.c#L1171` [XNU]
[14] `osfmk/x86_64/lowglobals.h#L54` [XNU]
[15] `osfmk/mach/i386/vm_param.h#LL198` [XNU]
[16] `EXTERNAL_HEADERS/mach-o/loader.h#L68/#L83` [XNU]

Thanks to FDP, this operation is done heuristically by setting a breakpoint that triggers on writes to the CR3 register, since this privileged control register should be accessed only by the kernel to store the physical memory address of the first page directory entry[17]. An alternative approach would simply be single-stepping the VM execution until the program counter is a kernel virtual address, but on average this is expected to be too slow to be practical.

Reading kernel virtual memory from user space

Loosely speaking, as part of the context switching mechanism to go from kernel to user space, XNU updates the page directory base address, stored in the CR3 register, with the address of the page tables of the new process, which never allow accessing the virtual memory of the kernel. Consequently, reading kernel memory with FDP fails if the VM is currently running in user space. This problem was first noticed when lldbmacros were loaded in LLDB and for any reason the VM was stopped in user space, since every time the control returns to the debugger lldbmacros try to read the kdp struct in kernel memory. To get round the problem, every time FDP fails to read virtual memory, the CR3 register is overwritten temporarily with the page directory base address of the kernel (retrieved and memorised during the attaching process), and the reading is tried one more time; if also this second reading fails, then the virtual address is considered invalid (not mapped). The original value for CR3 is restored before proceeding.

Listing 4.10: `read_virtual_memory()` from `LLDBagility/stubvm.py` [LLDBagility]

```python
    @lldbagilityutils.indented(logger)
    @lldbagilityutils.synchronized
    def read_virtual_memory(self, vaddr, nbytes):
        logger.debug(
            "read_virtual_memory(vaddr=0x{:016x}, nbytes=0x{:x})".format(vaddr, nbytes)
        )
        data = self.stub.ReadVirtualMemory(vaddr, nbytes)

        if not data and not _in_kernel_space(self.read_register("rip")):
            # if reading fails, it could be the case that we are trying to read kernel
            # virtual addresses from user space (e.g. when LLDB stops in user land and
            # the user loads or uses lldbmacros)
            # in this case, we try the read again but using the kernel pmap
            logger.debug(">  using kernel pmap")
            process_cr3 = self.read_register("cr3")
            # switch to kernel pmap
            self.write_register("cr3", self.kernel_cr3)
            # try the read again
            data = self.stub.ReadVirtualMemory(vaddr, nbytes)
            # switch back to the process pmap
            self.write_register("cr3", process_cr3)

        if self._kdp_vaddr and vaddr <= self._kdp_vaddr <= vaddr + nbytes:
            # this request has very likely been generated by LLDBmacros
            logger.debug(">  fake kdp struct")
            assert data is not None
```

[17] `osfmk/i386/i386_init.c#L288` [XNU]

```
            # fill some fields of the empty (since the boot-arg "debug" is
        ↪ probably not set) kdp struct
            saved_state = lldbagilityutils.p64(NULL)
            kdp_thread = lldbagilityutils.p64(self._get_active_thread_vaddr())
            fake_partial_kdp_struct = b"".join((saved_state, kdp_thread))
            kdp_struct_offset = self._kdp_vaddr - vaddr
            data = (
                data[:kdp_struct_offset]
                + fake_partial_kdp_struct
                + data[kdp_struct_offset + len(fake_partial_kdp_struct) :]
            )

        data = data if data else b""
        logger.debug(">␣␣len(data):␣0x{:x}".format(len(data)))
        return data
```

Populating the kdp struct

To retrieve information about the debugging session, lldbmacros rely on some fields of the kdp struct[18]. This struct is populated by the kernel only when the KDP stub is enabled[19], and this is usually not the case when using LLDBagility. The problem is obviated by hooking the memory reads from the debugger to area occupied by the uninitialised struct, and returning instead a patched memory chunk in which all the necessary fields (e.g. `kdp_thread`) are filled with proper values. Note that the location of the KDP struct in memory is known since RELEASE kernels seem to always contain its symbol.

Finding the virtual address of the active thread

One of the fields of the kdp struct that needs to be populated is `kdp_thread`, containing the virtual address of the active thread. This can be found in the `cpu_data` structure[20], whose base address is normally written in the model specific register `MSR_IA32_GS_BASE`; but when the execution switches from kernel space to user space, the content of this register is swapped with the content of `MSR_IA32_KERNEL_GS_BASE`. Hence, when the VM is running in user space the address of `cpu_data` must be retrieved from the second register.

Returning an incremented program counter at breakpoint hits

According to the LLDB sources[21], when a breakpoint is hit KDP doesn't decrement the program counter, so the debugger does this operation by itself. On the opposite side, when a breakpoint fires in FDP, the program counter is decremented so to contain the address that generated the trap. Thus, in order for LLDB to register the breakpoint correctly (e.g. incrementing the hit count and executing callbacks), after raising a KDP exception to communicate the breakpoint trap to the debugger LLDBagility must then make sure that the next read of the program counter (as-

[18] `tools/lldbmacros/core/operating_system.py#L777` [XNU]
[19] `osfmk/kdp/kdp_udp.c#L1349` [XNU]
[20] `osfmk/i386/cpu_data.h#L150` [XNU]
[21] `source/Plugins/Process/MacOSX-Kernel/ThreadKDP.cpp#LL157` [LLDB]

sumed to be generated by LLDB in response to the exception) returns the value read with FDP incremented by one.

Listing 4.11: `state()` from `LLDBagility/stubvm.py` [LLDBagility]

```python
    @lldbagilityutils.indented(logger)
    @lldbagilityutils.synchronized
    def state(self):
        logger.debug("state()")
        if self.is_breakpoint_hit():
            logger.debug(">  state breakpoint hit")
            self._exception = (EXC_BREAKPOINT, EXC_I386_BPTFLT, 0x0)
            # the following assumes that the next call to STUBVM.read_register("rip")
            # will be made by LLDB in response to this EXC_BREAKPOINT exception
            self._return_incremented_at_next_read_register_rip = True
        state = (self.stub.GetState(), self._exception)
        self._exception = None
        return state
```

Hooking changes to FLAGS for single-stepping

In KDP debugging, to single-step the kernel LLDB simply sets the `TF` flag of the `FLAGS` register and then resumes the execution, which will automatically stop after one instruction with a type-1 interrupt. With FDP, single-stepping can be done only through the provided API method. So, when the debugger tries to set the `TF` flag, LLDBagility acknowledges the change but does not commit it, and at the next debugger request for resuming execution LLDBagility will then just single-step through FDP and send a stopping notification to LLDB.

Implementing hardware breakpoints

The x86 architecture provides hardware breakpoint capabilities through dedicated registers: `DR0` to `DR3` are used to specify the addresses to break at, while `DR7` is used to control the breaking conditions. In particular, the low-order eight bits of `DR7` selectively enable the four breakpoints, either locally for the current process or globally; higher bits define instead whether breakpoints should break on execution, data write, data read or write. Contrary to KDP, FDP permits these registers to be set, and implementing hardware breakpoints is then straightforward.

Listing 4.12: `set_hard_breakpoint()` from `LLDBagility/stubvm.py`

```python
    @lldbagilityutils.indented(logger)
    @lldbagilityutils.synchronized
    def set_hard_breakpoint(self, trigger, nreg, vaddr):
        logger.debug(
            "set_hard_exec_breakpoint(trigger='{}', nreg=0x{:016x}, vaddr=0x{:016x})".format(
                trigger, nreg, vaddr
            )
        )
        assert self.is_state_halted()
        assert trigger in ("e", "w", "rw")
        assert 0 <= nreg <= 3
        trigger_bitshifts = {nreg: 16 + nreg * 4 for nreg in range(4)}
        status_bitshifts = {nreg: nreg * 2 for nreg in range(4)}
```

```python

        ctrl_mask = self.read_register("dr7")
        # reset trigger entry for the chosen register to 0b00
        ctrl_mask &= ~(0b11 << trigger_bitshifts[nreg])
        # set new entry
        if trigger == "e":
            trigger_entry = 0b00
        elif trigger == "w":
            trigger_entry = 0b01
        elif trigger == "rw":
            trigger_entry = 0b11
        else:
            raise NotImplementedError
        ctrl_mask |= trigger_entry << trigger_bitshifts[nreg]
        # enable breakpoint globally
        ctrl_mask |= 0b10 << status_bitshifts[nreg]
        logger.debug(">␣␣ctrl_mask:␣0b{:032b}".format(ctrl_mask))

        self.write_register("dr{}".format(nreg), vaddr)
        self.write_register("dr7", ctrl_mask)
```

4.5 Using lldbmacros with kernels lacking debug information

As noted in section 3.2, Apple does not publish Kernel Debug Kits for many macOS releases, and this absence makes kernel debugging more difficult since the process cannot benefit from full symbolic information and consequently from lldbmacros. To alleviate the problem, this section explores some ideas about reusing debug information and lldbmacros from any released KDK for debugging a macOS kernel lacking its own debug kit. The proposed solution is relatively limited in power but it has also been proven useful.

To do introspection, lldbmacros rely on the debug information contained in the DWARF companion file of the debugged kernel; for instance, the `showversion` macro[22] requires locating the global `version` string in memory.

Listing 4.13: The `version` global string, as defined in `config/version.c` [XNU]

```c
const char version[] = OSTYPE "␣Kernel␣Version␣###KERNEL_VERSION_LONG###:␣###KERNEL_BUILD_DATE###;␣###KERNEL_BUILDER###:###KERNEL_BUILD_OBJROOT###";
```

Assume to have the DWARF file and lldbmacros for kernel build A, but not for build B because its KDK has not been released. The goal is then to extract part of A's debugging information to create a new DWARF file for kernel B that allows loading A's lldbmacros in LLDB for debugging B. This new DWARF must contain the minimum amount of debugging information necessary to execute the desired lldbmacros. The factors suggesting that this could be possible to some extent are:

- The debug information required by the most important macros to work correctly comprises only a relatively small number of global variables and data types.

[22] `tools/lldbmacros/xnu.py#L554` [XNU]

- The definition in XNU sources of these fundamental variables and types does not seem to change much between kernel versions, especially if these are successive releases.
- RELEASE kernels contain partial symbolic information for many symbols used by lldbmacros.

Listing 4.14: Retrieving symbol values with the `/usr/bin/nm` utility

```
$ nm /System/Library/Kernels/kernel | grep -w _version
ffffff8000b0c440 S _version
```

- Being Python scripts, lldbmacros are easy to modify and adapt to different kernels, if needed.

Two cases are now to be distinguished. In the first, all variables and data types that have to be extracted from A's DWARF have coincidentally the exact same definition in kernel B. This is generally not possible to know in advance unless the sources of the two kernels have been published. If this holds true, a DWARF for B can be created by patching a copy of A's in such a way that:

- Its Mach-O UUID matches the UUID of kernel B instead of A's; this simply requires to find the offset of the `LC_UUID` load command[23] in the Mach-O headers of the DWARF file and update the `uuid` field accordingly.
- Its `__TEXT`, `__DATA` and `__LINKEDIT` segments are relocated and resized according to the structure of kernel B instead of A's; again, this simply requires to find the offsets of the three `LC_SEGMENT_64` commands[24] in the Mach-O headers of the DWARF file and update the `vmaddr` and `vmsize` fields with the corresponding values taken from B's executable.
- The `AT_location` field of each symbol used by the macros is updated with the value of same symbol as retrieved from the symbol table of B's executable (assuming all required symbols are present in the table).

In the second case instead, the definition of at least one required data type differs between A and B (for instance, a field is added or removed from a struct). If so, the simple patching described above is no longer sufficient, since it becomes necessary to actually modify data type definitions in the DWARF file, which is difficult due to the complexity of the format. For such circumstances, it is possible to create B's DWARF from scratch:

- First, A's DWARF file is parsed to extract information of all variables and data types used by lldbmacros.
- Then, this information is used to generate C source files containing the definitions of the extracted variables and types. These sources can easily be edited by hand to modify types as desired.
- Lastly, sources are compiled with the debugging switch (e.g. the `-g` option for both Clang[25] and GCC[26]) so to create a companion DWARF file.

[23] `EXTERNAL_HEADERS/mach-o/loader.h#L1140` [XNU]
[24] `EXTERNAL_HEADERS/mach-o/loader.h#L349` [XNU]
[25] *Clang: a C language family frontend for LLVM*. URL: `https://clang.llvm.org`.
[26] *GCC, the GNU Compiler Collection*. URL: `https://gcc.gnu.org`.

Before loading it in LLDB for debugging, the resultant DWARF file must be patched as described above for the first case.

LLDBagility provides several scripts for automatising the processes presented in this section; their usage and the effectiveness of the approach is demonstrated in the next chapter.

4.6 Comparison with the other debugging methods

The approach adopted by LLDBagility, thanks in part to the powerful capabilities of the FDP API, allowed to improve over the limitations of both KDP and the GDB stub in VMware Fusion. In particular:

- Since debugging at the hypervisor level completely bypasses the need for the KDP stub, enabling the debugging capabilities of XNU is not required anymore.
- More generally, since XNU doesn't need to be configured for debugging, the OS can be debugged while running with a default configuration.
- With no need to wait for the KDP stub to initialise, debugging can now start at the first instruction of the kernel (or even before in the `boot.efi` boot loader).
- All the many side effects on the whole system due to KDP debugging are eliminated, and detecting that the OS is being debugged should be more difficult.
- Hardware breakpoints and watchpoints are now available.
- The execution of macOS can be paused (and resumed) at will from the debugger.
- The state of the VM can be saved and restored from the debugger.
- The kernel can be reattached at any time.
- Debugging with FDP is much more reliable than with KDP and presents no timing issues or packet drops.
- FDP is considerably faster than the GDB protocol.
- LLDBagility (and FDP) are free and open-source.

Chapter 5

Case study

This chapter presents an example macOS kernel debugging session carried out with LLDBagility. Many of the outputs included were edited or truncated for presentation.

The demonstration utilises Ned Williamson's PoC for CVE-2019-8605[1], a use-after-free in XNU that opened the way for the 'SockPuppet' kernel exploit for iOS[2]. The PoC, shown in listing 5.1, panics vulnerable macOS kernels by triggering a NULL pointer dereference. The bug has been fixed with the macOS Mojave 10.14.6 Supplemental Update[3].

Listing 5.1: Ned Williamson's PoC for CVE-2019-8605

```
#define IPPROTO_IP 0

#define IN6_ADDR_ANY { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 }
#define IN6_ADDR_LOOPBACK { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1 }

int main() {
    int s = socket(AF_INET6, SOCK_RAW, IPPROTO_IP);
    printf("res0: %d\n", s);
    struct sockaddr_in6 sa1 = {
        .sin6_len = sizeof(struct sockaddr_in6),
        .sin6_family = AF_INET6,
        .sin6_port = 65000,
        .sin6_flowinfo = 3,
        .sin6_addr = IN6_ADDR_LOOPBACK,
        .sin6_scope_id = 0,
    };
    struct sockaddr_in6 sa2 = {
        .sin6_len = sizeof(struct sockaddr_in6),
        .sin6_family = AF_INET6,
        .sin6_port = 65001,
```

[1] *Issue 1806: XNU: Use-after-free due to stale pointer left by in6_pcbdetach*. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=1806.

[2] Ned Williamson. *SockPuppet: A Walkthrough of a Kernel Exploit for iOS 12.4*. URL: https://googleprojectzero.blogspot.com/2019/12/sockpuppet-walkthrough-of-kernel.html.

[3] Apple. *About the security content of macOS Mojave 10.14.6 Supplemental Update*. URL: https://support.apple.com/en-in/HT210548.

```
        .sin6_flowinfo = 3,
        .sin6_addr = IN6_ADDR_ANY,
        .sin6_scope_id = 0,
    };

    int res = connect(s, (const sockaddr*)&sa1, sizeof(sa1));
    printf("res1:␣%d\n", res);

    unsigned char buffer[4] = {};
    res = setsockopt(s, 41, 50, buffer, sizeof(buffer));
    printf("res1.5:␣%d\n", res);

    res = connect(s, (const sockaddr*)&sa2, sizeof(sa2));
    printf("res2:␣%d\n", res);

    close(s);
    printf("done\n");
}
```

The target virtual machine used for the demonstration installed macOS 10.14.3 Mojave build 18D109, chosen both because vulnerable to the PoC and because no KDK was available for it. The host machine run instead macOS 10.15.3 Catalina build 19D76, with LLDB 1001.0.13.3 from Xcode 10.3, VirtualBox 6.0.8 built with the FDP patch, and a development version of LLDBagility between the released 1.0.0 and the future 1.1.0. In preparation for debugging, the RELEASE kernel of the target machine (located at `/System/Library/Kernels/kernel`) was downloaded to the host via the `scp` utility and saved to `/Users/fcagnin/kernel-18D109`, and similarly a copy of the C source code of the PoC was uploaded to the VM to `/Users/tcook/in6_selectsrc.cc`.

5.1 Part 1: Testing the fdp- commands

The debugging session started by firing up the target macOS VM through the VirtualBox GUI. Shortly afterwards, in a terminal window LLDB was started and attached to the machine using the LLDBagility command `fdp-attach`:

```
(lldb) fdp-attach macos-mojave-18D109
LLDBagility
* Attaching to the VM
* Resuming the VM execution until reaching kernel code
* Kernel load address: 0xffffff801de00000
* Kernel slide:        0x1dc00000
* Kernel cr3:          0x6c704000
* Kernel version:      Darwin Kernel Version 18.2.0: Thu Dec 20 20:46:53 PST
     ↪ 2018; root:xnu-4903.241.1~1/RELEASE_X86_64
* VM breakpoints deleted
* LLDB breakpoints deleted
Version: Darwin Kernel Version 18.2.0: Thu Dec 20 20:46:53 PST 2018;
     ↪ root:xnu-4903.241.1~1/RELEASE_X86_64; stext=0xffffff801de00000
Kernel UUID: 1970B070-E53F-3178-83F3-1B95FA340695
Load Address: 0xffffff801de00000
WARNING: Unable to locate kernel binary on the debugger system.
* Unable to find the 'kdp' symbol. Did you specify the target to debug?
Process 1 stopped
* thread #1, stop reason = signal SIGSTOP
```

```
    frame #0: 0xffffff801e0d56b5
->  0xffffff801e0d56b5: retq
    0xffffff801e0d56b6: movq   %cr3, %rcx
    0xffffff801e0d56b9: andq   $-0x1000, %rcx            ; imm = 0xF000
    0xffffff801e0d56c0: movq   (%r8,%r9,8), %rax
Target 0: (No executable module.) stopped.
```

Since LLDB wasn't able to find the kernel executable automatically in the file system (mainly because no KDK for the debugged kernel was installed), LLDBagility warned that the target to debug should be specified manually. This was done using the LLDB command `target create`:

```
(lldb) target create /Users/fcagnin/kernel-18D109
Current executable set to '/Users/fcagnin/kernel-18D109' (x86_64).
```

After specifying the kernel binary, the VM was reattached:

```
(lldb) fdp-attach macos-mojave-18D109
LLDBagility
* Attaching to the VM
* Resuming the VM execution until reaching kernel code
. . .
Load Address: 0xffffff801de00000
Kernel slid 0x1dc00000 in memory.
Loaded kernel file /Users/fcagnin/kernel-18D109
Loading 94 kext modules warning: Can't find binary/dSYM for
     ↪ com.apple.kec.corecrypto (46F3B625-86D1-3761-9603-34835A98AA49)
.warning: Can't find binary/dSYM for com.apple.kec.Libm
     ↪ (2DF6EF8D-C4B1-3754-883E-192ABD2743DB)
.warning: Can't find binary/dSYM for com.apple.kec.pthread
     ↪ (F4714573-8F64-35BD-9C41-5D4BDCBFAD1C)
. . .
.warning: Can't find binary/dSYM for com.apple.driver.AppleHWSensor
     ↪ (88083746-B4CC-38FC-9DB2-81D03592CBD5)
.warning: Can't find binary/dSYM for com.apple.fileutil
     ↪ (5E0468C0-F2DE-37EF-BB2A-0796BA8311B9)
. done.
Process 1 stopped
* thread #1, stop reason = signal SIGSTOP
    frame #0: 0xffffff801e0d56b5 kernel`___lldb_unnamed_symbol992$$kernel + 181
kernel`___lldb_unnamed_symbol992$$kernel:
->  0xffffff801e0d56b5 <+181>: retq
    0xffffff801e0d56b6 <+182>: movq   %cr3, %rcx
    0xffffff801e0d56b9 <+185>: andq   $-0x1000, %rcx            ; imm = 0xF000
    0xffffff801e0d56c0 <+192>: movq   (%r8,%r9,8), %rax
Target 1: (kernel) stopped.
```

At this point, the execution of the VM was resumed to let the system finish booting:

```
(lldb) continue
Process 1 resuming
```

After a few moments, macOS was fully up and running. In a terminal window inside the target VM, the PoC was compiled:

```
Tims-iMac:~ tcook$ clang -o in6_selectsrc in6_selectsrc.cc
Tims-iMac:~ tcook$ file in6_selectsrc
in6_selectsrc: Mach-O 64-bit executable x86_64
```

Before executing the PoC which panics the machine, the VM was paused with `fdp-interrupt` and its state saved in a handful of seconds with `fdp-save`:

```
(lldb) fdp-interrupt
Process 1 stopped
* thread #1, stop reason = signal SIGINT
    frame #0: 0xffffff801e0dbf40 kernel`machine_idle + 480
kernel`machine_idle:
->  0xffffff801e0dbf40 <+480>: cli
    0xffffff801e0dbf41 <+481>: movq   %gs:0x0, %rax
    0xffffff801e0dbf4a <+490>: andq   $-0x2, 0x100(%rax)
    0xffffff801e0dbf52 <+498>: callq  0xffffff801df5b140        ; do_mfence
Target 1: (kernel) stopped.
(lldb) fdp-save
* Saving the VM state
* State saved
```

The execution of the VM was then resumed, and the PoC was run:

```
(lldb) continue
Process 1 resuming
```

```
Tims-iMac:~ tcook$ while true; do sudo ./in6_selectsrc; done
Password:
res0: 3
res1: 0
res1.5: -1
res2: 0
done
res0: 3
res1: 0
res1.5: -1
res2: 0
done
. . .
```

After a few attempts the bug was successfully triggered, and the machine panicked and initiated rebooting. But since a previous state of the VM was saved, it was possible to quickly restore it with `fdp-restore`:

```
(lldb) fdp-restore
* Restoring the last saved VM state
* State restored
LLDBagility
* Attaching to the VM
* Resuming the VM execution until reaching kernel code
. . .
.warning: Can't find binary/dSYM for com.apple.driver.AppleHWSensor
     ↪ (88083746-B4CC-38FC-9DB2-81D03592CBD5)
.warning: Can't find binary/dSYM for com.apple.fileutil
     ↪ (5E0468C0-F2DE-37EF-BB2A-0796BA8311B9)
. done.
Process 1 stopped
* thread #1, stop reason = signal SIGSTOP
    frame #0: 0xffffff801e0dbf40 kernel`machine_idle + 480
kernel`machine_idle:
->  0xffffff801e0dbf40 <+480>: cli
    0xffffff801e0dbf41 <+481>: movq   %gs:0x0, %rax
    0xffffff801e0dbf4a <+490>: andq   $-0x2, 0x100(%rax)
```

```
    0xffffff801e0dbf52 <+498>: callq  0xffffff801df5b140        ; do_mfence
Target 1: (kernel) stopped.
```

To pause the execution of the VM before the system rebooted, for demonstration purposes a read-write watchpoint on the `panicDebugging` global variable was set with `fdp-hbreakpoint`:

```
(lldb) fdp-hbreakpoint set rw 0 &panicDebugging
  * Hardware breakpoint set: address = 0xffffff801e896814
(lldb) continue
Process 1 resuming
```

The PoC was then executed again. When the bug was triggered after a couple of seconds, the watchpoint fired:

```
Process 1 stopped
* thread #1, stop reason = EXC_BREAKPOINT (code=3, subcode=0x0)
    frame #0: 0xffffff801dfaed01 kernel`handle_debugger_trap + 1665
kernel`handle_debugger_trap:
->  0xffffff801dfaed01 <+1665>: jne    0xffffff801dfaed25        ; <+1701>
    0xffffff801dfaed03 <+1667>: leaq   0x773a78(%rip), %rdi      ; "Attempting
     ↪ system restart..."
    0xffffff801dfaed0a <+1674>: xorl   %eax, %eax
    0xffffff801dfaed0c <+1676>: callq  0xffffff801dfc3860        ; printf
Target 1: (kernel) stopped.
```

5.2 Part 2: Loading and executing lldbmacros

As mentioned above, no KDK was released for the kernel being debugged. Trying, as an example, to load the lldbmacros distributed with the KDK for kernel build 18C54 resulted in 'FATAL FAILURE':

```
(lldb) command script import /Library/. . ./KDK_10.14.2_18C54.kdk/. .
     ↪ ./kernel.dSYM/Contents/Resources/Python/kernel.py
Loading kernel debugging from /Library/. . ./KDK_10.14.2_18C54.kdk/. .
     ↪ ./kernel.dSYM/Contents/Resources/Python/kernel.py
LLDB version lldb-1001.0.13.3
  Swift-5.0
settings set target.process.python-os-plugin-path "/Library/. .
     ↪ ./KDK_10.14.2_18C54.kdk/. . ./lldbmacros/core/operating_system.py"
Target arch: x86_64
Instantiating threads completely from saved state in memory.
settings set target.trap-handler-names hndl_allintrs hndl_alltraps
     ↪ trap_from_kernel hndl_double_fault hndl_machine_check _fleh_prefabt
     ↪ _ExceptionVectorsBase _ExceptionVectorsTable _fleh_undef _fleh_dataabt
     ↪ _fleh_irq _fleh_decirq _fleh_fiq_generic _fleh_dec
FATAL FAILURE: Unable to find kdp_thread state for this connection.
command script import "/Library/. . ./KDK_10.14.2_18C54.kdk/. .
     ↪ ./kernel.dSYM/Contents/Resources/Python/lldbmacros/xnu.py"
xnu debug macros loaded successfully. Run showlldbtypesummaries to enable type
     ↪ summaries.
settings set target.process.optimization-warnings false
```

Examination of lldbmacros sources in correspondence of the error revealed it to be consequence of the lack of kernel debug information, and in particular to the missing definition of the kdp global variable; further inspection of lldbmacros' initialisation process showed usages also of the `version`, `kernel_stack_size`, `threads_count`, `processor_list`, `threads` and `initproc` global variables.

This second part of the demonstration shows how it was possible to create a companion DWARF for kernel 18D109 with some debug information extracted from the DWARF for kernel 18C54, using the helper scripts provided by LLDBagility. The procedure was explained in detail in section 4.5. First of all, the information to be extracted from the DWARF had to be located in the file itself. To find the offset where a particular variable or structure is defined, the `dwarfdump` utility was used:

```
$ DWARFutils/dwarfdump --name kdp /Library/. . ./KDK_10.14.2_18C54.kdk/. .
    ↪ ./kernel.dSYM/Contents/Resources/DWARF/kernel
. . .
Searching .debug_info for 'kdp'... 1 match:

0x00029b5a: TAG_variable [2]
              AT_name( "kdp" )
              AT_type( {0x00029b6f} ( kdp_glob_t ) )
              AT_external( true )
              AT_decl_file( "/BuildRoot/.␣.␣./xnu-4903.231.4/osfmk/kdp/kdp.c" )
              AT_decl_line( 99 )
              AT_location( [0xffffff8000c96008] )
```

In this case, the `kdp` variable was defined at offset 0x29b5a. The same procedure was repeated to retrieve the offsets of `version`, `kernel_stack_size` and the other variables listed above, and results were written to the `demo-18D109.vars` configuration file as shown in listing 5.2, used later as input to the LLDBagility scripts.

Listing 5.2: The `demo-18D109.vars` configuration file

```
#!/usr/bin/env bash

KDKUTILS_SOURCE_KERNEL_DWARF="/Library/.␣.␣./KDK_10.14.2_18C54.kdk/.␣.␣
    ↪ ./kernel.dSYM/Contents/Resources/DWARF/kernel"
KDKUTILS_SOURCE_KERNEL_DIEOFFSETS=(
    `#FindGlobalVariables` \
    0x01478819 `#version` \
    0x001a4241 `#kernel_stack_size` \
    0x00029b5a `#kdp` \
    0x001793d9 `#threads_count` \
    0x00179419 `#processor_list` \
    0x001793c4 `#threads` \
    `#kern.globals` \
    0x00dcd769 `#initproc` \
)
KDKUTILS_GENERATED_KERNEL="/tmp/kernel-18D109-DWARF"

KDKUTILS_TARGET_KERNEL="/Users/fcagnin/kernel-18D109"
KDKUTILS_TARGET_KERNEL_DWARF="$KDKUTILS_GENERATED_KERNEL"
KDKUTILS_RELOCATESYMBOLS=(
    `#FindGlobalVariables` \
    "version" \
    "kernel_stack_size" \
    "kdp" \
    "threads_count" \
    "processor_list" \
```

```
    "threads" \
    `#kern.globals` \
    "initproc" \
)

KDKUTILS_LLDBMACROS="/Library/.␣.␣./KDK_10.14.2_18C54.kdk/.␣.␣
    ↪ ./kernel.dSYM/Contents/Resources/Python/kernel.py"
LLDBAGILITY_VMNAME="macos-mojave-18D109"
```

By executing the KDKutils/1-create-DWARF.sh [LLDBagility] script (see listing D.3), the debug information relative to all variables added to the configuration file was extracted from the DWARF for kernel 18C54 in the form of C sources, shown in part in listing 5.3, which were then recompiled to generate a new DWARF, saved as /tmp/kernel-18D109-DWARF:

```
$ ./1-create-DWARF.sh /Users/fcagnin/demo-18D109.vars
VARSFILE="/Users/fcagnin/demo-18D109.vars"
KDKUTILS_SOURCE_KERNEL_DWARF="/Library/.␣.␣./KDK_10.14.2_18C54.kdk/.␣.␣
    ↪ ./kernel.dSYM/Contents/Resources/DWARF/kernel"
KDKUTILS_SOURCE_KERNEL_DIEOFFSETS="0x01478819␣0x001a4241␣0x00029b5a␣0x001793d9␣
    ↪ 0x00179419␣0x001793c4␣0x00dcd769"
KDKUTILS_GENERATED_KERNEL="/tmp/kernel-18D109-DWARF"
0x014787ec: TAG_compile_unit [1] *
. . .
0x00e89874: TAG_structure_type [32] *
0x00e89875:   AT_byte_size( 0x10 )
0x00e89876:   AT_decl_file( 0x09 ( "/BuildRoot/.␣.␣
    ↪ ./xnu-4903.231.4/bsd/sys/eventvar.h" ) )
0x00e89877:   AT_decl_line( 0x6d ( 109 ) )
Using cached DIEPointer(ttype=(15243282, 'knote_lock_ctx'))
Using cached DIEPointer(ttype=(15243277, 'knote_lock_ctx*'))
. . .
```

Listing 5.3: Excerpt of autogenerated C sources. Data types were extracted from the DWARF for kernel 18C54.

```
. . .
typedef struct ipc_object *ipc_object_t; /* die=0x2e5a5 */

typedef enum {
  PSET_SMP = 0,
} pset_cluster_type_t; /* die=0x2807b */

typedef struct pset_node *pset_node_t; /* die=0x2dee9 */

struct pset_node {
  processor_set_t psets;     /* off=0x0000 */
  pset_node_t     nodes;     /* off=0x0008 */
  pset_node_t     node_list; /* off=0x0010 */
  pset_node_t     parent;    /* off=0x0018 */
};                           /* size=0x20 die=0x2def9 */
. . .
typedef struct {
  void *          saved_state;          /* off=0x0000 */
  thread_t        kdp_thread;           /* off=0x0008 */
  int             kdp_cpu;              /* off=0x0010 */
  uint32_t        session_key;          /* off=0x0014 */
  unsigned int    conn_seq;             /* off=0x0018 */
  unsigned short  reply_port;           /* off=0x001c */
```

```
  unsigned short exception_port;        /* off=0x001e */
  boolean_t      is_conn;               /* off=0x0020 */
  boolean_t      is_halted;             /* off=0x0024 */
  unsigned char  exception_seq;         /* off=0x0028 */
  boolean_t      exception_ack_needed;  /* off=0x002c */
} kdp_glob_t;                           /* die=0x29b6f */

kdp_glob_t kdp;
```

Next, using the `KDKutils/2-fake-DWARF.sh` [LLDBagility] script (see listing D.4), the new DWARF was patched, so that its UUID and the `__TEXT`, `__DATA` and `__LINKEDIT` segments all matched the structure of the kernel 18D109:

```
$ ./2-fake-DWARF.sh /Users/fcagnin/demo-18D109.vars
VARSFILE="/Users/fcagnin/demo-18D109.vars"
KDKUTILS_TARGET_KERNEL="/Users/fcagnin/kernel-18D109"
KDKUTILS_TARGET_KERNEL_DWARF="/tmp/kernel-18D109-DWARF"
KDKUTILS_RELOCATESYMBOLS="version"
Relocating /Users/fcagnin/kernel-18D109 version 0xffffff8000b1c560
Relocating /Users/fcagnin/kernel-18D109 kernel_stack_size 0xffffff8000c9c000
Relocating /Users/fcagnin/kernel-18D109 kdp 0xffffff8000c96008
Relocating /Users/fcagnin/kernel-18D109 threads_count 0xffffff8000c9b014
Relocating /Users/fcagnin/kernel-18D109 processor_list 0xffffff8000c9b000
Relocating /Users/fcagnin/kernel-18D109 threads 0xffffff8000c9afd0
Relocating /Users/fcagnin/kernel-18D109 initproc 0xffffff8000e17538
```

The patched DWARF was then loaded in LLDB:

```
(lldb) target symbols add /tmp/kernel-18D109-DWARF
symbol file '/tmp/kernel-18D109-DWARF' has been added to
     ↪ '/Users/fcagnin/kernel-18D109'
```

lldbmacros from 18C54 were imported anew, this time without errors:

```
(lldb) command script import /Library/. . ./KDK_10.14.2_18C54.kdk/. .
     ↪ ./kernel.dSYM/Contents/Resources/Python/kernel.py
Loading kernel debugging from /Library/. . ./KDK_10.14.2_18C54.kdk/. .
     ↪ ./kernel.dSYM/Contents/Resources/Python/kernel.py
LLDB version lldb-1001.0.13.3
  Swift-5.0
settings set target.process.python-os-plugin-path "/Library/. .
     ↪ ./KDK_10.14.2_18C54.kdk/. . ./lldbmacros/core/operating_system.py"
Target arch: x86_64
Instantiating threads completely from saved state in memory.
settings set target.trap-handler-names hndl_allintrs hndl_alltraps
     ↪ trap_from_kernel hndl_double_fault hndl_machine_check _fleh_prefabt
     ↪ _ExceptionVectorsBase _ExceptionVectorsTable _fleh_undef _fleh_dataabt
     ↪ _fleh_irq _fleh_decirq _fleh_fiq_generic _fleh_dec
command script import "/Library/. . ./KDK_10.14.2_18C54.kdk/. .
     ↪ ./kernel.dSYM/Contents/Resources/Python/lldbmacros/xnu.py"
xnu debug macros loaded successfully. Run showlldbtypesummaries to enable type
     ↪ summaries.
settings set target.process.optimization-warnings false
```

The `showversion` macro worked as expected:

```
(lldb) showversion
Darwin Kernel Version 18.2.0: Thu Dec 20 20:46:53 PST 2018;
     ↪ root:xnu-4903.241.1~1/RELEASE_X86_64
```

But `allproc` failed!

```
(lldb) allproc

************ LLDB found an exception ************
There has been an uncaught exception. A possible cause could be that remote
     ↪ connection has been disconnected.
However, it is recommended that you report the exception to lldb/kernel debugging
     ↪ team about it.
************ Please run 'xnudebug debug enable' to start collecting logs.
     ↪ ************

Traceback (most recent call last):
  File "/Library/. . ./KDK_10.14.2_18C54.kdk/. .
     ↪ ./kernel.dSYM/Contents/Resources/Python/lldbmacros/xnu.py", line 116, in
     ↪ _internal_command_function
    obj(cmd_args=stream.target_cmd_args)
  File "/Library/. . ./KDK_10.14.2_18C54.kdk/. .
     ↪ ./kernel.dSYM/Contents/Resources/Python/lldbmacros/process.py", line 115,
     ↪ in AllProc
    for proc in kern.procs :
  File "/Library/. . ./KDK_10.14.2_18C54.kdk/. .
     ↪ ./lldbmacros/core/kernelcore.py", line 547, in __getattribute__
    proc_val = cast(all_proc_head.lh_first, 'proc *')
  File "/Library/. . ./KDK_10.14.2_18C54.kdk/. . ./lldbmacros/core/cvalue.py",
     ↪ line 88, in __getattr__
    raise AttributeError("No field by name: "+name )
AttributeError: No field by name: lh_first
```

Once more, type information were missing, and inspection of lldbmacros sources suggested that the `allproc` global variable should be included in the DWARF for this macro to work. The definition of the variable in DWARF for kernel 18C54 was again found with `dwarfdump`:

```
$ DWARFutils/dwarfdump --name allproc /Library/. . ./KDK_10.14.2_18C54.kdk/. .
     ↪ ./kernel.dSYM/Contents/Resources/DWARF/kernel
. . .
0x00e8b6a5: TAG_variable [2]
             AT_name( "allproc" )
             AT_type( {0x00e8b6ba} ( proclist ) )
. . .
```

The `demo-18D109.vars` configuration file was updated to include `allproc`, and then the DWARF was regenerated and patched using the two previous scripts.

To automatise the process of starting LLDB, specifying the kernel binary, adding the generated symbols, attaching to the VM, and loading lldbmacros, a third script `KDKutils/3-attach-DWARF.sh` [LLDBagility] (see listing D.5) was used:

```
$ ./3-attach-DWARF.sh /Users/fcagnin/demo-18D109.vars
VARSFILE="/Users/fcagnin/demo-18D109.vars"
KDKUTILS_TARGET_KERNEL="/Users/fcagnin/kernel-18D109"
KDKUTILS_TARGET_KERNEL_DWARF="/tmp/kernel-18D109-DWARF"
KDKUTILS_LLDBMACROS="/Library/. . ./KDK_10.14.2_18C54.kdk/. .
     ↪ ./kernel.dSYM/Contents/Resources/Python/kernel.py"
LLDBAGILITY_VMNAME="macos-mojave-18D109"
(lldb) target create "/Users/fcagnin/kernel-18D109"
Current executable set to '/Users/fcagnin/kernel-18D109' (x86_64).
(lldb) target symbols add "/tmp/kernel-18D109-DWARF"
```

```
symbol file '/tmp/kernel-18D109-DWARF' has been added to
     ↪ '/Users/fcagnin/kernel-18D109'
(lldb) fdp-attach macos-mojave-18D109
LLDBagility
* Attaching to the VM
* Resuming the VM execution until reaching kernel code
. . .
Process 1 stopped
* thread #1, stop reason = signal SIGSTOP
    frame #0: 0xffffff801dfaed01 kernel`handle_debugger_trap + 1665
kernel`handle_debugger_trap:
->  0xffffff801dfaed01 <+1665>: jne    0xffffff801dfaed25        ; <+1701>
    0xffffff801dfaed03 <+1667>: leaq   0x773a78(%rip), %rdi      ; "Attempting
     ↪ system restart..."
    0xffffff801dfaed0a <+1674>: xorl   %eax, %eax
    0xffffff801dfaed0c <+1676>: callq  0xffffff801dfc3860        ; printf
Target 0: (kernel) stopped.
(lldb) command script import "/Library/. . ./KDK_10.14.2_18C54.kdk/. .
     ↪ ./kernel.dSYM/Contents/Resources/Python/kernel.py"
Loading kernel debugging from /Library/. . ./KDK_10.14.2_18C54.kdk/. .
     ↪ ./kernel.dSYM/Contents/Resources/Python/kernel.py
. . .
(lldb) showversion
Darwin Kernel Version 18.2.0: Thu Dec 20 20:46:53 PST 2018;
     ↪ root:xnu-4903.241.1~1/RELEASE_X86_64
```

The `allproc` macro was executed again, without errors:

```
(lldb) allproc
Process 0xffffff80281fce20
  name in6_selectsrc
  pid:486     task:0xffffff802a7f6cc0   p_stat:2      parent pid: 485
Cred: euid 0 ruid 0 svuid 0
Flags: 0x4006
  0x00000002 - has a controlling tty
  0x00000004 - process is 64 bit
  0x00004000 - process has called exec
State: Run
Process 0xffffff80281fc540
  name sudo
  pid:485     task:0xffffff802a7f7e00   p_stat:2      parent pid: 300
Cred: euid 0 ruid 0 svuid 0
Flags: 0x4106
  0x00000002 - has a controlling tty
  0x00000004 - process is 64 bit
  0x00000100 - has set privileges since exec
  0x00004000 - process has called exec
State: Run
. . .
Process 0xffffff80255757f0
  name launchd
  pid:1       task:0xffffff8024dc8cc0   p_stat:2      parent pid: 0
Cred: euid 0 ruid 0 svuid 0
Flags: 0x4004
  0x00000004 - process is 64 bit
  0x00004000 - process has called exec
State: Run
Process 0xffffff801ea168e8
  name kernel_task
  pid:0       task:0xffffff8024dc9840   p_stat:2      parent pid: 0
```

```
Cred: euid 0 ruid 0 svuid 0
Flags: 0x204
  0x00000004 - process is 64 bit
  0x00000200 - system process: no signals, stats, or swap
State: Run
```

Lastly, the `paniclog` macro was executed, which too crashed because of some missing information:

```
(lldb) paniclog

************ LLDB found an exception ************
There has been an uncaught exception. A possible cause could be that remote
     ↪ connection has been disconnected.
However, it is recommended that you report the exception to lldb/kernel debugging
     ↪ team about it.
************ Please run 'xnudebug debug enable' to start collecting logs.
     ↪ ************

Traceback (most recent call last):
    File "/Library/. . ./KDK_10.14.2_18C54.kdk/. .
     ↪ ./kernel.dSYM/Contents/Resources/Python/lldbmacros/xnu.py", line 114, in
     ↪ _internal_command_function
    obj(cmd_args=stream.target_cmd_args, cmd_options=stream.target_cmd_options)
    File "/Library/. . ./KDK_10.14.2_18C54.kdk/. .
     ↪ ./kernel.dSYM/Contents/Resources/Python/lldbmacros/xnu.py", line 614, in
     ↪ ShowPanicLog
    panic_buf = Cast(kern.globals.panic_info, 'char *')
    File "/Library/. . ./KDK_10.14.2_18C54.kdk/. .
     ↪ ./lldbmacros/core/kernelcore.py", line 279, in __getattr__
    raise ValueError('No such global variable by name: %s '%str(name))
ValueError: No such global variable by name: panic_info
```

But, after recreating the DWARF file including information for the `panic_info` and `debug_buf_ptr` global variables, also this macro worked correctly:

```
$ ./3-attach-DWARF.sh /Users/fcagnin/demo-18D109.vars
VARSFILE="/Users/fcagnin/demo-18D109.vars"
. . .
(lldb) fdp-attach macos-mojave-18D109
LLDBagility
* Attaching to the VM
* Resuming the VM execution until reaching kernel code
. . .
(lldb) paniclog
panic(cpu 0 caller 0xffffff801e0da29d): Kernel trap at 0xffffff801e411764, type
     ↪ 13=general protection, registers:
CR0: 0x00000000c0010033, CR2: 0x00007fa61f001000, CR3: 0x000000005f29d000, CR4:
     ↪ 0x00000000000006e0
RAX: 0x0000000000000001, RBX: 0xdeadbeefdeadbeef, RCX: 0x0000000000000000, RDX:
     ↪ 0x0000000000000000
RSP: 0xffffff8880c0bd30, RBP: 0xffffff8880c0bdc0, RSI: 0x0000000000000000, RDI:
     ↪ 0x0000000000000001
R8:  0x0000000000000000, R9:  0xffffff8880c0bde0, R10: 0xffffff801ea4de20, R11:
     ↪ 0x0000000000000000
R12: 0x0000000000000000, R13: 0xffffff802a791e00, R14: 0xffffff8026ed33a8, R15:
     ↪ 0x0000000000000000
RFL: 0x0000000000010282, RIP: 0xffffff801e411764, CS:  0x0000000000000008, SS:
     ↪ 0x0000000000000010
```

```
Fault CR2: 0x00007fa61f001000, Error code: 0x0000000000000000, Fault CPU: 0x0
     ↪ VMM, PL: 0, VF: 0

Backtrace (CPU 0), Frame : Return Address
0xffffff801dd4c290 : 0xffffff801dfaeb0d mach_kernel : _handle_debugger_trap +
     ↪ 0x48d
0xffffff801dd4c2e0 : 0xffffff801e0e8653 mach_kernel : _kdp_i386_trap + 0x153
0xffffff801dd4c320 : 0xffffff801e0da07a mach_kernel : _kernel_trap + 0x4fa
0xffffff801dd4c390 : 0xffffff801df5bca0 mach_kernel : _return_from_trap + 0xe0
0xffffff801dd4c3b0 : 0xffffff801dfae527 mach_kernel : _panic_trap_to_debugger +
     ↪ 0x197
0xffffff801dd4c4d0 : 0xffffff801dfae373 mach_kernel : _panic + 0x63
0xffffff801dd4c540 : 0xffffff801e0da29d mach_kernel : _kernel_trap + 0x71d
0xffffff801dd4c6b0 : 0xffffff801df5bca0 mach_kernel : _return_from_trap + 0xe0
0xffffff801dd4c6d0 : 0xffffff801e411764 mach_kernel : _in6_selectsrc + 0x114
0xffffff8880c0bdc0 : 0xffffff801e443015 mach_kernel : _nd6_setdefaultiface + 0xd75
0xffffff8880c0be20 : 0xffffff801e520274 mach_kernel : _soconnectlock + 0x284
0xffffff8880c0be60 : 0xffffff801e5317bf mach_kernel : _connect_nocancel + 0x20f
0xffffff8880c0bf40 : 0xffffff801e5b62bb mach_kernel : _unix_syscall64 + 0x26b
0xffffff8880c0bfa0 : 0xffffff801df5c466 mach_kernel : _hndl_unix_scall64 + 0x16

BSD process name corresponding to current thread: in6_selectsrc
Boot args: "fs4:\System\Library\CoreServices\boot.efi" keepsyms=1 -v

Mac OS version:
18D109

Kernel version:
Darwin Kernel Version 18.2.0: Thu Dec 20 20:46:53 PST 2018;
     ↪ root:xnu-4903.241.1~1/RELEASE_X86_64
Kernel UUID: 1970B070-E53F-3178-83F3-1B95FA340695
Kernel slide:     0x000000001dc00000
Kernel text base: 0xffffff801de00000
__HIB  text base: 0xffffff801dd00000
System model name: iMac11,3 (Iloveapple)

System uptime in nanoseconds: 134478762979
last loaded kext at 24981853631: @fileutil      18.306.12 (addr
     ↪ 0xffffff7f9f0ae000, size 114688)
loaded kexts:
@fileutil       18.306.12
>!AHWSensor     1.9.5d0
. . .
@kec.Libm       1
@kec.corecrypto 1.0
```

The debugging session was then terminated.

Chapter 6

Conclusion

This work presented LLDBagility, a new software tool for macOS kernel debugging that allows to connect LLDB to any macOS virtual machine running on a patched version of the VirtualBox hypervisor, thanks to the capabilities of virtual machine introspection offered by the Fast Debugging Protocol. The tool was developed to overcome the limitations of the other two methods for kernel debugging in macOS, namely the Kernel Debugging Protocol and the GDB stub in VMware Fusion.

Another novel contribution of this work was a solution for alleviating the absence of lldbmacros for most kernel builds. In this regard, reusing debug information from available DWARF companion files for kernels that lack their own revealed to be effective, as demonstrated in the case study.

Room for improvement and future work has been identified. First, LLDBagility has been implemented for Python 2 because LLDB was based on this version when the development of the tool started. Now, the very latest versions of LLDB shipped with the Command Line Tools are based instead on Python 3, and so LLDBagility must be upgraded as well; this fix is already planned for the next release of the tool.

Second, as made clear by the case study, identifying the symbols required by lldbmacros to work correctly is for now a manual process, as is extracting symbols' offsets in the DWARF file; these procedures could certainly be automated at least to some extent, so that switching to different lldbmacros or DWARF files would be seamless.

Third, the last part of the DWARF patching procedure requires to know the virtual load addresses of all symbols that will be used later by lldbmacros. This information is assumed to be found in the symbol table of the RELEASE kernel binary to debug, and while this has always been the case for the tests conducted during LLDBagility's development, such assumption may not hold for some other symbols and kernels; then, some heuristics to find the missing symbols in kernel memory have to be devised.

Lastly, rumours have it that Apple will sooner or later switch its Mac products to

Arm-based CPUs, and currently both VirtualBox and FDP only work with x86. If this transition happens, then to debug a macOS compiled for Arm FDP will have to be ported either to a different hypervisor or to an Arm emulator; in minor part, LLDBagility will have to be updated as well, since some of its code is based on x86 features (e.g. hardware breakpoints).

Appendix A

Interactions between XNU and LLDB during KDP initialisation

This appendix gives an overview of the initial phase of the kernel debugging process via KDP as seen by both XNU and LLDB. All references to source code files are provided for XNU 4903.221.2[1] from macOS 10.14.1 Mojave and LLDB 8.0.0[2].

Assuming a debug build of the kernel has been correctly set up for use in macOS and the debug boot-arg has been set to `DB_HALT`, at some point during XNU startup an `IOKernelDebugger` object calls `kdp_register_send_receive()`. This routine, after parsing the debug boot-arg, executes `kdp_call()` to generate a CPU exception of type `EXC_BREAKPOINT`, which in turn triggers `trap_from_kernel()`, `kernel_trap()` and `kdp_i386_trap()`. This last calls `handle_debugger_trap()` and eventually `kdp_raise_exception()`, dropping into `kdp_debugger_loop()`. Since no debugger is connected yet, the kernel stops at `kdp_connection_wait()`, printing the string 'Waiting for remote debugger connection.' and then waiting to receive a `KDP_REATTACH` request followed by a `KDP_CONNECT`.

Listing A.1: The kernel's call stack as examined just after KDP initialisation, thanks to `DB_HALT`

```
(lldb) bt
* thread #1, stop reason = signal SIGSTOP
  * frame #0: 0xffffff801a4e3bf5 kernel.debug`kdp_call at kdp_machdep.c:331:1
    frame #1: 0xffffff801a25e28f kernel.debug`kdp_register_send_receive. . . at
     ↪ kdp_udp.c:490:3
    frame #2: 0xffffff7f9b3b82df
     ↪ IONetworkingFamily`IOKernelDebugger::registerHandler. . . at
     ↪ IOKernelDebugger.cpp:664:9 [opt]
. . .
```

On the LLDB side, the kdp-remote plug-in[3] handles the logic for connecting to a

[1] Apple. *XNU 4903.221.2 Source*. URL: https://opensource.apple.com/source/xnu/xnu-4903.221.2/.

[2] LLVM Developer Group. *LLDB 8.0.0 Source*. URL: http://releases.llvm.org/8.0.0/lldb-8.0.0.src.tar.xz.

[3] source/Plugins/Process/MacOSX-Kernel/ [LLDB]

remote KDP server. When the kdp-remote command is executed by the user, the debugger initiates the connection to the specified target by executing the routine ProcessKDP::DoConnectRemote(), which sends in sequence the two initial requests KDP_REATTACH and KDP_CONNECT.

In XNU, after the two requests are received kdp_connection_wait() exits and shortly after kdp_handler() is entered; here, requests from the KDP client (LLDB in this case) are processed using a dispatch table and responded in a loop until either KDP_RESUMECPUS or KDP_DISCONNECT is received.

Completed the initial handshake, LLDB then sends three more requests, namely KDP_VERSION, KDP_HOSTINFO and KDP_KERNELVERSION, to retrieve information on the debuggee. If the kernel version string (e.g. 'Darwin Kernel Version 16.0.0 . . . root:xnu-3789.1.32 3/DEVELOPMENT_X86_64; . . .') is recognised as coming from a Darwin kernel, then the darwin-kernel dynamic loader plug-in is loaded. At this point, the connection to the remote target is established and the attach phase is completed by instantiating said plug-in, which tries to locate the kernel load address and the kernel image. Finally, the Darwin kernel module is loaded, which first searches the local file system for an on-disk file copy of the kernel using its UUID and then eventually loads all kernel extensions.

After attaching, LLDB waits for commands from the user, which will be translated into KDP requests and sent to XNU. For example:

- Commands register read and register write generate respectively KDP_READREGS and KDP_WRITEREGS requests.
- Commands memory read and memory write generate KDP_READMEM and KDP_WRITEMEM requests (KDP_READMEM64 and KDP_WRITEMEM64 for 64-bit targets).
- Commands breakpoint set and breakpoint delete generate KDP_BREAKPOINT_SET and KDP_BREAKPOINT_REMOVE requests (KDP_BREAKPOINT_SET64 and KDP_BREAKPOINT_REMOVE64 for 64-bit targets).
- Commands continue, step and variants like stepi all generate KDP_RESUMECPUS requests. In case of single-stepping, this request is preceded by a KDP_WRITEREGS for setting the TF bit of the FLAGS register, so to cause a type-1 interrupt to be raised by the CPU after the execution of the next instruction.

Upon receiving a KDP_RESUMECPUS request, both routines kdp_handler() and kdp_debugger_loop() exit and the machine resumes normal execution. When the CPU hits a breakpoint, a trap is generated, and from trap_from_kernel() the control flow reaches kdp_debugger_loop() again; but this time LLDB is connected, and thus a KDP_EXCEPTION notification is generated to inform it about the interruption. With a call to kdp_handler(), the KDP stub is then ready to receive and handle new debugging requests.

Appendix B

Excerpts from XNU sources

This appendix contains selected source code excerpts from XNU 4903.221.2[1].

Listing B.1: kdp_req_t from osfmk/kdp/kdp_protocol.h [XNU]

```
/*
 * Requests
 */
typedef enum {
        /* connection oriented requests */
        KDP_CONNECT,    KDP_DISCONNECT,

        /* obtaining client info */
        KDP_HOSTINFO,   KDP_VERSION,    KDP_MAXBYTES,

        /* memory access */
        KDP_READMEM,    KDP_WRITEMEM,

        /* register access */
        KDP_READREGS,   KDP_WRITEREGS,

        /* executable image info */
        KDP_LOAD,       KDP_IMAGEPATH,

        /* execution control */
        KDP_SUSPEND,    KDP_RESUMECPUS,

        /* exception and termination notification, NOT true requests */
        KDP_EXCEPTION,  KDP_TERMINATION,

        /* breakpoint control */
        KDP_BREAKPOINT_SET, KDP_BREAKPOINT_REMOVE,

        /* vm regions */
        KDP_REGIONS,

        /* reattach to a connected host */
        KDP_REATTACH,
```

[1]Apple. *XNU 4903.221.2 Source*. URL: https://opensource.apple.com/source/xnu/xnu-4903.221.2/.

```

	/* remote reboot request */
	KDP_HOSTREBOOT,

	/* memory access (64-bit wide addresses). Version 11 protocol */
	KDP_READMEM64,	KDP_WRITEMEM64,

	/* breakpoint control (64-bit wide addresses). Version 11 protocol */
	KDP_BREAKPOINT64_SET, KDP_BREAKPOINT64_REMOVE,

	/* kernel version string, like "xnu-1234.5~6". Version 11 protocol */
	KDP_KERNELVERSION,

	/* physical memory access (64-bit wide addresses). Version 12 protocol */
	KDP_READPHYSMEM64,	KDP_WRITEPHYSMEM64,

	/* ioport access (8-, 16-, and 32-bit) */
	KDP_READIOPORT, KDP_WRITEIOPORT,

	/* msr access (64-bit) */
	KDP_READMSR64,	KDP_WRITEMSR64,

	/* get/dump panic/corefile info */
	KDP_DUMPINFO,

	/* keep this last */
	KDP_INVALID_REQUEST
} kdp_req_t;
```

Listing B.2: kdp_debugger_loop() from osfmk/kdp/kdp_udp.c [XNU]

```
static void
kdp_debugger_loop(
	unsigned int		exception,
	unsigned int		code,
	unsigned int		subcode,
	void			*saved_state)
{
	int			index;

	if (saved_state == 0)
		printf("kdp_raise_exception_with_NULL_state\n");

	index = exception;
	if (exception != EXC_BREAKPOINT) {
		if (exception > EXC_BREAKPOINT || exception < EXC_BAD_ACCESS) {
			index = 0;
		}
		printf("%s_exception_(%x,%x,%x)\n",
				exception_message[index],
				exception, code, subcode);
	}

	kdp_sync_cache();

	/* XXX WMG it seems that sometimes it doesn't work to let kdp_handler
	 * do this. I think the client and the host can get out of sync.
	 */
	kdp.saved_state = saved_state;
	kdp.kdp_cpu = cpu_number();
```

```
	kdp.kdp_thread = current_thread();

	if (kdp_en_setmode)
		(*kdp_en_setmode)(TRUE); /* enabling link mode */

	if (pkt.input)
		kdp_panic("kdp_raise_exception");

	if (((kdp_flag & KDP_PANIC_DUMP_ENABLED)
	     || (kdp_flag & PANIC_LOG_DUMP))
	    && panic_active()) {
		kdp_panic_dump();
		if (kdp_flag & REBOOT_POST_CORE && dumped_kernel_core())
			kdp_machine_reboot();
	} else {
		if ((kdp_flag & PANIC_CORE_ON_NMI) && !panic_active()
			&& !kdp.is_conn) {

			disableConsoleOutput = FALSE;
			kdp_panic_dump();
			if (kdp_flag & REBOOT_POST_CORE && dumped_kernel_core())
				kdp_machine_reboot();

			if (!(kdp_flag & DBG_POST_CORE))
				goto exit_debugger_loop;
		}
	}

again:
	if (!kdp.is_conn) {
		kdp_connection_wait();
	} else {
		kdp_send_exception(exception, code, subcode);
		if (kdp.exception_ack_needed) {
			kdp.exception_ack_needed = FALSE;
			kdp_remove_all_breakpoints();
			printf("Remote debugger disconnected.\n");
		}
	}

	if (kdp.is_conn) {
		kdp.is_halted = TRUE;		/* XXX */
		kdp_handler(saved_state);
		if (!kdp.is_conn)
		{
			kdp_remove_all_breakpoints();
			printf("Remote debugger disconnected.\n");
		}
	}
	/* Allow triggering a panic core dump when connected to the machine
	 * Continuing after setting kdp_trigger_core_dump should do the
	 * trick.
	 */

	if (1 == kdp_trigger_core_dump) {
		kdp_flag |= KDP_PANIC_DUMP_ENABLED;
		kdp_panic_dump();
		if (kdp_flag & REBOOT_POST_CORE && dumped_kernel_core())
			kdp_machine_reboot();
		kdp_trigger_core_dump = 0;
```

```
        }

        /* Trigger a reboot if the user has set this flag through the
         * debugger.Ideally, this would be done through the HOSTREBOOT packet
         * in the protocol,but that will need gdb support,and when it's
         * available, it should work automatically.
         */
        if (1 == flag_kdp_trigger_reboot) {
                kdp_machine_reboot();
                /* If we're still around, reset the flag */
                flag_kdp_trigger_reboot = 0;
        }

        if (kdp_reentry_deadline) {
                kdp_schedule_debugger_reentry(kdp_reentry_deadline);
                printf("Debugger re-entry scheduled in %d milliseconds\n",
 kdp_reentry_deadline);
                kdp_reentry_deadline = 0;
        }

        kdp_sync_cache();

        if (reattach_wait == 1)
                goto again;

exit_debugger_loop:
        if (kdp_en_setmode)
                (*kdp_en_setmode)(FALSE); /* link cleanup */
}
```

Listing B.3: kdp_raise_exception() from osfmk/kdp/kdp_udp.c [XNU]

```
#if !CONFIG_KDP_INTERACTIVE_DEBUGGING
extern __attribute__((noreturn)) void panic_spin_forever(void);

__attribute__((noreturn))
void
kdp_raise_exception(
                __unused unsigned int           exception,
                __unused unsigned int           code,
                __unused unsigned int           subcode,
                __unused void                   *saved_state
                )
#else
void
kdp_raise_exception(
                unsigned int            exception,
                unsigned int            code,
                unsigned int            subcode,
                void                    *saved_state
                )
#endif
{
#if CONFIG_EMBEDDED
        assert(PE_i_can_has_debugger(NULL));
#endif

#if CONFIG_KDP_INTERACTIVE_DEBUGGING

        kdp_debugger_loop(exception, code, subcode, saved_state);
```

```
#else /* CONFIG_KDP_INTERACTIVE_DEBUGGING */
        assert(current_debugger != KDP_CUR_DB);

        panic_spin_forever();
#endif /* CONFIG_KDP_INTERACTIVE_DEBUGGING */
}
```

Listing B.4: `kdp_call()` from `osfmk/kdp/ml/x86_64/kdp_machdep.c` [XNU]

```
void
kdp_call(void)
{
        __asm__ volatile ("int $3");   /* Let the processor do the work */
}
```

Listing B.5: Excerpt of `kernel_trap()` from `osfmk/i386/trap.c` [XNU]

```
        switch (type) {

            case T_NO_FPU:
                fpnoextflt();
                return;

            case T_FPU_FAULT:
                fpextovrflt();
                return;

            case T_FLOATING_POINT_ERROR:
                fpexterrflt();
                return;

            case T_SSE_FLOAT_ERROR:
                fpSSEexterrflt();
                return;

            case T_INVALID_OPCODE:
                fpUDflt(kern_ip);
                goto debugger_entry;

            case T_DEBUG:
                    if ((saved_state->isf.rflags & EFL_TF) == 0 && NO_WATCHPOINTS)
                    {
                            /* We've somehow encountered a debug
                             * register match that does not belong
                             * to the kernel debugger.
                             * This isn't supposed to happen.
                             */
                            reset_dr7();
                            return;
                    }
                    goto debugger_entry;
            case T_INT3:
              goto debugger_entry;
```

Appendix C

Excerpts from LLDB sources

This appendix contains selected source code excerpts from LLDB 8.0.0[1].

Listing C.1: The `DoConnectRemote()` attach routine from `source/Plugins/Process/MacOSX-Kernel/ProcessKDP.cpp` [LLDB]

```
Status ProcessKDP::DoConnectRemote(Stream *strm, llvm::StringRef remote_url) {
  Status error;

  // Don't let any JIT happen when doing KDP as we can't allocate memory and we
  // don't want to be mucking with threads that might already be handling
  // exceptions
  SetCanJIT(false);

  if (remote_url.empty()) {
    error.SetErrorStringWithFormat("empty␣connection␣URL");
    return error;
  }

  std::unique_ptr<ConnectionFileDescriptor> conn_ap(
      new ConnectionFileDescriptor());
  if (conn_ap.get()) {
    // Only try once for now.
    // TODO: check if we should be retrying?
    const uint32_t max_retry_count = 1;
    for (uint32_t retry_count = 0; retry_count < max_retry_count;
         ++retry_count) {
      if (conn_ap->Connect(remote_url, &error) == eConnectionStatusSuccess)
        break;
      usleep(100000);
    }
  }

  if (conn_ap->IsConnected()) {
    const TCPSocket &socket =
        static_cast<const TCPSocket &>(*conn_ap->GetReadObject());
    const uint16_t reply_port = socket.GetLocalPortNumber();

```

[1] LLVM Developer Group. *LLDB 8.0.0 Source*. URL: `http://releases.llvm.org/8.0.0/lldb-8.0.0.src.tar.xz`.

```
    if (reply_port != 0) {
      m_comm.SetConnection(conn_ap.release());

      if (m_comm.SendRequestReattach(reply_port)) {
        if (m_comm.SendRequestConnect(reply_port, reply_port,
                                      "Greetings␣from␣LLDB...")) {
          m_comm.GetVersion();

          Target &target = GetTarget();
          ArchSpec kernel_arch;
          // The host architecture
          GetHostArchitecture(kernel_arch);
          ArchSpec target_arch = target.GetArchitecture();
          // Merge in any unspecified stuff into the target architecture in
          // case the target arch isn't set at all or incompletely.
          target_arch.MergeFrom(kernel_arch);
          target.SetArchitecture(target_arch);

          /* Get the kernel's UUID and load address via KDP_KERNELVERSION
           * packet.  */
          /* An EFI kdp session has neither UUID nor load address. */

          UUID kernel_uuid = m_comm.GetUUID();
          addr_t kernel_load_addr = m_comm.GetLoadAddress();

          if (m_comm.RemoteIsEFI()) {
            // Select an invalid plugin name for the dynamic loader so one
            // doesn't get used since EFI does its own manual loading via
            // python scripting
            static ConstString g_none_dynamic_loader("none");
            m_dyld_plugin_name = g_none_dynamic_loader;

            if (kernel_uuid.IsValid()) {
              // If EFI passed in a UUID= try to lookup UUID The slide will not
              // be provided. But the UUID lookup will be used to launch EFI
              // debug scripts from the dSYM, that can load all of the symbols.
              ModuleSpec module_spec;
              module_spec.GetUUID() = kernel_uuid;
              module_spec.GetArchitecture() = target.GetArchitecture();

              // Lookup UUID locally, before attempting dsymForUUID like action
              module_spec.GetSymbolFileSpec() =
                  Symbols::LocateExecutableSymbolFile(module_spec);
              if (module_spec.GetSymbolFileSpec()) {
                ModuleSpec executable_module_spec =
                    Symbols::LocateExecutableObjectFile(module_spec);
                if (FileSystem::Instance().Exists(
                        executable_module_spec.GetFileSpec())) {
                  module_spec.GetFileSpec() =
                      executable_module_spec.GetFileSpec();
                }
              }
              if (!module_spec.GetSymbolFileSpec() ||
                  !module_spec.GetSymbolFileSpec())
                Symbols::DownloadObjectAndSymbolFile(module_spec, true);

              if (FileSystem::Instance().Exists(module_spec.GetFileSpec())) {
                ModuleSP module_sp(new Module(module_spec));
                if (module_sp.get() && module_sp->GetObjectFile()) {
                  // Get the current target executable
```

```
         ModuleSP exe_module_sp(target.GetExecutableModule());

         // Make sure you don't already have the right module loaded
         // and they will be uniqued
         if (exe_module_sp.get() != module_sp.get())
          target.SetExecutableModule(module_sp, eLoadDependentsNo);
        }
       }
      }
     } else if (m_comm.RemoteIsDarwinKernel()) {
      m_dyld_plugin_name =
        DynamicLoaderDarwinKernel::GetPluginNameStatic();
      if (kernel_load_addr != LLDB_INVALID_ADDRESS) {
       m_kernel_load_addr = kernel_load_addr;
      }
     }

     // Set the thread ID
     UpdateThreadListIfNeeded();
     SetID(1);
     GetThreadList();
     SetPrivateState(eStateStopped);
     StreamSP async_strm_sp(target.GetDebugger().GetAsyncOutputStream());
     if (async_strm_sp) {
      const char *cstr;
      if ((cstr = m_comm.GetKernelVersion()) != NULL) {
       async_strm_sp->Printf("Version:␣%s\n", cstr);
       async_strm_sp->Flush();
      }
      //           if ((cstr = m_comm.GetImagePath ()) != NULL)
      //           {
      //             async_strm_sp->Printf ("Image Path:
      //             %s\n", cstr);
      //             async_strm_sp->Flush();
      //           }
     }
    } else {
     error.SetErrorString("KDP␣REATTACH␣failed");
    }
   } else {
    error.SetErrorString("KDP␣REATTACH␣failed");
   }
  } else {
   error.SetErrorString("invalid␣reply␣port␣from␣UDP␣connection");
  }
 } else {
  if (error.Success())
   error.SetErrorStringWithFormat("failed␣to␣connect␣to␣'%s'",
                  remote_url.str().c_str());
 }
 if (error.Fail())
  m_comm.Disconnect();

 return error;
}
```

Listing C.2: `SendRequestReattach()` as defined in `source/Plugins/Process/MacOSX-Kernel/CommunicationKDP.cpp` [LLDB]

```
bool CommunicationKDP::SendRequestReattach(uint16_t reply_port) {
```

```cpp
  PacketStreamType request_packet(Stream::eBinary, m_addr_byte_size,
                                  m_byte_order);
  const CommandType command = KDP_REATTACH;
  // Length is 8 bytes for the header plus 2 bytes for the reply UDP port
  const uint32_t command_length = 8 + 2;
  MakeRequestPacketHeader(command, request_packet, command_length);
  // Always send connect ports as little endian
  request_packet.SetByteOrder(eByteOrderLittle);
  request_packet.PutHex16(htons(reply_port));
  request_packet.SetByteOrder(m_byte_order);
  DataExtractor reply_packet;
  if (SendRequestAndGetReply(command, request_packet, reply_packet)) {
    // Reset the sequence ID to zero for reattach
    ClearKDPSettings();
    lldb::offset_t offset = 4;
    m_session_key = reply_packet.GetU32(&offset);
    return true;
  }
  return false;
}
```

Listing C.3: `RemoteIsDarwinKernel()` from `source/Plugins/Process/MacOSX-Kernel/CommunicationKDP.cpp` [LLDB]

```cpp
bool CommunicationKDP::RemoteIsDarwinKernel() {
  if (GetKernelVersion() == NULL)
    return false;
  return m_kernel_version.find("Darwin Kernel") != std::string::npos;
}
```

Listing C.4: `HardwareSingleStep()` from `source/Plugins/Process/Utility/RegisterContextDarwin_i386.cpp` [LLDB]

```cpp
bool RegisterContextDarwin_i386::HardwareSingleStep(bool enable) {
  if (ReadGPR(false) != 0)
    return false;

  const uint32_t trace_bit = 0x100u;
  if (enable) {
    // If the trace bit is already set, there is nothing to do
    if (gpr.eflags & trace_bit)
      return true;
    else
      gpr.eflags |= trace_bit;
  } else {
    // If the trace bit is already cleared, there is nothing to do
    if (gpr.eflags & trace_bit)
      gpr.eflags &= ~trace_bit;
    else
      return true;
  }

  return WriteGPR() == 0;
}
```

Appendix D

Excerpts from LLDBagility sources

This appendix contains selected source code excerpts from LLDBagility 1.0.0[1].

Listing D.1: The example KDPClient class for retrieving the kernel version from a KDP server, as defined in kdputils/examples/kdpclient.py [LLDBagility]

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
import argparse
import socket

import kdputils.protocol
import kdputils.requests
from kdputils.protocol import KDPRequest


class KDPClient:
    def __init__(self, kdpserver_host):
        self.sock_reply = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock_reply.bind(("0.0.0.0", 0))
        _, self.req_reply_port = self.sock_reply.getsockname()

        self.sock_exc = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock_exc.bind(("0.0.0.0", 0))
        _, self.exc_note_port = self.sock_exc.getsockname()

        self.kdpserver_host = kdpserver_host
        self.seqid = 0
        self.sesskey = 0x1337

    def __enter__(self):
        self._reattach()
        self._connect()
        return self

    def __exit__(self, *exc):
        self._reattach()

```

[1]Francesco Cagnin. *LLDBagility 1.0.0 Source*. URL: https://github.com/quarkslab/LLDBagility/tree/v1.0.0.

```python
    def send_req_and_recv_reply(self, reqpkt):
        kdputils.protocol.send(
            self.sock_reply,
            (self.kdpserver_host, kdputils.protocol.KDP_REMOTE_PORT),
            reqpkt,
            self.seqid,
            self.sesskey,
        )
        replypkt, _ = kdputils.protocol.recv(self.sock_reply)
        return replypkt

    def _reattach(self):
        self.seqid = 0
        replypkt = self.send_req_and_recv_reply(
            kdputils.requests.kdp_reattach(self.req_reply_port)
        )
        assert replypkt["is_reply"] and replypkt["request"] == KDPRequest.KDP_REATTACH

    def _connect(self):
        replypkt = self.send_req_and_recv_reply(
            kdputils.requests.kdp_connect(
                self.req_reply_port, self.exc_note_port, b"<o/"
            )
        )
        self.seqid += 1
        assert replypkt["is_reply"] and replypkt["request"] == KDPRequest.KDP_CONNECT

    def get_kernelversion(self):
        replypkt = self.send_req_and_recv_reply(kdputils.requests.kdp_kernelversion())
        assert (
            replypkt["is_reply"] and replypkt["request"] == KDPRequest.KDP_KERNELVERSION
        )
        self.seqid += 1
        return replypkt["version"]


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("host")
    args = parser.parse_args()

    with KDPClient(args.host) as kdpclient:
        kernelversion = kdpclient.get_kernelversion()
        print(kernelversion)
```

Listing D.2: The KDPServer class from `LLDBagility/kdpserver.py` [LLDBagility]

```python
class KDPServer:
    def __init__(self):
        self.sv_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sv_sock.setblocking(False)
        self.sv_sock.bind(("127.0.0.1", 0))

        self._cl_host = None
        self._cl_reply_port = None
        self._cl_exception_port = None
```

```python
        self._cl_exception_seq = None
        self._cl_session_key = 0x1337
        self._cl_connected = False

        self._continue_debug_loop = True

    def _process(self, vm, reqpkt, cl_addr):
        if reqpkt["request"] == KDPRequest.KDP_CONNECT:
            assert self._cl_host and self._cl_reply_port
            self._cl_exception_port = reqpkt["exc_note_port"]
            self._cl_exception_seq = itertools.cycle(range(256))
            self._cl_connected = True
            vm.halt()
            vm.unset_all_breakpoints()
            replypkt = kdputils.replies.kdp_connect(KDPError.KDPERR_NO_ERROR)

        elif reqpkt["request"] == KDPRequest.KDP_DISCONNECT:
            assert self._cl_connected
            self._cl_connected = False
            self._continue_debug_loop = False
            vm.unset_all_breakpoints()
            replypkt = kdputils.replies.kdp_disconnect()

        elif reqpkt["request"] == KDPRequest.KDP_HOSTINFO:
            assert self._cl_connected
            cpus_mask, cpu_type, cpu_subtype = vm.get_host_info()
            replypkt = kdputils.replies.kdp_hostinfo(cpus_mask, cpu_type, cpu_subtype)

        elif reqpkt["request"] == KDPRequest.KDP_VERSION:
            assert self._cl_connected
            version, feature = KDP_VERSION, KDP_FEATURE_BP
            replypkt = kdputils.replies.kdp_version(version, feature)

        elif reqpkt["request"] == KDPRequest.KDP_READREGS:
            assert self._cl_connected
            if reqpkt["flavor"] == x86_THREAD_STATE64:
                regs = vm.read_registers(_STRUCT_X86_THREAD_STATE64)
                replypkt = kdputils.replies.kdp_readregs(KDPError.KDPERR_NO_ERROR, regs)
            elif reqpkt["flavor"] == x86_FLOAT_STATE64:
                raise NotImplementedError
            else:
                regs = {}
                replypkt = kdputils.replies.kdp_readregs(
                    KDPError.KDPERR_BADFLAVOR, regs
                )

        elif reqpkt["request"] == KDPRequest.KDP_WRITEREGS:
            assert self._cl_connected
            if reqpkt["flavor"] == x86_THREAD_STATE64:
                regs = {
                    k: v for k, v in reqpkt.items() if k in _STRUCT_X86_THREAD_STATE64
                }
                vm.write_registers(regs)
                replypkt = kdputils.replies.kdp_writeregs(KDPError.KDPERR_NO_ERROR)
            elif reqpkt["flavor"] == x86_FLOAT_STATE64:
                raise NotImplementedError
```

```
            else:
                replypkt = kdputils.replies.kdp_writeregs(KDPError.KDPERR_BADFLAVOR)

        elif reqpkt["request"] == KDPRequest.KDP_RESUMECPUS:
            assert self._cl_connected
            vm.resume()
            replypkt = kdputils.replies.kdp_resumecpus()

        elif reqpkt["request"] == KDPRequest.KDP_REATTACH:
            assert not self._cl_connected
            self._cl_host, self._cl_reply_port = cl_addr
            replypkt = kdputils.replies.kdp_reattach()

        elif reqpkt["request"] == KDPRequest.KDP_READMEM64:
            assert self._cl_connected
            if reqpkt["nbytes"] > MAX_KDP_DATA_SIZE:
                data = b""
                replypkt = kdputils.replies.kdp_readmem64(
                    KDPError.KDPERR_BAD_NBYTES, data
                )
            else:
                data = vm.read_virtual_memory(reqpkt["address"], reqpkt["nbytes"])
                if len(data) != reqpkt["nbytes"]:
                    replypkt = kdputils.replies.kdp_readmem64(
                        KDPError.KDPERR_BAD_ACCESS, data
                    )
                else:
                    replypkt = kdputils.replies.kdp_readmem64(
                        KDPError.KDPERR_NO_ERROR, data
                    )

        elif reqpkt["request"] == KDPRequest.KDP_WRITEMEM64:
            assert self._cl_connected
            if reqpkt["nbytes"] > MAX_KDP_DATA_SIZE:
                replypkt = kdputils.replies.kdp_writemem64(KDPError.KDPERR_BAD_NBYTES)
            else:
                assert reqpkt["nbytes"] == len(reqpkt["data"])
                vm.write_virtual_memory(reqpkt["address"], reqpkt["data"])
                replypkt = kdputils.replies.kdp_writemem64(KDPError.KDPERR_NO_ERROR)

        elif reqpkt["request"] == KDPRequest.KDP_BREAKPOINT64_SET:
            assert self._cl_connected
            vm.set_soft_exec_breakpoint(reqpkt["address"])
            replypkt = kdputils.replies.kdp_breakpoint64_set(KDPError.KDPERR_NO_ERROR)

        elif reqpkt["request"] == KDPRequest.KDP_BREAKPOINT64_REMOVE:
            assert self._cl_connected
            vm.unset_soft_breakpoint(reqpkt["address"])
            replypkt = kdputils.replies.kdp_breakpoint64_remove(
                KDPError.KDPERR_NO_ERROR
            )

        elif reqpkt["request"] == KDPRequest.KDP_KERNELVERSION:
            assert self._cl_connected
            kernel_version = vm.get_kernel_version()
            replypkt = kdputils.replies.kdp_kernelversion(kernel_version)
```

```python

        elif reqpkt["request"] == KDPRequest.KDP_EXCEPTION:
            assert self._cl_connected
            assert reqpkt["is_reply"]
            replypkt = None

        else:
            raise NotImplementedError

        return replypkt

    def debug(self, vm):
        # it is implicitly assumed the first two KDP requests received are
        # KDP_REATTACH and KDP_CONNECT (this is always true when LLDB connects)
        while self._continue_debug_loop:
            time.sleep(0.003)

            try:
                # receive a request
                reqpkt, cl_addr = kdputils.protocol.recv(self.sv_sock)
            except socket.error:
                pass
            else:
                # process the request
                replypkt = self._process(vm, reqpkt, cl_addr)
                if replypkt:
                    # send the response
                    cl_addr = (self._cl_host, self._cl_reply_port)
                    kdputils.protocol.send(
                        self.sv_sock, cl_addr, replypkt, reqpkt["seq"], reqpkt["key"]
                    )

            if self._cl_connected and vm.is_state_changed():
                _, exception = vm.state()
                if exception:
                    (exception, code, subcode) = exception
                    reqpkt = kdputils.requests.kdp_exception(
                        n_exc_info=0x1,
                        cpu=0x0,
                        exception=exception,
                        code=code,
                        subcode=subcode,
                    )
                    cl_addr = (self._cl_host, self._cl_exception_port)
                    kdputils.protocol.send(
                        self.sv_sock,
                        cl_addr,
                        reqpkt,
                        next(self._cl_exception_seq),
                        self._cl_session_key,
                    )
```

Listing D.3: KDKutils/1-create-DWARF.sh [LLDBagility]

```bash
#!/usr/bin/env bash
set -e
dirname () { python -c "import os; print(os.path.dirname(os.path.realpath('$0')))"; }
cd "$(dirname "$0")"
```

```
: ${1?"Usage:␣$0␣VARSFILE"}
VARSFILE="$1"
echo "VARSFILE=\"$VARSFILE\""

source "$VARSFILE"

echo "KDKUTILS_SOURCE_KERNEL_DWARF=\"$KDKUTILS_SOURCE_KERNEL_DWARF\""
echo "KDKUTILS_SOURCE_KERNEL_DIEOFFSETS=\"${KDKUTILS_ . . . _DIEOFFSETS[@]}\""
echo "KDKUTILS_GENERATED_KERNEL=\"$KDKUTILS_GENERATED_KERNEL\""

# from the input DWARF file, extract the structures/variables at the specified
    ↪ offsets
DWARFUTILS_SRCDIRECTORY=$(../DWARFutils/parse-dwarf-types-to-c-source.py
    ↪ "$KDKUTILS_SOURCE_KERNEL_DWARF" ${KDKUTILS_SOURCE_KERNEL_DIEOFFSETS[@]} \
    | python -c 'import␣re,␣sys;␣print(re.search("Output␣directory:␣.(.+?).$",␣
    ↪ sys.stdin.read()).group(1))' )
# compile the extracted structures/variables into a new DWARF file
cd "$DWARFUTILS_SRCDIRECTORY" >/dev/null
    command -v clang-format >/dev/null && clang-format -i
    ↪ -style="{AlignConsecutiveDeclarations:␣true}" *.c
    clang -g -x c -shared *.c
    mkdir -p "$(dirname␣"$KDKUTILS_GENERATED_KERNEL")"
    cp "a.out.dSYM/Contents/Resources/DWARF/a.out" "$KDKUTILS_GENERATED_KERNEL"
    rm -r "a.out" "a.out.dSYM"
    file "$KDKUTILS_GENERATED_KERNEL"
cd - >/dev/null
```

Listing D.4: KDKutils/2-fake-DWARF.sh [LLDBagility]

```
#!/usr/bin/env bash
set -e
dirname () { python -c "import␣os;␣
    ↪ print(os.path.dirname(os.path.realpath('$0')))"; }
cd "$(dirname␣"$0")"

: ${1?"Usage:␣$0␣VARSFILE"}
VARSFILE="$1"
echo "VARSFILE=\"$VARSFILE\""

source "$VARSFILE"

echo "KDKUTILS_TARGET_KERNEL=\"$KDKUTILS_TARGET_KERNEL\""
echo "KDKUTILS_TARGET_KERNEL_DWARF=\"$KDKUTILS_TARGET_KERNEL_DWARF\""
echo "KDKUTILS_RELOCATESYMBOLS=\"$KDKUTILS_RELOCATESYMBOLS\""

# update the UUID of the generated DWARF so that it matches the UUID of the
    ↪ kernel to debug
DEBUGGEEKERNEL_UUID=$(dwarfdump -u "$KDKUTILS_TARGET_KERNEL" | python -c 'import␣
    ↪ re,␣sys;␣print(re.match(r"UUID:␣(.+?)␣",␣sys.stdin.read()).group(1))')
./set-macho-uuid.py "$KDKUTILS_TARGET_KERNEL_DWARF" "$DEBUGGEEKERNEL_UUID"

# relocate the "__TEXT", "__DATA" and "__LINKEDIT" segments of the generated
    ↪ DWARF so that
# their location matches the location of the same segments of the kernel to debug
vmaddr () {
    SEGNAME="$1"
    otool -l "$KDKUTILS_TARGET_KERNEL" | grep -A2 "segname␣$SEGNAME" | head -n 3
    ↪ | python -c 'import␣re,␣sys;␣print(re.search(r"vmaddr␣(0x[0-9a-f]+)",␣
    ↪ sys.stdin.read()).group(1))'
```

```
}
vmsize () {
    SEGNAME="$1"
    otool -l "$KDKUTILS_TARGET_KERNEL" | grep -A2 "segname␣$SEGNAME" | head -n 3
     ↪ | python -c 'import␣re,␣sys;␣print(re.search(r"vmsize␣(0x[0-9a-f]+)",␣
     ↪ sys.stdin.read()).group(1))'
}
./set-segments-vmaddr-and-vmsize.py "$KDKUTILS_TARGET_KERNEL_DWARF" \
    --text      "$(vmaddr␣__TEXT),$(vmsize␣__TEXT)" \
    --data      "$(vmaddr␣__DATA),$(vmsize␣__DATA)" \
    --linkedit  "$(vmaddr␣__LINKEDIT),$(vmsize␣__LINKEDIT)"

# relocate the symbols in the generated DWARF so that their location matches the
     ↪ location
# of the same symbols in the symbol table of the kernel to debug
relocate () {
    SYMBOL="$1"
    ADDRESS=$(nm "$KDKUTILS_TARGET_KERNEL" \
        | grep -E "␣_$SYMBOL\$" \
        | echo "0x$(awk␣'{print␣$1;}')")
    ../DWARFutils/relocate-dwarf-variable.py "$KDKUTILS_TARGET_KERNEL_DWARF"
     ↪ "$SYMBOL" "$ADDRESS"
}
for SYMBOL in "${KDKUTILS_RELOCATESYMBOLS[@]}"
do
    relocate "$SYMBOL"
done
```

Listing D.5: KDKutils/3-attach-DWARF.sh [LLDBagility]

```
#!/usr/bin/env bash
set -e
dirname () { python -c "import␣os;␣
     ↪ print(os.path.dirname(os.path.realpath('$0')))"; }
cd "$(dirname␣"$0")"

: ${1?"Usage:␣$0␣VARSFILE"}
VARSFILE="$1"
echo "VARSFILE=\"$VARSFILE\""

source "$VARSFILE"

echo "KDKUTILS_TARGET_KERNEL=\"$KDKUTILS_TARGET_KERNEL\""
echo "KDKUTILS_TARGET_KERNEL_DWARF=\"$KDKUTILS_TARGET_KERNEL_DWARF\""
echo "KDKUTILS_LLDBMACROS=\"$KDKUTILS_LLDBMACROS\""
echo "LLDBAGILITY_VMNAME=\"$LLDBAGILITY_VMNAME\""

# attach and debug the VM
env PATH="/usr/bin:/bin:/usr/sbin:/sbin" LOGLEVEL="DEBUG" lldb \
    -o "target␣create␣\"$KDKUTILS_TARGET_KERNEL\"" \
    -o "target␣symbols␣add␣\"$KDKUTILS_TARGET_KERNEL_DWARF\"" \
    -o "fdp-attach␣$LLDBAGILITY_VMNAME" \
    -o "command␣script␣import␣\"$KDKUTILS_LLDBMACROS\"" \
    -o "showversion"
```

Glossary

address sanitisation a technique to dynamically detect memory corruption bugs, such as use-after-free and out-of-bounds accesses to heap and stack, based on compiler instrumentation.

address space layout randomisation a technique for hindering the exploitation of memory corruption vulnerabilities by randomising the memory location of key data areas, such as the position of the stack, heap and the base of the executable.

binary a computer file that is not a text file, in some contexts used as synonym for executable.

boot-arg an Extensible Firmware Interface (EFI) firmware variable stored in NV-RAM, used to configure the system boot.

device file an interface to a device driver implemented as an ordinary file, so to be interacted with regular input/output system calls.

device driver a computer program for controlling a device attached to the computer, allowing to access the device functionalities without knowing how they are implemented in hardware.

DTrace a dynamic tracing framework to instrument the kernel and troubleshoot problems on production systems in real time.

DWARF a standardized debugging data format, used to store information about a compiled computer program for use by debuggers.

exception an error condition in the CPU occurring while this executes an instruction, such as division by zero.

executable a file containing a computer program, often encoded in machine language.

Extended Page Tables Intel's implementation of the Second Level Address Translation (SLAT), a hardware-assisted virtualisation technology for accelerating the translation of guest physical memory addresses to host physical addresses.

Extensible Firmware Interface a partition on a data storage device containing the bootloaders and applications to be launched at system boot by the Unified Extensible Firmware Interface (UEFI) firmware.

Fast Debugging Protocol an API for virtual machine introspection and debugging. Described in section 4.2.

hypervisor a computer program that creates and manages the execution of virtual machines. Described in section 2.2.

Internet Protocol the principal communication protocol used in the Internet.

interrupt an input signal to the CPU indicating the occurrence of an event.

kernel the core of an operating system, which controls everything that runs in the system by managing directly the hardware resources and allocating them to running processes.

kernel space the memory area where the kernel execute.

Kernel Debug Kit a collection of useful material for XNU debugging. Described in section 3.2.

Kernel Debugging Protocol the remote kernel debugging mechanism implemented in XNU. Described in section 3.2.

kext a macOS bundle containing additional code to be loaded into the kernel at run time, without the need to recompile or relink.

LLDB the debugger component of the LLVM project. Described in section 2.1.4.

lldbmacros a set of scripts for debugging Darwin kernels in LLDB. Described in section 3.2.1.

Mach-O a file format for executables, object code, shared libraries, dynamically-loaded code, and core dumps.

non-maskable interrupt a hardware interrupt ignored by standard masking techniques.

non-volatile random-access memory random-access memory that retains data even without a power supply.

random-access memory a type of computer memory in which items can be read or written in almost the same amount of time regardless of their physical location in the memory chip.

software development kit a collection of software development tools in one installable package.

superuser a special user account in possess of the highest privileges necessary for system administration, commonly referred to as 'root'.

System Integrity Protection a security mechanism for limiting the power of the superuser in macOS. Described in section 2.3.1.

system call a mechanism implemented by operating system kernels to allow processes to interface with the OS and request for services.

translation lookaside buffer a cache that stores recent translations of virtual to physical memory addresses.

trap an exception that is reported immediately after the execution of the trapping instruction.

universally unique identifier a 128-bit number used to identify information in computer systems, typically generated in such a way that the probability it will be a duplicate is close enough to zero to be negligible.

Unix-like any operating system either explicitly based on Unix or behaving similarly to it.

use-after-free a class of memory corruption bugs that involves a computer program using a memory area after this has been already freed.

user space the memory area where applications (e.g. user processes) execute.

User Datagram Protocol a connectionless, message-oriented protocol for communications over IP.

virtual machine introspection a technique for monitoring the state of a running system-level VM. Described in section 2.2.1.

virtual machine (system-level) a virtual representation of a real computer system. Described in section 2.2.

watchdog timer a hardware timer that automatically generates a system reset if it's not reset periodically.

x86 a family of complex instruction set architectures with variable instruction length, developed by Intel starting with the 8086 and 8088 microprocessors.

x86-64 the 64-bit version of the x86 instruction set.

XNU the kernel of the macOS and Darwin operating systems, among others. Short for 'X is Not Unix'. Described in section 2.3.

Acronyms

API application programming interface.

CPU central processing unit.

CVE Common Vulnerabilities and Exposures.

EFI Extensible Firmware Interface.

EPT Extended Page Tables.

EULA end-user license agreement.

FDP Fast Debugging Protocol.

GUI graphical user interface.

IP Internet Protocol.

KDK Kernel Debug Kit.

KDP Kernel Debugging Protocol.

MAC media access control.

NMI non-maskable interrupt.

NVRAM non-volatile random-access memory.

OS operating system.

PoC proof of concept.

RAM random-access memory.

RSP remote serial protocol.

SDK software development kit.

SIP System Integrity Protection.

TLB translation lookaside buffer.

UDP User Datagram Protocol.

UUID universally unique identifier.

VM virtual machine.

VMI virtual machine introspection.

VMM virtual machine monitor.

Online sources

[1] National Museum of American History. *Computer Oral History Collection, 1969-1973, 1977. Jean J. Bartik and Frances E. (Betty) Snyder Holberton Interview*. URL: https://amhistory.si.edu/archives/AC0196_bart730427.pdf (visited on 26/11/2019) (see p. 4).

[2] *Apache License, Version 2.0*. URL: https://www.apache.org/licenses/LICENSE-2.0 (visited on 13/01/2020) (see p. 33).

[3] Apple. *About macOS Recovery*. Published on 12 October 2018. URL: https://support.apple.com/en-us/HT201314 (visited on 21/09/2019) (see p. 11).

[4] Apple. *About System Integrity Protection on your Mac*. Published on 25 September 2019. URL: https://support.apple.com/en-us/HT204899 (visited on 16/01/2020) (see p. 10).

[5] Apple. *About the security content of macOS Mojave 10.14.6 Supplemental Update*. URL: https://support.apple.com/en-in/HT210548 (visited on 08/02/2020) (see p. 49).

[6] Apple. *Apple Open Source*. URL: https://opensource.apple.com (visited on 02/02/2020) (see p. 10).

[7] Apple. *IONetworkingFamily 129.200.1 Source*. URL: https://opensource.apple.com/source/IONetworkingFamily/IONetworkingFamily-129.200.1/IOKernelDebugger.cpp.auto.html (visited on 09/02/2020) (see p. 14).

[8] Apple. *Kernel Programming Guide. Building and Debugging Kernels*. Last updated on 8 August 2013. URL: https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/build/build.html (visited on 18/09/2019) (see pp. 20, 26, 27).

[9] Apple. *Kernel Programming Guide. Kernel Architecture Overview*. Last updated on 8 August 2013. URL: https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/Architecture/Architecture.html (visited on 19/09/2019) (see p. 10).

[10] Apple. *Kernel Programming Guide. Security Considerations*. Last updated on 8 August 2013. URL: https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/security/security.html (visited on 07/05/2019) (see p. 28).

[11] Apple. *Kernel Programming Guide. Mach Overview*. Last updated on 8 August 2013. URL: https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/Mach/Mach.html (visited on 29/11/2019) (see p. 6).

[12] Apple. *LLDB Quick Start Guide. About LLDB and Xcode*. Last updated on 18 September 2013. URL: https://developer.apple.com/library/archive/documentation/IDEs/Conceptual/gdb_to_lldb_transition_guide/document/Introduction.html (visited on 07/05/2019) (see p. 7).

[13] Apple. *LLVM Compiler Overview*. Last updated on 13 December 2012. URL: https://developer.apple.com/library/archive/documentation/CompilerTools/Conceptual/LLVMCompilerOverview/ (visited on 07/05/2019) (see p. 7).

[14] Apple. *More Software Downloads - Apple Developer*. URL: https://developer.apple.com/download/more/?=Kernel%5C%20Debug%5C%20Kit (visited on 20/09/2019) (see p. 18).

[15] Apple. *Reset NVRAM or PRAM on your Mac*. Published on 8 November 2018. URL: https://support.apple.com/en-us/HT204063 (visited on 21/09/2019) (see p. 11).

[16] Apple. *System Integrity Protection Guide*. Last updated on 16 September 2015. URL: https://developer.apple.com/library/archive/documentation/Security/Conceptual/System_Integrity_Protection_Guide/ (visited on 21/09/2019) (see p. 10).

[17] Apple. *Technical Note TN2339: Building from the Command Line with Xcode FAQ*. Last updated on 19 June 2017. URL: https://developer.apple.com/library/archive/technotes/tn2339/_index.html (visited on 03/01/2020) (see p. 7).

[18] Apple.
Technical Q&A QA1264: Generating a Non-Maskable Interrupt (NMI).
Last updated on 04 June 2013. URL: https://developer.apple.com/library/archive/qa/qa1264/_index.html (visited on 27/01/2020) (see p. 26).

[19] Apple. *XNU 1456.1.26 Source*. Browsable mirror at https://github.com/apple/darwin-xnu/tree/xnu-1456.1.26/. URL: https://opensource.apple.com/source/xnu/xnu-1456.1.26/ (visited on 27/08/2019) (see p. 15).

[20] Apple. *XNU 1699.26.8 Source*. Browsable mirror at https://github.com/apple/darwin-xnu/tree/xnu-1699.26.8/. URL: https://opensource.apple.com/source/xnu/xnu-1699.26.8/ (visited on 20/08/2019) (see p. 27).

[21] Apple. *XNU 2050.7.9 Source*. Browsable mirror at https://github.com/apple/darwin-xnu/tree/xnu-2050.7.9/. URL: https://opensource.apple.com/source/xnu/xnu-2050.7.9/ (visited on 20/08/2019) (see p. 19).

[22] Apple. *XNU 4903.221.2 Source*. Browsable mirror at https://github.com/apple/darwin-xnu/tree/xnu-4903.221.2/. URL: https://opensource.apple.com/source/xnu/xnu-4903.221.2/ (visited on 20/08/2019) (see pp. 13, 63, 65).

[23] Apple. *XNU 4903.241.1 Source*.
URL: https://opensource.apple.com/source/xnu/xnu-4903.241.1/ (visited on 18/11/2019) (see p. 13).

[24] Apple. *XNU 6153.11.26 Source*.
URL: https://opensource.apple.com/tarballs/xnu/xnu-6153.11.26.tar.gz (visited on 17/02/2020) (see p. 13).

[25] Jeremy Bennett. *Howto: GDB Remote Serial Protocol*.
URL: https://www.embecosm.com/appnotes/ean4/embecosm-howto-rsp-server-ean4-issue-2.html (visited on 01/02/2020).

[26] Francesco Cagnin. *An overview of macOS kernel debugging*.
Posted on 7 May 2019. URL: https://blog.quarkslab.com/an-overview-of-macos-kernel-debugging.html (visited on 30/10/2019) (see p. 31).

[27] Francesco Cagnin. *LLDBagility 1.0.0 Source*. Released on 18 June 2019.
URL: https://github.com/quarkslab/LLDBagility/tree/v1.0.0 (visited on 30/10/2019) (see pp. 31, 75).

[28] Francesco Cagnin. *LLDBagility: practical macOS kernel debugging*.
Posted on 18 June 2019.
URL: https://blog.quarkslab.com/lldbagility-practical-macos-kernel-debugging.html (visited on 30/10/2019) (see p. 31).

[29] *Clang: a C language family frontend for LLVM*.
URL: https://clang.llvm.org (visited on 22/10/2019) (see p. 47).

[30] The MITRE Corporation. *CVE 2018 entries*.
URL: https://cve.mitre.org/data/downloads/allitems-cvrf-year-2018.xml (visited on 06/11/2019) (see p. 3).

[32] Zach Cutlip. *Source Level Debugging the XNU Kernel*.
Posted on 24 October 2018.
URL: https://shadowfile.inode.link/blog/2018/10/source-level-debugging-the-xnu-kernel/ (visited on 27/01/2020)
(see pp. 18, 26, 28).

[33] *ddb(4) - NetBSD Manual Pages*. URL:
https://netbsd.gw.com/cgi-bin/man-cgi?ddb+4+NetBSD-current
(visited on 24/01/2020) (see p. 27).

[34] *ddb(4) - OpenBSD manual pages*.
URL: https://man.openbsd.org/ddb (visited on 24/01/2020)
(see p. 27).

[35] *Debugging with GDB: Remote Stub*. URL: https://sourceware.org/gdb/current/onlinedocs/gdb/Remote-Stub.html
(visited on 29/01/2020).

[36] Damien DeVille. *Kernel debugging with LLDB and VMware Fusion*.
Posted on 15 August 2015. URL: http://ddeville.me/2015/08/kernel-debugging-with-lldb-and-vmware-fusion (visited on 22/08/2019)
(see pp. 22, 26).

[37] Damien DeVille.
Using the VMware Fusion GDB stub for kernel debugging with LLDB.
Posted on 18 August 2015.
URL: http://ddeville.me/2015/08/using-the-vmware-fusion-gdb-stub-for-kernel-debugging-with-lldb (visited on 28/01/2020)
(see p. 28).

[38] *Driver x64 Restrictions*. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/driver-x64-restrictions
(visited on 01/02/2020) (see p. 33).

[39] eskimo. *Re: debugging kernel drivers*. Posted on 16 July 2015.
URL: https://forums.developer.apple.com/message/28317#27581
(visited on 07/05/2019) (see p. 7).

[40] eskimo. *Re: Where can I find Kernel Debug Kit for 10.11.6 (15G22010)?*
Posted on 6 March 2019.
URL: https://forums.developer.apple.com/thread/108732#351881
(visited on 07/05/2019) (see p. 18).

[41] Etnus. *MIPS Delay Slot Instructions*.
URL: http://www.jaist.ac.jp/iscenter-new/mpc/old-machines/altix3700/opt/toolworks/totalview.6.3.0-1/doc/html/ref_guide/MIPSDelaySlotInstructions.html (visited on 28/11/2019) (see p. 6).

[42] Landon Fuller. *Mach Exception Handlers*.
URL: https://www.mikeash.com/pyblog/friday-qa-2013-01-11-mach-exception-handlers.html (visited on 29/11/2019) (see p. 6).

[43] *GCC, the GNU Compiler Collection*.
URL: https://gcc.gnu.org (visited on 22/10/2019) (see p. 47).

[44] *GDB: The GNU Project Debugger*.
URL: https://www.gnu.org/software/gdb/ (visited on 02/02/2020) (see p. 5).

[45] GeoSn0w. *Debugging macOS Kernel For Fun*. Posted on 2 December 2018.
URL:
https://geosn0w.github.io/Debugging-macOS-Kernel-For-Fun/
(visited on 11/09/2019) (see p. 20).

[46] *Getting Started with WinDbg (Kernel-Mode)*.
URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/getting-started-with-windbg--kernel-mode- (visited on 20/01/2020) (see p. 13).

[47] *GNU General Public License, version 2*. URL:
https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html
(visited on 15/02/2020) (see p. 10).

[48] LLVM Developer Group. *LLDB 8.0.0 Source*. Browsable mirror at https://github.com/llvm/llvm-project/tree/llvmorg-8.0.0/lldb/.
URL: http://releases.llvm.org/8.0.0/lldb-8.0.0.src.tar.xz
(visited on 16/10/2019) (see pp. 63, 71).

[49] The Open Group.
Mac OS X Version 10.5 Leopard on Intel-based Macintosh computers. URL:
https://www.opengroup.org/openbrand/register/brand3555.htm
(visited on 08/11/2019) (see p. 10).

[50] The Open Group.
macOS version 10.15 Catalina on Intel-based Mac computers. URL:
https://www.opengroup.org/openbrand/register/brand3653.htm
(visited on 08/11/2019) (see p. 10).

[51] *Issue 1806: XNU: Use-after-free due to stale pointer left by in6_pcbdetach*.
URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=1806 (visited on 08/02/2020) (see p. 49).

[52] Scott Knight. *macOS Kernel Debugging*. Posted on 15 August 2018.
URL: https://knight.sc/debugging/2018/08/15/macos-kernel-debugging.html (visited on 11/09/2019) (see p. 20).

[53] *KVM*. URL: https://www.linux-kvm.org/page/Main_Page (visited on 09/02/2020) (see p. 34).

[54] Xiang Lei. *XNU kernel debugging via VMWare Fusion*.
Last updated on 19 February 2014. URL:
http://trineo.net/p/17/06_debug_xnu.html (visited on 30/01/2020) (see pp. 20, 22).

[55] Jonathan Levin. *jtool*.
URL: http://www.newosxbook.com/tools/jtool.html (visited on 15/02/2020).

[56] LightBulbOne. *Introduction to macOS Kernel Debugging*. Posted on 4 October 2016. URL: https://lightbulbone.com/posts/2016/10/intro-to-macos-kernel-debugging/ (visited on 11/09/2019) (see p. 20).

[57] Kedy Liu. *Debugging macOS Kernel using VirtualBox*. Posted on 10 April 2017. URL: https://klue.github.io/blog/2017/04/macos_kernel_debugging_vbox/ (visited on 22/08/2019) (see p. 22).

[58] John Lockwood. *OSX installing on virtual machine (legal issues)*. Posted on 30 October 2015. URL: https://discussions.apple.com/thread/7312791?answerId=29225237022#29225237022 (visited on 22/08/2019) (see p. 22).

[59] *MacPmem - OS X Physical Memory Access*. URL: https://github.com/google/rekall/tree/master/tools/osx/MacPmem (visited on 03/02/2020) (see p. 40).

[60] Alexander B. Magoun and Paul Israel. *Did You Know? Edison Coined the Term "Bug"*. URL: https://spectrum.ieee.org/the-institute/ieee-history/did-you-know-edison-coined-the-term-bug (visited on 06/11/2019) (see p. 3).

[61] *man page kextcache section 8*. URL: http://www.manpagez.com/man/8/kextcache/ (visited on 27/01/2020) (see p. 21).

[62] Max108. *Enabling parts of System Integrity Protection while disabling specific parts?* Posted on 14 September 2015. URL: https://forums.developer.apple.com/thread/17452#thread-message-52814 (visited on 07/05/2019) (see p. 11).

[63] Spencer Michaels. *xendbg*. URL: https://github.com/nccgroup/xendbg (visited on 03/02/2020) (see p. 9).

[65] *New LLVM Project License Framework*. URL: https://llvm.org/docs/DeveloperPolicy.html%5C#new-llvm-project-license-framework (visited on 02/02/2020) (see p. 7).

[66] *On-Line Kernel Debugging Using DDB*. URL: https://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/kerneldebug-online-ddb.html (visited on 24/01/2020) (see p. 27).

[67] *Oracle VM VirtualBox*. URL: https://www.virtualbox.org (visited on 02/02/2020) (see p. 9).

[68] *Poor performance in visual mode during remote debugging via gdbserver*. URL: https://github.com/radareorg/radare2/issues/3808 (visited on 29/01/2020) (see p. 28).

[70] *Quarkslab*. URL: https://www.quarkslab.com (visited on 09/02/2020) (see p. 31).

[71] *Remote Debugging*. URL: https://lldb.llvm.org/use/remote.html (visited on 03/02/2020).

[73] RedNaga Security. *Remote Kext Debugging*. Posted on 9 April 2017. URL: https://rednaga.io/2017/04/09/remote_kext_debugging/ (visited on 11/09/2019) (see p. 20).

[75] snare. *Debugging the Mac OS X kernel with VMware and GDB*. Posted on 14 February 2012. URL: http://ho.ax/posts/2012/02/debugging-the-mac-os-x-kernel-with-vmware-and-gdb/ (visited on 22/08/2019) (see p. 22).

[76] snare. *VMware debugging II: "Hardware" debugging*. Posted on 18 February 2012. URL: http://ho.ax/posts/2012/02/vmware-hardware-debugging/ (visited on 29/01/2020) (see p. 28).

[77] Mathieu Tarral. *pyvmidbg*. URL: https://github.com/Wenzel/pyvmidbg (visited on 03/02/2020) (see p. 9).

[78] *The LLDB Debugger*. URL: https://lldb.llvm.org (visited on 02/02/2020) (see p. 7).

[79] *VMware*. URL: https://www.vmware.com (visited on 02/02/2020) (see p. 28).

[80] Jason Wessel. *Using kgdb, kdb and the kernel debugger internals*. URL: https://www.kernel.org/doc/html/v4.17/dev-tools/kgdb.html (visited on 20/01/2020) (see p. 13).

[81] *Why is debugging of native shared libraries used by Android apps slow?* Posted on 10 November 2011. URL: https://stackoverflow.com/questions/8051458/why-is-debugging-of-native-shared-libraries-used-by-android-apps-slow (visited on 29/01/2020) (see p. 28).

[82] Ned Williamson. *SockPuppet: A Walkthrough of a Kernel Exploit for iOS 12.4*. Published on 10 December 2019. URL: https://googleprojectzero.blogspot.com/2019/12/sockpuppet-walkthrough-of-kernel.html (visited on 08/02/2020) (see p. 49).

[83] *Xen Project*. URL: https://xenproject.org (visited on 09/02/2020) (see p. 34).

Printed sources

[31] Nicolas Couffin.
Winbagility: Débogage furtif et introspection de machine virtuelle.
Project website at https://winbagility.github.io. 2016
(see pp. 28, 33, 34).

[64] Charlie Miller et al. *iOS Hacker's Handbook*. John Wiley & Sons, 2012
(see p. 14).

[69] Gerald J Popek and Robert P Goldberg.
'Formal requirements for virtualizable third generation architectures'.
In: *Communications of the ACM* 17.7 (1974), pp. 412–421 (see p. 8).

[72] Gene Sally. *Pro Linux embedded systems*. 2010 (see p. 28).

[74] Amit Singh. *Mac OS X internals: a systems approach*.
Addison-Wesley Professional, 2006 (see pp. 6, 14, 27).