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

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 Kernel Debugging Protocol (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 **??**, 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.

1.1 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 **??**), 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 1.1 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 1.1: 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 1.2, 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 **??**.
- 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 1.2: 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]

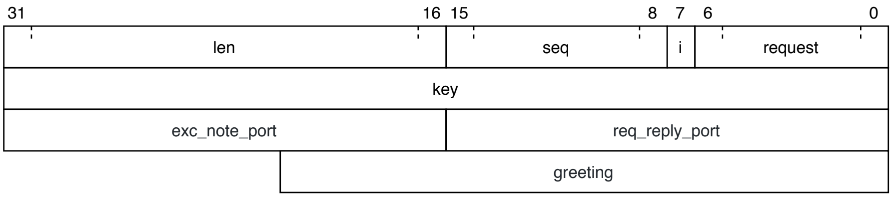
[11] Apple. *XNU 1456.1.26 Source*. URL: https://opensource.apple.com/source/xnu/xnu-1456.1.26/.

[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 1.1: 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 1.3 as an example for KDP_READMEM64 packets.

Listing 1.3: 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 **??**.

1.1.1 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 1.2) 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 1.4: 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 operating system (OS) execution.

Listing 1.5: 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 System Integrity Protection (SIP) is disabled, see **??**).

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.

1.2 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 1.2.1), 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 1.6: 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.

1.2.1 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 1.7: 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]

1.3 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 1.8: 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 non-volatile random-access memory (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 1.9. Proper values for `kcsuffix` are typically 'development', 'debug' or 'kasan'; valid bitmasks for debug were listed in listing 1.1. 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 1.9: 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:

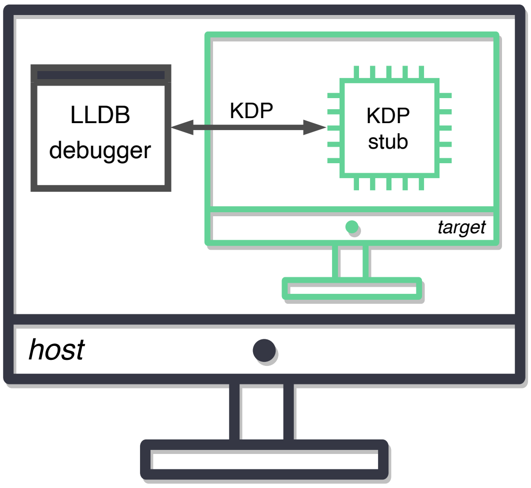
- All resources cited above suggest to disable SIP, but this is not required at all: installing the debug binaries and setting boot-args can be done directly from macOS Recovery without disabling SIP for macOS (operation that requires a reboot into macOS Recovery anyway).
- Similarly, no resource points 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 1.1.1), 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 1.2: 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 virtual machines (VMs) are much easier to set up, debug, reuse and dispose than physical machines. The process is illustrated in fig. 1.2. 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.

1.4 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

[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.

machines installed the Kernel Debug Kit build 18F132. The target machine was configured as explained in section 1.3: 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.

1.5 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 1.1.1), 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 1.10: 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 1.11: 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 1.2.

1.6 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.

1.6.1 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].

1.6.2 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.

1.6.3 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 **??**), no matter the guest OS. The process is represented in fig. 1.3. 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].

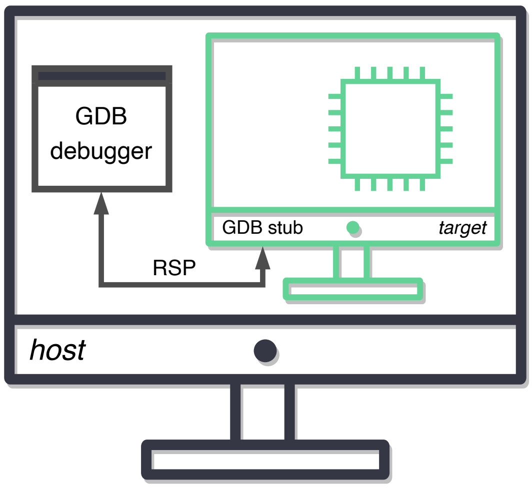
[46] Apple. *Kernel Programming Guide. Security Considerations*. URL: https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/security/security.html.

[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 1.3: 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.



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 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.

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

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 application programming interface (API) for virtual machine introspection and debugging.

hypervisor a computer program that creates and manages the execution of virtual machines.

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.

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

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.

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

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

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

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

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

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

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.

virtual machine (system-level) a virtual representation of a real computer system.

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'.

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.

RSP remote serial protocol.

SIP System Integrity Protection.

TLB translation lookaside buffer.

UDP User Datagram Protocol.

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).

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

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

[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).

[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).

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

[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. 2).

[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. 8, 14, 15).

[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).

[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. 16).

[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).

[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).

[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).

[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. 6).

[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).

[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).

[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).

[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. 14).

[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. 3).

[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. 15).

[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. 7).

[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 p. 1).

[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. 1).

[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. 1).

[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).

[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).

[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).

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

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

[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. 6, 14, 16).

[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. 15).

[34] *ddb(4) - OpenBSD manual pages*. URL: https://man.openbsd.org/ddb (visited on 24/01/2020) (see p. 15).

[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. 10, 14).

[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. 16).

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

[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).

[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. 6).

[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).

[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).

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

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

[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. 8).

[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. 1).

[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).

[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).

[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).

[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).

[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).

[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. 8).

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

[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. 8, 10).

[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. 8).

[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. 10).

[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. 10).

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

[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).

[61] *man page kextcache section 8*. URL: http://www.manpagez.com/man/8/kextcache/ (visited on 27/01/2020) (see p. 9).

[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).

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

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

[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. 15).

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

[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. 16).

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

[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. 8).

[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. 10).

[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. 16).

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

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

[79] *VMware*. URL: https://www.vmware.com (visited on 02/02/2020) (see p. 16).

[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. 1).

[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. 16).

[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).

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

Printed sources

[31] Nicolas Couffin.
Winbagility: Débogage furtif et introspection de machine virtuelle.
Project website at `https://winbagility.github.io`. 2016 (see p. 16).

[64] Charlie Miller et al. *iOS Hacker's Handbook*. John Wiley & Sons, 2012 (see p. 2).

[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.

[72] Gene Sally. *Pro Linux embedded systems*. 2010 (see p. 16).

[74] Amit Singh. *Mac OS X internals: a systems approach*.
Addison-Wesley Professional, 2006 (see pp. 2, 15).