A

`disarm(j)` - the missing manual page

This appendix provides a tutorial and documentation for using `disarm`. While seemingly a simple CLI tool, `disarm` has quite a few advanced features, which put it on par (and, in many cases, beyond) with expensive commercial disassemblers.

It is the aim of `disarm` to be an all-in one tool for multiple binary formats, in much the same way as `jtool2(j)` was to Mach-O. This was accomplished by refactoring `jtool2` so that its Mach-O functionality was separated from the main, and opening up an interface to register the other file formats (presently, ELF and PE). `jtool2` matched (and often, vastly improved on) Apple's own Mach-O tools (`pagestuff(1)`, `size(1)`, `segedit(1)`, `dyldinfo(1)`, and of course `otool(1)`). Similarly, `disarm` emulates some functions of the mighty `objdump(1)`, `readelf(1)` and Windows' `dumpbin.exe`. This is especially important on platforms where these tools are not installed, notably Android devices.

Note, that `disarm` does not purport to compete or replace with commercial disassemblers developed by crews of people with better funding. It was developed for specific command-line workflows, and has grown immensely useful for its Author, who thought it would be a good idea to freely share it with researchers and other enthusiasts.

1. Basic Information

When faced with a known binary format (Mach-O, ELF or PE), `disarm` can be used to display format-encoded information.

Table A/1-1: `disarm(j)` switches for basic information

Switch	Purpose
`-i`	File format agnostic information
`-I`	File format specific information
`-l`	File format agnostic regions (segments/sections)
`-I`	File format specific regions, and detailed header

1.1. Header/Format

1.1.1. File format agnostic information

Mach-O, ELF and PE all have proprietary headers, but largely encode similar information. Using `-i` will display format agnostic information. That is, irrespective of the file format you can expect (largely) the same output. This makes it useful for `grep(1)`ing individual lines (i.e. attributes):

Output A/1-2(a): Demonstrating `disarm -i` (on a Mach-O binary)

```
/bin/ls File-format agnostic information:
File type:        executable
Format:           Mach-O
Target OS:        macOS 11.0.0
Architecture:     ARM64e (ARMv8.3)
File Size:        0x15af0 (88816) bytes
Linker:           /usr/lib/dyld
Text region(s):   0x37dc-0x7420 (@0x1000037dc-0x100007420) (15428/0x3c44 bytes)
         Entry:   0x3a3c
Data region(s):   0xc000-0x10000 (16384/0x4000 bytes)
String region(s): 0x7a34-0x7f28 (1268/0x4f4 bytes)
Linker stubs:     0x7420-0x7960 (1344/0x540 bytes)
BuildID/UUID:     6C62FCB2-5D18-3692-8775-334D211AEB3C
Binary is digitally signed (use --signature to view)
```

Output A/1-2(b): Demonstrating `disarm -i` (on an ELF binary)

```
samples/ls File-format agnostic information:
File type:        shared library
Format:           ELF
Target OS:        Linux 3.0
Architecture:     x86_64
File Size:        0x285c8 (165320) bytes
Linker:           /lib64/ld-linux-x86-64.so.2
Text region(s):   0x4730-0x17072 (76098/0x12942 bytes)
Data region(s):   0x22000-0x22248 (584/0x248 bytes)
String region(s): 0x10a8-0x16a2 (1530/0x5fa bytes)
Linker stubs:     0x4020-0x4730 (1808/0x710 bytes)
Initializers:     0x20f70-0x20f78
Terminators:      0x20f78-0x20f80
```

Output A/1-2(c): Demonstrating `disarm -i` (on a PE binary)

```
samples/PE/samples/ntoskrnl.exe File-format agnostic information:
File type:        executable
Format:           PE32+
Target OS:        Windows 0.0.0
Architecture:     ARM64
File Size:        0x9f55e8 (10442216) bytes
Text region(s):   0x0-0x133000 (@0x140acd000-0x140c00000) (1257472/0x133000 bytes)
         Entry:   0x9dd2d0
Data region(s):   0x998a00-0xab1364 (1149284/0x118964 bytes)
```

As the above shows, most of the entries are applicable for all file types. Some particular fields do vary by binary, as in the above ELF binary that has initializers/terminators, or the Mach-O, which is code signed.

1.1.2. File format specific information

Using –I (note uppercase) will display format specific information. This will parse and display individual header fields:

Output A/1-3(a): Demonstrating `disarm –I` (on a Mach-O, matching `jtool2(j)` output)

```
morpheus@eM1nent (~)$ disarm –I /bin/ls
Mach-O header information:
Magic:  64-bit Mach-O
Type:   executable
CPU:    ARM64e (ARMv8.3)
Cmds:   19
Size:   1728
Flags:  0x200085 (NOUNDEFS DYLDLINK TWOLEVEL PIE)
```

Output A/1-3(b): Demonstrating `disarm –I` (on an ELF, matching `readelf –h(1)` output)

```
morpheus@QiLin (~)$ disarm –I /bin/ls
ELF header information:
  Version:                           1
  Type:                              shared object
  Machine:                           Advanced Micro Devices x86-64
  Entry point address:               0x61c0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          163400 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 29

PT_DYNAMIC section @0x219d8, 0x210 bytes (33 entries):
  DT_NEEDED       (Shared library dependency)                 libselinux.so.1
  DT_NEEDED       (Shared library dependency)                 libcap.so.2
        ...
  DT_NULL         (Terminator)                                0x0
```

1.2. Note about Fat Files

Many files in Darwin, particularly in macOS, are fat ("universal") binaries. To directly work on the ARM64e slice, specify ARCH=arm64e in the command line, or (recommended) put it into an exported environment variable. `disarm(j)` intentionally does not support its predecessor's (`jtool2`'s) –arch argument, since it's easier to set the environment variable and forget about the excess fat.

`disarm(j)` can also emulate Darwin's `lipo(1)` to "suck out" the relevant slice, with –e arch. This will automatically take out the active architecture (i.e. the one specified in the ARCH variable). This method can be used to extract any architecture slice, though any subsequent processing requiring disassembly will work only on `arm64[e]`.

Output A/1-4: `disarm(j)`'s handling of fat files

```
#
# When encountering a fat file, if ARCH is not set, disarm complains:
#
morpheus@eM1nent (~) % disarm /bin/ls
Fat file with 2 architectures (x86_64,arm64e). Use ARCH=... to pick desired architecture
morpheus@eM1nent (~) % export ARCH=arm64e
#
# Extract active architecture (lipo(1))
#
morpheus@eM1nent (~) % disarm -e arch /bin/ls
Not performing any fixups!
Extracted to /tmp/ls.arm64e
```

1.3. Segmenting/Sectioning

The `-i` switch will display basic regions of the file - text, data, strings and stubs, as shown above. Sometimes, however, there is a need to get into more detail, as the file may be sectioned (ELF) or segmented (Mach-O) into different areas, which may be mapped to different memory addresses.

1.3.1. File format agnostic segmentation

`disarm(j)` automatically processes the file segmentation - whether from PE sections, ELF PT_LOADs, or Mach-O Load commands - into generalized "regions". Using `-l` (lowercase) will display these regions, keeping the presentation to as file format agnostic form as possible:

Output A/1-5(a): Demonstrating `disarm -l` (on a Mach-O)

```
$ disarm -l __/bin/ls
0x0-0x8000 -> 0x100000000-0x100008000 __TEXT    (TEXT,READ-ONLY, 32768 bytes)
   0x37dc-0x7420 -> 0x1000037dc-0x100007420 __TEXT.__text       (TEXT,READ-ONLY, 15428 bytes)
   0x7420-0x7960 -> 0x100007420-0x100007960 __TEXT.__auth_stubs (TEXT,READ-ONLY, 1344 bytes)
   0x7960-0x7a34 -> 0x100007960-0x100007a34 __TEXT.__const      (DATA,READ-ONLY, 212 bytes)
0x7a34-0x7f28 -> 0x100007a34-0x100007f28 __TEXT.__cstring       (1268 bytes)
   0x7f28-0x8000 -> 0x100007f28-0x100008000 __TEXT.__unwind_info        (TEXT,READ-ONLY, 216
0x8000-0xc000 -> 0x100008000-0x10000c000 __DATA_CONST   (DATA, 16384 bytes)
        0x8000-0x82a0 -> 0x100008000-0x1000082a0 __DATA_CONST.__auth_got        (DATA, 672 by
        0x82a0-0x82d0 -> 0x1000082a0-0x1000082d0 __DATA_CONST.__got     (DATA, 48 bytes)
        0x82d0-0x8538 -> 0x1000082d0-0x100008538 __DATA_CONST.__const   (DATA, 616 bytes)
0xc000-0x10000 -> 0x10000c000-0x100010000 __DATA        (DATA, 16384 bytes)
        0xc000-0xc020 -> 0x10000c000-0x10000c020 __DATA.__data  (DATA, 32 bytes)
0x10000-0x15af0 -> 0x100010000-0x100015af0 __LINKEDIT   (DATA,READ-ONLY, 23280 bytes)
0x104a0-0x104f0 -> 0x1000104a0-0x1000104f0 function starts      (80 bytes)
0x104f0-0x10ab0 -> 0x1000104f0-0x100010ab0 symbol table (1472 bytes)
0x10ab0-0x10d68 -> 0x100010ab0-0x100010d68 indirect symbols     (696 bytes)
0x10d68-0x11150 -> 0x100010d68-0x100011150 string table (1000 bytes)
0x11150-0x15af0 -> 0x100011150-0x100015af0 code signature       (18848 bytes)
```

Output A/1-5(b): Demonstrating `disarm -l` (on a PE)

```
$ disarm -l /samples/PE/ntdll.dll
0x0400-0xdc00 → 0x180001000-0x18000e800 .hexpthk         (TEXT, 55296 bytes)
0xdc00-0x2b3400 → 0x18000f000-0x1802b4800 .text          (TEXT, 2775040 bytes)
0x2b3400-0x2b4c00 → 0x1802b5000-0x1802b6800 PAGE         (TEXT, 6144 bytes)
0x2b4c00-0x2b7000 → 0x1802b7000-0x1802b9400 RT           (TEXT, 9216 bytes)
0x2b7000-0x2b8000 → 0x1802ba000-0x1802bb000 fothk        (TEXT, 4096 bytes)
0x2b8000-0x33d600 → 0x1802bb000-0x180340600 .rdata       (546304 bytes)
0x33d600-0x33fc00 → 0x180341000-0x180343600 .data        (DATA, 9728 bytes)
0x33fc00-0x369400 → 0x180352000-0x18037b800 .pdata       (169984 bytes)
0x369400-0x373000 → 0x18037c000-0x180385c00 .mrdata      (39936 bytes)
0x373000-0x373200 → 0x180386000-0x180386200 .00cfg       (512 bytes)
0x373200-0x378000 → 0x180387000-0x18038be00 .a64xrm      (19968 bytes)
0x378000-0x3ed200 → 0x18038c000-0x180401200 .rsrc        (479744 bytes)
0x3ed200-0x3ee600 → 0x180402000-0x180403400 .reloc       (5120 bytes)
```

1.3.2. File format specific segmentation

To see the file format specific sections, use –L. As one can expect, output here will be significantly different, with `disarm` emulating `jtool2(j)`, `objdump(1)/readelf(1)` or even Windows' `dumpbin.exe`, according to the binary type. It is recommended to use –v here for more verbosity:

Output A/1-6(a): Demonstrating `disarm –L –v` on a Mach-O (matching `jtool2(j)` output)

```
morpheus@eM1nent (~)% disarm –L –v /bin/ls
LC  0: LC_SEGMENT_64  Mem: 0x000000000–0x100000000  File: Not Mapped  ---/---  __PAGEZERO
LC  1: LC_SEGMENT_64  Mem: 0x100000000–0x100008000  File: 0x0–0x8000  r–x/r–x  __TEXT
   Mem: 0x1000037dc–0x100007420 File: 0x000037dc–0x00007420      __TEXT.__text  (Normal)
   Mem: 0x100007420–0x100007960 File: 0x00007420–0x00007960      __TEXT.__auth_stubs    (Symb
   Mem: 0x100007960–0x100007a34 File: 0x00007960–0x00007a34      __TEXT.__const
   Mem: 0x100007a34–0x100007f28 File: 0x00007a34–0x00007f28      __TEXT.__cstring       (C–St
   Mem: 0x100007f28–0x100008000 File: 0x00007f28–0x00008000      __TEXT.__unwind_info
...
LC  4: LC_SEGMENT_64    Mem: 0x100010000–0x100018000    File: 0x10000–0x15af0   r––/r–– __LIN
LC  5: LC_DYLD_INFO
        No rebase info
           Bind info:   1152  bytes at offset 65536 (0x10000–0x10480)
        No Weak info
        No Lazy info
           Export info: 32    bytes at offset 66688 (0x10480–0x104a0)
LC  6: LC_SYMTAB               Symtab: 92 entries @0x104f0(66800), Strtab is 1000 bytes @0x1
LC  7: LC_DYSYMTAB
            1 local symbols at index     0
            1 external symbols at index  1
           90 undefined symbols at index 2
           No TOC
           No modtab
          174 Indirect symbols at offset 0x10ab0
           No External relocations
LC  8: LC_LOAD_DYLINKER         /usr/lib/dyld
LC  9: LC_UUID                  UUID: 6C62FCB2–5D18–3692–8775–334D211AEB3C
LC 10: LC_BUILD_VERSION         Build Version:           Platform: macOS 11.0.0 SDK: 13
LC 11: LC_SOURCE_VERSION        Source Version:          400.0.0.0.0
LC 12: LC_MAIN                  Entry Point:             0x3a3c (Mem: 0x100003a3c)
LC 13: LC_LOAD_DYLIB            /usr/lib/libutil.dylib (compat ver 1.0.0, current ver 1.0.0)
LC 14: LC_LOAD_DYLIB            /usr/lib/libncurses.5.4.dylib (compat ver 5.4.0, current ver
LC 15: LC_LOAD_DYLIB            /usr/lib/libSystem.B.dylib (compat ver 1.0.0, current ver 131
LC 16: LC_FUNCTION_STARTS       Offset:     66720, Size:     80 (0x104a0–0x104f0) with 57 fur
LC 17: LC_DATA_IN_CODE          Offset:     66800, Size:      0 (0x104f0–0x104f0)
LC 18: LC_CODE_SIGNATURE        Offset:     69968, Size:  18848 (0x11150–0x15af0)
```

Output A/1-6(b): Demonstrating `disarm –L –v` on an ELF (matching `readelf(1)` output)

```
disarm –L ~/Documents/OSXBook/2nd/src/disarm/samples/ELF/ls                7:45
30 sections:
  (unnamed) (inactive) 0x0–0x0 (File: 0x0–0x0)
  .interp (program defined) 0x318–0x334 (File: 0x318–0x334)
  .note.ABI–tag (note section) 0x334–0x354 (File: 0x334–0x354)
  .note.gnu.property (note section) 0x358–0x388 (File: 0x358–0x388)
  .note.gnu.build–id (note section) 0x388–0x3ac (File: 0x388–0x3ac)
...
  .gnu_debugdata (program defined) 0x0–0xeb8 (File: 0x26e58–0x27d10)
  .shstrtab (string table section) 0x0–0x135 (File: 0x27d10–0x27e45)
```

1.4. Looking up regions/addresses

The –A and –0 switches can be used with address or offset arguments, in order to perform a quick lookup of one to the other. This will also resolve the name of the containing region, which is the agnostic way of saying section (PE), section (ELF) or segment/section (Mach-O):

Output A/1-7: Demonstrating `disarm –0 and –A`

```
$ disarm –A 0x100007a49 /bin/ls
Address 0x100007a49 is at offset 0x7a49 (__TEXT.__cstring)
$ disarm –O 0x104f4 /bin/ls
Offset 0x104f4 maps to address 0x1000104f4 (Region 17: symbol table)
#
# ELF (Android): Find containing ELF section
#
tokay:/ $ disarm –O 0x35e4 /system/bin/toybox
Offset 0x35e4 maps to address 0x35e4 (Region 12: .rela.dyn)
tokay:/ $ disarm –A 0x24242 /system/bin/toybox
Address 0x24242 is at offset 0x24242 (.eh_frame)
```

2. Dumping

Disarm doubles as a file dumper. This functionality is very similar to `od(1)`, `xxd(1)`, or `hexdump(1)`, and makes `disarm` a good replacement for platforms where one or another of the above tools is not found by default (e.g. Android's `toybox`). The basic usage is simple: `disarm -d`, which produces output similar to `hexdump -C`. This is useful for simple file inspection.

2.1. Data Inspection

Often times raw data, dumped in hexadecimal, is nearly nonsensical to the untrained eye. Though all hexdumpers helpfully display ASCII as part of the dump, that may not avail when the data is in some other representation.

`disarm` can take any 32-bit or 64-bit quantity, specified in hexadecimal, and attempt to interpret it in a variety of forms. The most basic one is simply as an instruction, if possible. If the instruction in question is from a particular ARMv8.x or ARMv9.x feature, this will also be indicated:

Output A/2-1: Command-Line quick disassembly of a binary instruction

```
$ disarm 0x8a989ff0
AND     X16, X31, X24, ASR #39
#
# LDG is a Load/Store instruction for memory tags, and requires MTE, from ARMv8.5
#
morpheus@eM1nent (~/.../disarm) % disarm 0xd965a31a
LDG     X26, [X24, #1440]       ; (FEAT_MTE)
```

When verbosity is requested (using -v), Decimal, Float, Big Endian, ASCII and Unicode are added to the display:

Output A/2-2: Command-Line disassembly and type parsing of a hexadecimal value

```
$ disarm -v 0x8a989ff0
AArch64:     AND        X16, X31, X24, ASR #39
Decimal:     -1969709072 (signed)/2325258224 (unsigned)
Binary:      10001010 10011000 10011111 11110000
Big endian:  -257976182 (0xf09f988a)
Float:       -0.000000
ASCII:       ? ? ? ?
String:      😊
```

2.2. Classic (`hexdump -C`) Dumping

Most often data needs to be inspected in bulk, and that is where hex dumping remains the most common form. There are multiple standard dumping tools, such as `xxd(1)` and `od(1)`. `disarm(j)`'s dumping was modeled after the Author's favorite, `hexdump(1)`, and specifically its common `-C` switch. Using `hexdump -C` on a file (or `stdin`) dumps the data in "canonical hex + ASCII display", which makes it an efficient presentation of both printable and non-printable characters.

Using `disarm -d` is intentionally entirely equivalent to `hexdump -C`, with the one exception that on encapsulated binaries (such as fat binaries, kernelcaches, or other IM4P), `disarm(j)` will apply itself on the inner file. There is not, by design, any way to disable this behavior (since one can always opt to use another tool for this).

Output A/2-3: Demonstrating classic dumping

```
# Hexdumping the kernelcache dumps the file verbatim
#
morpheus@eM1nent (~)$ hexdump -C ~/Downloads/kernelcache.release.ipad16 | head -5
00000000  30 84 01 24 a9 1d 16 04  49 4d 34 50 16 04 6b 72  |0..$....IM4P..kr|
00000010  6e 6c 16 1f 4b 65 72 6e  65 6c 4d 61 6e 61 67 65  |nl..KernelManage|
00000020  6d 65 6e 74 5f 68 6f 73  74 2d 34 32 33 2e 31 30  |ment_host-423.10|
00000030  30 2e 35 04 84 01 24 a7  a4 62 76 78 32 26 6d 01  |0.5...$..bvx2&m.|
00000040  00 04 1d 40 b2 01 9e 0e  20 83 8f 7f fc fa 55 1e  |...@.... .....U.|
#
# disarm cuts through the IM4P container, expands the LZFSE (bvx2) in memory,
# and displays the contents of the payload. Because this is a Mach-O, disarm
# also uses the address display, rather than offset
#
morpheus@eM1nent (~)$ disarm -d ~/Downloads/kernelcache.release.ipad16 | head -4
/Users/morpheus/Downloads/kernelcache.release.ipad16: This is an IM4P... with a BVX2 payload.
fffffe0007004000 cf fa ed fe 0c 00 00 01  02 00 00 00 0c 00 00 00  |................|
fffffe0007004010 31 01 00 00 40 55 00 00  00 00 00 00 00 00 00 00  |1...@U..........|
fffffe0007004020 1b 00 00 00 18 00 00 00  b8 b1 d2 87 48 a8 fc 65  |............H..e|
```

2.3. Smart Dumping

Matching other hex dumpers in functionality is only the beginning. `disarm(j)` is capable of "Smart Dumping", meaning that it can (within reasonable limits) interpret the data being dumped, according to several heuristics:

- The dump format is determined by the corresponding region metadata. This is usually derived from segment/section flags (e.g. S_CSTRING_LITERALS for a Mach-O), but may also be deduced through well-known or reserved names (e.g. in the case of Swift or Objective-C sections).
- "Smart dumping" nests, so that hex values are also interpreted according to corresponding region metadata. When hex dumping, if a 64-bit hex value is found to be a valid offset or pointer to another region, its format can be determined by that region's metadata.
- If a given data item can be interpreted as a structure, it will be displayed accordingly. This is commonly the case with Mach-O MIG, Objective-C and Swift binaries (and may be open in the future for third party extension).

The smart dump feature is automatically used when dumping a region (i.e. one of those reported by `disarm -l` on the binary), and is both very powerful and highly flexible. `disarm`'s core provides a format agnostic hexdump, but each file format parser internally extends it to its format idiosyncrasies. Mach-O, for example, provides CFString, Objective-C, Swift, MIG and other extensions.

A good example of the versatility of smart dumping can be seen when looking at Apple's binaries. `amfid`, for example, contains blocks, MIG, and vtables, all mixed in its `__DATA_CONST.__const`:

Output A/2-4: Demonstrating Smart Dumping

```
morpheus@eM1nent (/) %ARCH=arm64e disarm -r __DATA_CONST.__const /usr/libexec/amfid
#
# Note that smart dumping dumps 8-bytes at a time, not 16.
# This is in order to better display metadata about the value
# rather than its ASCII representation, which would anyway be
# nonsensical in the case of pointers
#
100020f80:    0xc0156ae10000019f .isa = __NSConcreteGlobalBlock
100020f88: 0x50000000   0x0       .flags = GLOBAL,SIG
100020f90:    0x100008c00         .func = _func_0x100008c00
100020f98:    0x100020f60         .desc
100020fa0:    0xc00c0000000001a1 libc++.1.dylib::__ZTVN10__cxxabiv120__si_class_type_infoE
100020fa8:    0x100019870    "12OSDictionary"
100020fb0:    0x100020e90    "?"
100020fb8:    0xc00c0000000001a1 libc++.1.dylib::__ZTVN10__cxxabiv120__si_class_type_infoE
100020fc0:    0x10001987f    "7OSArray"
100020fc8:    0x100020e90    "?"
100020fd0:    0xc00c0000000001a1 libc++.1.dylib::__ZTVN10__cxxabiv120__si_class_type_infoE
100020fd8:    0x100019888    "9OSBoolean"
100020fe0:    0x100020e90    "?"
100020fe8:    0xc00c0000000001a1 libc++.1.dylib::__ZTVN10__cxxabiv120__si_class_type_infoE
100020ff0:    0x100019893    "8OSNumber"
100020ff8:    0x100020e90    "?"
_autodetected_MIG_1000_Subsystem:  (12 routines, 1000-1012, msg size 4228)
100021000:      0x100009634    (subsystem server)
100021008:    0x3e8 0x3f4      Messages: 1000-1012
100021010:      0x1084         Max Size: 4228
100021018:      0x0
```

Many more examples of smart dumping can be found throughout this book, under the relevant discussion of whichever segment/section. Smart dumping is vastly superior to classic, "dumb" dumping, and so it is set to be the default, with no need for any additional switches (other than `-r …`, to specify the region). In case of interest in the raw data, using `-d` in conjunction with `-r …` will override the behavior, and dump the region specified in `hexdump -C` mode.

When coupled with regions, smart dumping also allows simplification of the command line. In this way, to dump XML (legacy) entitlements in a Mach-O one can simply use "`disarm -r entitlements`" (obviating the need for a dedicated `--ent`, as was in `jtool2(j)`).

3. Searching

An important functionality of `disarm(j)` is in searching for patterns in the binary. This harnesses standard memory matching with the file format specifics and disassembly, which allows not just presenting the search results, but also their context. `disarm` is designed to show the context by displaying the memory region (segment or section) where the pattern was found, and data surrounding the string.

3.1. Locating Strings and other data

Textual strings are often kept in their raw form - in order to be presented to the user, or as left over debugging/tracing messages. Such strings are invaluable in reverse engineering, as they provide hints as to the purpose of the functions using them, or even leak out otherwise stripped or redacted symbol names.

The –f switch can be used to find all instances of a string in a given binary. Since strings are primarily the main patterns sought, non-printable characters must be escaped, in the usual '\x##' notation (keeping in mind that '\' is a shell escape as well, so quotes are recommended. If the '\x##' escapes a NUL byte, the matching will include it as well, taking the string length to be at the end of the "natural" NUL byte terminator.

3.2. References

An important and common practice in reverse engineering is finding references to symbols or addresses. Addresses may be found in the data sections, but are often computed by register operations.

The ––`refs` option is especially suitable to look for references, since it covers both data and text sections. For the data sections, it amounts to a simple pattern matcher, (equivalent to sifting through `hexdump –C` in search of a hex pattern). For the text section, however, it will disassemble *and* emulate the register values, and - if the requested value is found - print the containing function as a reference.

Adding –v will also show you the snippet of code where the reference was found. This is especially useful for function references, since `disarm` will follow most register assignments, and display the function called including its arguments.

3.3. Gadgets

Gadgets are snippets of assembly code, commonly at the end of functions, which are used in exploitation. Various exploitation techniques repurpose `gadgets`, by subverting program flow to one or more gadgets, with the attacker knowing or even controlling the value of key registers used by the gadget. Although such practices (called ROP (Return Oriented Programming) or JOP (Jump Oriented Programming)) are mitigated by ARMv8.3 PAC, finding gadgets can also be useful when reverse engineering.

The –g switch can be used to instruct `disarm` to locate gadgets in the code. This requires the stating of the opcodes only, in a comma delimited manner. An example was shown in Listing 13/2-36, and is redisplayed here:

Listing A/3-1: PMAP calls, via the GENTER wrapper

```
morpheus@eM1nent (~) %JCOLOR=1 disarm -g BTI,MOVK,MOVK,MOVK,B /tmp/extracted/kernel.rebuilt
_sptm_lockdown_probably:
fffffff0088ebbf4     d503245f    BTI     c       ;
fffffff0088ebbf8     f2e00010    MOVK    X16, #0, LSL #48      ; X16 := 0xf
fffffff0088ebbfc     f2c00010    MOVK    X16, #0, LSL #32        ; X16 := 0xf
fffffff0088ebc00     f2a00010    MOVK    X16, #0, LSL #16        ; X16 := 0xf
fffffff0088ebc04     f2800010    MOVK    X16, #0 ; X16 := 0x0
fffffff0088ebc08     14892541    B       0xfffffff00ab3510c      ; __GENTER_wrapper_to_SPTM
        __GENTER_wrapper_to_SPTM(0);
_sptm_retype_probably:
fffffff0088ebc0c     d503245f    BTI     c       ;
fffffff0088ebc10     f2e00010    MOVK    X16, #0, LSL #48        ; X16 := 0x0
fffffff0088ebc14     f2c00010    MOVK    X16, #0, LSL #32        ; X16 := 0x0
fffffff0088ebc18     f2a00010    MOVK    X16, #0, LSL #16        ; X16 := 0x0
fffffff0088ebc1c     f2800030    MOVK    X16, #1 ; X16 := 0x1
fffffff0088ebc20     1489253b    B       0xfffffff00ab3510c      ; __GENTER_wrapper_to_SPTM
          __GENTER_wrapper_to_SPTM(0);
...
```

4. Disassembly

Disassembly is `disarm`'s raison d'être. The tool originally started its life as a CLI for quick disassembly of 32-bit ARMv8 opcodes. It was a (short) while later that it was extended to work on files, and later still - with version 2.0 - absorbed `jtool2(j)`'s Mach-O parsing abilities, updating them to catch up with Darwin changes, while further incorporating ELF and PE support.

4.1. The core

The disassembler core is the eponymously named `libdisarm`, which is statically compiled into the tool. It is a custom disassembly engine built from scratch, which offers both advantages and disadvantages.

The main disadvantage is lack of support for some (albeit rare) opcodes. The tool was built around the need to reverse specific binaries, and so grew to encompass most of the ARMv8 standard (general purpose register) opcodes. SIMD is largely (but not fully) supported, and SVE/SME even less so. On the other hand, since many of the binaries I reversed were kernels or boot loaders, there is support for all …_ELx registers (in `MSR`/`MRS` arguments), and even some of the (many) Apple Silicon specific registers.

The main advantage* is that `disarm`'s engine is highly flexible. The core is a `disassembleLoop(…)`, which deciphers the opcodes, and follows register values using lightweight emulation. This allows even the basic dump to annotate (as comments to the right of the output) the values of the registers, especially function arguments.

Additionally, `disarm` allows the registration of callbacks, to handle register operations, or function calls. Callbacks are the "secret sauce" which enables the more advanced features of `disarm`, as used internally. When registered externally, callbacks allow third parties to extend `disarm(j)` with their own logic, for better symbolic execution or reversing needs.

4.2. Common workflows

The workflows of disassembling with `disarm` commonly revolve around using the tool along with `grep(1)`. Using `grep(1)` as a filter simplifies `disarm(j)`'s logic by avoiding a much more complicated command-line syntax. Further, it allows `disarm(j)` to be run just once, output into a file, and then perform different workflows in a much faster and more scalable manner.

4.2.1. Quick disassembly

For quick disassembly, matching the capabilities of other disassemblers but without any advanced features, the `-q` switch can be specified. On an Apple M1 this disassembles the 2.2 million instructions of the XNU kernel (proper) in under 1.7 seconds, and the full kernel cache (11.6 million instructions) in under 7 seconds.

4.2.2. Example: Cursory (no flow control) decompilation:

When disassembling and emulating register state, `disarm(j)` also reconstructs function calls, by printing the arguments from the registers and on the stack. Being a "dry run", this functionality is obviously limited by inability to deduce any pass through argument values,** but works very well with strings and other values.

Further, `disarm(j)` is designed so that such function reconstruction lines start with white space, rather than an offset or an address, of the disassembly. Combined with the `-r` switch's ability to compile to the end of the function, and `grep(1)`, this provides a quick, albeit partial decompilation, focusing on callouts from the current function.

---

* - A personal advantage was for your Author, in that there is no better way to get intimately familiar with assembly than to implement a disassembler from scratch. Reading the 12,000+ ARM spec is already an endeavor. Implementing it even more so.
** - at least, for the moment.

5. Program Analysis

Over time, common reverse engineering work flows have been incorporated into `disarm(j)`, to allow automated symbolication of programs. The `--analyze` command line option will exercise these work flows, and record their findings so they can be automatically applied during future analysis. This integrates with another feature of `disarm(j)` - companion files - which allow the user to quickly and easily build a separate companion file with symbols and notes.

5.1. Analysis mode

When encountering a new binary, a cursory disassembly with `disarm(j)` will often already give a high level view of its function and code. To go deeper, it's recommended to run analysis mode, using `--analyze`. This must be the only option specifed (excluding `-v[v]`), and will automatically record results in a companion file (see later).

Analysis mode runs two sets of code flows:

- **Static rules:** are hard-coded into `disarm(j)`. Those include automated workflows for Objective-C binaries, and for kernels - both XNU and Linux. When the file types can be reliably detected, `disarm(j)` will perform the analysis steps to reconstruct symbols based on known metadata and patterns in these files.
- **Dynamic rules:** which the user can specify using **matchers**. These open up nearly limitless possibilities, harnessing `disarm(j)`'s argument tracking and limited symbolic execution, to evaluate and apply user-supplied rules in a simple textual format.

If no matcher rules are specified, only the static rules will be applied. `disarm(j)` will issue a notice on this. The following output demonstrates analysis mode on a random Objective-C binary from Darwin:

Output A/5-1: `disarm(j)` Objective-C binary analysis

```
morpheus@eM1nent (/tmp) % ARCH=arm64e disarm --analyze /usr/libexec/keybagd
Objective-C detected - running analysis
Analyzing Objective-C classes..
Analyzing Objective-C methods
Finding functions
Not loading matchers (…/keybagd.matchers not found) - anaylsis will be limited to function starts
Generated companion file: keybagd.ARM64.A6B8CA6C-600B-3910-87B0-D9CB33695B19
#
# Example: Displaying Objective-C methods discovered from the file's __DATA_CONST.__objc*
#
morpheus@eM1nent (/tmp) % cat keybagd.ARM64.A6B8CA6C-600B-3910-87B0-D9CB33695B19 | grep '\['
0x100006578|[KBXPCListener listener:shouldAcceptNewConnection:]
0x100006704|[KBXPCService remoteProcessHasBooleanEntitlement:]
0x100006764|[KBXPCService remoteServiceName]
0x1000067e8|[KBXPCService retrievePasscodeFromFileHandle:ofLength:withbaseaddress:]
...
#
# Example: Counting how many  symbols (function starts, imports and objective-C) were discovered:
#
morpheus@eM1nent (/tmp) % wc -l keybagd.ARM64.A6B8CA6C-600B-3910-87B0-D9CB33695B19
     967 keybagd.ARM64.A6B8CA6C-600B-3910-87B0-D9CB33695B19
```

5.2. Companion Files

As a binary is inspected and reversed, more and more details of its operation are uncovered. Stripped functions can be given more meaningful names, and other significant addresses, such as globals, can be found and symbolicated. `disarm` provides **companion files** to help you symbolicate binaries. These will be auto-created in analysis mode, but can also be created manually.

5.2.1. Naming convention

Companion files follow the naming convention of:

binaryName.*arch*[.*UUID*]

For example:

ls.ARM64.6C62FCB2-5D18-3692-8775-334D211AEB3C

Since `disarm` (presently) only handles ARM64 binaries, *`arch`* is ARM64. The *`UUID`* is auto-deduced from the file (based on the LC_UUID of Mach-O files, or `.gnu.hash` of ELFs). If one cannot be deduced, it is omitted. Using a UUID ensures that, should you examine another version of the binary (which likely has a different address space layout), the wrong symbols would not be used.

You need not concern yourself too much with the companion file naming convention: Specifying `--companion` will instruct `disarm` to create an empty companion file, either in the directory where the target binary is, or in the present working directory. Alternatively, when running analysis (`--analyze`), the companion file will be generated automatically.

5.2.2. File format

The companion file format is very simple, and can be summed up by the following rules:

1. Empty lines are not allowed. Each line must contain a comment or an entry.
2. Comment lines, beginning with # are silently ignored. These enable you to annotate the file.
3. Entries are of the form

`Address``:`*`Symbol`*

Using a text file makes companion files human-readible and less prone to corruption (unlike Mach-O `.dSym` or IDA `.idb` files, for example). This also promotes their sharing between researchers. A common workflow is to use two terminal sessions (or `screen(1)/tmux`), with one for your favorite text-editor, and another for using `disarm`. As your symbolication efforts unravel more symbols you update with the editor, repeated invocations of `disarm` become clearer and more informative.

5.3. Matchers

The most powerful analysis capabilities `disarm` offers lie in its **matchers**. Matchers are rules which enable `disarm`'s disassembly engine to auto-symbolicate a binary. These rules can be specified in a fairly simple text file. `disarm` will automatically look for this file, and (unless a companion file is already present) will automatically load it and analyze the binary.

As this section will demonstrate, matcher syntax is easy to write and maintain, and it is hoped that this functionality will be adopted by other disassemblers as well.

5.3.1. Argument Matchers

Among other capabilities, `disarm(j)` tracks register values, which enables it to figure out arguments to functions according to the ARM PCS. This enables the creation of simple rules for function calls - when argument #x has value y. The syntax is made simple:

- The rule format is pipe ('|') delimited:

`#|matching pattern|caller_function|called_function|comments`

- Strings: can be specified as is (provided they don't start with '`0x`', with no need for quotes, but (please) no "|"s. The matching is performed on a substring.
- Integers: start with a '`0x`', and **must** be specified in hex.

An example of both matchers is shown here:

Listing A/5-2: Demonstrating argument matchers

```
#
## Matchers for argument 0:
#
0|zone_require failed: address in unexpe|_zone_require_panic|_panic
0|0x55aa0101|_write_legacy_header|_copy_cpu_map
#
## Matchers for argument 1:
#
1|idle_thread_create|_kernel_bootstrap_thread|_strncpy|PACIBSP|osfmk/kern/startup.c
```

5.3.2. Region Matchers

Region Matchers extend the power of matchers into data segments. There are often plenty of pointers and magic values in data, as constants, global values, string initializers, and more.

5.3.2.1. Matching by string value

Many structures have `char *` fields pointing to well known names. Considering XNU again, examples of this abound - `sysctl` names, filesystem names, network domains, to name but a few.

Assuming you found such as character string reference, it's likely to be embedded inside a structure. Considering the case of a `sysctl` MIB, as discussed in 14/2.24, the `sysctl name` field is 40 bytes into the structure. The following rule will therefore apply:

Listing A/5-3: Demonstarting a region string matcher

```
#
# String Region rules: These apply to data. Format is:
# _SECTION/_SEGMENT:"string"|symbol|type|+/-offset
#
__DATA_CONST|"slide"|_sysctl_kern_slide|sysctl|-40
```

5.3.2.2. Matching by 64-bit integer value

A more common case is where hard coded and other "magic" values are embedded in the data section. Once again, a region matcher can be used here. This time, instead of specifying the string in quotes, specify `val=0x…`, allowing for the full 64-bit value, and keeping in mind it must be aligned (that is, be at address or offset `0x…0` or `0x…8`. This can be a portion of a bigger value - this won't matter, so long as unicity is assured:

Listing A/5-4: Demonstarting a region string matcher

```
#
# The  well known Apple NVRAM Guid 7C436110-AB2A-4BBB-A880-FE41995C9F8
# Assign it the c++ mangled "gAppleNVRAMGuid" at offset 0 (GUID beginning)
#
$ ...
__TEXT.__const|val=0xbb4b2aab1061437c|__ZL15gAppleNVRAMGuid||0
```

As another example of two simple but far reaching rules, the following two matchers can be used to reliably locate XNU's `sysent` and `mach_trap_table`. Not only are these important symbols by themselves, but they also positively identify the binary analyzed as a XNU kernel, which can kick off additional analysis rules.

Listing A/5-5: Demonstrating XNU's `sysent` and `mach_trap_table` matchers

```
#
# Example for XNU's two most important tables - _sysent and _mach_trap_table
# Our "magic" patterns are at varying offsets (-40, -240, respectively)
#
__DATA_CONST|val=0x0004000100000000|_sysent|_sysent|-40
__DATA_CONST|val=0x0000000000000504|_mach_trap_table|_mach_trap_table|-240
```

5.3.3. Opcode Matchers

Functions will often refer to globals, which are loaded using specific instruction sequences (commonly, ADRP/ADD). When a function is known, it's possible to run an **opcode matchers**. Due to their performance overhead, these can only defined in the scope of a previously defined or discovered symbol. The matching disregards arguments, looking only at the opcodes themselves, but since `disarm(j)` emulates the instructions, the final value matched can be specified as the value of a register.

As an example, let's assume your previous analysis of XNU revealed the address of `_zone_require_panic`, after setting an argument matcher for its `panic` message. The disassembly starts like this:

Listing A/5-6: Interesting disassembly...

```
_zone_require_panic:
fffffff008607368  d503237f    PACIBSP
fffffff00860736c  d10203ff    SUBi    SP, SP, #128    ; SP = SP - 0x80
fffffff008607370  a9045ff8    STP     X24, X23, [X31, #64]    ; *[SP +64] = [X24, X23]
fffffff008607374  a90557f6    STP     X22, X21, [X31, #80]    ; *[SP +80] = [X22, X21]
fffffff008607378  a9064ff4    STP     X20, X19, [X31, #96]    ; *[SP +96] = [X20, X19]
fffffff00860737c  a9077bfd    STP     X29, X30, [X31, #112]   ; *[SP +112] = [X29, X30]
fffffff008607380  9101c3fd    MOVr    X29, X31, #112  ; FP = SP + 112
fffffff008607384  aa0103f3    MOVr    X19, X1         ;  X19 = X1
fffffff008607388  79406808    LDRH_i  W8, [X0, #52]   ;  X8 = *(X0 + 0x68)
fffffff00860738c  b0ff9b69    ADRP    X9, #-3219      ;  X9 = 0xfffffff007974000-
fffffff008607390  91152129    ADD     X9, X9, #1352   ;  X9 = 0xfffffff007974548
fffffff008607394  9100212a    ADD     X10, X9, #8     ;  X9 + 0x8 = 0xfffffff007974550
fffffff008607398  a9402929    LDP     X9, X10, [X9]   ; [X9, X0] = *[X9]
fffffff00860739c  8b010108    ADDsr   X8, X8, X1      ; R8 = R8 + R1 = 0x74736e6f635f5f xx
fffffff0086073a0  eb0a011f    CMPsr   X8, X10 ;
fffffff0086073a4  fa419122    CCMP    X9, X1, #2, LS  ;
fffffff0086073a8  fa499100    CCMP    X8, X9  ;
fffffff0086073ac  54000122    B.CS    0xfffffff0086073d0      ;
fffffff0086073b0  5280b3a8    MOVZ    W8, #1437       ; X8 = 0x59d
fffffff0086073b4  d0ff5209    ADRP    X9, #-5566      ; X9 = 0xfffffff007049000-
fffffff0086073b8  9128dd29    ADD     X9, X9, #2615   ; X9 = X9 ='zalloc.c'
fffffff0086073bc  a900a3e9    STP     X9, X8, [X31, #8]       ; *[SP +8] = [X9, X8]
fffffff0086073c0  f90003f3    STRi    X19, [X31]      ; *0x0 = 0x0
fffffff0086073c4  d0ff5200    ADRP    X0, #-5566      ; X0 = 0xfffffff007049000-
fffffff0086073c8  91359800    ADD     X0, X0, #3430   ; X0  ='zone_require failed: address …"
    _panic("zone_require failed: address not in a zone (addr: %p) @%s:%d",(stack)>"zalloc.c");
...
```

Since XNU is open source, we can see that the beginning of the function looks like this:

Listing A/5-7: ...and its corresponding source

```
/* This function just formats the reason for the panics by redoing the checks */
__abortlike
static void
zone_require_panic(zone_t zone, void *addr)
{
        uint32_t zindex;
        zone_t other;

        if (!from_zone_map(addr, zone_elem_inner_size(zone))) {
                panic("zone_require failed: address not in a zone (addr: %p)", addr);
        }
  ...
```

A bit of digging (or looking through 12/2) would show that `from_zone_map` looks at the `zone_info.zi_map_range`. This is the first field in the `zone_info` structure, and therefore has the same address in memory. In the assembly, that would be in X_9, the X_t of the `ADD` instruction. Crafting an opcode rule for this is therefore:

Listing A/5-8: Demonstrating opcode matchers

```
#
# Opcode matching: Specify the symbol to run in (an exported symbol or a previously
# discovered one), and then use opcodes:opcode, ... . The value
# you want to match and symbolicate will can be specified in the Xt, Xd, Xn, etc
# of the instruction (per the ARM specification
#
_zone_require_panic|opcodes:ADRP,ADD|Xt=zone_array|
```

Opcode matchers are run at the last stage of matching, since they may rely on function symbols defined in other matchers.

B

`jtrace(j)` - the missing manual page

This appendix provides a tutorial and documentation for using `jtrace(j)`. Designed as a drop-in replacement to the highly useful `strace(1)` utility, `jtrace(j)` not only matches its counterpart's functionality, but augments and exceeds it using new switches, and a powerful plugin architecture.

1. Basic Usage (`strace(1)` compatibility)

Users of `strace(1)` will find `jtrace(j)` fully compatible with it. Although the tools share no common code, care has been taken to ensure that the latter can serve as a drop in replacement for the former, by honoring the following command line options:

Table B/1-1: The `strace(1)` compatible options of `jtrace(j)`

Option	Purpose
`-o` *outputFile*	Output to *outputFile*, rather than `stdout`
`-f`	Follow Forks
`-t[t[t]]`	Timestamping
`-T`	Time spent in system call
`-p` *pid*	Attach to a process/thread by *pid*
`-v`	Verbose output
`-y`	Resolve file descriptors (default behavior, so ignored)

2. Enhancements (and `jtrace(j)`-specific options)

In addition to the above, `strace(1)`-compatible options, `jtrace(j)` provides specific options, offering extended functionality. These are summarized in the following table, and then detailed.

Table B/2-1: The specific command-line options of `jtrace(j)`

Option	Purpose
`-F`	(deprecated by `strace(1)` - Follow threads, but **NOT** processes
`-O`	Redirect output to jtrace.out
`--color`	Enable color (also: `JCOLOR=1` in environment)
`--thread`	Attach to a process/thread by name
`--plugin` *pluginName*	Load *pluginName* (if not loaded automatically)

2.1. Color

As with all the other J-tools, ANSI colors provide an important aspect of detail in the output. `jtrace`'s output is voluminous, and colors really helps to focus on the parts which matter: system call/function names, arguments, and timestamps.

ANSI colors, however, work by means of escape sequences (a.k.a "curses"), which are not always legible, for example on incompatible terminal emulators, or when piping ('|') to some other program, notably `more(1)` (but not `less(1)`, which can support curses with `-R`). Color is therefore not enabled by default, but may be enabled in one of several ways:

- **The `--color` argument:** when supplied, will explicitly enable colors.
- **The `JCOLOR` environment variable:** Setting this lets you choose between a one time colored invocation (if specified on the command line), or as a default (if `exported` into the shell environment.
- **The `color` keyword:** in a .jtc file, will apply color (when used by itself or with 'on') or disable color (when used with 'off') for all invocations of `jtrace` using that configuration.

2.2. Marks

In many instances, tracing will require you to invoke some command or condition while tracing a running process (e.g. some daemon). The problem here is that the target won't necessarily sit idly waiting for that condition, and may perform other calls in the meanwhile, causing `jtrace` to spew output.

Marking enables you to tag the output of `jtrace` with a simple textual mark - optionally colorized in red - by pressing the 'M' key in an active `jtrace` session. Subsequent presses of 'M' will embed additional marks - simply titled "Mark #" with a monotonically increasing number. This can then be useful as you sift through the copious output, as you will be able to jump to the exact spots where marks were placed by a simple find operation in your favorite terminal or editor.

2.3. Freeze/Thaw

Considering again the challenges of tracing busily running processes, another approach is to selectively freeze the process. This can be used while setting up a test condition.

Pressing 'F' will 'F'reeze all threads of the target process currently being traced. Pressing 'T' will 'T'haw them. Quitting `jtrace` will automatically thaw any frozen processes. Note, however, the caveats:

- There is an inherent delay between `jtrace`'s operation and that of the process. This opens up a small window (and possibly, race conditions) between the request to 'F'reeze, and the actual suspension of the target(s).
- If `jtrace` is not started with `–f`, it will only freeze the specific thread it is attached to.
- Freezing a process is not without its own complications, and may adversely affect system stability (especially if the process is an important one, like Android's `system_server`).

2.4. Config (.jtc) files

Tracing the same processes or threads usually requires the same, or very similar command line. In order to ease use, common options for a trace can be specified in a config file. These are simple textual files, which can be dropped in `${JTRACE_DIR}` (default: /data/local/tmp, since it's the only writable location by `shell`). The files will be auto-loaded if their name matches the process/thread name being traced, or - if named "default.jtc (for all tracees).

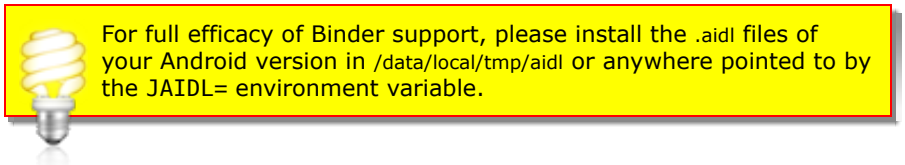
The vocabulary of the configuration files is presently very limited, and consists of the following keywords:

- `color`
- `threads` (to automatically trace all threads, like –F)
- `suppress` (with a comma-delimited list of system call names to suppress. All others enabled.
- `enable` (with a comma-delimited list of system call names to enable. All others suppressed.

2.5. Androidisms

`jtrace(j)` will run on any Linux system, either ARM64 or the legacy X86_64. First and foremost, however, it was designed for Android. As such, it contains built-in extensions specific to that platform. These include:

- **Binder message support:** Binder goes three levels deep (sometimes more) to obtain the parcel data pointed to from the `struct binder_transaction`, which itself is pointed to from the `binder_transaction`'s data pointer, which was pointed to from the `binder_write_read`, the `ioctl(2)` argument (*whew!*). This is technically done through an `ioctl(2)` plugin (see next), but - because Binder is nigh-omnipresent on Android, has been compiled into the binary. The logic only gets activated if the *fd* of the `ioctl(2)` is detected to be any of the /dev/*binder flavors. The Binder hard-coded plugin itself allows further plugins, to allow extending functionality to look directly at parcels, or filter certain transaction codes.



- **System property requests:** Currently in protocol version 2, are passed through `sendmsg(2)` calls over /dev/socket/property_service. They will automatically be reconstructed to the property/value setting, rather than showing the raw message, unless –v is specified.

3. Plugins

Unlike `strace(1)`, or `ltrace(1)`, whose system call and library hook logic is fixed, `jtrace(j)`'s is dynamic. `jtrace(j)` offers a simple but capable API, allowing developers to extend the default handlers, block them, or replace them altogether. This brings an entirely new dimension of functionality to the tool, extending it from a passive tracer to an active one, capable of fuzzing, memory-manipulation and fault injection.

3.1. System call handlers

JTrace plugins can install any number of system call handlers. They are expected to do so in their constructor, since this code is guaranteed to be called by `jtrace(j)` on plugin load. Only one API is required here, in order to register the system call handler. This is kept simple, as follows:

Listing B/3-1: The JTrace register API

```
typedef enum {
        DO_NOTHING = 0, // You want JTrace to continue as normal
        BLOCK      = 1, // You want JTrace to block this system call
        NOT_HANDLED = 2, // You didn't handle the syscall
        DO_EXIT         =3 , // exit jtrace
} syscall_handler_ret_t;

typedef syscall_handler_ret_t (*syscall_handler_func) (int exit, int pid);

enum syscallHandlerFlags {

        HANDLER_ENTRY_ONLY = 1,
        HANDLER_EXIT_ONLY  = 2,
        HANDLER_OVERRIDE_DEFAULT = 1024,
};

// piece de resistance
int registerSyscallHandler (char *SyscallName, // must exist or you'll get -1
             syscall_handler_func Handler,     // Your callback, per above
                                  int Bitness, // 32, 64 or 0 (both)
                                  int Flags);  // as per flags, above

#define ERR_SYSCALL_NOT_FOUND   (-1)
```

The system call registration requires the system call **name**, which is intentional, since the numbering of system calls on Linux tends to change across architectures. The *Handler* argument is a function taking in two arguments: A boolean, indicating if called on entry or exit (since the handler may be called on either or on both), and the *pid* (actually, the **thread** identifier) in which the system call executes.

The handler is then free to do whatever is needed, since at this point `jtrace(j)` is blocking the system call and is running the plugin code. Please observe the following:

- Register values can and should be accessed only through the **Register API** (`[get/set]RegValue(…)`, discussed next)
- The plugin code **SHOULD NOT** use `printf(3)`, or, in fact, write any output to `stdout`/`stderr`. This is because in a multithreaded environment, the <stdio.h> functions use locking, which slows down `jtrace(j)`, and thus also the inferior.
- Instead, the plugin code can use three functions here:
  - **`getOutputBuffer(void);`** will return a pointer to a thread-specific output buffer, that the plugin can use. This is often part of the larger per-thread buffer `jtrace(j)` uses for output. Do not underflow it.
  - **`getOutputBufferSize(void);`** will return the output buffer size. Note that the buffer size for **ALL** thread buffered output is currently hard coded at around 1MB. Depending on how much output has amassed, you might get to use a lot less (but still a guaranteed `0(256)` bytes. Friendly advice - be sparse, and save copious output to other files. If you overrun this buffer, `jtrace(j)` ***will*** crash, but the overflow will be yours, not `jtrace(j)`.
  - **`updateOutputBufferPos(howMuch);`** please keep count (for example, by collecting the return value of `sprintf(3)`) of how many bytes you write into the buffer. At the end of your output, remember to call this function in order to have `jtrace(j)` use your output. Otherwise, your plugin code will return, but `jtrace(j)` might overwrite it with other syscalls' output.

- If you need Thread Local Storage (TLS) for the plugin - call `getPluginPrivateData(void)`. This returns a void pointer to at least a page of data. If you need more than that, use pointers.

The very specific case of Binder parcels is best served by a special kind of plugin:

Listing B/3-2: The JTrace binder-specific API

```
// Hook function to call: Parcel, Size of Parcel, Code (from binder_transaction)
// and Reply - a boolean indicating if this is a reply Parcel (or a request)
// Parcel bytes may be modified, but not extended past Size
typedef int binderHook_t (char *Parcel , size_t Size, int Code, int Reply);

// This will call binderHook_t for all messages of Interface
int registerBinderTransactionHook(char *Interface, // UTF-16, per AIDL
                                  binderHook_t  Hook);
```

This enables invocation of the handler for a specific interface name (NULL for all). The *Hook* is then called with the *parcel* (which will contain the interface), its *Size*, the method *Code* (from the `struct binder_transaciton`) and a boolean indicating if this is a request or a reply. The pointer to the parcel is a copy of the incoming or outgoing parcel. At present, the original cannot be modified (by design, not for lack of capability).

3.2. The Register API

One obvious omission from the system call handler arguments are the registers. This is intentional as of JTrace v2. The rationale here is to maintain plugin portability, even between different architectures, by forcing register access through an architecture agnostic API:

Listing B/3-3: The JTrace register API

```
//
// Abstracting the calling convention:
//
// JTrace will automatically translate these register "numbers" into the corresponding
// registers of the underlying architecture, as per the ABI. In the ARM64 case, these
// are straightforward noops - ARG_0...ARG_7 are X0..X7. In ARM32, X0..X3 are 0..3,
// but jtrace is smart enough to get the other "register" values from the stack. In the
// Intel case, the mapping is between rdi/rsi/rdx/rcx/r8/r9 or the stack.
//
// Likewise, the syscall number - in Intel, that's eax/rax. In ARM32/ARM64 r7/r8 respectively
//
// This enables you to get the arguments in a plugin without bothering to consider how the
// calling convention works!

#define REG_ARG_0       0
#define REG_ARG_1       1
#define REG_ARG_2       2
#define REG_ARG_3       3
#define REG_ARG_4       4 // ARM32: stack
#define REG_ARG_5       5
#define REG_ARG_6       6 // Probably won't need more than this since syscalls have max 6..
#define REG_ARG_7       7

// these are intentionally values greater than ARM64's registers,
// so we are guaranteed to map them by ABI
#define REG_ARG_RETVAL          33   // ARM: X0   x86_64: RAX
#define REG_ARG_SYSCALL_NUM     34   // r7, X8

// The idea is to then leave the regs_t opaque, and use accessors
// And to ensure that's the case, JTrace no longer exports regs.
long getRegValue (int num);

// setRegValue - overwrite REG_ARG_* to 64-bit value
// Can actually be used on any general purpose register, by number
void setRegValue (int num, uint64_t value);

// Saved register values, as they were on entry. May be different from the ones on exit,
// particularly for REG_ARG_0 on ARM (since X0 is return value)
uint64_t getSavedRegValue (uint32_t Reg);
```

Target register access is abstracted by two main calls - `[get/set]RegValue`. Registers are encoded by their position as arguments. This provides the abstraction of the architecture's calling convention, allowing the same plugin code to work across different architectures.

Table B/3-4: jtrace's [get/set]RegValue constants

Constant	ARM64	X86_64
REG_ARG_0	X0	RDI
REG_ARG_1	X1	RSI
REG_ARG_2	X2	RDX
REG_ARG_3	X3	RCX
REG_ARG_4	X4	R8
REG_ARG_5	X5	R9

In ARM architectures, using values higher than REG_ARG_5 (i.e. 6..29) will retrieve the other general purpose registers. This is generally discouraged, but deliberately left supported for advanced users.

3.3. The Memory API

Plugin code will often need to read or write to an inferior's memory. For this, `jtrace(j)` provides a simple memory API:

Listing B/3-5: The JTrace memory API

```
// This conveniently hides process_vm_readv, /proc/pid/mem, or ptrace
int readProcessMemory (int Pid, arch_int Addr, int Size, char *MyAddr);

// And this can set the process memory, writing Size bytes from MyAddr to Addr!
int writeProcessMemory (int Pid, arch_int Addr, int Size, char *MyAddr);

// For handling sendmsg(2)/recvmsg(2)
char *getIOVecDataFromMsgHdr (struct msghdr *MsgHdr,
                              unsigned int *BytesRead,
                              pid_t    Pid);
```

The `[read/write]ProcessMemory` are straightforward. They hide the underlying implementation - which may be `ptrace(…[PEEK/POKE]…)`, `process_vm_[read/write]v` or /proc/*pid*//mem - and just access the memory in the most efficient of the three ways found to be possible (usually, the last).

`getIOVecDataFromMsgHdr` is a convenience function left for those cases wherein `sendmsg(2)` or `recvmsg(2)` are encountered, which would require multiple `read(2)`s to get to the `struct iovecs` in the message header, and then their buffer contents.

3.4. The FD API

I/O operations will work on file descriptors. Information on these can oe obtained through the /proc/*pid*/fd entries, but `jtrace(j)` provides a convenience API to do that:

Listing B/3-6: The JTrace FD API

```
// Get an FD name. This is useful for operating in read/write hooks where
// the first argument is the fd number. Returns NULL if fd is not conencted

char *getFDName(int fd); // for the thread we are tracing
char *getFDForPidName(int fd, pid_t Pid); // if tracing multiple processes

int setFDName(int fd, char *Name); // New - Set a name, up to MAX_NAME chars
                                   // For example, as result of SYNC_FILE_INFO

int setFDForPidName(int fd, pid_t Pid, char *Name); // up to MAX_NAME chars
```

3.5. See Also

The downloadable .tgz of `jtrace(j)` provides plenty open source examples of plugins, and a simple script to compile them, using the Android NDK.