Kernel Tracing via Corellium

Kernel tracing is focused on mapping the exact execution paths inside the kernel, which can be useful for diagnosing kernel issues or integrated into coverage-guided fuzzers to assist with finding new inputs.

Tools

Corellium provides five tools to faciliate tracing:

- `btgen`: Processes a kernelcache to produce a list of tracepoints in a text file (`.bt`) and a reconstruction file (`.btrec`)
- `btasm`: Converts the tracepoints list from text format to a binary format understood by the hypervisor (`.btbin`)
- `hyptrace`: Runs inside a Corellium device to load the binary tracepoints file and collect the results in a binary format (`.btrace`)
- `btrace`: Uses the reconstruction file generated by `btgen` to convert the binary trace data (`.btrace`) to plaintext with flow information (`.trace`)
- `trace_to_lighthouse.py`: Converts the plaintext trace data (`.trace`) to a format understood by Lighthouse. Note that this is only block-level coverage data and removes program flow.

Usage

1. Ensure that the kernelcache is not a universal/fat binary

Kernels downloaded directly from Corellium (on the `Connect` tab) may be universal binaries, which `btgen` does not currently support. Use the `lipo` tool provided by Xcode to extract the ARM64 slice:

```
$ lipo kernel-iPhone9,1-19E258 -thin arm64 -output kernel-iPhone9,1-19E258.arm64
```

Generally, arm64e kernels will not need this step.

2. Identify boundaries of CoreCrypto’s text section

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	T	DS
com.apple.kec.corecrypto:__text	FFFFFFF0059DD000	FFFFFFF005A243EC	R	.	X	.	L	align_64	78	public	CODE	64	00	00
com.apple.kec.corecrypto:__stubs	FFFFFFF005A243EC	FFFFFFF005A25000	R	.	X	.	L	dword	79	public	CODE	64	00	00

- This can be done easily in IDA Pro by opening the Segments tab
- Build a range that excludes this region e.g., `0-0xFFFFFFF0059DD000`, `0xFFFFFFF005A25000-0xFFFFFFFFFFFFFFFF` for the entire kernel

3. Run `btgen`, providing the following:

- The path to the kernelcache binary
- The path to save the reconstruction file, which will be needed later
- A sequence of ranges specifying what should be measured
    - This is a sequence of pairs, i.e. “`<start va> <end va> [<start va> <end va> ...]`”
- Example: `./btgen kernel-iPhone9,1-19E258.arm64 kernel-iPhone9,1-19E258.btrec 0 0xFFFFFFF0059DD000 0xFFFFFFF005A25000 0xFFFFFFFFFFFFFFFF > kernel-iPhone9,1-19E258.bt`

4. Add a header to the top of the `.bt` file specifying the base of the kernel in both virtual and physical addresses

For devices with 16K pages (iPhone 6s and newer):

```
vbase 0xfffffff007004000
pbase 0x0000000803004000
```

For devices with 4K pages (iPhone 6/6 Plus):

```
vbase 0xfffffff007004000
pbase 0x0000000802004000
```

Note that each line in the file (after the newly-added header) contains:

- The address of a tracepoint
- Some number of commands

Lines that start with # are comments which may indicate a tracepoint that isn’t supported.

The first command is typically `f0`, which enables a thread filter. For full-system traces (meaning all threads), see the note below.

The rest of the commands tell the hypervisor which other pieces of information are needed to reconstruct the flow at that tracepoint, such as the target register of an indirect branch instruction.

5. Run `btasm`, providing the following:

- The path to the generated `.bt` file
- The path to the output file, e.g. `kernel-iPhone9,1-19E258.btbin`
- Example: `./btasm kernel-iPhone9,1-19E258.bt kernel-iPhone9,1-19E258.btbin`

6. Capture the trace

- Upload `hyptrace` and the `.btbin` file generated by `btasm` to the target Corellium device.
- Within the virtual device, run: `hyptrace /path/to/kernel.btbin /path/to/output.btrace 16777216 /bin/ls`

This will run the specified command (here `/bin/ls`) and produce up to 16MB of kernel traces from it, saved into the output `.btrace` file. Only the main thread of the command will be traced (assuming the `f0` command was left in place, see above).

7. Convert to a human-readable trace

Run `./btrace /path/to/output.btrace /path/to/kernel.btrec > /path/to/output.trace` where:

- `output.btrace` is the file generated by `hyptrace` inside the virtual device
- `kernel.btrec` is the reconstruction file generated by `btgen` in step 3
- `output.trace` is the output file for the human-readable program flow

This will generate a readable text file. Each line (except comments starting with #) contains a pair of addresses, designating start and end of execution block. Then it describes how the basic block ended:

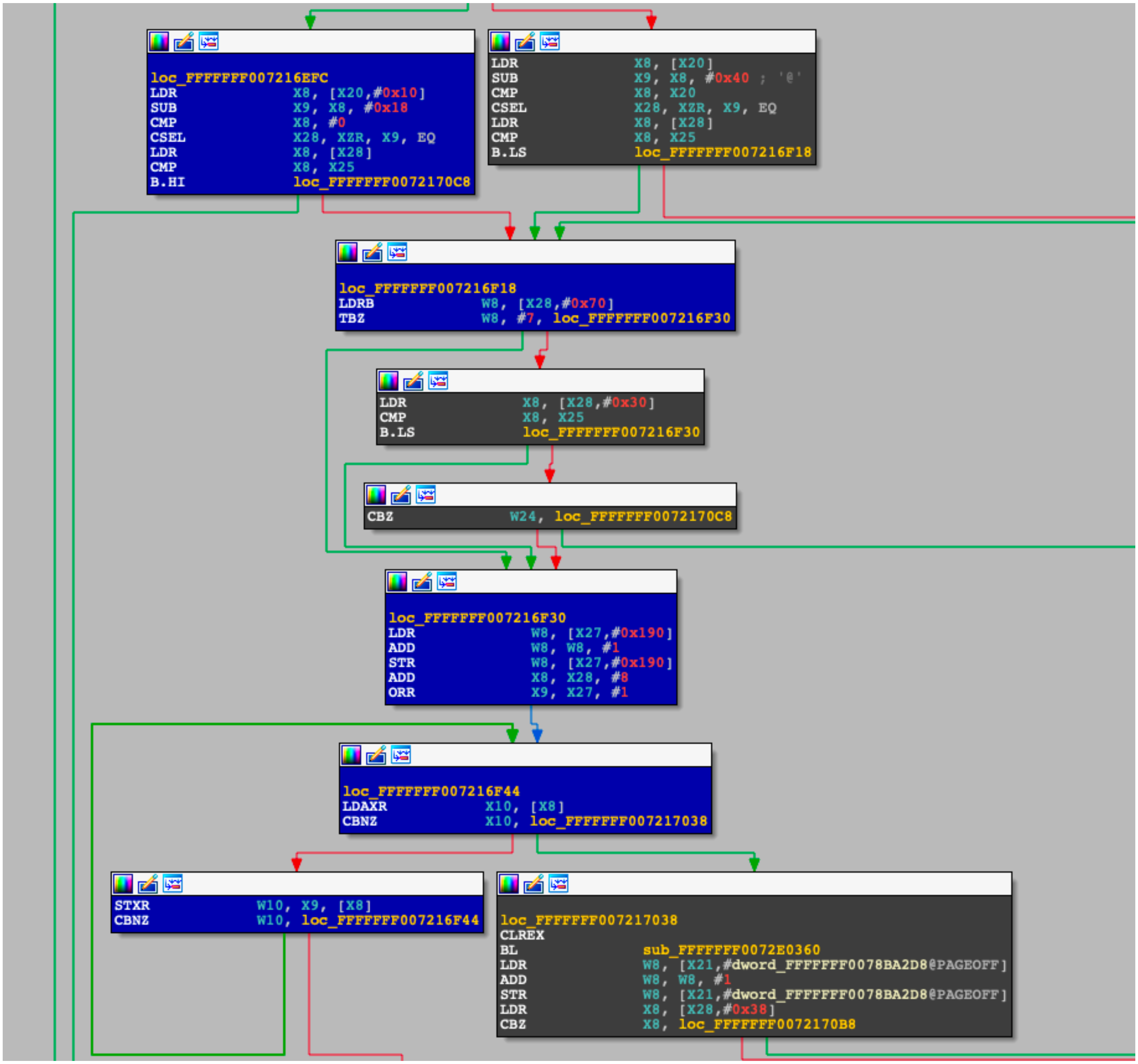
Keyword	Description
`any`	Basic block ended in a regular instruction (no control transfer)
`stx`	Basic block ended in any of the store-exclusive instructions
`ldx`	Basic block ended in any of the load-exclusive instructions
`jump <address>`	Basic block ended in an unconditional jump to a specific address
`jump-ind <address>`	Basic block ended in an unconditional indirect jump; actual jump target address printed
`ret <address>`	Basic block ended in a function return; actual return address printed
`call <address>`	Basic block edned in a function call at a specified address
`call-ind <address>`	Basic block ended in an indirect function call; actual function address printed
`branch <address> <taken>`	Basic block ended in a conditional branch to a specified address; either `taken` or `not taken` printed depending on result
`eret <address>`	Basic block ended in an exception return; actual return address printed
`invalid`	Basic block ended in an invalid opcode, which really should not happen in normal operation

8. Import into Lighthouse

`trace_to_lighthouse.py` is a simple Python script to take a set of trace files and produce a file that can be read by Lighthouse by limiting the output to just the addresses that were executed during the trace. Note that this is a lossy operation as it is only coverage data and not full program flow.

Run `python3 trace_to_lighthouse.py /path/to/output.trace > /path/to/output.coverage`.

With Lighthouse installed in IDA Pro, click `File->Load file->Code coverage file` and select the `output.coverage` file. Blocks that were executed will be colored blue.



A Note on CoreCrypto

The CoreCrypto kernel extension is subject to integrity checks for FIPS 140 compliance and therefore cannot be patched. Since the tracepoints are replaced at runtime with instructions that trap to the hypervisor, this would fail the integrity checks and crash the device. CoreCrypto must be excluded from the tracepoints when running `btgen`.

A Note on STX/LDX Instructions

Performing dynamic instrumentation of ARM/Aarch64 has some issues when load-exclusive/store-exclusive instructions are involved. Consequently, `btgen` contains special handling for these instructions that prevents installing tracepoints between an `LDX` instruction and its corresponding `STX` instruction. As these are used as a primitive on which locks are built, they are used frequently throughout the iOS kernel. Should a tracepoint inadvertently be placed in between these instructions, the kernel will loop indefinitely and likely panic shortly thereafter due to a lock timeout. This is something to keep in mind if re-implementing or expanding `btgen` to support other kernels.

Capturing Traces from Custom Applications

The functionality of `hyptrace` is fairly simple and can be extracted and used by other applications. The necessary calls are implemented in `hyptrace_hvc.s` and `hyptrace_hvc.h`. The key steps are as follows:

1. Allocate an output buffer. Note that this should be page-aligned.
2. Call `hyptrace_set_buffer`, providing the address of the buffer and its size.
3. For single-thread tracing, set the thread filter by calling `hyptrace_set_filter` and providing an array of thread IDs to trace. Generally this will be the result of calling `hyptrace_get_threadid`. Skip this for full-system tracing.
4. Set the trace points by calling `hyptrace_set_points` providing a buffer and size. The buffer must contain the compiled tracepoints as generated by the `btasm` utility.
5. Execute whatever operations are desired for tracing, e.g. making system calls, interating with IOKit objects, etc.
6. Un-set the output/tracepoints buffers by calling `hyptrace_set_buffer(NULL, 0)` and `hyptrace_set_points(NULL, 0)`.
7. Save or process the output buffer.

Capturing Full-System Traces

`btgen` and `hyptrace` check environment variables to determine whether to apply a thread filter. When running in full-system mode, all threads will be traced, capturing a full-system trace. Provide `BTGEN_FULL_SYSTEM=1` when running `btgen` and `HYPTRACE_FULL_SYSTEM=1` when running `hyptrace` to enable this functionality.