Mini Dumps

Efficient core dumps for FlashBlade

Outline

- Motivation
- Simple fixes that don’t fix it (MADV_DONTDUMP)
- Core file, signal handlers
- Summary of solution
- The devil is in the details
- Caveats and future work

Motivation

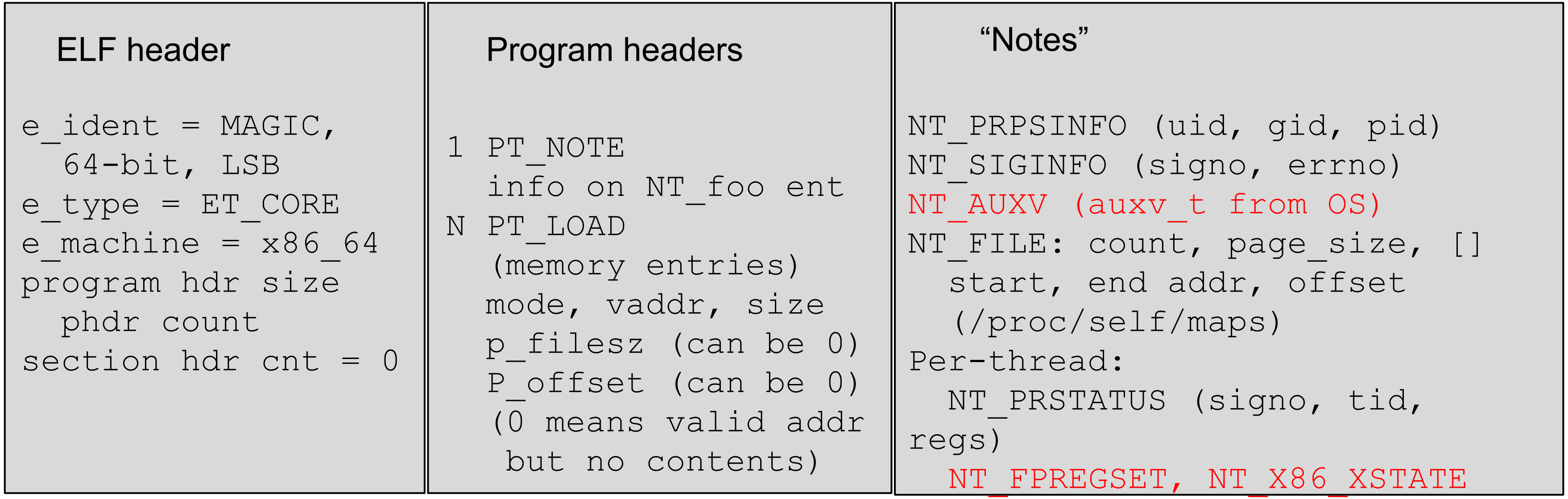
- Core dumps are super useful for debugging
- But kernel-driven core dumps dump everything in memory:
  - Customer data, secret keys, etc.
  - 30+ seconds to dump
  - 54GB dumps (before compression)
- Desire for faster dumps
- Desire for smaller dumps (with little loss of debuggability)
- Desire for eliminating regions from the dump
- But it still needs to be readable using gdb

Why not MADV_DONTDUMP?

- Not a safe function call at dump time
    - So we would pay a cost at runtime for an event that may never happen
    - Kernel lock to protect data structure
- Interactions with memory allocator
    - We allocate 2MB pages in 64MB chunks and re-use them
    - No-dump regions change quickly
- Applies at at least 4k granularity
    - We may want finer-grained control since important data and customer data could share a page
- Still many GB dumped we probably don’t need

What’s in a core file?

- `readelf -a core` shows us the contents



ELF header

```
e_ident = MAGIC,
  64-bit, LSB
e_type = ET_CORE
e_machine = x86_64
program hdr size
  phdr count
section hdr cnt = 0
```

Program headers

```
1 PT_NOTE
  info on NT_foo ent
N PT_LOAD
   (memory entries)
  mode, vaddr, size
  p_filesz (can be 0)
  P_offset (can be 0)
   (0 means valid addr
    but no contents)
```

“Notes”

```
NT_PRPSINFO (uid, gid, pid)
NT_SIGINFO (signo, errno)
NT_AUXV (auxv_t from OS)
NT_FILE: count, page_size, []
  start, end addr, offset
   (/proc/self/maps)
Per-thread:
  NT_PRSTATUS (signo, tid,
regs)
  NT_FPREGSET, NT_X86_XSTATE
```

Signal handlers

- Program errors: `SIGSEGV`, `SIGINT`, `SIGILL`, possibly others
- `std::abort`: `SIGABRT`
- We can catch all of these
- But can we do enough in a signal handler?
    - `man 7 signal`
    - Safe function calls: `open()`, `read()`, `write()`, `mkdir()`, etc.
        - Most file I/O can be done in a signal handler
    - No memory allocation though
- We can do file I/O, and we know what a `core` file looks like

Summary: What to dump?

- Need Per-thread state
    - Registers at dump time
    - Signal info for crashing thread
- “Interesting” memory contents
    - Stack contents (how far back?)
    - Memory near any register value that looks like a pointer
    - Memory near any stack value that looks like a pointer
    - Also chase anything in those blocks that may be a pointer?
    - How much around each “anchor” address?

Summary: What to dump?

- Main thread info comes from signal handler
- Send signal to other threads to dump their per-thread info
- One thread must dump the memory contents and memory map

So that’s really it -- we have a signal handler which pokes all the other threads (`SIGUSR1`); we read `/proc/self/maps`, and we write out interesting memory contents

Details: Opt-in

- Signal handler only registered via `ps_init_platform()` call to `setup_signal_handlers()`
    - Need copy of `argv` for `NT_PRPSINFO`'s `pr_psargs` field
- Need to reserve memory for unknown quantities
    - `PT_LOAD` info for dump contents, `NT_FILE` info for memory map
        - Need one per line in /proc/self/maps, plus splitting when not fully dumped
        - Assume we have at most 128GB memory and map in 2MB chunks; 64k entries max
    - Space for per-thread info
        - 2 * k_max_execution_count
    - Bounce buffer for memory contents that can't be written directly: 4kB
    - String storage for file names from `/proc/self/maps` since we read that file in chunks

Details: signal_received()

1. Check for cascading signals and exit if reentrant too many times
2. fflush() stdout, stderr; setvbuf to non-buffered
   - Not documented as signal safe but seems to work
3. Signal other threads (only on first entrance to handler)
4. Dump registers, stack to stdout for the logs
   - Need `sig_printf()` [format to local buffer then `write(STDOUT_FILENO)`] to be signal safe
   - Internally we also tell minidump about stack range, registers
5. One thread dumps /proc/self/maps to logs
6. Wait for all threads to tell minidump about their registers + stack
7. Write minidump file
8. Wait for all threads before re-signalling or returning so we get correct exit status

Details: signalling other threads

- Need to know all threads that are running, so we can signal at dump time
  - Since we start threads, we can memoize `pthread_t` and OS thread id (`gettid()`) on start
  - `pthread_kill()` other threads
- `SIGUSR1` is unused for other purposes

1. Print registers to log
2. Tell minidump about our `siginfo_t`
3. Print stack to log
4. Print some stack contents to log
5. Determine valid stack range and tell minidump
6. Spin forever, `sleep(100)`

Details: per-thread valid stack range

- Limit to 64kB
    - We just need some reasonable number
    - Allocated stack is 8MB; sigaltstack is 512kB
- We could memoize valid stack addresses on thread start
- Currently we “probe” by writing to a nullfd
    - Use /dev/random instead of /dev/null so the handler isn’t a no-op
    - write() returns >= 0, it was a valid address

Details: /proc/self/maps

- We parse each line of `/proc/self/maps`
    - Read 1kB at a time (need some fixed buffer size)
    - Print to log on crash
    - Pass to minidump to parse text contents
- Each line looks like
    - `<hex begin>-<hex end> <permissions> <hex offset> <dev-id> <inode> <filename>`
    - Filename is optional, could be special tags like `[stack]`
    - Permissions are `rwx[ps]` with – for no permission; ‘p’rivate or ‘s’hared
    - Offsets indicate different mapped portions of a file
    - Ignore devid and inode

Details: /proc/self/maps

- Memoize the range and name, determine initial dump plan
    - Don’t save filename for `/anon_hugepage`
    - Determine “real” file based on leading `/` (e.g. `/lib/x86_64-linux-gnu/libc-2.1.19.so`)
    - Never dump unreadable or shared mappings (we don’t use shared mmap)
    - Always dump files
        - in theory file content can be determined from symbols, but it’s only a few 100kB
    - Always dump special regions, e.g. `[stack]`, `[vdso]`, `[vsyscall]`
        - `[vsyscall]` requires memcpy then write(2)
    - Assume entries not 2MB aligned are ‘interesting’ and always dump
        - I don’t recall what’s here but it’s not too many bytes
    - Default no-dump everything else
        - DMA memory, malloc contents
        - We will only dump things from anchor addresses

Details: anchor addresses

- From a value that looks like a memory address, ensure data from [addr - before, addr + after) is in the dump
  - `add_anchor_addr()` for each register, adding 1k before and 8k or 1M after
  - `add_anchor_addr()` for each word in the stack, adding 128 bytes before and 1k or 4k after
  - Larger ranges for the faulting thread
- Keep all known memory mappings in sorted order; look for entry overlapping anchor address (4k aligned)
  - We don’t use convenient containers like std::map because they allocate on insert
    - We could provide an allocator that draws from a pre-allocated pool of nodes
  - Split entry if needed
  - Check if next entry is adjacent and shift 4k from one to the next (common case)
  - Be careful with `offset` field for files

Details: write minidump file in order

- `core` file has an ordering for each section
    - We emit per thread and per process info in the same order
    - Crashing thread first
- Log how much data we kept from anchor addresses, how long minidump took

```
INFO: Keeping 59344 kB from mapped files (e.g. shared libs) and 259292 kB
from other sources; elided 1232792 kB
INFO: Adding anchor addresses for 15 threads in 252 maps
INFO: Added 1796 kB from anchor addresses
INFO: Cannot open directory /ssd/core (errno 2); falling back to "."
INFO: Writing minidump for 15 threads, 456 mappings
INFO: Emit NT_SIGINFO for LWP 21449
INFO: Emit NT_FILE for 130 entries, 9398 bytes
INFO: minidump took 1639 ms
```

Details: process monitoring

- Process may appear up even if heartbeat is non-responsive
  - It takes non-zero time to write a minidump
  - `kill -9` and restart during this time would be tragic
- Give a 10 second extra buffer for dump to succeed before kill + restart

Caveats

- We currently have no sparse virtual addresses
    - So if it’s in `/proc/self/maps`, and it’s one of our 2MB chunks, it’s also physical
    - This is a solvable problem since we own the memory allocator

Future work

- API to mark ranges as “never dump” or “always dump”
    - Desire to only mark things “always dump” at startup, not forward operation
        - Per-server or per-authority data structures that are of high value
        - And to not mark too much as “always dump”
    - Desire to rarely mark regions as “never dump”
        - AES keys no longer in process
        - Mark DMA buffers and messaging buffers as no-dump?
        - Lock-free crash-safe table? We can’t afford to segfault when iterating
- Save `NT_FPREGSET` and `NT_X86_XSTATE` regions too?
    - So far these haven’t been missed

Summary

- Signal handler can write a valid `core` file
- Signal other threads to get their registers and stack at crash time

From an early crash in a runtestsD test:

```
$ ls -l minidump core
-rw------- 1 mdf mdf 1.4G Jun 18 18:22 core
-rw-r--r-- 1 mdf mdf 313M Jun 18 18:22 minidump
```

- Usable core dumps in an order-of-magnitude less space and time