Post-Mortem Kernel Debugging with netdump(4)

BY MARK JOHNSTON

FreeBSD kernel panics are a hopefully rare occurrence, but they do happen from time to time. You may have been unlucky and hit a kernel bug on a production system, or perhaps you are developing a kernel patch and uncovered a bug while testing. In such situations, a reboot will bring the system back online, but the contents of RAM will be lost, making it impossible to find the root cause of the panic.

FreeBSD supports both live debugging and post-mortem debugging of kernel panics. While live debugging is often simpler, its nature means that the panicked system cannot be rebooted until the developer has finished debugging. This is often impractical, so post-mortem debugging via core dumps is a common debugging activity. For a long time, the FreeBSD kernel has had the ability to save a core dump, often called a "kernel dump," following a panic; once a kernel dump has been saved, the panicked system can be rebooted and brought back online, and the dump can then be used to diagnose the problem.

FreeBSD supports both live debugging and post-mortem debugging of kernel panics.

When a userspace program crashes and dumps core, the operating system saves its state in a regular file somewhere in the file system. When the kernel crashes, though, life is not as straightforward: the kernel itself is responsible for mediating access to its file systems, and following a panic the kernel is by definition in an inconsistent state, so writing data to a file is a fraught endeavor. A kernel panic is bad enough, but it'd be much worse if the kernel went on to corrupt its own file systems!

FreeBSD's traditional solution to this problem is to write the kernel dump to a raw disk partition, often the same one used for swap space. Doing so is much simpler than modifying a file system, and since swapped-out data does not persist between reboots, there is little risk of overwriting important data.

Configuring kernel dumps is easy: in `/etc/rc.conf`, set the dumpdev variable to the name of the disk device to which kernel dumps should be saved, or set it to the string `"AUTO"` if the swap partition is to be used. Under the hood, this mechanism uses `dumpon(8)` to tell the kernel which disk device to use. When booting up following a panic, FreeBSD will automatically run

`savecore(8)`, which reads the saved kernel dump and places it in `/var/crash` for later use.

Disk-based kernel dumps work well so long as the system has a spare partition where they can be saved. This is not always the case, though: systems might boot disklessly and not have any persistent storage at all, or, as is common with embedded devices, there might not be any disk space to spare. In these situations one historically had to resort to live debugging, or a one-off hack like using a USB thumb drive to store the kernel dump. However, as of FreeBSD 12.0 there's a better way!

Introduction to `netdump`

`netdump(4)` is a relatively new feature which lets a panicked FreeBSD transmit a kernel dump over a network before rebooting. In short, it uses a custom UDP-based protocol to transmit the contents of RAM to a server, implemented by `netdumpd(8)` (`ftp/netdumpd` in the FreeBSD ports system). This lets one get a dump from a panicked kernel without requiring any local storage on the system.

It should be stated up front that `netdump` does not perform any encryption or authentication, so the contents of kernel memory are transmitted directly over the network. Since kernel memory typically contains secret information, it is important to use `netdump` only on trusted networks.

`netdump` has a long history: it started life circa 2000 as FreeBSD 4 patch by Darrell Anderson at Duke University, and was ported forward over the years by developers at several FreeBSD-using companies. It was finally committed to the FreeBSD src repository in 2018 and first became available in FreeBSD 12.0.

Internally, `netdump` is built on top of `debugnet`, a standalone IPv4/UDP implementation which is specialized to be usable in a panicked kernel. In particular, `debugnet`'s UDP stack runs in a single thread, does not perform any heap memory allocations, and does not block (e.g., to wait for an interrupt or a mutex). These constraints come from a need to minimize the complexity of kernel code which executes after a panic: since the kernel has already crashed, `netdump` must avoid making the situation worse while it does its job.

Because `debugnet` transmits and receives packets, it needs to be able to talk to network interface controller (NIC) hardware. Thus, individual NIC drivers require modification in order to be used by `netdump`. Typically this modification consists of adding a "polling" mode to the driver's packet transmission and receive paths. In practice the required modifications are straightforward to implement and typically involve adding less than 100 lines of C code to a given driver. Many widely used drivers implement `debugnet` support today, including all of the Intel drivers (in fact, all drivers implemented using the `iflib` framework), modern Mellanox drivers, the VirtIO network driver, and several drivers for GigE NICs often found in desktop systems or server management ports; a full list is given in the `netdump(4)` manual page.

Finally, `debugnet` hooks into the kernel's packet buffer allocator. This is because driver code will continue to use the standard `mbuf(9)` allocator interface to allocate buffers after a panic, but `netdump` needs to avoid relying on the standard allocator. During system initialization, `debugnet` pre-allocates and reserves memory for use after the kernel panics, thus ensuring that mbuf allocations will be successful and won't perturb the state of the kernel more than necessary.

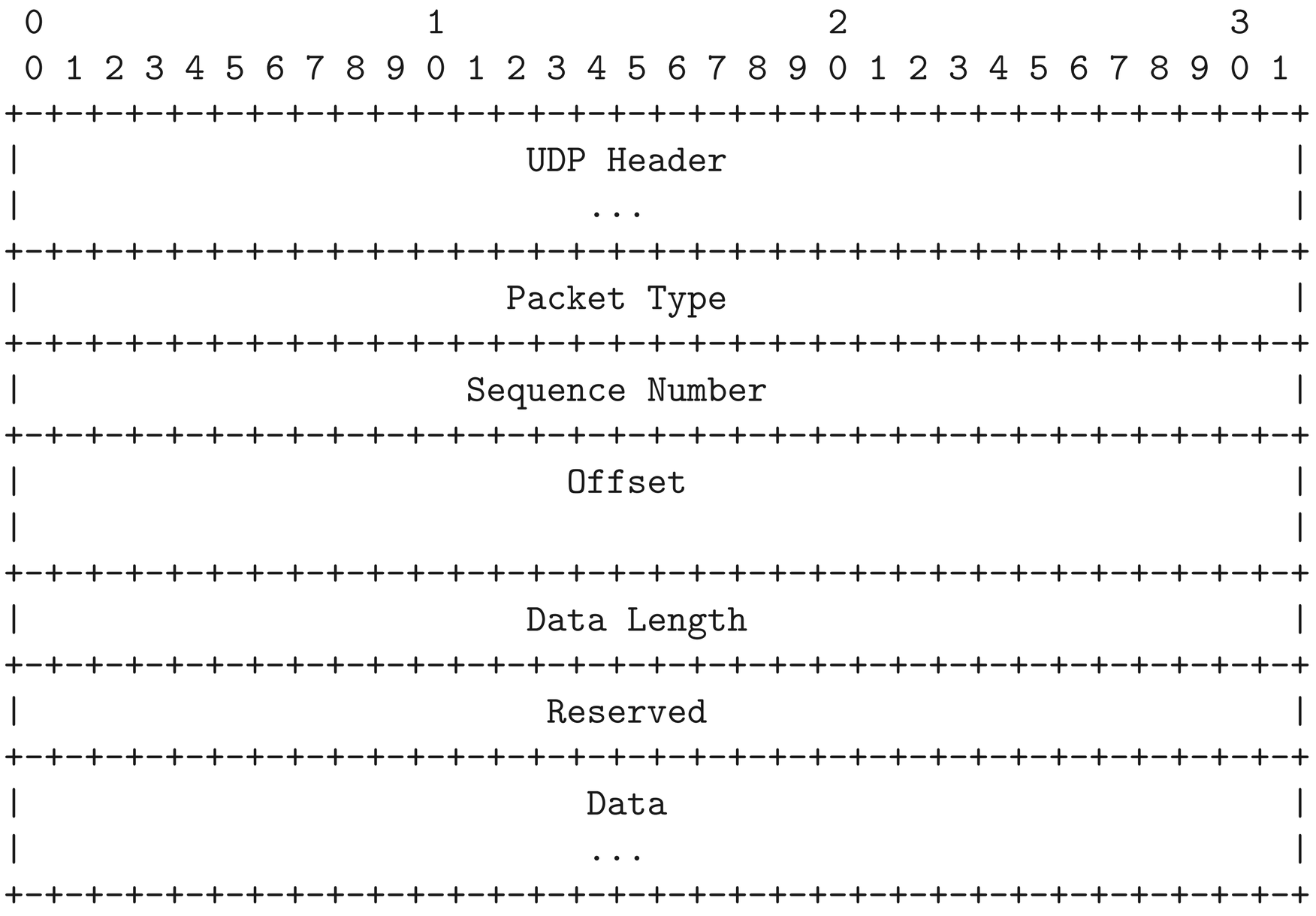
The `debugnet` Protocol

The `debugnet` protocol, in keeping with the requirements of `netdump`, is very simple and specialized to its task. It is implemented on top of UDP and currently uses only IPv4; IPv6 could be supported as well, but so far this has not been implemented.

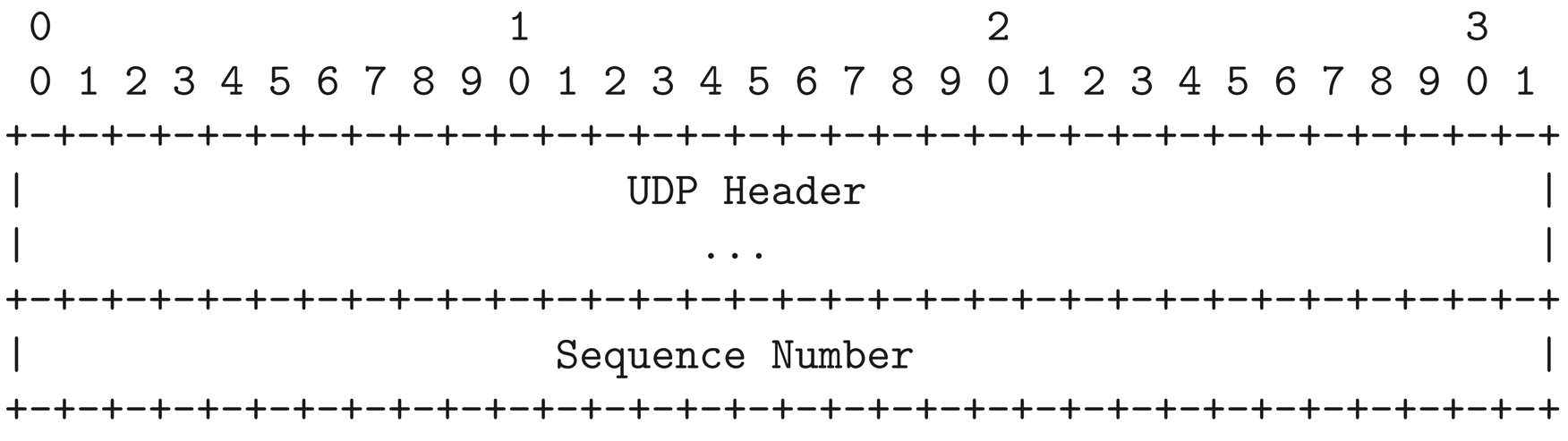
`netdump`s are initiated by the panicking system, which acts as a client, with the server imple-

mented by `netdumpd`. The `debugnet` protocol has two packet types: client messages, and acknowledgements:

Message:



Acknowledgement:



When initiating a `netdump`, the client first has to discover the MAC address of the next-hop router. To do so, its configuration includes a "gateway" IP, for which `debugnet` broadcasts ARP requests.

Once the router address is known, the client first sends a message with type `NETDUMP_HERALD` (1) to the server on port 20023. This establishes a session with the server, which binds to an ephemeral port and sends an acknowledgement to the client at port 20024. All subsequent packets sent by the client go to this ephemeral port. All client messages are acknowledged by the server.

Once the session is fully established, the client begins transmitting kernel dump data. Messages containing this data have type `NETDUMP_VMCORE` (3). Each message gets a unique sequence number and specifies the offset and length of the data relative to the beginning of the kernel dump file. Upon receipt of a `NETDUMP_VMCORE` message, the server writes the data at

the corresponding offset in the dump, and then transmits an acknowledgement. The client will typically transmit a burst of chunks of data and wait for acknowledgements to arrive for all of them before continuing.

Once all of the kernel dump data has been transmitted and acknowledged, the client provides some metadata describing the panic in a `NETDUMP_KDH` (4) message, and then completes the session with a `NETDUMP_FINISHED` (2) message. At this point, the kernel dump is available on the server's file system and can be used for debugging.

Configuring `netdump`

Armed with some knowledge of how `netdump` works under the hood, we can explore its configuration. There are, effectively, four configuration variables that `netdump` needs to work:

1. the client IP address
2. the server IP address
3. the gateway IP address
4. the interface to use (e.g., `em0`)

Just as with traditional disk-based kernel dumps, `netdump` can be configured with `dumpon`(8). For example, with a client at 10.0.1.157 on `vtnet0`, the server at 10.0.1.236, and a gateway at 10.0.1.1, one can configure `netdump` like so:

```
# dumpon -c 10.0.1.157 -s 10.0.1.236 -g 10.0.1.1 vtnet0
```

Then, on the server, `netdumpd` can be run as a foreground program

```
$ netdumpd -d . -D -P ./netdumpd.pid
netdumpd: default: listening on all interfaces
Waiting for clients.
```

This will cause kernel dumps to be saved in the current directory, as specified by the `-d` flag. To test the setup, we can manually trigger a panic and tell the kernel to dump core:

```
# sysctl debug.kdb.panic=1
debug.kdb.panic: 0panic: kdb_sysctl_panic
cpuid = 1
time = 1655412790
KDB: stack backtrace:
db_trace_self_wrapper() at db_trace_self_wrapper+0x2b/frame 0xfffffe007c573af0
vpanic() at vpanic+0x151/frame 0xfffffe007c573b40
panic() at panic+0x43/frame 0xfffffe007c573ba0
kdb_sysctl_panic() at kdb_sysctl_panic+0x61/frame 0xfffffe007c573bd0
sysctl_root_handler_locked() at sysctl_root_handler_locked+0x9c/frame
0xfffffe007c573c20
sysctl_root() at sysctl_root+0x213/frame 0xfffffe007c573ca0
userland_sysctl() at userland_sysctl+0x187/frame 0xfffffe007c573d50
sys___sysctl() at sys___sysctl+0x5c/frame 0xfffffe007c573e00
amd64_syscall() at amd64_syscall+0x12e/frame 0xfffffe007c573f30
fast_syscall_common() at fast_syscall_common+0xf8/frame 0xfffffe007c573f30
--- syscall (202, FreeBSD ELF64, sys___sysctl), rip = 0x8011a773a, rsp =
```

```
0x7fffffffd938, rbp = 0x7fffffffd970 ---
KDB: enter: panic
[ thread pid 784 tid 100098 ]
Stopped at      kdb_enter+0x32: movq    $0,0x1279963(%rip)
db> dump
debugnet: overwriting mbuf zone pointers
debugnet_connect: searching for gateway MAC...
netdumping to 10.0.1.236 (02:9a:88:79:b5:0a)
Dumping 257 out of 4057 MB:..7%..13%..25%..32%..44%..56%..63%..75%..81%..94%
netdump finished.
debugnet: restoring mbuf zone pointers

Dump complete
```

On the server, we should see something like the following:

```
New dump from client devvm [10.0.1.157] (to ./vmcore.devvm.0)
.................(KDH from devvm [10.0.1.157])
Completed dump from client devvm [10.0.1.157]
```

Now we have a kernel dump in the directory specified by the `-d` flag!

In this example, the client and server are on the same link. The gateway parameter is thus redundant and can be omitted:

```
# dumpon -c 10.0.1.157 -s 10.0.1.236 vtnet0
```

Configuring `netdump` using `/etc/rc.conf` is a bit trickier. If the relevant IP addresses are static, then they can be passed using the `dumpon_flags rc.conf` variable. If not, one may instead use a hook in the system's DHCP client to invoke `dumpon` once the client address is known. The `dumpon.8` manual page provides an example of how to do this with `dhclient(8)`.

Starting in FreeBSD 14.0 and 13.2, `debugnet` will be able to infer a client address in most situations, simplifying configuration.

`netdump` On The Fly

One limitation of `netdump` was the need to configure it in advance of a panic. Starting in FreeBSD 13.0, it is possible to configure `netdump` *after* a panic, from DDB (the in-kernel debugger). This is done using DDB's `netdump` command:

```
# sysctl debug.kdb.panic=1
...
Stopped at      kdb_enter+0x32: movq    $0,0x1279963(%rip)
db> netdump -s 10.0.1.236
debugnet: overwriting mbuf zone pointers
debugnet_connect: searching for server MAC...
netdumping to 10.0.1.236 (02:9a:88:79:b5:0a)
Dumping 258 out of 4057 MB:..7%..13%..25%..31%..44%..56%..62%..75%..81%..93%
netdump finished.
```

```
debugnet: restoring mbuf zone pointers

Dump complete
```

Next Steps

A kernel dump on its own is not very useful: debuggers require that a core dump be paired with an exact copy of the kernel and its debug info. When packaging up a core dump to send to a developer, be sure to include the matching kernel. By default, kernel debug info is split into separate files under `/usr/lib/debug`. Thus, it is usually safest to include the following:

1. the kernel dump file (usually `vmcore.<something>`)
2. the contents of `/boot/kernel/`
3. the contents of `/usr/lib/debug/boot/kernel/`

`netdumpd` sports a `-i` flag which can be used to specify a script to run after a `netdump` completes. This can be used to perform post-processing of kernel dumps. A discussion of kernel debugging itself is outside the scope of this article, but a past article provides lots of information.

`netdump` can be very useful for debugging the kernel in certain environments, but has some limitations. A few mentioned already include the lack of confidentiality, missing IPv6 support, and fixed port numbers. If you find yourself running into such limitations (or bugs, for that matter!), please be sure to report the problem in the FreeBSD project's bug tracker or on the project mailing lists.

MARK JOHNSTON is a software developer and FreeBSD src committer living in Toronto, Ontario, Canada. He currently works for the FreeBSD Foundation and is interested in most aspects of operating system development. When not sitting in front of a computer he enjoys playing in a city dodgeball league with friends.