Pretty-printing of kernel data structures

Daniel Lovasko
Charles University in Prague
lovasko@freebsd.org

Abstract

One of the key features of a debugger is the ability to examine memory and the associated data structures. For a long time, DDB, the FreeBSD kernel debugger, has been shipping with limited functionality in this area. By borrowing a core part of the DTrace technology, the Compact C Type Format (CTF), we aim to provide a convenient and self-maintainable way for DDB to pretty-print all C data structures used in the currently loaded kernel image. In order to satisfy technological and licensing criteria we developed a custom library implementation of the CTF. While implementing the idea, we faced many challenges, such as intelligent printing of recursive data structures, avoiding the need to perform disk I/O inside the debugger, supporting cross-compilation by being endian-independent or designing the library to achieve predictable speed and memory consumption. We conclude with musings about other potential usage of the format in FreeBSD and we discuss the possibility to extend the format to support C++ classes.

1 CTF

While developing the DTrace software, team at Sun Microsystems was presented with a challenge to find a way to represent C data types. The existing formats such as the DWARF or STABS are either too bloated with debugging information outside the needed scope or do not store the data in an efficient way so that it could be stored on a production system.

The result was the Compact C Type Format (CTF) that had few goals in mind: space minimalism which is present in every aspect of the technology, and the ability to represent complex C types (e.g. constant pointer to a function). This paper discusses the second version of the format.

1.1 Overview

Due to historical reasons, the CTF data is located in an ELF section named `.SUNW_ctf`. The whole binary blob can be divided into 6 sections: format header, label section, type section, function section, data object section and the string table.

The header contains the offsets of the consequent sections, parent relations, and a flag field that currently indicates only whether the rest of the data is compressed.

As mentioned and established above, the CTF is a very efficient way to store the C type information. Essentially, the format distinguishes between the following objects: integers (which contain characters and `void`), floating point numbers, `struct`s and `union`s, pointers, arrays, functions, `typedef`s, forward declarations, and qual-

ifiers `const`, `restrict` and `volatile`. Each object is assigned a non-negative two-byte integer ID which is used by other objects to reference it (e.g. a pointer is always referencing an another object). An algorithm that follows such references has to be careful not to follow cyclic references and therefore avoid hanging.

Apart from storing the pure type information, CTF contains data that is interrelated with the symbol table - namely the types of functions and data objects (global variables). CTF associates this data with the symbol table by using the same order of objects.

The CTF string table is defined to be complementary to the ELF string table, effectively preventing data duplication.

1.2 Labels and merging

After a quick observation of the kernel code, it is obvious that a large portion of the types is used in both the kernel and its modules (e.g. commonly used integer types `intXX_t`, `cpu_t`, ...). Since kernel modules are always present with the kernel (but not necessarily the other way around) we can safely assume that they can depend on the types defined in the kernel. By substituting the actual type data from the kernel module CTF section with references to the CTF data of the kernel file, we are able to save space. This process is called merging.

Taking the inheritance to more extreme lengths by creating a system-wide parent-child CTF relationships between shared objects and binaries is discussed in the section 7.3.

The label section consists of label records - a pair of associated ID and a name. They serve a purpose of protecting type entries that are referenced from outside of the data set to be overwritten in case of a parent update.

1.3 Toolset

In order to examine, create and manipulate the CTF data on a higher level a toolset of command-line utilities was created. The most basic tool used to inspect the CTF data is the `ctfdump` utility. Its purpose is to print the data information along with internals such as IDs. Optionally, it can compute statistics about the types - average number of members of a `struct`, highest function argument count, etc. In our adaptation of this tool, we separated this functionality into the standalone application called `ctfstats` to be more aligned with the UNIX philosophy. Since there is no support for generating the CTF data in compilers, another way had to be found: the `ctfconvert`. Traditionally, all major compilers are able to produce `STABS` or `DWARF` data. Fortunately, DWARF is a superset of CTF and therefore we are able to obtain the required data. Last but certainly not least, `ctfmerge` is designed to perform the merging and copying of data sets.

1.4 Contemporary presence

The presence of the CTF data for the FreeBSD kernel is motivated largely by the DTrace software. During the build of the kernel the `make` script calls the `ctfconvert` repeatedly to convert each kernel and kernel module object so that it contains the CTF data too. Afterwards, all the CTF data is merged against the kernel file. This procedure happens if and only if the `WITH_CDDL` and `WITH_CTF` options are selected.

2 DDB

DDB is an interactive kernel debugger that supports unique features such as live single-stepping

the kernel code or inserting breakpoints to the kernel code. It can be used to inspect processes, threads, global variables, and many more objects. Apart from examining specific objects, DDB can be used to investigate hangs caused either by a deadlock or a livelock. Thanks to being an integral part of the kernel code, DDB is always available (except when explicitly excluded from the kernel configuration file).

This paper focuses on the DDB's ability to examine memory, specifically reading the of values of variables and symbols.

3 Problem

One of the crucial parts of debugging any algorithm or code in general is the content of variables. Being able to spot a erroneous value might be a solid lead to the problematic code. Providing the kernel developers with a facility to inspect the memory with greater freedom from hardship might speed up the bug fixing process along with implementation of new features.

4 Current situation

The problem of pretty-printing kernel data structures was tackled before with an approach that requires linear amount of work to the number of structures present. Examples of such commands in the current DDB version are: `show bio`, `show buffer`, `show domain` and others. In case of a different structure that is not included in this list, user needs to leave the comfort of such commands and retrieve back to the raw commands `print` and `examine`. Using these commands to study contents of a memory that contains complex (possibly even nested) data structure takes tremendous amount of time and can even introduce some concentration-related mistakes. Such finite list approach combined with raw word-sized memory reading is a painful and insufficient.

5 Solution

The available CTF data is a perfect fit for such task: its minimal memory requirements allow it to be stored on disk and loaded to the main memory when necessary, and contains information about the all used types to a great detail. One of the obvious benefits is the universality of such approach, where no additional code needs to be written to support new types introduced in new iterations of the kernel code. Important addition is also the total reduction of human-induced mistakes, since the CTF data is generated directly from the DWARF data that come from the source.

6 Implementation

6.1 Library

We developed a BSD-licensed implementation of the CTF called `libctf`. It aims to replace the old `libctf` written at Sun. Technical reasons for such change are described below.

6.1.1 Well defined type sizes

The original Sun `libctf` implementation is using the classic C types (`int`, `short`, ...) to represent all CTF internal information. Thanks to vague definitions of the C type sizes, the CTF data will have different size on a different platform. Our implementation uses the standard `stdint.h` types - `uint8_t`, `int32_t`, etc. This

has two main benefits: the size is strongly defined beforehand and the same CTF data can be manipulated or even created on a platform with an `int` of a different size.

6.1.2 Endianness

The old Sun implementation of the `libctf` does not handle the endianness at all. While cross-compiling a kernel on a machine with a different endian than the target machine, user is interested in generating the CTF data too. Even though it will not produce any error, problems will arise upon reading the data on the target machine. Our `libctf` uses big-endian for storing the data on the disk and converts them to hosts endianness upon reading, much like the TCP/IP network stack.

6.1.3 Kernel space

Kernel space usage of the `libctf` was planned from the beginning of the project. Reading and parsing of the CTF data works in both kernel and user space without any change in the user-presented API.

6.1.4 Unit and integration tests

Taking the inspiration from the modern software movement, `libctf` contains unit and integration tests.

6.2 Debugger

We introduced a new command for DDB - `prettyprint` with a short `pp` alternative. The command takes two main arguments: an address, and a name of a data structure that should reside at the address. Optional features include hexadecimal output instead of the decimal and enabling the typedef chain solving.

6.2.1 Caching

Performing I/O operations inside the kernel debugger is very tricky due to its own subsystems that do not support disk reading/writing. Since the CTF data is located inside the ELF section of the kernel binary `/boot/kernel/kernel`, it must be loaded into the main memory before entering the debugger. An ideal place for such procedure is the function that resides in file and is triggered after setting `sysctl` entry `debug.kdb.enter` to 1.

Fortunately, the already present function `link_elf_ctf_get` loads the right byte blocks from the kernel file using kernel space methods and even implemented caching, so that the data can be loaded once (in our case before entering the debugger) and repeated calls do not perform any I/O due to the cached result.

6.2.2 Pretty-formatting

The output of the `prettyprint` command aspires to be as similar to the actual C declaration. The major difference is excluding the final semicolon in favor of an equals sign followed by the value of the variable.

6.2.3 Typedef chain

A common practice is to create `typedef`ed variants of basic types without any change, to introduce a more clearer type naming. This technique can be applied in multiple layers, creating a `typedef` chain. The `prettyprint` command aims to solve these chains, by presenting each link.

6.2.4 Common data structures

If the pretty-printing algorithm would ignore specific implementation details of even a little complex data structures, like linked lists, it would become unusable. Without any alteration the algorithm would include new level of indentation with every element in the list and therefore make it unusable for lists of length 4 or more, considering the 80 characters limit of the DDB terminal.

An approach that ignores this would still be correct, but also the exact opposite of user friendly. As it turns out, not only linked lists demand special attention, but almost every recursive data structure. The design decision was to include this distinctive method for all the data structures commonly used in kernel that are defined in header files `sys/queue.h` and `sys/tree.h`.

7 Future use of CTF

7.1 Kernel modules

In the present-day implementation, the search algorithm is looking for types only in the kernel CTF data. Extending it to include the modules will broaden the debugging possibilities in a significant way.

7.2 System-wide linking

Relationships like the one between the kernel and its modules exist in many instances: `libc.so` and base userland utilities, Gtk+ shared objects with Gtk+ applications, Apache and its modules, etc. With the new `libctf` implementation even multilevel inheritance is supported. This means that a Gtk+ application written in C could be CTF-merged against Gtk+ shared objects which will be in turn CTF-merged with the `libc.so`. This will shrink the memory requirements even more and allow for broader adaptation.

7.3 Symbol table

Current implementation of the pretty-printing does not interact with the symbol table, which might be improved. Combining the information from the symbol table, CTF sections regarding function and data objects might result in improved ease of debugging of these objects.

7.4 Compilers

After adding support for writing the CTF data in `libctf`, its inclusion due to the simplicity of use and permissive license can be planned in compiler suites so that they can emit the CTF data and the `ctfconvert` utility can be deprecated.

7.5 Machine-readable output

Possible addition to the CTF command-line toolset, mainly `ctfdump`, is the `libxo` library from Juniper Networks that provides a structured machine-readable output in various formats.

8 Extensions of CTF

Prevailing limitation of the CTF comes directly from the name of the format - it is focused exclusively on the C language and its types. Therefore its very cumbersome to use it with other languages, e.g. C++. In order to inspect a simple `std::string`, user needs to replicate the

memory structure used in his C++ implementation. Such use in DTrace leads to a code that is compiler suite specific which leads to limited portability. Possible solution would be to create a next version of CTF, named CTF++, along with updated toolset that will support C++ classes, their access restrictions, inheritance relationships and member methods and data fields. This change would enable to use DTrace to inspect C++ applications like Firefox, Octave, MongoDB and many others.

Acknowledgements

- George Neville-Neil
- Pedro Giffuni
- Robert Mustacchi
- John-Mark Gurney
- Robert Watson