CTF implementation and use in the kernel debugger

Daniel Lovasko

Dear Reader,
this is my proposal to The FreeBSD Project for the Google Summer of Code 2014.

General Information

Name

Daniel Lovasko

E-Mail

daniel.lovasko@gmail.com

Phone

+421 904 329 363
+420 774 525 128

IM/IRC

GTalk
`dlovasko` on freenode and EFnet

Bio

- vim user
- UNIX aficionado
- The Beatles fan
- C enthusiast
- open source evangelist
- tea savourer
- MacBook owner
- Haskell experimenter
- bullet list devotee

Motivation

I was a long time GNU/Linux user, but for more than a year now my most used operating systems are FreeBSD and OS X used on desktop and MacBook, respectively. My enlightening experience with FreeBSD and passion for systems programming in C were the impulses for this proposal.

University

Charles University in Prague
Faculty of Mathematics and Physics
Fourth year bachelor student in Informatics
Subject field Programming

Erasmus Exchange University

Saarland University
Faculty of Natural Sciences and Technology I (Mathematics and Computer Science)

Open Source Involvement & Work Experience

SUSE Linux Between January 2012 and July 2013 I was employed as a Software Developer. My work revolved around packages along with maintenance and development of internal tools in Python and Perl that supported the packaging work-flow. I occasionally committed to open-source projects, e.g. `quilt`. Also, my bug-fixing patches for the set of packages I was responsible for were accepted upstream.

German Artificial Intelligence Research Center From December 2013 to May 2014 while staying as an exchange student at the Saarland University, I was granted an Assistant Researcher internship position in the Talking Robots project. My main responsibilities consist of improving existing applications in Java, both interface and underlying algorithms and management of internal git repositories. I consider a big achievement that I talked my coworkers into open-sourcing the project, which is now actively in-progress.

Relevant Works

Lightweight x86 ELF debugger for FreeBSD and Linux written in C
https://www.github.com/lovasko/Kvetoslavov

- single-instruction stepping
- code breakpoints and breakpoint stepping
- attaching to and detaching from running processes
- type and variable name dumping

Other Works

- Sudoku solver written in Prolog *https://www.gitorious.org/praha*
- Feature-rich ray-tracer in Java *https://www.gitorious.org/lucenec*
- Various small projects for my needs, e.g. todo-manager written in Python
  https://www.gitorious.org/olomouc

Relevant Completed University Courses

- Programming I, II
- Introduction to UNIX
- Selected topics on UNIX
- Programming in UNIX

Useful Completed University Courses

- Algorithms and Data Structures I, II
- Principles of Computer Architectures
- Operating Systems

Version Control Systems

- Git - both user and internals knowledge
- SVN - user knowledge

Availability

My work in the German AI Research Center ends at the end of May. That means that the first week from Summer of Code would overlap with this commitment. I would like to offer a compensation, by working a week longer after the deadline. I have no other plans for the summer and I would be able to work on this project from 1. June, around 8 hours per day.

Possible Mentor

Robert N. M. Watson (rwatson@freebsd.org)

Project Information

Project Title

CTF implementation and use in the kernel debugger

Project Description

When somebody compiles the kernel with the `WITH_CDDL` and `WITH_CTF` options, the `ctfconvert` utility is used to convert the `gdb` STABS debugging symbols to the CTF data. This is apparently done to enable the D-Trace utility to retrieve the type information data. Since we already have this information inside the kernel at our disposal, it would be a shame not to use it also for other purposes, in our case the kernel on-line debugger DDB.

The main idea behind this project is to *add the capability of pretty printing data structures* contained inside the FreeBSD kernel source based on the CTF data contained in the `SUNW_ctf` section of the kernel binaries. The following example will illustrate the feature. The names of variables and data structures are fictional for better intuition.

Example

```
>pretty_print 0xfa6a4890 thread
struct thread {
  unsigned int id = 12
  unsigned char on_cpu = 0
  int flags = 34
  enum state = THREAD_STOPPED (2)
  union color = {
    unsigned int rgba = 4288086271
    unsigned char[4] components =
      [0] = 255
      [1] = 170
      [2] = 0
      [3] = 255
  }
  struct ucred *p_ucred = {
    unsigned int ref = 2
    uid_t uid = 1900
    struct jail j = {
      char *name = "Azkaban"
    }
  }
}
```

This project would tackle a few problems, mainly:

Accessing the CTF data Due to licensing problems, which are too off-topic for this proposal to explain, the existing library code for parsing the CTF data is not applicable. Thus, as a part of the project, a fresh new CTF data parsing library is needed.

Pretty printer output format There is no universally good way to display the requested data and therefore this section may appear very disputable (see Community) and I am very open to all suggestions. As seen in the example, I propose to respect the output style of the existing `ctfdump` utility and the kernel source code style where the opening parenthesis is on the same line as the struct or union. Unions and structs are indented by one more level compared to their parent data structure. Every field is prefixed with its type (for non-basic simple data types see the section below). By default the command prints out the whole data structure using a depth-first approach. Nonetheless, limiting the depth should definitely be an option. Enumerations are displayed with the actual name followed by the corresponding integer values inside parenthesis.

The example showcases an existing coding technique: typedefing the basic data types into named types that are just the same inside. To make this clearer, the output format could contain a *typedef chain* e.g. `uid_t -> __uuid_t -> __uint32_t -> unsigned int`. These chains would end when the primitive well-known data type is discovered. As it may use a lot of space, this feature can be optional.

Integrating the changes with the DDB UI In my opinion the command should look compatible with the existing `examine` DDB command. The option names and format of the option arguments should be decided ad hoc, with the help of the community, to achieve maximal intuitivity and user compatibility. Some examples are discussed in sections above and below.

Apart from the maximum depth selection, user should be able to print only specific field of the structure. An example:

```
>pretty_print 0xfa6a4890 thread.state
THREAD_STOPPED (2)
```

The same principle can be applied also to arrays by being able to select n-th component.

Another useful options is, as already implemented inside the existing `examine` command, the ability to select the number base selecting one from binary, octal, decimal, hexadecimal or hexadecimal with spaces between bytes. This may render crucially useful in many cases.

Adapt the layout based to common design patterns A very common pattern that we can see in programming are recursive data structures, in C particularly the linked lists. One, usually the first or the last field of a struct is a pointer to the next element in the list. With the indentation output model we would end up with all elements of a linked list more and more indented, which can become really unuseful. Instead, the plausible solution would be to keep the indentation the same and provide a *link arrow* between them, as in this example:

```
struct linked_list {
  int data = 7
}
   |
   | next
   v
struct linked_list {
  int data = 69
}
   |
   | next
   v
struct linked_list {
  int data = 40
  linked_list *next = NULL;
}
```

A special attention should be paid to the recursive data structures, as they may contain loops. Therefore, the pretty print code should contain an algorithm to detect such data structure defects, such as *the tortoise and the hare*.

The second improvement discusses second common design pattern: the true/false flags represented as bits inside a integer variable. The general example above also contains an illustrative sample: `flags = 34`. It is obvious, that only two binary positions are 1, 32 and 2. The pretty print could automatically print variables that names include "flag" or "flg" (or any other suggested heuristic) in a binary format, e.g. `00010010`.

Benefits for The FreeBSD Project

Clearly, the group of people that would benefit the most from these changes are the FreeBSD kernel developers. Being able to dump the variable contents is one of the basic principles of debugging and improving one of the most used functions is always a change for the better. This change would allow more insightful look inside the kernel internals and save loads of time for the DDB users. Not to mention, these changes only add possibilities and do not interfere with existing features of the DDB.

Comparison to other approaches

It is always possible to create a custom pretty printer for every kernel data structure, but this is obviously a never-ending struggle to keep up with the kernel code and also a heavy burden that can be actually automated, like this proposal states.

Deliverables

Throughout the time of making, the following three units will be created:

BSD licensed `libctf`

The FreeBSD compatibly-licensed `libctf` implementation. Since this library may be used for various other projects, it would be best to keep it as compatible as possible for writing bindings and using it in C/C++ programs. For this purpose, I propose to use the ANSI C standard (C89).

Changes to the DDB codebase

All changes to the DDB codebase must respect the existing code style and be very well documented inside a wiki page and commit messages.

Documentation

I understand the value of a good documentation and I would like to keep it always up-to-date.

man pages The DDB man pages within the section 4 should be updated with the description and examples how to use the newly added `pretty_print` command.

Web page As mentioned earlier, the `libctf` library should have it's own website.

Test Plan

The DDB improvement alone require personal testing by actually using the code to pretty print various data structures and verify the printed results. The very parsing of the CTF data is a responsibility of the `libctf`.

Testing the `libctf`

Correctness is the main goal of this component. An automatic testing suite would be created: generate a random oriented tree graph of data structures, each edge representing a dependency. After compilation, run the `ctfconvert` utility on the created object files. Finally, parse the result of the output generated by simple own `ctfdump` program and the created graph.

Speed may also be a vital concern of the library, since it can be used in a rather big setup, the ctfconvert utility converting the kernel ELF tables. While not being used that often or in a speed-critical setup, it would be still nice to keep the times healthy. Also, comparison to existing CDDL libraries would be a nice benchmark.

Project Schedule

In my view, this project should fit the planned time span perfectly. If I finished the proposed plan unexpectedly earlier, I would continue to improve already created codebase. The `libctf` API can be enriched to support more of the CTF standard, improved documentation, various bug-fixes that may appear to be needed. In case of being behind schedule, I would still fulfill my commitment to The FreeBSD Project in my free time and finish the project.

Initial phase

During the first phase, an `libctf` API design and an actual implementation should take place. After all the documentation is written, this project should have its own website, either having an own domain, e.x. *www.libctf.org* or some subdomain inside the *freebsd.org* domain.

Between $\frac{1}{4}$ and the Midterm

The testing of the library should take place. After passing all the correctness tests and undergoing the speed comparisons, the tests and speed graphs may be posted to the library web page. Also, until the actual midterm, a code that does the basic pretty printing of all structures inside the `/boot/kernel/kernel` file.

Pre-final phase

Inclusion of the stand-alone code to the DDB codebase. All the argument and option parsing should be done during this period. Every non-basic feature such as the linked list adaption is ought to be finished. Also, implementation of any new ideas that popped up during the making should be done.

Final phase

While presenting the changes to the community along the way, some interesting ideas may come along. Implementing or discussing these can be done during this time. Last step is pushing the code to the official repositories. This phase may seem a bit emptier than the others, but it is intended to be so. Being realistic, the plan is not always the same as the reality, so this space is intended to be a time reserve.

Community

As with every other decision, the more insightful and various ideas and opinions, the more illuminated the final resolution is. I would like to read the reflections on my work and I am open to suggestions about various parts of the project, e.x. the pretty print format, options and DDB integration. A proper place for this communication would be the official FreeBSD forum.

Blog

I would like to put down my experience along the way to the blogosphere.

Future

I would like to continue to maintain all of my created code, especially if any issues or feature requests appeared. Future plans may include diving deeper into the DDB code and implementing other interesting ideas.