Type-aware kernel virtual memory access for utilities

Daniel Lovasko

Dear Reader,
this is my proposal to the FreeBSD Project for the Google Summer of Code 2015. Following pages are dedicated to the concept ideas, implementation details and the general motivation behind the project. Set of relevant personal information can be found at the end of the document.

Introduction

About `libctf`[1]

`libctf` is a BSD-licensed implementation of the Compact C Type Format I developed during the last year's Summer of Code that is used to store type information with respect to the C language type system. Its main strengths are low memory consumption, future ubiquity in the base system and available robustness on demand.

About `libkvm`[2]

`libkvm` provides the access to kernel virtual memory of live kernels and crash dumps. It supports reading and writing kernel memory, along with reading symbol addresses and gather process information.

Main Idea

Many userland utilities, such as `ps(1)` or `netstat(1)` use `libkvm` to obtain specific data in order to fulfil user's inquiry regarding the kernel state. In an unfortunate case of a kernel and the userland utility not being compiled with the same structures in terms of type sizes, endians or memory alignment, these tools do not work properly. By proposing a specific joint interface of `libctf` and `libkvm` that would relay on the machine-independent information provided by the CTF format and already existing access to the memory image, we aim to overcome the current architecture-related limitation. Furthermore, to demonstrate the functionality and usability of such library API, a predefined set of basic userland utilities will be enhanced as a proof of concept.

Potential Mentor

John-Mark Gurney (jmg@freebsd.org)

[1] https://github.com/lovasko/libctf
[2] https://www.freebsd.org/cgi/man.cgi?query=kvm&sektion=3

The problem

Important and ubiquitous types `size_t` and `long` vary in their bit widths on many platforms, while the same applies to the pointer types. Another significant change in the memory layout of a structure is the memory alignment.

```
root@64bsd:~ # uname -p
amd64
root@64bsd:~ # clang -dM -E -x c /dev/null | grep __SIZEOF_SIZE_T__
#define __SIZEOF_SIZE_T__ 8
```

```
root@32bsd:~ # uname -p
i386
root@32bsd:~ # clang -dM -E -x c /dev/null | grep __SIZEOF_SIZE_T__
#define __SIZEOF_SIZE_T__ 4
```

This means that every `struct` containing `size_t` will inflate (in this example) by 4 bytes, which in turn shifts every consecutive structure member. Any program trying to map a 64-bit structure as a mirror image of a memory on top of a 32-bit image will be affected negatively, leading to possible data corruption.

Example

By compiling the following simple example on both architectures we demonstrate the impact on a size of a `struct`.

```
#include <stdlib.h>
#include <stdio.h>

struct city {
  float rec_jan_high;
  long population;
  char name[8];
};

int
main(void)
{
  struct city khartoum = {39.2f, 31921, "Khartoum"};

  printf("%u\n", sizeof (struct city)); /* %ul on 64bit */
  return EXIT_SUCCESS;
}
```

We get different results:

```
root@32bsd:~ # ./a.out
16
```

```
root@64bsd:~ # ./a.out
24
```

CTF

Running `ctfdump`[3] on both compiled binaries, we get following type information.

```
root@32bsd:~ # ctfdump ./a.out
        ...
       ID: 2
     Kind: int
     Root: no
     Name: long
     Size: 32
   Offset: 0
   Signed: yes
  Content: number
        ...
       ID: 5
     Kind: struct
     Root: yes
     Name: city
       float record_jan_high | 0
       long population | 32
       char [8] name | 64
        ...
```

```
root@64bsd:~ # ctfdump ./a.out
        ...
       ID: 2
     Kind: int
     Root: no
     Name: long
     Size: 64
   Offset: 0
   Signed: yes
  Content: number
        ...
```

[3] https://github.com/lovasko/ctfdump

```
  ID: 5
Kind: struct
Root: yes
Name: city
  float record_jan_high | 0
  long population | 64
  char [8] name | 128
   ...
```

Note the difference of the `population` offset caused by the memory alignment and the `name` offset caused by the `long` being 8 bytes wide.

Practical Impact

By extrapolating the issue onto structures used to describe I/O operations, mutexes or processes in the kernel, we can concur the presence of the matter in all utilities that work with crash dumps coming from different architectures.

The solution

Introduction

Based on the `struct city` example that contained the `ctfdump` output we can conclude that CTF is a valid tool that provides sufficient information independently on the underlying architecture, vanquishing issues with bit width and memory alignment. Knowing the difference in offsets, utilities would be able to handle the data properly. To summarize the task: we want to be able to copy data from the kernel image/core file to a userland `struct` without being affected by architecture-dependent essentials. Following sections discuss the API and possible styles of usage of this format in utilities.

Mapping Procedure

By creating a mock structure in userland that vaguely maps to the kernel structure we establish a target for the copy procedure. This structure needs to contain only those members of the source `struct` which are needed for the utility to work with. Another way of picturing this is that the mock structure is a query description handed to the algorithm as a fill request. An example of such structure for a source structure `struct sema`[4] might be:

```
struct my_sema {
  intmax_t sema_waiters;
  intmax_t sema_value;
};
```

As compared to the original structure:

[4] `/usr/src/sys/sys/sema.h`

```
struct sema {
  struct mtx sema_mtx;
  struct cv sema_cv;
  int sema_waiters;
  int sema_value;
};
```

The contract, or ABI, is a triplet `<struct name, member name, type>`. The copy procedure is using run-time *reflection-like* method - obtaining CTF data of the mock structure from the caller's binary file. In order to have the full information needed for the mapping of members between the source and the target structures, the library needs CTF data of the target `struct` too. Afterwards, it tries to map the member names of both structures, while making sure that the type does not change, otherwise the member is not mappable. The only allowed change is in the byte width of an integral type.

Maximal Types

To future-proof the mock structures, it is strongly advised to use maximal `stdint.h` integer types, `intmax_t` and `uintmax_t`. This way adding support for a next generation of architectures (128-bit or 256-bit) is only a matter of updating the `stdint.h` file (which will be done system-wide anyway) and recompiling the tool in order to use it, since `intmax_t` will already point to the maximal integral type by definition.

Mapping Metadata

As the mapping might not yield success for every member we need to handle errors properly. Clamping values or providing defaults is unwanted, as there are no guaranteed defaults for types such as `float`, `int` or even pointers, because `NaN`, `0` and `NULL` are perfectly valid values too. This implies a need for a parallel data structure that would carry the necessary metadata about the mapping result.

The mapping can result only in two possible states: success or failure. Therefore, only one bit per member is needed and we can pack this information densely (internal representation) and provide a clean interface on top of it.

Proposed API

It is important to note that this particular API is subject to future discussion, perfect for the Community Bonding Phase of the Summer of Code. Since this is only a proposal, a real need for a specific API function or a need for a radical change to a proposed one might emerge later during the development process.

Since all the API calls form a new set of functionality, it is appropriate to store all of them in a library. We chose the name Type-aware KVM Library, `libtak` for short.

```
tak_t*
tak_open(kvm_t* kd)
```

Open a Type-aware KVM connection.

```
void
tak_close(tak_t* td)
```
Close a Type-aware KVM connection.

```
int
tak_read_sym(tak_t *td,
             char* symbol,
             char* source_type,
             char* target_type,
             void** target_value,
             tak_md_t* md)
```
Read a specified *symbol* that has type *source_type* and map it to the *target_type* structure, while saving the actual data to *target_value* and mapping metadata to *md*.

```
int
tak_read_addr(tak_t *td,
              unsigned long addr,
              char* source_type,
              char* target_type,
              void** target_value,
              tak_md_t* md)
```
Read a specified *addr* that has type *source_type* and map it to the *target_type* structure, while saving the actual data to *target_value* and mapping metadata to *md*.

```
int
tak_md_valid(tak_md_t* td, char* member_name)
```
Check if the *member_name* was mapped correctly.

```
int
tak_md_valid_all(tak_md_t* td)
```
Check if all data was mapped correctly.

More Complex Example

To explain the idea on a specific data structure, we will aim to obtain the number of threads of a specific process identified by a PID as an input.

First, we need to know which symbol is pointing at the process list managed by the kernel - the `allproc` symbol that is defined as a global variable of type `struct proclist`.
By examining the source file containing the type declaration[5] we can create a mimicry `struct`:

```
struct my_proclist {
```

[5] `/usr/src/sys/sys/proc.h`

```
  struct my_proc* lh_first;
};
```

It is important to note that the structure is defined by the `sys/queue.h` macro `LIST_HEAD` which is quite common situation and the `libtak` might improve on this common pattern.

Following the declaration of `struct my_proclist`, we create a mimicry structure for the `struct proc`. Since we care only about the field `p_pid` in order to identify the process, the expanded form of the `LIST_ENTRY` macro in order to be able to loop over all processes, and the actual data stored in `p_numthreads`, the structure will appear as follows:

```
struct my_proc {
  struct {
    struct my_proc* le_next;
    struct my_proc** le_prev;
  }

  intmax_t p_pid;
  intmax_t p_numthreads;
}
```

Having all necessary structures declared, we can employ the `tak_read_sym` function to retrieve the structure `struct my_proclist` and proceed with many calls to `tak_read_addr` followed by metadata checking via `tak_md_valid` in a matching `struct my_proc`.

Benefits

for The FreeBSD Project and Users

Results of this project would improve the system introspection and monitoring across various platforms with a unified set of tools depending on a minimal set of information. Sysadmins and power-users solving issues can spend less time on their toolset and brush up their troubleshooting workflow.

for Others

The license and implementation of the library and its API does not prevent other open and closed source operating systems (such as NetBSD, illumos or Darwin) to borrow the functionality freely.

Deliverables

Cross-architecture `libctf`

One of the key software components is the CTF implementation, `libctf`. It currently supports only x86 and x86_64 systems. In order to facilitate cross-architecture examination support, we propose to finalize the library so that it supports all major architectures (ARM, MIPS, SPARC64) supported by FreeBSD. Such changes will take endian and word size related issues in mind.

Cross-architecture `libtak`

As a part of the project, an implementation of the structure mapping algorithm will be created along with defining every API function mentioned earlier.

Proof-of-concept Implementation of Utilities

In order to test the idea, a separate, much simpler version of certain utilities will be created. These utilities are chosen based on the area of their usage, uniformly distributed between network, I/O or process information. Subsequently, the new code can be either ported back to the original utilities or replace them all together. Proposed utilities are:

- `ps(1)`
- `ipcs(1)`
- `w(1)`
- `vmstat(1)`

Project Schedule

Initial Phase

- ARM, MIPS and SPARC64 support for `libctf` along with unit testing
- `libtak` API design

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

- `libtak` API implementation
- `ipcs(1)` proof-of-concept
- `ps(1)` proof-of-concept

Pre-final Phase

- `w(1)` proof-of-concept
- `vmstat(1)` proof-of-concept

Final Phase

Depending on the progress achieved in the previous phases there are more options available: either finish the selected proof-of-concept utilities (by merging the new code or borrowing the command line option parsing from the existing ones) or create a proof-of-concept version of another utility, e.g. `iostat(1)` or `top(1)`.

Personal Information

Contact

Name	Daniel Lovasko
E-Mail	lovasko@freebsd.org
Phone	+421 904 329 363
IRC	`lovasko` on EFnet and Freenode

Motivation

I find the project topic to be very useful with regards to system administration and troubleshooting across multiple platforms. Working and engaging with the FreeBSD community is a joyous experience and would like to continue the collaboration, mainly concerning the CTF format and its usage. CTF and its utilisation in the FreeBSD system is also the topic of my final university thesis and therefore this project has an academic importance for me personally too.

Availability

- 2-4 hours/day during 25 May - 31 June (exam period)
- 6-8 hours/day during 1 July - 24 August

Open Source Involvement & Work Experience

Google Summer of Code 2014 — 5/2014 - 8/2014

The goal of my summer project was to add the capability of pretty-printing data structures to the kernel debugger DDB. First step was to create a BSD-licensed library that implements the Compact C Type Format used in DTrace and mdb to represent data types. This involved reverse-engineering the format, writing unit tests, thorough documentation of both the format itself and the library code, and following strict code-style guidelines set by the project's community. Final piece was the utilisation of the library in the kernel debugger to achieve the set goal.

German Artificial Intelligence Research Center — 12/2013 - 5/2014

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.

SUSE Linux 1/2012 - 7/2013

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.

Talks

AsiaBSDCon 3/2015

The main track presentation, and a corresponding conference paper, discuss importance of the Compact C Type Format for debugging purposes and describe innovative ways for its further employment, as well as introduces the library implementation to potentially interested developers.

FOSDEM 1/2015

After successfully finishing the Google Summer of Code, I delivered a lightning talk to present the improvements to the kernel debugger made along with the free-licensed CTF implementation to a wider audience outside of the FreeBSD community. This included developers from various Linux and illumos distributions.

SUSE Labs Conference 2/2013

Presentation of the Packaging Toolkit, an internal set of helper utilities that I co-created while working at SUSE, aiming to improve the packaging workflow. The talk had a positive impact and generated new contributions to the project.

Education

University

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

Erasmus Exchange University

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

Relevant Completed University Courses

- Programming I, II

- Introduction to UNIX
- Selected topics on UNIX
- Programming in UNIX
- Algorithms and Data Structures I, II
- Principles of Computer Architectures
- Operating Systems