Kernel Dump Analysis made Quick and Easy!

Alex Sidorenko <asid@hpe.com> *(HPE)*
Milan P. Gandhi <mgandhi@redhat.com> *(Red Hat)*

What is kernel dump analysis?

- kernel dump/vmcore can be obtained during crash/panic or hang
- It allows developers to see what was happening on system in depth as it is essentially a dump of memory contents (kernel pages plus possibly some userspace pages)
- In many situations it is impossible to determine root cause of the problem without a dump captured at the time of crash or hang
- Standard open-source tool for kernel dump/vmcore analysis is ‘crash’

Tools for kernel dump analysis

- ‘crash’ provides a number of commands and options to analyze various information from a vmcore
- But effective use of ‘crash’ requires expertise and knowledge of kernel internals
  - It’s desirable to extend the functionality available in *‘crash’* environment to process many data structures programmatically and present a summary
  - In some situations doing a manual review is very time consuming and impractical

    e.g. analyzing hundreds of hung tasks, lvm volumes, SCSI devices and multipath device maps etc.

Extensions for *'crash'*

- *'crash'* can be dynamically extended by writing programs in C and linking them in a special way. After that, the extensions can be loaded/unloaded by builtin 'extend' command.

```
crash>  extend  <path-to-loadable-crash-extension-file>

e.g.
crash>  extend  /cores/mpykdump.so
```

- Crash project page lists extensions that can be compiled and then used with crash:
  - https://people.redhat.com/anderson/extensions.html
- In addition to that, there are extensions languages, that let you write programs without a need to compile/link them

Extensions for *'crash'*

- Extension languages for *'crash'*:

  The extensions for crash are typically written in one of the following two languages

  - **EPPIC** (Embeddable Pre-Processor and Interpreter for C)
    - A replacement of an older project, SIAL (Simple Image Access Language)
    - Useful for writing small tools, but problematic for big projects. It is included as part of *'crash'* distributions
  - ***PyKdump*** - Python bindings to GDB/crash internals

  EPPIC is very useful for kernel developers as they already know C. But it is not very good for big projects.

  In contrast, *PyKdump* needs learning Python language but provides a number of features that make it a premier choice even for very large projects.

Features of *'PyKdump'*

- More scalable:
  - Programs written in *pykdump* could be split in multiple files and/or libraries, and time required to load/unload these scripts is reasonable even for the huge projects
  - No need to recompile the whole extension just for a small change in program created under this framework
- Based upon Python 3, so all the features available in Python can be utilized in programs
- Error processing is very easy due to the exception handling mechanism derived from Python
- An ability to execute *'crash'* commands and parse their output within programs for further processing

Features of *'PyKdump'*

- Programs can make runtime decisions based upon symbolic info from vmlinux
  - Same extension and programs can process vmcores collected from different Linux kernel versions

    Currently the programs available in PyKdump framework can process dumps collected from RHEL 5.6+/RHEL 6/7, Fedora 17+, OEL 6/7, SLES 10+, Ubuntu, and most latest upstream kernel versions
  - Programs written under other frameworks usually need much more effort to cope with data structure changes across different kernel versions
- Allows developers to automate their analysis steps, so most of the analysis for a new vmcore can be done by program itself
- Developers can add custom checks in programs and warn users about potential problems that are easy to miss while inspecting vmcore manually

'PyKdump' Design

- *'PyKdump'* maps C-structures used with Linux kernel to Python objects

  For example:

  - It maps C *'struct'* and *'union'* by creating corresponding python objects with attributes matching the respective field names of C *struct/union*

    These Python objects are used within programs created in PyKdump framework to analyze the struct/union internals, its fields etc.

  - Other C data types are mapped to similar Python types e.g. C *'int'* is mapped to Python *'integer'*

  - Most operators in C are mapped to similar Python operators

'PyKdump' Design

- In "C" we have two dereference operators: '.' and '->'
- As Python does not have a concept of pointers, there is just '.' operator. But we can understand what is needed based on data type and context.

  Assuming that '*ptr*' is a pointer to struct with its fields being other structs/pointers, a dereference chain might look like:

For 'C'

```
ptr→a.b.c→d
```

For 'Python' *– it will deduce what is needed based on fields definitions*

```
ptr.a.b.c.d
```

'PyKdump' built-in functions

- To ease the analysis of different structures within kernel, *'PyKdump'* provides built-in functions according to mapping rules
- These functions allows programmers to read kernel objects, iterate through linked lists/trees, execute crash commands, etc.
- Two most important pykdump API functions are:
  - **readSU**
    - Read struct/union based on its type and address provided as arguments
  - **readSymbol**
    - Get a Python object representing a kernel object defined as a global variable in C. Depending on C-definition, this subroutine can return a struct/union, array, pointer, string etc.

'PyKdump' built-in functions

- Some other useful PyKdump API subroutines:
  - **exec_crash_command_bg**
    - Allows the programs to execute crash commands and parse its output. They are executed in background and a timeout value can be specified (for big vmcores and pathologic cases, some crash commands can take up to 30 minutes of CPU)
  - **EnumInfo**
    - Checks the enum declaration and returns the value from specified position
  - **member_size**
    - Used to inspect size of structure/union member
  - **member_offset**
    - Used to find an offset of structure/union member

'PyKdump' built-in functions

- How to use PyKdump API subroutines?
- Let us look at an example for 'readSU' function mentioned in previous slides

Syntax:

readSU("<<struct_name>>", <<address>>)

e.g

readSU("struct hd_struct", int(bd_disk.part0))

- **1st argument:** The 1st argument to this function is struct name that we want to retrieve. For example, in above line, we are trying to retrieve '*struct hd_struct*' from the address pointed by '*bd_disk.part0*'
- **2nd argument:** This is an address of struct or variable from which we want to get the structure mentioned in 1st argument.
- **Returns:** Structure mentioned in 1st argument.

'PyKdump' built-in functions

- *'readSU'*

  Example:

  Below are the typical 'crash' commands used to get address of 'struct hd_struct' from the 'gendisk' structure:

```
    crash> dev -d|head
    MAJOR  GENDISK            NAME  REQUEST_QUEUE      ...
        8  ffff8a3304a1ec00    sda  ffff8a32f5842f58   ...
        8  ffff8a3308462000    sdb  ffff8a32f5bf0000   ...
        ...

'hd_struct' is embedded inside 'gendisk' structure:

    crash> struct gendisk -o|grep -i hd_struct
           [0x48] struct hd_struct part0;

 And now we can get address of' hd_struct' from 'gendisk':

    crash> p &((struct gendisk *) 0xffff8a3304a1ec00)->part0
    $2 = (struct hd_struct *) 0xffff8a3304a1ec48
```

'PyKdump' built-in functions

Example (cont'd)

- Using 'readSU' function for retrieving 'hd_struct' from 'gendisk' structure:

*A pointer to '**gendisk**' structure is stored in '**block_device**':*

```
crash> struct block_device -o|grep -i bd_disk
  [0x98] struct gendisk *bd_disk;
```

*And '**hd_struct**' is embedded inside '**gendisk**' :*

```
crash> struct gendisk -o|grep -i hd_struct
   [0x48] struct hd_struct part0;
```

*From the **bd_disk** pointer **hd_struct** could be retrieved as:*

```
hd_struct = readSU("struct hd_struct", long(bd_disk.part0))
```

'PyKdump' built-in functions

Example (cont'd)

- A Python object is initialized to hold the contents of struct/union returned by built-in functions. This Python object could then be used in programs to print its member variables, inspect its contents etc:

```
    hd_struct = readSU("struct hd_struct", long(bd_disk.part0))

Above line will initialize Python object named 'hd_struct' that is mapped
to C structure 'hd_struct'
```

- Information about *'hd_struct'* can be printed as:

```
    print(hd_struct)

To print structure address in hexadecimal:

    print("{:x}".format(hd_struct))
```

'PyKdump' built-in functions

- exec_crash_command_bg:

  Syntax:

  exec_crash_command_bg("<<crash_command>>")

  e.g

  exec_crash_command_bg("sys")

  - **1st argument:** One of the built-in crash commands
  - **Returns:** Output of the crash command as a string

```
# Parsing the output of 'sys' command to verify
# kernel version string:

for l in exec_crash_command_bg('sys').splitlines()[1:]:
    try:
        if (('RELEASE: ' in l) and ('2.6.32-' in l)):
            version = 'rhel6'
        elif (('RELEASE: ' in l) and ('3.10.0-' in l)):
            version = 'rhel7'
    except:
        pylog.warning("cannot parse:", l)
```

Checking structure definitions

- During Linux kernel development, definition of some structures can change
- Crash extension programs need to be written in a way that can deal with these changes, to be able to work with different kernels
- PyKdump solves this problem by providing subroutines for inspecting structure/union properties, e.g. check whether a specific member exists or not
- This could be done in several ways, e.g. using ***member_size*** function discussed in next slides
- In addition, there are more advanced mechanisms such as ‘pseudoattributes’

Checking structure definitions

- For example, older version of Linux kernels had different name for member variable ***'elevator_type'*** present in ***'struct elevator_queue'***

From latest upstream:

```
    $ less include/linux/elevator.h
    ...
    struct elevator_queue
    {
            struct elevator_type *type;
    ...
```

*Below patch had changed the definition for '**elevator_queue**' as below:*

```
    $ git show 22f746e235a5c
    ...
    @@ -90,10 +90,9 @@ struct elevator_type
      */
     struct elevator_queue
     {
    -       struct elevator_ops *ops;
    +       struct elevator_type *type;
            void *elevator_data;
            struct kobject kobj;
    -       struct elevator_type *elevator_type;
```

Checking structure definitions

- **member_size** function could be used to verify if ***'struct elevator_queue'*** has a member named ***'elevator_type'*** or ***'type'***
- If the specific member does not exist in structure, then **member_size** returns *'-1'*

```
sdev_q = readSU("struct request_queue", long(sdev.request_queue))

if (member_size("struct elevator_queue", "elevator_type") != -1):
    elevator_name = sdev_q.elevator.elevator_type.elevator_name

elif(member_size("struct elevator_queue", "type") != -1):
    elevator_name = sdev_q.elevator.type.elevator_name

else:
    elevator_name = "<Unknown>"
```

- This allows the same program written in PyKdump framework to be usable with kernel versions shipped across multiple Linux distributions, and upstream kernels

Exception handling

- PyKdump uses exception handling mechanism derived from Python
- A well-written program should be able to handle unexpected situations, reporting potential issues instead of just terminating
- For example: during some of the storage failure events, SCSI paths to the multipath device are lost, and these paths may be in process of deletion from system. Some of the structures used by these failed paths may be invalid, or already freed due to kernel bug, race conditions etc.
- An access to such unreliable structure could case failure/errors in program. Such events could be handled easily by *'try', 'except'* blocks

e.g

```
try:
    temp_scsi_device = readSU("struct scsi_device",
                              long(temp_pgpath.path.dev.bdev.bd_disk.queue.queuedata))
except:
    pylog.warning("Error in processing sub paths for multipath device:", name)
    return
```

Ready to use programs

- The *‘PyKdump’* project currently provides number of very useful mature programs for vmcore analysis that can be used immediately:
  - **xportshow** - Displays information about connections and sockets
  - **crashinfo** - Shows general information about kernel dump
  - **scsishow** - Shows SCSI subsystem information from the dump
  - **dmshow** - Shows device-mapper, multipath, lvm information
  - **taskinfo** - Prints processes status information as captured at the time of crash
  - **nfsshow** - Prints NFS information from kernel dump
  - **hanginfo** - Summarizes information about hung tasks detected in vmcore

Obtaining PyKdump

- A latest stable and ready to use binary for PyKdump is available for download from it's project page
  - https://sourceforge.net/projects/pykdump/
- If you want to build PyKdump from sources, you need the following:
  - Source code for Python-3 or above
  - Source code for 'crash' utility
  - ZIP utility
- Detailed steps to manually build the PyKdump binary

  https://sourceforge.net/p/pykdump/wiki/Building%20From%20GIT/

  Once the compilation process is completed successfully, the ready to use *'mpykdump.so'* binary would be available in *'Extension'* directory

Thank you!