Programmatic Kernel Dump Analysis On Linux

Alex Sidorenko
Hewlett-Packard
asid@hp.com

Abstract

Companies providing Linux support rely heavily on kernel dumps created on customers' hosts. Kernel dump analysis is an art and it is impossible to make it fully automatic. The standard tool used for dump-analysis, 'crash', provides a number of useful commands. But when we need to enhance it or to analyze several thousand similar structures, we need programmatic API.

In this paper we describe Python bindings to *crash*[1] and compare it to C-like SIAL extension language. After a general framework discussion we look at some practical tools developed on top of PyKdump, such as `xportshow`. This tool works on kernels 2.4.21-2.6.28 and provides many useful features, such as printing routing tables, emulating `netstat` and summarizing networking system status.

1 Why Do We Need Programmatic API?

- adding new features and enhancing functionality quickly
- there are have too many structures to look through all of them ourselves
- running a number of useful tests — each of them can be executed manually, but there are many of them
- running programs on a customer's site if for some reason he cannot send us vmcores
- we can use an already developed tool on live kernels instead of writing new DLKM or Systemtap script

An ability to run scripted tests quickly is extremely important for support organizations. Even though in theory customers should provide a detailed description of the problem, in reality it is not unusual to get vmcore with just the generic description of "the host is unresponsive."

[1]`http://sourceforge.net/projects/pykdump`

It can mean many different things, for example:

- a critical userspace application (e.g. Oracle) stopped responding
- network connectivity is lost
- the system is just overloaded
- the system is out of memory
- there is a bug in the kernel leading to CPUs executing kernel code forever, with interrupts disabled

In such cases it makes sense to run a number of standard tests to narrow down the problem. For example:

- how much memory is used and whether it is fragmented
- check load averages and runqueues (e.g. are there any RT processes)
- when was the last time NICs transmitted and received data
- is syslogd hanging (this will make all processes doing `syslog()` unresponsive)

It makes sense to run all such tests programmatically to save time and effort. Furthermore, even those lacking the proper skills to do dump analysis themselves can run automated tasks.

1.1 Extensions Available for Crash

Crash [1] is a standard tool used for dump analysis. There is similar another tool, ***lcrash*** but we will not discuss it here.

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 using builtin `extend` command. But developing in C is rather time-consuming and unpractical, especially if we need to write a custom code quickly. It is much better to use special extensions providing bindings of ***crash*** to higher-level languages. Using such an "extension language," we can develop new programs quickly without a need to compile/link every time we need to modify our script. Here are some known extension languages:

- SIAL–C-like language. Very handy for writing small tools, but problematic for big projects. Is included as part of ***crash*** distribution
- Alicia–Perl wrapper driving ***crash*** via stdin, retrieving results from stdout. Quite slow, as a result. There was no activity for this project on SF site during last 3 years
- *PyKdump*–Python bindings to ***GDB/crash*** internals

From these three frameworks, SIAL is probably the easiest to use for kernel hackers as they already know C. However, *PyKdump* provides a number of features that makes it better than SIAL for big projects:

- better scalability–a program can be split into many files and/or libraries and loading/execution time is reasonable even for huge programs
- Python standard library is extremely powerful
- Python is a high-level language with efficient lists, dictionaries and other useful classes
- error processing is easier because of exception mechanism
- more features for making runtime decisions based on symbolic info from vmlinux
- an ability to run ***crash*** commands and parse their output efficiently

1.2 Writing Programs That Work With Different Kernel Revisions

Linux kernel is a moving target. The definitions of kernel structures, global variables and algorithms are different from version to version. If we want to write a program that works for kernel dumps obtained from different kernels, this needs to be taken into account. Some possible approaches are:

- check for kernel version explicitly, use a different code for different versions
- check whether certain global variables exist
- check whether a structure has a specific fieldname

This means that we need to make runtime decisions. C is a strongly typed language:the variable type needs to be explicitly declared and cannot be changed afterwards. Let us consider the following case. There is a global variable. In an older kernel it was declared as

```
struct one var;
```

In a newer kernel the name of the struct has been changed even though its definition is the same:

```
struct two var;
```

We want to access `var.field`

In C-like languages (e.g. SIAL) a possible approach is the following:

```
{"LINUX_2_2_16",
   "(LINUX_RELEASE==0x020210)"},
{"LINUX_2_2_17",
  "(LINUX_RELEASE==0x020211)"},
{"LINUX_2_4_0",
   "(LINUX_RELEASE==0x020400)"},
...
```

Then in some include file crossSupport.h:

```
#if LINUX_2_6_X
    #define TYPEX struct one
#else
    #define TYPEX struct two
#endif
```

Then in the code:

```
#include <crossSupport.h>
void func(...)
{
TYPEX *s=(TYPEX *)var;
    if(var->field ...) {

    }
}
```

This is not very elegant and is rather unreliable. Most commercial distributions base their major release on a specific kernel and then backport bugfixes/features from recent kernels as needed. As a result, variables and structures definition on 2.6.9-based RHEL4 might change even though the kernel is still reported as 2.6.9. A better approach would be to retrieve variable types from vmlinux and use them as they are. In *PyKdump* we can do the following:

```
var = readSymbol("var")
f = var.field
```

Another approach is to base runtime decisions on explicit type information. That is, to check whether a struct has a specific member or what its type is. At this moment SIAL lacks this functionality but it might be added in the future.

2 PyKdump Design

Python is a very powerful and extremely popular programming language, at least among userspace application developers. Unfortunately, many kernel hackers only know well C and assembler. There are excellent books available and outstanding documentation provided on Python website [2]. But the syntax of Python operators is close enough to C, so there should be no problem in understanding all examples provided in this paper even for those who know nothing about Python.

2.1 Mapping C-structures To Python Objects

The Linux kernel is written in C (plus a bit of assembly). To be able to write useful dump-analysis scripts easily, we need as a minimum:

- to be able to read memory, global variables and struct/union contents
- to be able to write Python code easily looking at related C-sources

For example, if we want to write a program printing routing tables from a dump, we start by looking at its kernel implementation of related /proc routines. It would be convenient to be able to copy and paste pieces of related C-sources to our script, but even with SIAL (using C-like syntax) this does not always work.

While developing Python bindings to ***crash*** internals we used the following approach:

- we map C `struct` and `union` by creating Python objects with attributes matching the respective C field names
- we map other C types to Python types that are close, e.g. C `int` to Python `integer`
- we map C operators to similar Python operators

Python passes everything by reference, there are no pointers. As a result, there are no `*`, `->`, and `&` operators. It is easy to mimic reading and accessing fields of C struct/union in Python as both C and Python have the dot `.` operator:

```
struct blk_major_name {
   struct blk_major_name *next;
   int major;
   char name[16];
} svar;
```

```
s = readSU('struct blk_major_name', addr)
major = s.major
print "%3d       %-11s" % (major, s.name)
```

Here we read `struct blk_major_name` from a given address and print the `major` field. Python has many built-in data types, including integers, floating-point numbers and strings. We return properly typed values automatically, without specifying the type explicitly every time. There is no special pointer type in Python but we can represent pointers by integers. In the example above we expect to get

- s.next as an integer
- s.major as an integer
- s.name as a string

There are some problems with this approach. If we meet `char name[10]` declaration, how do we know whether it is intended to be used as a string or an array of 1-byte integers? We cannot know this from the symbolic information available in vmcore. To work around this, we introduce a special 'SmartString' type which mimics null-terminated strings but lets you access info as if it was a normal array. So if `name` is a SmartString, printing it will result in truncation on NULL byte but we still be able to access any byte using array access:

```
name="abc\0\5\6\7\8\9\10"
print s.name  # will print abc
print s.name[5] # will print 5
```

In most cases you can work with these SmartStrings just like with normal Python strings, but sometimes Python library functions check type explicitly (e.g. you cannot pass *SmartString* to regular expressions functions). You can convert `SmartString s.name` to a normal string using `str()` function, e.g.

```
str(s.name)
```

By default, struct/union members that are defined as char pointers or char arrays, are returned as *SmartString* type. If they have explicit *signed* or *unsigned* specifiers, they are returned as integer arrays.

2.2 Dereferencing Pointers in Structs and Unions (Emulating * and -> Operators)

What if we want to follow the 'next' pointer in the example above? The attribute dereference operator `->` in C is really just a syntax sugar that combines pointer dereference with attribute access:

```
/* The same as (*svar).next  */
svar->next;
/* The same as (*(*svar).next).next */
svar->next->next;
```

There is neither `*` nor `->` operators in Python but we still can dereference using alternative approaches. For example, for a pointer dereference we can use `Deref()` function. In C:

```
struct blk_major_name *sptr;
int major = (*sptr).major;
in major1 = sptr->major;
```

In Python (assuming that sptr is an object representing a pointer to structure):

```
major = Deref(sptr).major  # Approach 1
major1 = sptr.major        # Approach 2
```

Please note how we used the dot operator without dereferencing first. In C, it would have failed at the compilation stage. In *PyKdump*, the framework finds that an object is a pointer to a structure, so obviously the dot operator is not a simple field dereference. Consequently it interprets it as `->`. That is, you can use the dot operator in both cases and it will be used in whatever way is needed automatically. For example:

```
/* in C */
s->f1.f2->f3.f4-f5

# In Python
s.f1.f2.f3.f4.f5
```

In C, using dot operator on a pointer would trigger a compilation error. In Python, we make life easier by trying to interpret the dot operator either as `.` or `->`, depending on the object type.

More than that, in *PyKdump* pointers to structures and structures themselves have the same object type. It is similar to Java's approach where we have just references and no pointers.

Please note that the description above is correct only for pointers to structures. Pointers to any other type are represented with a different object class. In particular:

```
/* in C */
struct test *sptr;
struct test **pptr;

# in Python
# sptr is the same as Deref(sptr)
# (the same type), so you can write
sptr.f1

# pptr is completely different,
# Deref(pptr) is not the same as pptr
Deref(pptr).f1
```

To emulate the missing features we can define special attributes (usually called "properties" in OOP). Accessing such an attribute triggers a function call. A potential problem exists in the shape of name collision between internal object attributes and C-attributes as mapped to Python. Luckily, this is a highly improbable event for kernel structures. The "Linux Coding Style" document [3] says: "mixed-case names are frowned upon" so using mixed-case attributes for our own purposes should be safe enough. The "internal" methods of Python classes are all named like `__aname__` and, to reiterate, we have never seen name collision between field names of Linux kernel structures and Python internals.

2.3 Emulating & Operator

We can get the address of a global variable using `sym2addr()` function, e.g.

```
addr = sym2addr("init_task")
```

In other cases we start from a struct/union and need to find the address of its member. For example, we have a field which is defined as a struct (not a pointer), e.g.

```
type = struct task_struct {

   volatile long int state;

   ...

   struct list_head tasks;

}
```

When we access `tasks` attribute, we obtain an object representing a structure. For such objects we can use Addr(obj) function to obtain the associated address, e.g.

```
init_task = readSymbol('init_task')
init_task_saddr = Addr(init_task.tasks)
```

This works for objects representing aggregates, strings or pointers as they store the needed address internally. However, for integer/floating type the objects are just native Python integers/floats, there is no address. At this moment the only way to get the address of such a field is to compute it manually using low-level functions, e.g.

```
dev_base = readSymbol("dev_base")
off = member_offset("struct net_device",
                     "next")
addr_next = Addr(dev_base) + off
```

In the future we might wrap integers and floats so that `Addr()` will work for them as well - but this is not implemented yet. (The main reason for this is that we still need to evaluate the impact of these extra wrappers on performance).

2.4 Some Special Types

We map all C integer types to Python native `long integer` type and we map C struct/union instances to `class StructResult` instances. Pointers and arrays are normally represented by Python integers and lists. But in some cases we would like to preserve additional information while returning values. As a result, we wrap integers and strings in Python classes. Please note that usually you do not construct objects of these types yourself, as they will be initialized and returned as needed when using readSymbol and similar functions. Two most important cases are tPtr (to represent pointers to different C-types) and SmartString (used for C `char *` pointers and char arrays). When you obtain these objects from readSymbol, they internally store additional useful information.

2.4.1 StructResult

This type represents struct/union and a pointer to struct/union. In kernel sources we usually don't need to "read" structures as they are already in memory. So if we want to access a structure at a given address, we just use the cast operator, e.g.

```
struct sock *s = (struct sock *)addr;
```

In *PyKdump* we read them (ultimately reading bytes from vmcore file...):

```
# read from address addr
s = readSU("struct sock", addr)
# similar to C s.socket
socket = s.socket
# similar to C &s
addr = Addr(s)
```

2.4.2 tPtr - A Typed Pointer

When a variable is a pointer, it is an *integer* (address) plus type information.

tPtr class inherits from *long* so it can be used as a normal *long* integer. For example, it's OK to use it in arithmetical expressions. In some rare cases the library functions check for type of passed object explicitly. You can always convert *tPtr* to a plain long integer by doing conversion explicitly,

```
i = long(tptr)
```

Please note that at this moment this class is intended mainly for internal use. Objects of this type are returned as needed, but you should not attempt to create them yourself.

The main reason for needing this special type is preserving information while reading global variables (see the description of *readSymbol*).

2.4.3 SmartString

This type is used to represent variables and structure fields declared in C as `char *` or `char []`. This is a subclass of the standard Python string, with additional data attached and some methods redefined. We subclass to preserve the pointer value and address of the variable. We might need this to access the pointer itself if `char *` is used as a generic pointer instead of more correct `void *`. Another use for it is mapping char arrays where they are used to store byte values, not to represent an ASCII string. For example, in sources a variable declared like this:

```
char *testvar;
```

If the first 7 bytes of it are `abc\0def` it probably makes sense to interpret it as an ASCII string `abc`. In most cases this would be acceptable, but sometimes we need to access other bytes. We'll be able to do the following in Python:

```
# Read the variable, return
# SmartString object
s = readSymbol("testvar")
# Print this string using C
# NULL-terminated convention (i.e."abc")
print s
# print 2 chars after NULL
print s[4:6]
# Print the address of testvar
print Addr(testvar)
# print the pointer value
print long(testvar)
```

By default, `readSymbol()` reads and stores just the first 256 chars. If you need to read more, you can use the pointer value (retrieved as shown above).

2.4.4 Supporting Different Kernels

We have already discussed this briefly in 1.2. Now let us revise it by looking at some examples from real program.

Each object representing a struct/union has a number of attributes, mapped from C. In addition to those attributes, we can add our own. This is usually a sensible approach to isolate the dependencies on a specific kernel. For example, in some kernels `spinlock_t` is declared as

```
typedef struct {
    volatile unsigned int lock;
#ifdef CONFIG_DEBUG_SPINLOCK
    unsigned magic;
#endif
} spinlock_t;
```

but in others the access to a field similar to `lock` is more complicated:

```
typedef struct {
unsigned int slock;
} raw_spinlock_t;

typedef struct {
    raw_spinlock_t raw_lock;
#ifdef CONFIG_GENERIC_LOCKBREAK
    unsigned int break_lock;
#endif
#ifdef CONFIG_DEBUG_SPINLOCK
    unsigned int magic, owner_cpu;
    void *owner;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map dep_map;
#endif
} spinlock_t;
```

We can declare a new attribute that will be equivalent to `lck.lock` for the first kernel but to `lck.raw_lock.slock` for the second kernel. We do this in the following way:

```
sn = "spinlock_t"
structSetAttr(sn, "Slock",
              ["raw_lock.slock", "lock"])
```

This should be called only once. At the moment when this is executed, the framework traverses the list `["raw_lock.slock", "lock"]` and verifies whether the needed structures/fields exist, that is — does a dereference chain specified in a list element make sense? As soon as we find a match, we add this attribute (implemented as a "property" to the class to be used to represent this typedef). If no match was found, `structSetProcAttr` returns `False`. In case of success, later we can do the following:

```
sl = lck.Slock
```

The result will be the value of the function associated with that attribute—there will be no runtime overhead for checking structure definitions. The default function returns the value for the dereference chain that matched. In addition to this, we can specify an alternative function, for example:

```
# Programmatic attrs
def getSrc6(tw):
    # Some code...
    #
    # which returns this
    return val

sn = "struct tcp_timewait_sock"
structSetProcAttr(sn, "Src6", getSrc6)
```

In this case every time we use `tw.Src6` where `tw` is a result of the type `struct tcp_timewait_sock`, the function `getSrc6` will be used

2.5 Performance

The performance of Python language itself is more than adequate for our purpose. Python uses two-stage process:

- compile to pseudocode (and write results to files)
- execute pseudocode using a virtual machine

This is similar to Java. There are two JIT compilers to further increase the performance but they are still rather experimental and not ready for production. Still, the performance is excellent — summing 10 million integers in a loop takes less than 3s on a 2-year old laptop. The main performance bottleneck is due to the fact that *PyKdump* sources are compiled without any knowledge about symbolic information from the kernel. This is good as everything is compiled in advance. Even if we write a huge (> 100,000 lines) program, it will be compiled once for all kernels and start time in ***crash*** will be reasonably small.

SIAL uses a different approach: the compilation to pseudocode is done while loading the script. This means that if SIAL script is huge (and it is unclear how scalable is SIAL), loading it will take significant time.

With *PyKdump* programs, loading is very fast. After starting ***crash*** we load a rather small extension. This extension (written in C) consists of an embedded Python interpreter and subroutines to interface ***crash***. Then, when we want to execute a program, the interpreter loads it from files that are already compiled to pseudocode.

Accessing symbolic information from vmlinux is rather slow. We do it only once for each type, after which this information stays in memory until we exit ***crash***. This means that if we need to run several programs using the same structures, they will share this information.

A problem specific to *PyKdump* implementation is that traversing the dereference chain is a rather expensive operation. Here is what happens when we do `s.a`:

- we check the type information for object `s` and verify that it has a member `s`
- we find the address of member `a` and its type
- we create and return a new object of the needed type

If we have a longer dereference chain such as `s.a.b.c` this process is repeated. This is much longer than in C or SIAL where all type analysis is made at the compilation stage and not during runtime. To improve the performance, we use a number of tricks:

- using efficient functions ("readers") to dereference specific member
- metaclasses to build new classes on the fly to represent specific C-type
- using pseudoattributes for long dereference chains—they analyze all needed symbolic info and generate an efficient function to return the result quickly

As a result of all these optimizations, the performance for dereference chains is on par (but still somewhat lower) with SIAL. Arithmetical/logical operations on base types are much faster than those for SIAL.

The performance of real tools is more than adequate. The first run is always slower than subsequent runs. For example, running ***xportshow*** on a live kernel and emulating "netstat -an":

- first run 1.06s (real) 1.00s (CPU)
- second run 0.15s (real) 0.13s (CPU)

2.6 Packaging And Usage

Building *PyKdump* from sources is described at `http://pykdump.wiki.sourceforge.net/Building`. It is recommended to build from SVN using the "testing" branch instead of "trunk". "testing" is where we copy recent versions when they are more or less tested; "trunk" is much more experimental. There are some prebuilt packages on SF site (they are rather old).

To use *PyKdump*, you need just a single extension file. It usually has a name ***mpykdump64.so*** on 64-bit hosts but you can rename it as you wish. You start your ***crash*** session as usual and after that load the extension by doing

```
crash32> extend /tmp/mpykdump32.so
/tmp/mpykdump32.so: shared object loaded
```

The extension file contains all needed components:

- embedded Python interpreter
- an interface module to ***crash*** internals
- a subset of Python Standard Library
- some standard tools built on top if *PyKdump*, such as ***xportshow*** and ***crashinfo***

The extension file is constructed as a shared library with ZIP-archive appended. It is acceptable to to add your own programs directly to the extension file by using ***zip*** command. This is mainly useful for distribution; normally you develop and run programs directly from Python files. For example:

```
-----------hello.py--------------
# This is a basic PyKdump program
from pykdump.API import *

print "Hello PyKdump"
---------------------------------
crash32> epython hello.py
Hello PyKdump
crash32> epython hello
Hello PyKdump
```

3 XPORTSHOW

xportshow is a tool written using *PyKdump*. It is interesting that in addition to using it for general troubleshooting (e.g. HP Linux support organizations) it has been deployed by computer security experts such as "Volatile Systems."

The tool uses short options similar to those of ***netstat*** plus many long options. You can use multiple '-v' to increase the verbosity, up to '-vvv'. `xportshow -h` lists all options and there is additional documentation at `http://pykdump.wiki.sourceforge.net/xportshow`

You can find some examples of ***xportshow*** outputs at the end of this paper.

References

[1] `http://people.redhat.com/anderson/`

[2] `http://www.python.org`

[3] `http://www.llnl.gov/linux/slurm/coding_style.pdf`

```
crash32> xportshow -at
tcp   0.0.0.0:42691            0.0.0.0:*                    LISTEN
tcp   127.0.0.1:9161           0.0.0.0:*                    LISTEN
tcp   0.0.0.0:8010             0.0.0.0:*                    LISTEN
tcp   127.0.0.1:9165           0.0.0.0:*                    LISTEN
tcp   0.0.0.0:111              0.0.0.0:*                    LISTEN
tcp6  :::22                    :::*                         LISTEN
tcp   127.0.0.1:631            0.0.0.0:*                    LISTEN
tcp   15.236.177.25:52414      16.236.16.79:5223            ESTABLISHED
tcp   15.236.177.25:53004      69.159.122.174:22            ESTABLISHED
tcp   15.236.177.25:35015      15.37.113.20:143             ESTABLISHED
tcp   127.0.0.1:47939          127.0.0.1:9165               ESTABLISHED
tcp   127.0.0.1:9165           127.0.0.1:47939              ESTABLISHED
tcp   127.0.0.1:9161           127.0.0.1:54388              TIME_WAIT
tcp   127.0.0.1:9161           127.0.0.1:54387              TIME_WAIT

crash32> xportshow -atv
------------------------------------------------------------------------------
<struct tcp_sock 0xf62c8000> TCP
tcp 0.0.0.0:42691 0.0.0.0:* LISTEN
family=PF_INET
backlog=0(16)
max_qlen_log=5 qlen=0 qlen_young=0
------------------------------------------------------------------------------
<struct tcp_sock 0xf7580980>            TCP
tcp   15.236.177.171:51095     16.236.16.79:5223            ESTABLISHED
windows: rcv=63480, snd=32767  advmss=1398 rcv_ws=0 snd_ws=0
nonagle=0 sack_ok=0 tstamp_ok=0
rmem_alloc=0, wmem_alloc=0
rx_queue=0, tx_queue=0
rcvbuf=87380, sndbuf=16384
rcv_tstamp=7.8 s, lsndtime=10.2 s  ago
------------------------------------------------------------------------------
<struct tcp_sock 0xf7954e40> TCP
tcp 127.0.0.1:9161             127.0.0.1:54393              TIME_WAIT
tw_timeout=15000, ttd=1730

crash32> xportshow -ltvv
<struct sock 0xd5c6c600>                TCP
tcp   0.0.0.0:7778             0.0.0.0:*                    LISTEN
         family=PF_INET
         backlog=129(128)
         max_qlen_log=10 qlen=69 qlen_young=1
    --- Accept Queue <struct open_request 0xf001e600>
          laddr=128.8.61.4 raddr=10.148.6.13
          laddr=128.8.61.4 raddr=10.148.2.101
          laddr=128.8.61.4 raddr=10.149.6.7
    --- SYN-Queue
          laddr=128.8.61.4           raddr=128.8.11.24
          laddr=128.8.61.4           raddr=10.148.16.12
          laddr=128.8.61.4           raddr=10.152.0.45
          laddr=128.8.61.4           raddr=10.149.4.8
```

Figure 1: TCP Connections Info

```
crash32> xportshow -uav
-------------------------------------------------------------------------------
<struct udp_sock 0xf791b280>            UDP
udp6  ::1:123                     :::*                       st=7
        rx_queue=0, tx_queue=0
        rcvbuf=110592, sndbuf=110592
        pending=0, corkflag=0, len=0
-------------------------------------------------------------------------------
<struct udp_sock 0xf791b000>            UDP
udp6  :::123                      :::*                       st=7
        rx_queue=0, tx_queue=0
        rcvbuf=110592, sndbuf=110592
        pending=0, corkflag=0, len=0
-------------------------------------------------------------------------------

crash32> xportshow -ax
unix   State          I-node  Path
----------------------------------
unix   LISTEN         17667   /var/run/acpid.socket
unix   LISTEN         17996   @/var/run/hald/dbus-eYQQ7ZQwSxe
unix   LISTEN         17928   /var/run/dbus/system_bus_socket
unix   LISTEN         19733   /dev/gpmctl

crash32> xportshow -awv
-------------------------------------------------------------------------------
<struct raw_sock 0xe7678600>            RAW
raw   0.0.0.0:1                   0.0.0.0:*                  st=7
        rx_queue=0, tx_queue=0
        rcvbuf=131072, sndbuf=2048
```

Figure 2: Other Protocols Info

```
crash32> xportshow --summary
TCP Connection Info
-------------------
        ESTABLISHED      7
          TIME_WAIT      2
             LISTEN      7
                        NAGLE disabled (TCP_NODELAY):     1
UDP Connection Info
-------------------
  13 UDP sockets, 0 in ESTABLISHED
Unix Connection Info
------------------------
        ESTABLISHED    331
              CLOSE     12
             LISTEN     21
Raw sockets info
--------------------
              CLOSE      1
Interfaces Info
---------------
  How long ago (in seconds) interfaces trasmitted/received?
          Name     RX          TX
          ----   ----------  ---------
           lo         1.9       7467.3
         eth0         4.2          7.2
        wmaster0      7467.3         57.3
         eth1      7467.3       7467.3
         tun0         7.2          7.2
```

Figure 3: Summary

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada