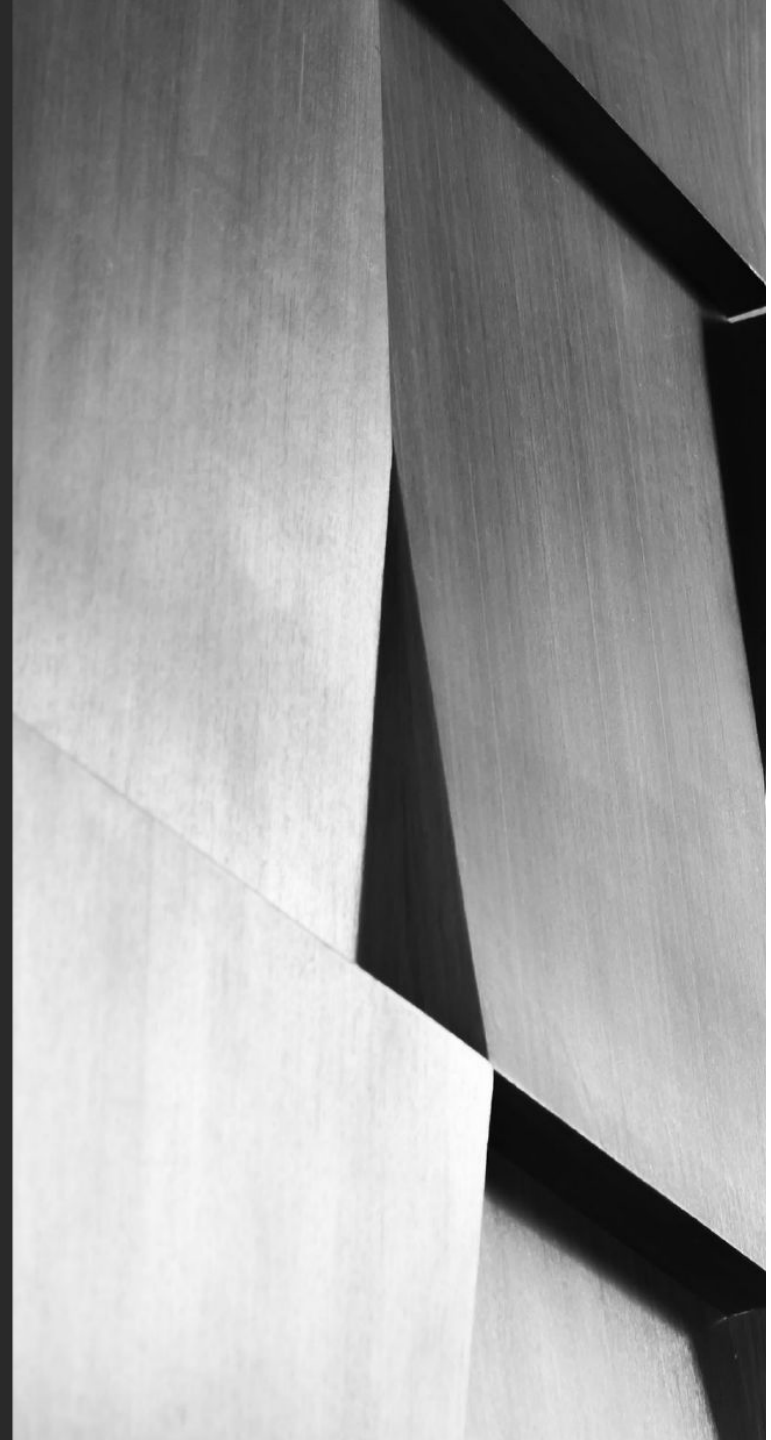


Kernel debugging approaches 2/2

Post-mortem

Vratislav Bendel
Principal Software Maintenance Engineer

Red Hat

What we'll cover today:

- Kernel vmcore collection
  - kexec mechanism & kdump.service
- Panic types
  - non-exhaustive list
- Crash tool
  - how to?
- Analysis tutorial
  - examples, tips & tricks

Kernel vmcore collection

kexec & kdump.service

Kexec

Boot into another kernel

Kexec is a systemcall that enables you to boot into another kernel.

- **kexec [-p]** the '-p' flag specifies a "panic kernel"
- **crashkernel=[auto]**

  The 'crashkernel=' kernel command line parameter specifies size of memory region reserved for a "panic kernel".

Kdump

Systemd.service automation

Kdump is a systemd.service that handles all component for kernel vmcore collection.

- **/etc/kdump.conf**

  Configuration file - specify dump target and method

- **kdump initramfs / initrd image**

  Contains software to mount the dump target and specialized service to save the vmcore.

Vmcore collection

How it works?

- Primary kernel reserves ‘crashekernel=’ region at boot
- Kdump.service sets up kdump-initramfs and loads panic kernel
- Once primary kernel calls panic(), kexec boots into panic kernel
- Panic kernel’s systemd-init sequence is set up to:
  - Mount the dump target FS
  - Special kdump.service saves the vmcore and performs *final_action* (default *‘reboot’*)

Vmcore collection

Testing

It's always recommended to test your kdump setup to be sure it works.

- Manual panic: ***# echo 'c' > /proc/sysrq-trigger*** (*)
- After the machine reboots, check the dump target. It should contain:
  - hostname-timestamp directory (default) containing:
    - vmcore
    - vmcore-dmesg.txt
    - [kexec-dmesg.txt] (depending on version)

(*) Sysrq facility needs to be enabled – see */proc/sys/kernel/sysrq* – *man proc*

Vmcore collection

Troubleshooting

Kdump.service failure >

- Check kdump.service logs - it should tell you what's wrong

Kdump.service OK, vmcore "incomplete" >

- Low disk space on target
- Unstable network connection to remote target

Vmcore collection

Troubleshooting

Kdump.service OK, no vmcore at all >

- Sufficient crashkernel= size ?
- Read-only target filesystem ?

You may use *'failure_action=shell'* to drop into a shell on the panic kernel and troubleshoot from there.

If you'd need extra tools you can add them via *'extra_bins'*.

Optionally you may utilize *'kdump_pre'* or *'kdump_post'* hooks.

Panic types

Keep your towel close!

Sysrq panic

Explicit user-issued panic

- Done by **panic sysrq** - either sys_write or magic key combo

- Commonly used by clustering software for “node fencing”
- Usually used when there’s problem with workload, not necessarily with the kernel

Kernel BUGs

Code-defined BUG()

Panics when a **BUG()** or conditional **BUG_ON()** defined in kernel source code gets executed. Usually represented as ‘ud2’ instruction.

- The precise source file+line are logged.

Optionally also WARN() and WARN_ON() can panic via

- */proc/sys/kernel/panic_on_warn* [0|1] default ‘0’

OOM

Out of memory

Optional panic when kernel's **OOM-killer** get invoked.

- */proc/sys/vm/panic_on_oom* [0|1|2]
  - 0 – no panic (default)
  - 1 – panic on global OOM
  - 2 – panic on every OOM (including cgroup limit OOM)

- The logged OOM report contains valuable information

Hung task panic

Blocked tasks, hung system

Optional panic when kernel reports “blocked task”

- */proc/sys/kernel/hung_task_panic* [0|1]

A task was in UNinterruptible sleep for longer than threshold:

- */proc/sys/kernel/hung_task_timeout* *default: 120 sec*

NULL deref

Dereferencing a bad pointer

Happens when kernel-space code attempts to dereference an inappropriate address pointer (0x0, 0xaf, 0xf002, ...)

- Most commonly an access to a struct member that is 0x0
- Memory corruptions manifest usually as NULL derefs

General protection fault

Bad read/write

General protection fault is a processor exception generated when the current “context” doesn’t have adequate permissions to perform the issued memory operation. For example:

- Reading a reserved/protected page “owned” by different “context”.
- Writing to a read-only page.
- Attempting to execute a non-executable page.

- The “type” is logged. Troubleshooting depends on type.

Soft lockup

CPU not rescheduling

Soft lockup is a condition when a CPU doesn't reschedule running tasks.

- Cannot happen in preemptible context.
- Locking bugs - note spinlocks
- Endless loops - easy fix = cond_resched()
- Starvation by SCHED_FIFO tasks

- Threshold = */proc/sys/kernel/watchdog_thresh* ***2** (default 20 sec)

- NOTE: spurious soft lockups due to **vCPU lags** in VMs

Hard lockup

CPU uncontrollable

Hard lockup is a condition when a CPU doesn't handle interrupt.

- Effectively makes the CPU uncontrollable.
- "soft lockup" in irq_disabled context
- HW malfunction

- Check the "interrupts enabled" CPU flag

- Threshold = */proc/sys/kernel/watchdog_thresh* (default 10 sec)

Lockup detection

Kernel watchdogs

Soft & hard lockup detection has 3 components:

- watchdog **kthread**
  - High sched prio. Increments a counter (Task was rescheduled).
- watchdog **interrupt** (high-resolution timer)
  - Checks if the watchdog kthread counter is being incremented.
  - Saves timestamp (Interrupt was handled).
- watchdog **NMI**
  - Implemented as perf_event
  - Checks if last watchdog interrupt timestamp is within threshold.

Understanding how lockups are detected is important in order to determine what exactly went wrong.

Lockup detection

sysctls

- */proc/sys/kernel/**softlockup_panic*** *[0|1]*
- */proc/sys/kernel/**hardlockup_panic*** *[0|1]*

- */proc/sys/kernel/**wachdog_thresh*** *default = 10*
  - *10 sec hard lockup* *20 sec soft lockup*

- */proc/sys/kernel/**nmi_watchdog*** *[0|1]* *Hard lockup detector*
- */proc/sys/kernel/**soft_watchdog*** *[0|1]* *Soft lockup detector*
- */proc/sys/kernel/**watchdog*** *[0|1]* *Both soft & hard*

- */proc/sys/kernel/**watchdog_cpumask***

Unknown NMI

Uhhuh..

- */proc/sys/kernel/panic_on_unrecovered_nmi* [0|1] default '0'

- "Unrecovered NMI" - NMI without a registered handler

- Can be used to panic server via hardware NMI button
- Useful to panic "hanged" server to get a vmcore to inspect

RCU stall panic

Synchronization problems

- */proc/sys/kernel/panic_on_rcu_stall* [0|1] default '0'

- May be useful for high-availability cluster fencing

Crash tool

Vmcore analysis

Crash tool

Start ‘crash’

- ***# crash <vmlinux> <vmcore>***

- Takes in “~/.crashrc”
  - Executes whatever is in crashrc in order
  - Useful for “default initial information” and loading scripts and plugins
  - Hint: add “set hex” to your crashrc =)

Crash tool

Interactive environment

- ***crash>*** prompt

- You may use general shell commands with '!' prefix
  - crash> !ls
  - crash> !cd ...
  - crash> !cat <file>

Crash tool

System info

- *crash>* ***sys***

  Prints basic system information

- *crash>* ***sys -i***

  Prints HW/FW-related information, similar to “dmidecode”

Crash tool

Kernel ring buffer log

- *crash>* ***log [-T]*** *[| less]*

  Inspect the kernel ring buffer log.

  '-T' translates timestamps to human time.

- Hint: Checking logs before the panic happened, as well as the panic report, is always a good place to start =)

Crash tool

Man pages

- *crash>* ***help [<command>]***

  Check what commands are available.

  Check man-pages for various commands.

Crash tool

Current context

Crash maintains a “current” context, which may be changed via ‘set’ command. By default this is set to the “panicking task” - the task that was active on the CPU which called panic().

Crash tool

CPU backtrace

- *crash>* ***bt***

  Prints stack backtrace of the **kernel stack** of the "current" task in a human readable form (gdb-like **).

- *crash>* ***bt [-c XYZ] [-r] [-f] [PID | *task_struct]***
  - '-c' specifies which CPUs' active tasks' kernel stacks to print ('-a' for "all")
  - '-f' interleaves the function returns with raw stack data
  - '-r' prints the raw stack
  - You may specify a PID or a task_struct pointer

(**) Crash is heavily inspired by *gdb*

Crash tool

CPU runqueue

- *crash>* ***runq***

  Prints “current” CPU runqueue.

- *crash>* ***runq -c XYZ***
  - ‘-c’ specifies which CPUs’ runqueue to print (‘-a’ for “all”)

- *crash>* ***runq -T***
  - ‘-T’ prints difference of runqueue timestamps to current time
  - Useful to understand if kernel timers were stuck on any CPUs

Crash tool

Timers

- *crash>* ***timer***

  Prints timers on standard timer bases

- *crash>* ***timer -r***

  Prints timers on high-resolution timer bases

Crash tool

Loaded modules

- *crash>* ***mod***

  Prints loaded modules

- *crash>* ***mod -t***

  Prints modules that cause *kernel taint*

Crash tool

Filesystems and Networks

- *crash>* ***mount***

  Prints mounted filesystems

  Useful to get superblock pointers.

- *crash>* ***net***

  Prints network interfaces

Crash tool

General memory stats

- *crash>* ***kmem -i***

  Check general memory stats

- *crash>* ***kmem -s***

  Print “slabinfo”

The *“kmem”* command is used to inspect memory metadata information about address pointers (more on a later slide). The ‘-i’ parameter makes it to consolidate information about whole memory into general stats.

Crash tool

Processes

- *crash>* ***ps [-S] [-m] [-y XYZ]***

  Print process list

  - ‘-S’ prints number of processes per state
  - ‘-m’ prints time how long is the process in the current state
  - ‘-y’ prints only processes with the specified scheduling policy

Crash tool

Opened file descriptors

- *crash>* ***files [PID | *task_struct]***

  Prints opened file descriptors

- *crash>* ***files [-d *dentry] [-p *inode]***
  - '-d' prints information about the file specified by *dentry
  - '-p' prints information about the file specified by *inode

Crash tool

Process' virtual memory

- *crash>* ***vm [PID | *task_struct]***

Prints virtual memory mappings, similar to */proc/PID/maps*

Crash tool

Simple print tools

- *crash>* ***eval [-b] <number>***

  Prints the given number of dec, oct, hex and binary.

  - '-b' also prints which specific bits are set to '1'

- *crash>* ***p***

  Standard "print" command like bash "echo". Useful for arithmetics.

Crash tool

Disassembly

- *crash>* ***dis <address | symbol>***

  Prints disassembly machine code.

  When a function symbol is inputted, prints the whole function.

  When address is inputted, interprets data on the address as code (*)

- *crash>* ***dis [-r] [-f] [-l]***
  - '-r' prints code from start of the function until the address
  - '-f' prints code from address till end of function
  - '-l' interleaves code with source code file+line references

(*) Crash won't verify if the data on the address are actually valid code...

Crash tool

Symbol information

- *crash>* ***sym <address | symbol>***

  Translates symbol to it's virtual address or vice versa.

- *crash>* ***sym -l***

  Lists all symbols ('kallsyms') - useful for grepping.

- *crash>* ***whatis <symbol>***

  Prints function header info

Crash tool

Struct contents

- *crash>* ***struct <struct_name> <address>***

  Interprets data starting at address as <struct_name> data.

- *crash>* ***struct <struct_name>.<member>[,<member>]***

  Print only specified struct members

- *crash>* ***struct <struct_name> -o [<address>]***

  Print member offsets.

Crash tool

Lists and Trees

- *crash>* ***list [-H] <address>***

  crash> ***tree [-t <type>] [-N] <address>***

  Interprets the given address as *list_head* or tree node of the given type and and prints the whole list or tree respectively.
  - '-H'/'-N' specifies the "head" of the list or the tree root node.

- These can be used also in a more complex construct with ***[-s struct [-l offset]]*** to print whole structs instead of just *list_head/tree_node pointers*.

Crash tool

Inspecting per-cpu data

- *crash>* ***kmem -o***

  Print CPU percpu base addresses

- *crash>* ***p <percpu_symbol>***

  Print percpu addresses for given symbol

- *crash>* ***p <symbol>[:cpuspec]***

  crash> ***struct <struct_name> <symbol|address>[:cpuspec]***

  Print specific percpu struct contents (':a' for "all")

Crash tool

Read raw data

- *crash>* ***rd <address> <number>***

  Prints data on given address and ‘number’ of subsequent 64-bit address pointers.

- *crash>* ***rd [-S[S]]***

  If the data contain known symbols, resp. slab objects, these options print those information.

  Also several other commands tend to have similar ‘-S[S]’ options.

Crash tool

Address information

- *crash>* ***vtop <address>***
- *crash>* ***ptov <address>***

  Translate given address from virtual to physical or vice versa

- *crash>* ***kmem <address>***

  Learn information about the address.

Crash tool

Search

- *crash>* ***search [-t] [-m 0xXYZ] <value>***

  Search for addresses which contain the <value> within the vmcore.

  - '-t' searches only kernel stacks of tasks
  - '-m' ignores specified bits in the <value>

Useful when looking for "which task was working with a given object", "what does this object belong to" or looking for a parent struct pointer.

Practical examples

Common procedures

Mindset

How to think when analyzing a vmcore

- Understanding state of system:
  - A round-trip personal routine of commands
- Checking something specific:
  - 1) **Define what you want to learn**
  - 2) Determine what data you need to inspect
  - 3) Use adequate commands to obtain the data

- **Avoid** “just looking around” if there’s **something**

For each

Working with sets of pointers

Crash provides a ***'foreach'*** command via which you can execute given command on specific group of processes. However sometimes it's beneficial to execute some command (ex. *'struct'*) on a set of addresses, which unfortunately cannot be done via the *'foreach'* command.
For that there's a **useful trick** - parse out the set of addresses into a file and the use the file as input for another command:

- *crash> 'command' | awk '{parser}' > my_parsed_data.txt*
  crash> 'command2' < my_parsed_data.txt

Longest blocked tasks

Checking “hung” system

The *‘ps -m’* nicely gives you times how long a task is in its current state.

This can be easily filtered based on task state:

- *crash> ps -m | grep UN*

You can then check what are the longest blocked tasks waiting for.

Process RSS

Analyzing memory usage

The *'ps' command prints out all threads, hence when simply adding up their RSS amounts, you may count thread-shared pages multiple times.*

To avoid that, you may print only the thread group leader:

- *crash> ps -G | ... | sort -nrk X | ...*

Interrupts enabled CPU flag

Confirming hard lockup relevance

Hard lockup is quite uncommon situation and may very well indicate hardware malfunction. One of the first things you should check is whether the hard lockup indeed is valid:

- Check the locked up CPU's flags if interrupts are disabled:
  - *crash> bt*

    ...

    RFLAGS: 00000246 <- 0x200 means "interrupts enabled"

    ...

Note if a CPU is stuck in IRQ context, it also can't handle another interrupt.

Checking lock owners

Analyzing lockups/hung-ups

Spinlocks:

The kernel should not reschedule when holding a spinlock - the routine needs to unlock it. So you should be able to find an active process on some CPU executing in context where it holds the spinlock.

- *crash> bt -a*
- *crash> search -t <spinlock_addr> (or addr of the parent struct)*

Checking lock owners

Analyzing lockups/hung-ups

Mutexes, RT_Mutexes, RW_Semaphores:

These locks contain an *"->owner"* member, so you should be able to find the task holding it quite easily (or at least the one which lastly locked it). The *"->owner"* member can have certain flag bits on the lowest bits.

Note that *rw_sem* has read and write lock mode. Write lock has standard single exclusive *"->owner"*, but read mode stores in *"->owner"* the **last** task which acquired the read lock.

- Use *"struct"* command to get the *"->owner"* member

Finding data on stack

Function arguments or other pointers

To identify specific data a task was working with, you need to understand how and where are data stored on stack and diligently follow the execution.

- 1) Identify the register where the data of interest were stored

  Function arguments passed in order via:

  RDI, RSI, RDX, RCX, R8, R9

  Sometimes (or if there's more than 6 arguments) the compiler may optimize to save a reference on stack and load from there

  You'd see that as offsetted loads from RBP or RSP

Finding data on stack

Function arguments or other pointers

- 2) Follow the machine code (backwards or forward) to find out where it got saved on stack.

  A great help is to see if the data of interest gets stored in ***non-volatile* register**. To maintain register non-volatility, the compiler commonly saves contents of those registers on stack in the function prologue.

- DEMO

Tip – save your crafts

Working efficiently

Whenever you find yourself using some command construct more often – save it under some alias in your *"~/.crashrc"*.

Whenever you spend time to craft a more complex command construct to obtain some information that could be considered relatively general, make sure to save your command construct.

Assignment

How to write your analysis?

Assignment

Intro

You will receive a vmcore file and relevant vmlinux so you can start crash.

You have list of Requirements you should elaborate in your solution.

Your solution should list explicit full crash commands you used to obtain data outputs, along with the outputs in unchanged form (you can trim).

Along with commands and data outputs you should elaborate what is relevant in the specific data output you included in your solution.

It should always be clear where did you get data that you are working with.

Assignment

Data output form

<Commentary what I want to learn>

~~~

crash> <full command>

…… (opt. trim)

<output as you get it from crash>

…… (opt. trim)

~~~

<Commentary what's relevant in the data above>

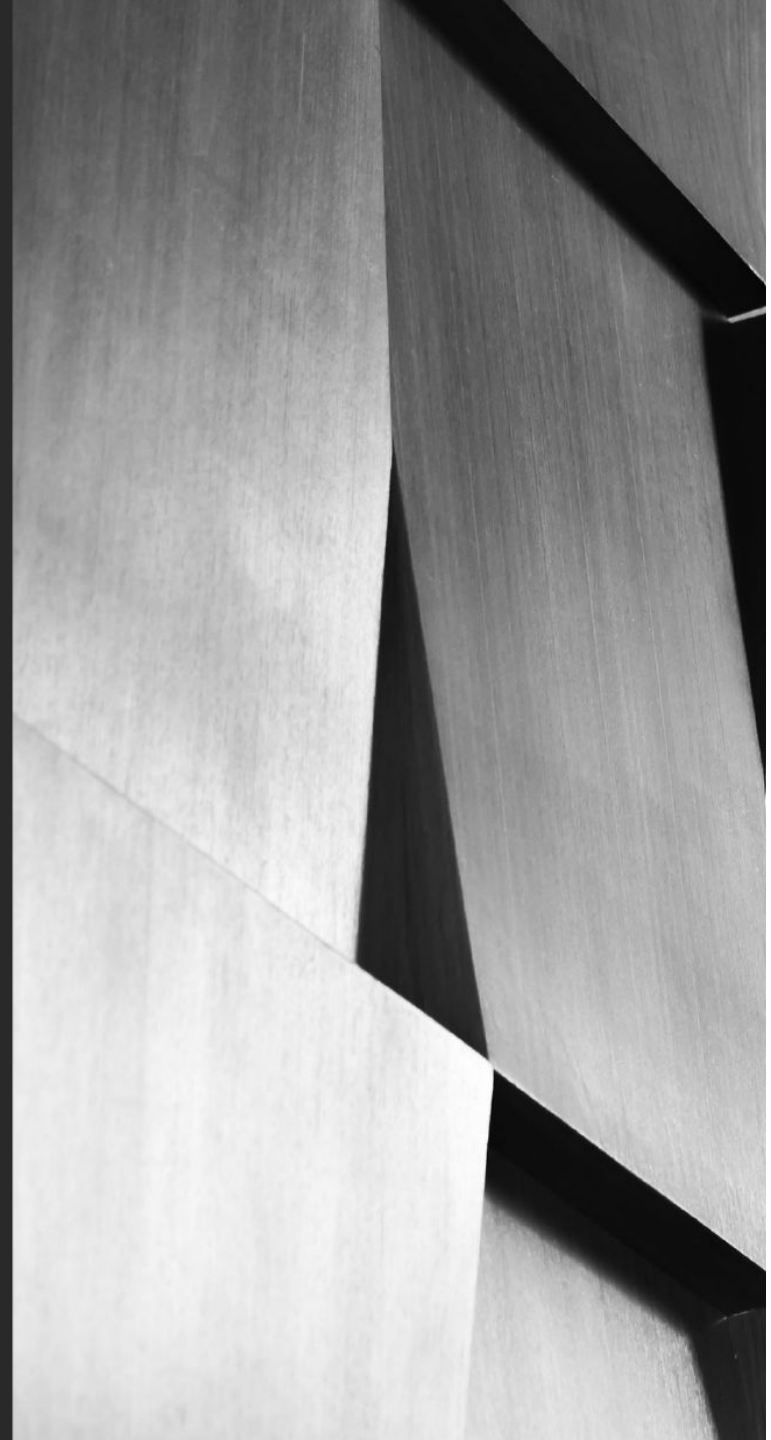
Assignment

Professionality

You may freely merge several mental steps into single output - no need to comment on every command you use.

The point it to create comprehensive “thought blocks” that have an idea at the start and outcome/resolution at the end.

Imagine you are writing and analysis to a client/customer - how would you write it in a professional manner, such that a technical person would understand what you found and how.



Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.

linkedin.com/company/red-hat

facebook.com/redhatinc

youtube.com/user/RedHatVideos

twitter.com/RedHat

Red Hat