Development tools for the Kernel

Release

The kernel development community

Nov 02, 2017

CONTENTS

This document is a collection of documents about development tools that can be used to work on the kernel. For now, the documents have been pulled together without any significant effot to integrate them into a coherent whole; patches welcome!

Table of contents

CHAPTER

ONE

COCCINELLE

Coccinelle is a tool for pattern matching and text transformation that has many uses in kernel development, including the application of complex, tree-wide patches and detection of problematic programming patterns.

Getting Coccinelle

The semantic patches included in the kernel use features and options which are provided by Coccinelle version 1.0.0-rc11 and above. Using earlier versions will fail as the option names used by the Coccinelle files and coccicheck have been updated.

Coccinelle is available through the package manager of many distributions, e.g. :

- Debian
- Fedora
- Ubuntu
- OpenSUSE
- Arch Linux
- NetBSD
- FreeBSD

You can get the latest version released from the Coccinelle homepage at http://coccinelle.lip6.fr/

Information and tips about Coccinelle are also provided on the wiki pages at http://cocci.ekstranet.diku.dk/wiki/doku.php

Once you have it, run the following command:

```
./configure
make
```

as a regular user, and install it with:

```
sudo make install
```

Supplemental documentation

For supplemental documentation refer to the wiki:

https://bottest.wiki.kernel.org/coccicheck

The wiki documentation always refers to the linux-next version of the script.

Using Coccinelle on the Linux kernel

A Coccinelle-specific target is defined in the top level Makefile. This target is named `coccicheck` and calls the `coccicheck` front-end in the `scripts` directory.

Four basic modes are defined: `patch`, `report`, `context`, and `org`. The mode to use is specified by setting the MODE variable with `MODE=<mode>`.

- `patch` proposes a fix, when possible.
- `report` generates a list in the following format: file:line:column-column: message
- `context` highlights lines of interest and their context in a diff-like style.Lines of interest are indicated with `-`.
- `org` generates a report in the Org mode format of Emacs.

Note that not all semantic patches implement all modes. For easy use of Coccinelle, the default mode is "report".

Two other modes provide some common combinations of these modes.

- `chain` tries the previous modes in the order above until one succeeds.
- `rep+ctxt` runs successively the report mode and the context mode. It should be used with the C option (described later) which checks the code on a file basis.

Examples

To make a report for every semantic patch, run the following command:

```
make coccicheck MODE=report
```

To produce patches, run:

```
make coccicheck MODE=patch
```

The coccicheck target applies every semantic patch available in the sub-directories of `scripts/coccinelle` to the entire Linux kernel.

For each semantic patch, a commit message is proposed. It gives a description of the problem being checked by the semantic patch, and includes a reference to Coccinelle.

As any static code analyzer, Coccinelle produces false positives. Thus, reports must be carefully checked, and patches reviewed.

To enable verbose messages set the V= variable, for example:

```
make coccicheck MODE=report V=1
```

Coccinelle parallelization

By default, coccicheck tries to run as parallel as possible. To change the parallelism, set the J= variable. For example, to run across 4 CPUs:

```
make coccicheck MODE=report J=4
```

As of Coccinelle 1.0.2 Coccinelle uses Ocaml parmap for parallelization, if support for this is detected you will benefit from parmap parallelization.

When parmap is enabled coccicheck will enable dynamic load balancing by using `--chunksize 1` argument, this ensures we keep feeding threads with work one by one, so that we avoid the situation where

most work gets done by only a few threads. With dynamic load balancing, if a thread finishes early we keep feeding it more work.

When parmap is enabled, if an error occurs in Coccinelle, this error value is propagated back, the return value of the make coccicheck captures this return value.

Using Coccinelle with a single semantic patch

The optional make variable COCCI can be used to check a single semantic patch. In that case, the variable must be initialized with the name of the semantic patch to apply.

For instance:

```
make coccicheck COCCI=<my_SP.cocci> MODE=patch
```

or:

```
make coccicheck COCCI=<my_SP.cocci> MODE=report
```

Controlling Which Files are Processed by Coccinelle

By default the entire kernel source tree is checked.

To apply Coccinelle to a specific directory, M= can be used. For example, to check drivers/net/wireless/ one may write:

```
make coccicheck M=drivers/net/wireless/
```

To apply Coccinelle on a file basis, instead of a directory basis, the following command may be used:

```
make C=1 CHECK="scripts/coccicheck"
```

To check only newly edited code, use the value 2 for the C flag, i.e.:

```
make C=2 CHECK="scripts/coccicheck"
```

In these modes, which works on a file basis, there is no information about semantic patches displayed, and no commit message proposed.

This runs every semantic patch in scripts/coccinelle by default. The COCCI variable may additionally be used to only apply a single semantic patch as shown in the previous section.

The “report” mode is the default. You can select another one with the MODE variable explained above.

Debugging Coccinelle SmPL patches

Using coccicheck is best as it provides in the spatch command line include options matching the options used when we compile the kernel. You can learn what these options are by using V=1, you could then manually run Coccinelle with debug options added.

Alternatively you can debug running Coccinelle against SmPL patches by asking for stderr to be redirected to stderr, by default stderr is redirected to /dev/null, if you’d like to capture stderr you can specify the DEBUG_FILE="file.txt" option to coccicheck. For instance:

```
rm -f cocci.err
make coccicheck COCCI=scripts/coccinelle/free/kfree.cocci MODE=report DEBUG_FILE=cocci.err
cat cocci.err
```

You can use SPFLAGS to add debugging flags, for instance you may want to add both –profile –show-trying to SPFLAGS when debugging. For instance you may want to use:

```
rm -f err.log
export COCCI=scripts/coccinelle/misc/irqf_oneshot.cocci
make coccicheck DEBUG_FILE="err.log" MODE=report SPFLAGS="--profile --show-trying" M=./drivers/
 ↪mfd/arizona-irq.c
```

err.log will now have the profiling information, while stdout will provide some progress information as Coccinelle moves forward with work.

DEBUG_FILE support is only supported when using coccinelle >= 1.2.

.cocciconfig support

Coccinelle supports reading .cocciconfig for default Coccinelle options that should be used every time spatch is spawned, the order of precedence for variables for .cocciconfig is as follows:

- Your current user's home directory is processed first
- Your directory from which spatch is called is processed next
- The directory provided with the –dir option is processed last, if used

Since coccicheck runs through make, it naturally runs from the kernel proper dir, as such the second rule above would be implied for picking up a .cocciconfig when using `make coccicheck`.

`make coccicheck` also supports using M= targets.If you do not supply any M= target, it is assumed you want to target the entire kernel. The kernel coccicheck script has:

```
if [ "$KBUILD_EXTMOD" = "" ] ; then
    OPTIONS="--dir $srctree $COCCIINCLUDE"
else
    OPTIONS="--dir $KBUILD_EXTMOD $COCCIINCLUDE"
fi
```

KBUILD_EXTMOD is set when an explicit target with M= is used. For both cases the spatch –dir argument is used, as such third rule applies when whether M= is used or not, and when M= is used the target directory can have its own .cocciconfig file. When M= is not passed as an argument to coccicheck the target directory is the same as the directory from where spatch was called.

If not using the kernel's coccicheck target, keep the above precedence order logic of .cocciconfig reading. If using the kernel's coccicheck target, override any of the kernel's .coccicheck's settings using SPFLAGS.

We help Coccinelle when used against Linux with a set of sensible defaults options for Linux with our own Linux .cocciconfig. This hints to coccinelle git can be used for `git grep` queries over coccigrep. A timeout of 200 seconds should suffice for now.

The options picked up by coccinelle when reading a .cocciconfig do not appear as arguments to spatch processes running on your system, to confirm what options will be used by Coccinelle run:

```
spatch --print-options-only
```

You can override with your own preferred index option by using SPFLAGS. Take note that when there are conflicting options Coccinelle takes precedence for the last options passed. Using .cocciconfig is possible to use idutils, however given the order of precedence followed by Coccinelle, since the kernel now carries its own .cocciconfig, you will need to use SPFLAGS to use idutils if desired. See below section "Additional flags" for more details on how to use idutils.

Additional flags

Additional flags can be passed to spatch through the SPFLAGS variable. This works as Coccinelle respects the last flags given to it when options are in conflict.

```
make SPFLAGS=--use-glimpse coccicheck
```

Coccinelle supports idutils as well but requires coccinelle >= 1.0.6. When no ID file is specified coccinelle assumes your ID database file is in the file .id-utils.index on the top level of the kernel, coccinelle carries a script scripts/idutils_index.sh which creates the database with:

```
mkid -i C --output .id-utils.index
```

If you have another database filename you can also just symlink with this name.

```
make SPFLAGS=--use-idutils coccicheck
```

Alternatively you can specify the database filename explicitly, for instance:

```
make SPFLAGS="--use-idutils /full-path/to/ID" coccicheck
```

See `spatch --help` to learn more about spatch options.

Note that the `--use-glimpse` and `--use-idutils` options require external tools for indexing the code. None of them is thus active by default. However, by indexing the code with one of these tools, and according to the cocci file used, spatch could proceed the entire code base more quickly.

SmPL patch specific options

SmPL patches can have their own requirements for options passed to Coccinelle. SmPL patch specific options can be provided by providing them at the top of the SmPL patch, for instance:

```
// Options: --no-includes --include-headers
```

SmPL patch Coccinelle requirements

As Coccinelle features get added some more advanced SmPL patches may require newer versions of Coccinelle. If an SmPL patch requires at least a version of Coccinelle, this can be specified as follows, as an example if requiring at least Coccinelle >= 1.0.5:

```
// Requires: 1.0.5
```

Proposing new semantic patches

New semantic patches can be proposed and submitted by kernel developers. For sake of clarity, they should be organized in the sub-directories of `scripts/coccinelle/`.

Detailed description of the `report` mode

`report` generates a list in the following format:

```
file:line:column-column: message
```

Example

Running:

```
make coccicheck MODE=report COCCI=scripts/coccinelle/api/err_cast.cocci
```

will execute the following part of the SmPL script:

```
<smpl>
@r depends on !context && !patch && (org || report)@
expression x;
position p;
@@

  ERR_PTR@p(PTR_ERR(x))

@script:python depends on report@
p << r.p;
x << r.x;
@@

msg="ERR_CAST can be used with %s" % (x)
coccilib.report.print_report(p[0], msg)
</smpl>
```

This SmPL excerpt generates entries on the standard output, as illustrated below:

```
/home/user/linux/crypto/ctr.c:188:9-16: ERR_CAST can be used with alg
/home/user/linux/crypto/authenc.c:619:9-16: ERR_CAST can be used with auth
/home/user/linux/crypto/xts.c:227:9-16: ERR_CAST can be used with alg
```

Detailed description of the `patch` mode

When the `patch` mode is available, it proposes a fix for each problem identified.

Example

Running:

```
make coccicheck MODE=patch COCCI=scripts/coccinelle/api/err_cast.cocci
```

will execute the following part of the SmPL script:

```
<smpl>
@ depends on !context && patch && !org && !report @
expression x;
@@

- ERR_PTR(PTR_ERR(x))
+ ERR_CAST(x)
</smpl>
```

This SmPL excerpt generates patch hunks on the standard output, as illustrated below:

```
diff -u -p a/crypto/ctr.c b/crypto/ctr.c
--- a/crypto/ctr.c 2010-05-26 10:49:38.000000000 +0200
+++ b/crypto/ctr.c 2010-06-03 23:44:49.000000000 +0200
@@ -185,7 +185,7 @@ static struct crypto_instance *crypto_ct
    alg = crypto_attr_alg(tb[1], CRYPTO_ALG_TYPE_CIPHER,
                          CRYPTO_ALG_TYPE_MASK);
    if (IS_ERR(alg))
-           return ERR_PTR(PTR_ERR(alg));
+           return ERR_CAST(alg);

    /* Block size must be >= 4 bytes. */
    err = -EINVAL;
```

Detailed description of the `context` mode

`context` highlights lines of interest and their context in a diff-like style.

> **NOTE**: The diff-like output generated is NOT an applicable patch. The intent of the `context` mode is to highlight the important lines (annotated with minus, `-`) and gives some surrounding context lines around. This output can be used with the diff mode of Emacs to review the code.

Example

Running:

```
make coccicheck MODE=context COCCI=scripts/coccinelle/api/err_cast.cocci
```

will execute the following part of the SmPL script:

```
<smpl>
@ depends on context && !patch && !org && !report@
expression x;
@@

* ERR_PTR(PTR_ERR(x))
</smpl>
```

This SmPL excerpt generates diff hunks on the standard output, as illustrated below:

```
diff -u -p /home/user/linux/crypto/ctr.c /tmp/nothing
--- /home/user/linux/crypto/ctr.c   2010-05-26 10:49:38.000000000 +0200
+++ /tmp/nothing
@@ -185,7 +185,6 @@ static struct crypto_instance *crypto_ct
    alg = crypto_attr_alg(tb[1], CRYPTO_ALG_TYPE_CIPHER,
                          CRYPTO_ALG_TYPE_MASK);
    if (IS_ERR(alg))
-           return ERR_PTR(PTR_ERR(alg));

    /* Block size must be >= 4 bytes. */
    err = -EINVAL;
```

Detailed description of the `org` mode

`org` generates a report in the Org mode format of Emacs.

Example

Running:

```
make coccicheck MODE=org COCCI=scripts/coccinelle/api/err_cast.cocci
```

will execute the following part of the SmPL script:

```
<smpl>
@r depends on !context && !patch && (org || report)@
expression x;
position p;
@@

  ERR_PTR@p(PTR_ERR(x))

@script:python depends on org@
p << r.p;
x << r.x;
@@

msg="ERR_CAST can be used with %s" % (x)
msg_safe=msg.replace("[","@(").replace("]",")")
coccilib.org.print_todo(p[0], msg_safe)
</smpl>
```

This SmPL excerpt generates Org entries on the standard output, as illustrated below:

```
* TODO [[view:/home/user/linux/crypto/ctr.c::face=ovl-face1::linb=188::colb=9::cole=16][ERR_
  ↪CAST can be used with alg]]
* TODO [[view:/home/user/linux/crypto/authenc.c::face=ovl-face1::linb=619::colb=9::cole=16][ERR_
  ↪CAST can be used with auth]]
* TODO [[view:/home/user/linux/crypto/xts.c::face=ovl-face1::linb=227::colb=9::cole=16][ERR_
  ↪CAST can be used with alg]]
```

CHAPTER

TWO

SPARSE

Sparse is a semantic checker for C programs; it can be used to find a number of potential problems with kernel code. See https://lwn.net/Articles/689907/ for an overview of sparse; this document contains some kernel-specific sparse information.

Using sparse for typechecking

"__bitwise" is a type attribute, so you have to do something like this:

```
typedef int __bitwise pm_request_t;

enum pm_request {
        PM_SUSPEND = (__force pm_request_t) 1,
        PM_RESUME = (__force pm_request_t) 2
};
```

which makes PM_SUSPEND and PM_RESUME "bitwise" integers (the "__force" is there because sparse will complain about casting to/from a bitwise type, but in this case we really _do_ want to force the conversion). And because the enum values are all the same type, now "enum pm_request" will be that type too.

And with gcc, all the "__bitwise"/"__force stuff" goes away, and it all ends up looking just like integers to gcc.

Quite frankly, you don't need the enum there. The above all really just boils down to one special "int __bitwise" type.

So the simpler way is to just do:

```
typedef int __bitwise pm_request_t;

#define PM_SUSPEND ((__force pm_request_t) 1)
#define PM_RESUME ((__force pm_request_t) 2)
```

and you now have all the infrastructure needed for strict typechecking.

One small note: the constant integer "0" is special. You can use a constant zero as a bitwise integer type without sparse ever complaining. This is because "bitwise" (as the name implies) was designed for making sure that bitwise types don't get mixed up (little-endian vs big-endian vs cpu-endian vs whatever), and there the constant "0" really _is_ special.

Using sparse for lock checking

The following macros are undefined for gcc and defined during a sparse run to use the "context" tracking feature of sparse, applied to locking. These annotations tell sparse when a lock is held, with regard to the annotated function's entry and exit.

__must_hold - The specified lock is held on function entry and exit.

__acquires - The specified lock is held on function exit, but not entry.

__releases - The specified lock is held on function entry, but not exit.

If the function enters and exits without the lock held, acquiring and releasing the lock inside the function in a balanced way, no annotation is needed. The tree annotations above are for cases where sparse would otherwise report a context imbalance.

Getting sparse

You can get latest released versions from the Sparse homepage at https://sparse.wiki.kernel.org/index.php/Main_Page

Alternatively, you can get snapshots of the latest development version of sparse using git to clone:

```
git://git.kernel.org/pub/scm/devel/sparse/sparse.git
```

DaveJ has hourly generated tarballs of the git tree available at:

```
http://www.codemonkey.org.uk/projects/git-snapshots/sparse/
```

Once you have it, just do:

```
make
make install
```

as a regular user, and it will install sparse in your ~/bin directory.

Using sparse

Do a kernel make with “make C=1” to run sparse on all the C files that get recompiled, or use “make C=2” to run sparse on the files whether they need to be recompiled or not. The latter is a fast way to check the whole tree if you have already built it.

The optional make variable CF can be used to pass arguments to sparse. The build system passes -Wbitwise to sparse automatically.

CHAPTER

THREE

KCOV: CODE COVERAGE FOR FUZZING

kcov exposes kernel code coverage information in a form suitable for coverage- guided fuzzing (randomized testing). Coverage data of a running kernel is exported via the "kcov" debugfs file. Coverage collection is enabled on a task basis, and thus it can capture precise coverage of a single system call.

Note that kcov does not aim to collect as much coverage as possible. It aims to collect more or less stable coverage that is function of syscall inputs. To achieve this goal it does not collect coverage in soft/hard interrupts and instrumentation of some inherently non-deterministic parts of kernel is disabled (e.g. scheduler, locking).

Usage

Configure the kernel with:

```
CONFIG_KCOV=y
```

CONFIG_KCOV requires gcc built on revision 231296 or later. Profiling data will only become accessible once debugfs has been mounted:

```
mount -t debugfs none /sys/kernel/debug
```

The following program demonstrates kcov usage from within a test program:

```
#include <stdio.h>
#include <stddef.h>
#include <stdint.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <fcntl.h>

#define KCOV_INIT_TRACE                     _IOR('c', 1, unsigned long)
#define KCOV_ENABLE                 _IO('c', 100)
#define KCOV_DISABLE                        _IO('c', 101)
#define COVER_SIZE                  (64<<10)

int main(int argc, char **argv)
{
    int fd;
    unsigned long *cover, n, i;

    /* A single fd descriptor allows coverage collection on a single
     * thread.
     */
    fd = open("/sys/kernel/debug/kcov", O_RDWR);
```

```
    if (fd == -1)
            perror("open"), exit(1);
    /* Setup trace mode and trace size. */
    if (ioctl(fd, KCOV_INIT_TRACE, COVER_SIZE))
            perror("ioctl"), exit(1);
    /* Mmap buffer shared between kernel- and user-space. */
    cover = (unsigned long*)mmap(NULL, COVER_SIZE * sizeof(unsigned long),
                                 PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if ((void*)cover == MAP_FAILED)
            perror("mmap"), exit(1);
    /* Enable coverage collection on the current thread. */
    if (ioctl(fd, KCOV_ENABLE, 0))
            perror("ioctl"), exit(1);
    /* Reset coverage from the tail of the ioctl() call. */
    __atomic_store_n(&cover[0], 0, __ATOMIC_RELAXED);
    /* That's the target syscal call. */
    read(-1, NULL, 0);
    /* Read number of PCs collected. */
    n = __atomic_load_n(&cover[0], __ATOMIC_RELAXED);
    for (i = 0; i < n; i++)
            printf("0x%lx\n", cover[i + 1]);
    /* Disable coverage collection for the current thread. After this call
     * coverage can be enabled for a different thread.
     */
    if (ioctl(fd, KCOV_DISABLE, 0))
            perror("ioctl"), exit(1);
    /* Free resources. */
    if (munmap(cover, COVER_SIZE * sizeof(unsigned long)))
            perror("munmap"), exit(1);
    if (close(fd))
            perror("close"), exit(1);
    return 0;
}
```

After piping through addr2line output of the program looks as follows:

```
SyS_read
fs/read_write.c:562
__fdget_pos
fs/file.c:774
__fget_light
fs/file.c:746
__fget_light
fs/file.c:750
__fget_light
fs/file.c:760
__fdget_pos
fs/file.c:784
SyS_read
fs/read_write.c:562
```

If a program needs to collect coverage from several threads (independently), it needs to open /sys/kernel/debug/kcov in each thread separately.

The interface is fine-grained to allow efficient forking of test processes. That is, a parent process opens /sys/kernel/debug/kcov, enables trace mode, mmaps coverage buffer and then forks child processes in a loop. Child processes only need to enable coverage (disable happens automatically on thread end).

CHAPTER

FOUR

USING GCOV WITH THE LINUX KERNEL

gcov profiling kernel support enables the use of GCC's coverage testing tool gcov with the Linux kernel. Coverage data of a running kernel is exported in gcov-compatible format via the "gcov" debugfs directory. To get coverage data for a specific file, change to the kernel build directory and use gcov with the `-o` option as follows (requires root):

```
# cd /tmp/linux-out
# gcov -o /sys/kernel/debug/gcov/tmp/linux-out/kernel spinlock.c
```

This will create source code files annotated with execution counts in the current directory. In addition, graphical gcov front-ends such as lcov can be used to automate the process of collecting data for the entire kernel and provide coverage overviews in HTML format.

Possible uses:

- debugging (has this line been reached at all?)
- test improvement (how do I change my test to cover these lines?)
- minimizing kernel configurations (do I need this option if the associated code is never run?)

Preparation

Configure the kernel with:

```
CONFIG_DEBUG_FS=y
CONFIG_GCOV_KERNEL=y
```

select the gcc's gcov format, default is autodetect based on gcc version:

```
CONFIG_GCOV_FORMAT_AUTODETECT=y
```

and to get coverage data for the entire kernel:

```
CONFIG_GCOV_PROFILE_ALL=y
```

Note that kernels compiled with profiling flags will be significantly larger and run slower. Also CONFIG_GCOV_PROFILE_ALL may not be supported on all architectures.

Profiling data will only become accessible once debugfs has been mounted:

```
mount -t debugfs none /sys/kernel/debug
```

Customization

To enable profiling for specific files or directories, add a line similar to the following to the respective kernel Makefile:

- For a single file (e.g. main.o):

```
GCOV_PROFILE_main.o := y
```

- For all files in one directory:

```
GCOV_PROFILE := y
```

To exclude files from being profiled even when CONFIG_GCOV_PROFILE_ALL is specified, use:

```
GCOV_PROFILE_main.o := n
```

and:

```
GCOV_PROFILE := n
```

Only files which are linked to the main kernel image or are compiled as kernel modules are supported by this mechanism.

Files

The gcov kernel support creates the following files in debugfs:

/sys/kernel/debug/gcov Parent directory for all gcov-related files.

/sys/kernel/debug/gcov/reset Global reset file: resets all coverage data to zero when written to.

/sys/kernel/debug/gcov/path/to/compile/dir/file.gcda The actual gcov data file as understood by the gcov tool. Resets file coverage data to zero when written to.

/sys/kernel/debug/gcov/path/to/compile/dir/file.gcno Symbolic link to a static data file required by the gcov tool. This file is generated by gcc when compiling with option `-ftest-coverage`.

Modules

Kernel modules may contain cleanup code which is only run during module unload time. The gcov mechanism provides a means to collect coverage data for such code by keeping a copy of the data associated with the unloaded module. This data remains available through debugfs. Once the module is loaded again, the associated coverage counters are initialized with the data from its previous instantiation.

This behavior can be deactivated by specifying the gcov_persist kernel parameter:

```
gcov_persist=0
```

At run-time, a user can also choose to discard data for an unloaded module by writing to its data file or the global reset file.

Separated build and test machines

The gcov kernel profiling infrastructure is designed to work out-of-the box for setups where kernels are built and run on the same machine. In cases where the kernel runs on a separate machine, special preparations must be made, depending on where the gcov tool is used:

1. gcov is run on the TEST machine

   The gcov tool version on the test machine must be compatible with the gcc version used for kernel build. Also the following files need to be copied from build to test machine:

from the source tree:

- all C source files + headers

from the build tree:

- all C source files + headers
- all .gcda and .gcno files
- all links to directories

It is important to note that these files need to be placed into the exact same file system location on the test machine as on the build machine. If any of the path components is symbolic link, the actual directory needs to be used instead (due to make's CURDIR handling).

2. gcov is run on the BUILD machine

The following files need to be copied after each test case from test to build machine:

from the gcov directory in sysfs:

- all .gcda files
- all links to .gcno files

These files can be copied to any location on the build machine. gcov must then be called with the -o option pointing to that directory.

Example directory setup on the build machine:

```
/tmp/linux:    kernel source tree
/tmp/out:      kernel build directory as specified by make O=
/tmp/coverage: location of the files copied from the test machine

[user@build] cd /tmp/out
[user@build] gcov -o /tmp/coverage/tmp/out/init main.c
```

Troubleshooting

Problem Compilation aborts during linker step.

Cause Profiling flags are specified for source files which are not linked to the main kernel or which are linked by a custom linker procedure.

Solution Exclude affected source files from profiling by specifying `GCOV_PROFILE := n` or `GCOV_PROFILE_basename.o := n` in the corresponding Makefile.

Problem Files copied from sysfs appear empty or incomplete.

Cause Due to the way seq_file works, some tools such as cp or tar may not correctly copy files from sysfs.

Solution Use `cat‘` to read `.gcda` files and `cp -d` to copy links. Alternatively use the mechanism shown in Appendix B.

Appendix A: gather_on_build.sh

Sample script to gather coverage meta files on the build machine (see 6a):

```
#!/bin/bash

KSRC=$1
KOBJ=$2
DEST=$3
```

```
if [ -z "$KSRC" ] || [ -z "$KOBJ" ] || [ -z "$DEST" ]; then
  echo "Usage: $0 <ksrc directory> <kobj directory> <output.tar.gz>" >&2
  exit 1
fi

KSRC=$(cd $KSRC; printf "all:\n\t@echo \${CURDIR}\n" | make -f -)
KOBJ=$(cd $KOBJ; printf "all:\n\t@echo \${CURDIR}\n" | make -f -)

find $KSRC $KOBJ \( -name '*.gcno' -o -name '*.[ch]' -o -type l \) -a \
                 -perm /u+r,g+r | tar cfz $DEST -P -T -

if [ $? -eq 0 ] ; then
  echo "$DEST successfully created, copy to test system and unpack with:"
  echo "  tar xfz $DEST -P"
else
  echo "Could not create file $DEST"
fi
```

Appendix B: gather_on_test.sh

Sample script to gather coverage data files on the test machine (see 6b):

```
#!/bin/bash -e

DEST=$1
GCDA=/sys/kernel/debug/gcov

if [ -z "$DEST" ] ; then
  echo "Usage: $0 <output.tar.gz>" >&2
  exit 1
fi

TEMPDIR=$(mktemp -d)
echo Collecting data..
find $GCDA -type d -exec mkdir -p $TEMPDIR/\{\} \;
find $GCDA -name '*.gcda' -exec sh -c 'cat < $0 > '$TEMPDIR'/$0' {} \;
find $GCDA -name '*.gcno' -exec sh -c 'cp -d $0 '$TEMPDIR'/$0' {} \;
tar czf $DEST -C $TEMPDIR sys
rm -rf $TEMPDIR

echo "$DEST successfully created, copy to build system and unpack with:"
echo "  tar xfz $DEST"
```

CHAPTER

FIVE

THE KERNEL ADDRESS SANITIZER (KASAN)

Overview

KernelAddressSANitizer (KASAN) is a dynamic memory error detector. It provides a fast and comprehensive solution for finding use-after-free and out-of-bounds bugs.

KASAN uses compile-time instrumentation for checking every memory access, therefore you will need a GCC version 4.9.2 or later. GCC 5.0 or later is required for detection of out-of-bounds accesses to stack or global variables.

Currently KASAN is supported only for the x86_64 and arm64 architectures.

Usage

To enable KASAN configure kernel with:

```
CONFIG_KASAN = y
```

and choose between CONFIG_KASAN_OUTLINE and CONFIG_KASAN_INLINE. Outline and inline are compiler instrumentation types. The former produces smaller binary the latter is 1.1 - 2 times faster. Inline instrumentation requires a GCC version 5.0 or later.

KASAN works with both SLUB and SLAB memory allocators. For better bug detection and nicer reporting, enable CONFIG_STACKTRACE.

To disable instrumentation for specific files or directories, add a line similar to the following to the respective kernel Makefile:

- For a single file (e.g. main.o):

  ```
  KASAN_SANITIZE_main.o := n
  ```

- For all files in one directory:

  ```
  KASAN_SANITIZE := n
  ```

Error reports

A typical out of bounds access report looks like this:

```
==================================================================
BUG: AddressSanitizer: out of bounds access in kmalloc_oob_right+0x65/0x75 [test_kasan] at addr␣
 ↪ffff8800693bc5d3
Write of size 1 by task modprobe/1689
==============================================================================
BUG kmalloc-128 (Not tainted): kasan error
```

```
---------------------------------------------------------------------

Disabling lock debugging due to kernel taint
INFO: Allocated in kmalloc_oob_right+0x3d/0x75 [test_kasan] age=0 cpu=0 pid=1689
 __slab_alloc+0x4b4/0x4f0
 kmem_cache_alloc_trace+0x10b/0x190
 kmalloc_oob_right+0x3d/0x75 [test_kasan]
 init_module+0x9/0x47 [test_kasan]
 do_one_initcall+0x99/0x200
 load_module+0x2cb3/0x3b20
 SyS_finit_module+0x76/0x80
 system_call_fastpath+0x12/0x17
INFO: Slab 0xffffea0001a4ef00 objects=17 used=7 fp=0xffff8800693bd728 flags=0x100000000004080
INFO: Object 0xffff8800693bc558 @offset=1368 fp=0xffff8800693bc720

Bytes b4 ffff8800693bc548: 00 00 00 00 00 00 00 00 5a 5a 5a 5a 5a 5a 5a 5a  ........ZZZZZZZZ
Object ffff8800693bc558: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object ffff8800693bc568: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object ffff8800693bc578: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object ffff8800693bc588: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object ffff8800693bc598: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object ffff8800693bc5a8: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object ffff8800693bc5b8: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object ffff8800693bc5c8: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b a5  kkkkkkkkkkkkkkk.
Redzone ffff8800693bc5d8: cc cc cc cc cc cc cc cc                          ........
Padding ffff8800693bc718: 5a 5a 5a 5a 5a 5a 5a 5a                          ZZZZZZZZ
CPU: 0 PID: 1689 Comm: modprobe Tainted: G    B          3.18.0-rc1-mm1+ #98
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS rel-1.7.5-0-ge51488c-20140602_
  ↪164612-nilsson.home.kraxel.org 04/01/2014
 ffff8800693bc000 0000000000000000 ffff8800693bc558 ffff88006923bb78
 ffffffff81cc68ae 00000000000000f3 ffff88006d407600 ffff88006923bba8
 ffffffff811fd848 ffff88006d407600 ffffea0001a4ef00 ffff8800693bc558
Call Trace:
 [<ffffffff81cc68ae>] dump_stack+0x46/0x58
 [<ffffffff811fd848>] print_trailer+0xf8/0x160
 [<ffffffffa00026a7>] ? kmem_cache_oob+0xc3/0xc3 [test_kasan]
 [<ffffffff811ff0f5>] object_err+0x35/0x40
 [<ffffffffa0002065>] ? kmalloc_oob_right+0x65/0x75 [test_kasan]
 [<ffffffff8120b9fa>] kasan_report_error+0x38a/0x3f0
 [<ffffffff8120a79f>] ? kasan_poison_shadow+0x2f/0x40
 [<ffffffff8120b344>] ? kasan_unpoison_shadow+0x14/0x40
 [<ffffffff8120a79f>] ? kasan_poison_shadow+0x2f/0x40
 [<ffffffffa00026a7>] ? kmem_cache_oob+0xc3/0xc3 [test_kasan]
 [<ffffffff8120a995>] __asan_store1+0x75/0xb0
 [<ffffffffa0002601>] ? kmem_cache_oob+0x1d/0xc3 [test_kasan]
 [<ffffffffa0002065>] ? kmalloc_oob_right+0x65/0x75 [test_kasan]
 [<ffffffffa0002065>] kmalloc_oob_right+0x65/0x75 [test_kasan]
 [<ffffffffa00026b0>] init_module+0x9/0x47 [test_kasan]
 [<ffffffff810002d9>] do_one_initcall+0x99/0x200
 [<ffffffff811e4e5c>] ? __vunmap+0xec/0x160
 [<ffffffff81114f63>] load_module+0x2cb3/0x3b20
 [<ffffffff8110fd70>] ? m_show+0x240/0x240
 [<ffffffff81115f06>] SyS_finit_module+0x76/0x80
 [<ffffffff81cd3129>] system_call_fastpath+0x12/0x17
Memory state around the buggy address:
 ffff8800693bc300: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
 ffff8800693bc380: fc fc 00 00 00 00 00 00 00 00 00 00 00 00 00 fc
 ffff8800693bc400: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
 ffff8800693bc480: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
 ffff8800693bc500: fc fc fc fc fc fc fc fc fc fc fc 00 00 00 00 00
>ffff8800693bc580: 00 00 00 00 00 00 00 00 00 00 03 fc fc fc fc fc
                                                 ^
 ffff8800693bc600: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
```

```
 ffff8800693bc680: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
 ffff8800693bc700: fc fc fc fc fb fb fb fb fb fb fb fb fb fb fb fb
 ffff8800693bc780: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
 ffff8800693bc800: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
==================================================================
```

The header of the report discribe what kind of bug happened and what kind of access caused it. It's followed by the description of the accessed slub object (see 'SLUB Debug output' section in Documentation/vm/slub.txt for details) and the description of the accessed memory page.

In the last section the report shows memory state around the accessed address. Reading this part requires some understanding of how KASAN works.

The state of each 8 aligned bytes of memory is encoded in one shadow byte. Those 8 bytes can be accessible, partially accessible, freed or be a redzone. We use the following encoding for each shadow byte: 0 means that all 8 bytes of the corresponding memory region are accessible; number N (1 <= N <= 7) means that the first N bytes are accessible, and other (8 - N) bytes are not; any negative value indicates that the entire 8-byte word is inaccessible. We use different negative values to distinguish between different kinds of inaccessible memory like redzones or freed memory (see mm/kasan/kasan.h).

In the report above the arrows point to the shadow byte 03, which means that the accessed address is partially accessible.

Implementation details

From a high level, our approach to memory error detection is similar to that of kmemcheck: use shadow memory to record whether each byte of memory is safe to access, and use compile-time instrumentation to check shadow memory on each memory access.

AddressSanitizer dedicates 1/8 of kernel memory to its shadow memory (e.g. 16TB to cover 128TB on x86_64) and uses direct mapping with a scale and offset to translate a memory address to its corresponding shadow address.

Here is the function which translates an address to its corresponding shadow address:

```
static inline void *kasan_mem_to_shadow(const void *addr)
{
    return ((unsigned long)addr >> KASAN_SHADOW_SCALE_SHIFT)
            + KASAN_SHADOW_OFFSET;
}
```

where `KASAN_SHADOW_SCALE_SHIFT = 3`.

Compile-time instrumentation used for checking memory accesses. Compiler inserts function calls (__asan_load*(addr), __asan_store*(addr)) before each memory access of size 1, 2, 4, 8 or 16. These functions check whether memory access is valid or not by checking corresponding shadow memory.

GCC 5.0 has possibility to perform inline instrumentation. Instead of making function calls GCC directly inserts the code to check the shadow memory. This option significantly enlarges kernel but it gives x1.1-x2 performance boost over outline instrumented kernel.

CHAPTER

SIX

THE UNDEFINED BEHAVIOR SANITIZER - UBSAN

UBSAN is a runtime undefined behaviour checker.

UBSAN uses compile-time instrumentation to catch undefined behavior (UB). Compiler inserts code that perform certain kinds of checks before operations that may cause UB. If check fails (i.e. UB detected) __ubsan_handle_* function called to print error message.

GCC has that feature since 4.9.x [1] (see `-fsanitize=undefined` option and its suboptions). GCC 5.x has more checkers implemented [2].

Report example

```
================================================================================
UBSAN: Undefined behaviour in ../include/linux/bitops.h:110:33
shift exponent 32 is to large for 32-bit type 'unsigned int'
CPU: 0 PID: 0 Comm: swapper Not tainted 4.4.0-rc1+ #26
 0000000000000000 ffffffff82403cc8 ffffffff815e6cd6 0000000000000001
 ffffffff82403cf8 ffffffff82403ce0 ffffffff8163a5ed 0000000000000020
 ffffffff82403d78 ffffffff8163ac2b ffffffff815f0001 0000000000000002
Call Trace:
 [<ffffffff815e6cd6>] dump_stack+0x45/0x5f
 [<ffffffff8163a5ed>] ubsan_epilogue+0xd/0x40
 [<ffffffff8163ac2b>] __ubsan_handle_shift_out_of_bounds+0xeb/0x130
 [<ffffffff815f0001>] ? radix_tree_gang_lookup_slot+0x51/0x150
 [<ffffffff8173c586>] _mix_pool_bytes+0x1e6/0x480
 [<ffffffff83105653>] ? dmi_walk_early+0x48/0x5c
 [<ffffffff8173c881>] add_device_randomness+0x61/0x130
 [<ffffffff83105b35>] ? dmi_save_one_device+0xaa/0xaa
 [<ffffffff83105653>] dmi_walk_early+0x48/0x5c
 [<ffffffff831066ae>] dmi_scan_machine+0x278/0x4b4
 [<ffffffff8111d58a>] ? vprintk_default+0x1a/0x20
 [<ffffffff830ad120>] ? early_idt_handler_array+0x120/0x120
 [<ffffffff830b2240>] setup_arch+0x405/0xc2c
 [<ffffffff830ad120>] ? early_idt_handler_array+0x120/0x120
 [<ffffffff830ae053>] start_kernel+0x83/0x49a
 [<ffffffff830ad120>] ? early_idt_handler_array+0x120/0x120
 [<ffffffff830ad386>] x86_64_start_reservations+0x2a/0x2c
 [<ffffffff830ad4f3>] x86_64_start_kernel+0x16b/0x17a
================================================================================
```

Usage

To enable UBSAN configure kernel with:

```
CONFIG_UBSAN=y
```

and to check the entire kernel:

```
CONFIG_UBSAN_SANITIZE_ALL=y
```

To enable instrumentation for specific files or directories, add a line similar to the following to the respective kernel Makefile:

- For a single file (e.g. main.o):

  ```
  UBSAN_SANITIZE_main.o := y
  ```

- For all files in one directory:

  ```
  UBSAN_SANITIZE := y
  ```

To exclude files from being instrumented even if CONFIG_UBSAN_SANITIZE_ALL=y, use:

```
UBSAN_SANITIZE_main.o := n
```

and:

```
UBSAN_SANITIZE := n
```

Detection of unaligned accesses controlled through the separate option - CONFIG_UBSAN_ALIGNMENT. It's off by default on architectures that support unaligned accesses (CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS=y). One could still enable it in config, just note that it will produce a lot of UBSAN reports.

References

CHAPTER

SEVEN

KERNEL MEMORY LEAK DETECTOR

Kmemleak provides a way of detecting possible kernel memory leaks in a way similar to a tracing garbage collector (https://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29#Tracing_garbage_collectors), with the difference that the orphan objects are not freed but only reported via /sys/kernel/debug/kmemleak. A similar method is used by the Valgrind tool (`memcheck --leak-check`) to detect the memory leaks in user-space applications. Kmemleak is supported on x86, arm, powerpc, sparc, sh, microblaze, ppc, mips, s390, metag and tile.

Usage

CONFIG_DEBUG_KMEMLEAK in "Kernel hacking" has to be enabled. A kernel thread scans the memory every 10 minutes (by default) and prints the number of new unreferenced objects found. To display the details of all the possible memory leaks:

```
# mount -t debugfs nodev /sys/kernel/debug/
# cat /sys/kernel/debug/kmemleak
```

To trigger an intermediate memory scan:

```
# echo scan > /sys/kernel/debug/kmemleak
```

To clear the list of all current possible memory leaks:

```
# echo clear > /sys/kernel/debug/kmemleak
```

New leaks will then come up upon reading `/sys/kernel/debug/kmemleak` again.

Note that the orphan objects are listed in the order they were allocated and one object at the beginning of the list may cause other subsequent objects to be reported as orphan.

Memory scanning parameters can be modified at run-time by writing to the `/sys/kernel/debug/kmemleak` file. The following parameters are supported:

- **off** disable kmemleak (irreversible)
- **stack=on** enable the task stacks scanning (default)
- **stack=off** disable the tasks stacks scanning
- **scan=on** start the automatic memory scanning thread (default)
- **scan=off** stop the automatic memory scanning thread
- **scan=<secs>** set the automatic memory scanning period in seconds (default 600, 0 to stop the automatic scanning)
- **scan** trigger a memory scan
- **clear** clear list of current memory leak suspects, done by marking all current reported unreferenced objects grey, or free all kmemleak objects if kmemleak has been disabled.

- **dump=<addr>** dump information about the object found at <addr>

Kmemleak can also be disabled at boot-time by passing `kmemleak=off` on the kernel command line.

Memory may be allocated or freed before kmemleak is initialised and these actions are stored in an early log buffer. The size of this buffer is configured via the CONFIG_DEBUG_KMEMLEAK_EARLY_LOG_SIZE option.

If CONFIG_DEBUG_KMEMLEAK_DEFAULT_OFF are enabled, the kmemleak is disabled by default. Passing `kmemleak=on` on the kernel command line enables the function.

Basic Algorithm

The memory allocations via `kmalloc()`, `vmalloc()`, `kmem_cache_alloc()` and friends are traced and the pointers, together with additional information like size and stack trace, are stored in a rbtree. The corresponding freeing function calls are tracked and the pointers removed from the kmemleak data structures.

An allocated block of memory is considered orphan if no pointer to its start address or to any location inside the block can be found by scanning the memory (including saved registers). This means that there might be no way for the kernel to pass the address of the allocated block to a freeing function and therefore the block is considered a memory leak.

The scanning algorithm steps:

1. mark all objects as white (remaining white objects will later be considered orphan)
2. scan the memory starting with the data section and stacks, checking the values against the addresses stored in the rbtree. If a pointer to a white object is found, the object is added to the gray list
3. scan the gray objects for matching addresses (some white objects can become gray and added at the end of the gray list) until the gray set is finished
4. the remaining white objects are considered orphan and reported via /sys/kernel/debug/kmemleak

Some allocated memory blocks have pointers stored in the kernel's internal data structures and they cannot be detected as orphans. To avoid this, kmemleak can also store the number of values pointing to an address inside the block address range that need to be found so that the block is not considered a leak. One example is __vmalloc().

Testing specific sections with kmemleak

Upon initial bootup your /sys/kernel/debug/kmemleak output page may be quite extensive. This can also be the case if you have very buggy code when doing development. To work around these situations you can use the 'clear' command to clear all reported unreferenced objects from the /sys/kernel/debug/kmemleak output. By issuing a 'scan' after a 'clear' you can find new unreferenced objects; this should help with testing specific sections of code.

To test a critical section on demand with a clean kmemleak do:

```
# echo clear > /sys/kernel/debug/kmemleak
... test your kernel or modules ...
# echo scan > /sys/kernel/debug/kmemleak
```

Then as usual to get your report with:

```
# cat /sys/kernel/debug/kmemleak
```

Freeing kmemleak internal objects

To allow access to previously found memory leaks after kmemleak has been disabled by the user or due to an fatal error, internal kmemleak objects won't be freed when kmemleak is disabled, and those objects may occupy a large part of physical memory.

In this situation, you may reclaim memory with:

```
# echo clear > /sys/kernel/debug/kmemleak
```

Kmemleak API

See the include/linux/kmemleak.h header for the functions prototype.

- `kmemleak_init` - initialize kmemleak
- `kmemleak_alloc` - notify of a memory block allocation
- `kmemleak_alloc_percpu` - notify of a percpu memory block allocation
- `kmemleak_vmalloc` - notify of a vmalloc() memory allocation
- `kmemleak_free` - notify of a memory block freeing
- `kmemleak_free_part` - notify of a partial memory block freeing
- `kmemleak_free_percpu` - notify of a percpu memory block freeing
- `kmemleak_update_trace` - update object allocation stack trace
- `kmemleak_not_leak` - mark an object as not a leak
- `kmemleak_ignore` - do not scan or report an object as leak
- `kmemleak_scan_area` - add scan areas inside a memory block
- `kmemleak_no_scan` - do not scan a memory block
- `kmemleak_erase` - erase an old value in a pointer variable
- `kmemleak_alloc_recursive` - as kmemleak_alloc but checks the recursiveness
- `kmemleak_free_recursive` - as kmemleak_free but checks the recursiveness

The following functions take a physical address as the object pointer and only perform the corresponding action if the address has a lowmem mapping:

- `kmemleak_alloc_phys`
- `kmemleak_free_part_phys`
- `kmemleak_not_leak_phys`
- `kmemleak_ignore_phys`

Dealing with false positives/negatives

The false negatives are real memory leaks (orphan objects) but not reported by kmemleak because values found during the memory scanning point to such objects. To reduce the number of false negatives, kmemleak provides the kmemleak_ignore, kmemleak_scan_area, kmemleak_no_scan and kmemleak_erase functions (see above). The task stacks also increase the amount of false negatives and their scanning is not enabled by default.

The false positives are objects wrongly reported as being memory leaks (orphan). For objects known not to be leaks, kmemleak provides the kmemleak_not_leak function. The kmemleak_ignore could also be used if the memory block is known not to contain other pointers and it will no longer be scanned.

Some of the reported leaks are only transient, especially on SMP systems, because of pointers temporarily stored in CPU registers or stacks. Kmemleak defines MSECS_MIN_AGE (defaulting to 1000) representing the minimum age of an object to be reported as a memory leak.

Limitations and Drawbacks

The main drawback is the reduced performance of memory allocation and freeing. To avoid other penalties, the memory scanning is only performed when the /sys/kernel/debug/kmemleak file is read. Anyway, this tool is intended for debugging purposes where the performance might not be the most important requirement.

To keep the algorithm simple, kmemleak scans for values pointing to any address inside a block's address range. This may lead to an increased number of false negatives. However, it is likely that a real memory leak will eventually become visible.

Another source of false negatives is the data stored in non-pointer values. In a future version, kmemleak could only scan the pointer members in the allocated structures. This feature would solve many of the false negative cases described above.

The tool can report false positives. These are cases where an allocated block doesn't need to be freed (some cases in the init_call functions), the pointer is calculated by other methods than the usual container_of macro or the pointer is stored in a location not scanned by kmemleak.

Page allocations and ioremap are not tracked.

CHAPTER

EIGHT

GETTING STARTED WITH KMEMCHECK

Vegard Nossum <vegardno@ifi.uio.no>

Introduction

kmemcheck is a debugging feature for the Linux Kernel. More specifically, it is a dynamic checker that detects and warns about some uses of uninitialized memory.

Userspace programmers might be familiar with Valgrind's memcheck. The main difference between memcheck and kmemcheck is that memcheck works for userspace programs only, and kmemcheck works for the kernel only. The implementations are of course vastly different. Because of this, kmemcheck is not as accurate as memcheck, but it turns out to be good enough in practice to discover real programmer errors that the compiler is not able to find through static analysis.

Enabling kmemcheck on a kernel will probably slow it down to the extent that the machine will not be usable for normal workloads such as e.g. an interactive desktop. kmemcheck will also cause the kernel to use about twice as much memory as normal. For this reason, kmemcheck is strictly a debugging feature.

Downloading

As of version 2.6.31-rc1, kmemcheck is included in the mainline kernel.

Configuring and compiling

kmemcheck only works for the x86 (both 32- and 64-bit) platform. A number of configuration variables must have specific settings in order for the kmemcheck menu to even appear in "menuconfig". These are:

- **CONFIG_CC_OPTIMIZE_FOR_SIZE=n** This option is located under "General setup" / "Optimize for size".

  Without this, gcc will use certain optimizations that usually lead to false positive warnings from kmemcheck. An example of this is a 16-bit field in a struct, where gcc may load 32 bits, then discard the upper 16 bits. kmemcheck sees only the 32-bit load, and may trigger a warning for the upper 16 bits (if they're uninitialized).

- **CONFIG_SLAB=y or CONFIG_SLUB=y** This option is located under "General setup" / "Choose SLAB allocator".

- **CONFIG_FUNCTION_TRACER=n** This option is located under "Kernel hacking" / "Tracers" / "Kernel Function Tracer"

  When function tracing is compiled in, gcc emits a call to another function at the beginning of every function. This means that when the page fault handler is called, the ftrace framework will be called before kmemcheck has had a chance to handle the fault. If ftrace then modifies memory that was tracked by kmemcheck, the result is an endless recursive page fault.

- **CONFIG_DEBUG_PAGEALLOC=n** This option is located under “Kernel hacking” / “Memory Debugging” / “Debug page memory allocations”.

In addition, I highly recommend turning on CONFIG_DEBUG_INFO=y. This is also located under “Kernel hacking”. With this, you will be able to get line number information from the kmemcheck warnings, which is extremely valuable in debugging a problem. This option is not mandatory, however, because it slows down the compilation process and produces a much bigger kernel image.

Now the kmemcheck menu should be visible (under “Kernel hacking” / “Memory Debugging” / “kmemcheck: trap use of uninitialized memory”). Here follows a description of the kmemcheck configuration variables:

- **CONFIG_KMEMCHECK** This must be enabled in order to use kmemcheck at all...
- **CONFIG_KMEMCHECK_``[``DISABLED | ENABLED | ONESHOT]‘‘_BY_DEFAULT‘‘** This option controls the status of kmemcheck at boot-time. “Enabled” will enable kmemcheck right from the start, “disabled” will boot the kernel as normal (but with the kmemcheck code compiled in, so it can be enabled at run-time after the kernel has booted), and “one-shot” is a special mode which will turn kmemcheck off automatically after detecting the first use of uninitialized memory.

  If you are using kmemcheck to actively debug a problem, then you probably want to choose “enabled” here.

  The one-shot mode is mostly useful in automated test setups because it can prevent floods of warnings and increase the chances of the machine surviving in case something is really wrong. In other cases, the one- shot mode could actually be counter-productive because it would turn itself off at the very first error – in the case of a false positive too – and this would come in the way of debugging the specific problem you were interested in.

  If you would like to use your kernel as normal, but with a chance to enable kmemcheck in case of some problem, it might be a good idea to choose “disabled” here. When kmemcheck is disabled, most of the run- time overhead is not incurred, and the kernel will be almost as fast as normal.

- **CONFIG_KMEMCHECK_QUEUE_SIZE** Select the maximum number of error reports to store in an internal (fixed-size) buffer. Since errors can occur virtually anywhere and in any context, we need a temporary storage area which is guaranteed not to generate any other page faults when accessed. The queue will be emptied as soon as a tasklet may be scheduled. If the queue is full, new error reports will be lost.

  The default value of 64 is probably fine. If some code produces more than 64 errors within an irqs-off section, then the code is likely to produce many, many more, too, and these additional reports seldom give any more information (the first report is usually the most valuable anyway).

  This number might have to be adjusted if you are not using serial console or similar to capture the kernel log. If you are using the “dmesg” command to save the log, then getting a lot of kmemcheck warnings might overflow the kernel log itself, and the earlier reports will get lost in that way instead. Try setting this to 10 or so on such a setup.

- **CONFIG_KMEMCHECK_SHADOW_COPY_SHIFT** Select the number of shadow bytes to save along with each entry of the error-report queue. These bytes indicate what parts of an allocation are initialized, uninitialized, etc. and will be displayed when an error is detected to help the debugging of a particular problem.

  The number entered here is actually the logarithm of the number of bytes that will be saved. So if you pick for example 5 here, kmemcheck will save 2^5 = 32 bytes.

  The default value should be fine for debugging most problems. It also fits nicely within 80 columns.

- **CONFIG_KMEMCHECK_PARTIAL_OK** This option (when enabled) works around certain GCC optimizations that produce 32-bit reads from 16-bit variables where the upper 16 bits are thrown away afterwards.

  The default value (enabled) is recommended. This may of course hide some real errors, but disabling it would probably produce a lot of false positives.

- **CONFIG_KMEMCHECK_BITOPS_OK** This option silences warnings that would be generated for bit-field accesses where not all the bits are initialized at the same time. This may also hide some real bugs.

  This option is probably obsolete, or it should be replaced with the kmemcheck-/bitfield-annotations for the code in question. The default value is therefore fine.

Now compile the kernel as usual.

How to use

Booting

First some information about the command-line options. There is only one option specific to kmemcheck, and this is called "kmemcheck". It can be used to override the default mode as chosen by the CONFIG_KMEMCHECK_*_BY_DEFAULT option. Its possible settings are:

- kmemcheck=0 (disabled)
- kmemcheck=1 (enabled)
- kmemcheck=2 (one-shot mode)

If SLUB debugging has been enabled in the kernel, it may take precedence over kmemcheck in such a way that the slab caches which are under SLUB debugging will not be tracked by kmemcheck. In order to ensure that this doesn't happen (even though it shouldn't by default), use SLUB's boot option slub_debug, like this: slub_debug=-

In fact, this option may also be used for fine-grained control over SLUB vs. kmemcheck. For example, if the command line includes kmemcheck=1 slub_debug=,dentry, then SLUB debugging will be used only for the "dentry" slab cache, and with kmemcheck tracking all the other caches. This is advanced usage, however, and is not generally recommended.

Run-time enable/disable

When the kernel has booted, it is possible to enable or disable kmemcheck at run-time. WARNING: This feature is still experimental and may cause false positive warnings to appear. Therefore, try not to use this. If you find that it doesn't work properly (e.g. you see an unreasonable amount of warnings), I will be happy to take bug reports.

Use the file /proc/sys/kernel/kmemcheck for this purpose, e.g.:

```
$ echo 0 > /proc/sys/kernel/kmemcheck # disables kmemcheck
```

The numbers are the same as for the kmemcheck= command-line option.

Debugging

A typical report will look something like this:

```
WARNING: kmemcheck: Caught 32-bit read from uninitialized memory (ffff88003e4a2024)
80000000000000000000000000000000000000088ffff0000000000000000
 i i i i u u u u i i i i i i i i u u u u u u u u u u u u u u u u
         ^

Pid: 1856, comm: ntpdate Not tainted 2.6.29-rc5 #264 945P-A
RIP: 0010:[<ffffffff8104ede8>]  [<ffffffff8104ede8>] __dequeue_signal+0xc8/0x190
RSP: 0018:ffff88003cdf7d98  EFLAGS: 00210002
RAX: 0000000000000030 RBX: ffff88003d4ea968 RCX: 0000000000000009
```

```
RDX: ffff88003e5d6018 RSI: ffff88003e5d6024 RDI: ffff88003cdf7e84
RBP: ffff88003cdf7db8 R08: ffff88003e5d6000 R09: 0000000000000000
R10: 0000000000000080 R11: 0000000000000000 R12: 000000000000000e
R13: ffff88003cdf7e78 R14: ffff88003d530710 R15: ffff88003d5a98c8
FS:  0000000000000000(0000) GS:ffff880001982000(0063) knlGS:00000
CS:  0010 DS: 002b ES: 002b CR0: 0000000080050033
CR2: ffff88003f806ea0 CR3: 000000003c036000 CR4: 00000000000006a0
DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
DR3: 0000000000000000 DR6: 00000000ffff4ff0 DR7: 0000000000000400
 [<ffffffff8104f04e>] dequeue_signal+0x8e/0x170
 [<ffffffff81050bd8>] get_signal_to_deliver+0x98/0x390
 [<ffffffff8100b87d>] do_notify_resume+0xad/0x7d0
 [<ffffffff8100c7b5>] int_signal+0x12/0x17
 [<ffffffffffffffff>] 0xffffffffffffffff
```

The single most valuable information in this report is the RIP (or EIP on 32- bit) value. This will help us pinpoint exactly which instruction that caused the warning.

If your kernel was compiled with `CONFIG_DEBUG_INFO=y`, then all we have to do is give this address to the addr2line program, like this:

```
$ addr2line -e vmlinux -i ffffffff8104ede8
arch/x86/include/asm/string_64.h:12
include/asm-generic/siginfo.h:287
kernel/signal.c:380
kernel/signal.c:410
```

The "`-e vmlinux`" tells addr2line which file to look in. **IMPORTANT:** This must be the vmlinux of the kernel that produced the warning in the first place! If not, the line number information will almost certainly be wrong.

The "`-i`" tells addr2line to also print the line numbers of inlined functions. In this case, the flag was very important, because otherwise, it would only have printed the first line, which is just a call to `memcpy()`, which could be called from a thousand places in the kernel, and is therefore not very useful. These inlined functions would not show up in the stack trace above, simply because the kernel doesn't load the extra debugging information. This technique can of course be used with ordinary kernel oopses as well.

In this case, it's the caller of `memcpy()` that is interesting, and it can be found in `include/asm-generic/siginfo.h`, line 287:

```
281 static inline void copy_siginfo(struct siginfo *to, struct siginfo *from)
282 {
283         if (from->si_code < 0)
284                 memcpy(to, from, sizeof(*to));
285         else
286                 /* _sigchld is currently the largest know union member */
287                 memcpy(to, from, __ARCH_SI_PREAMBLE_SIZE + sizeof(from->_sifields._
  ↪sigchld));
288 }
```

Since this was a read (kmemcheck usually warns about reads only, though it can warn about writes to unallocated or freed memory as well), it was probably the "from" argument which contained some uninitialized bytes. Following the chain of calls, we move upwards to see where "from" was allocated or initialized, `kernel/signal.c`, line 380:

```
359 static void collect_signal(int sig, struct sigpending *list, siginfo_t *info)
360 {
...
367         list_for_each_entry(q, &list->list, list) {
368                 if (q->info.si_signo == sig) {
369                         if (first)
370                                 goto still_pending;
```

```
371                         first = q;
...
377         if (first) {
378 still_pending:
379                 list_del_init(&first->list);
380                 copy_siginfo(info, &first->info);
381                 __sigqueue_free(first);
...
392         }
393 }
```

Here, it is `&first->info` that is being passed on to `copy_siginfo()`. The variable `first` was found on a list – passed in as the second argument to `collect_signal()`. We continue our journey through the stack, to figure out where the item on "list" was allocated or initialized. We move to line 410:

```
395 static int __dequeue_signal(struct sigpending *pending, sigset_t *mask,
396                         siginfo_t *info)
397 {
...
410                 collect_signal(sig, pending, info);
...
414 }
```

Now we need to follow the `pending` pointer, since that is being passed on to `collect_signal()` as `list`. At this point, we've run out of lines from the "addr2line" output. Not to worry, we just paste the next addresses from the kmemcheck stack dump, i.e.:

```
[<ffffffff8104f04e>] dequeue_signal+0x8e/0x170
[<ffffffff81050bd8>] get_signal_to_deliver+0x98/0x390
[<ffffffff8100b87d>] do_notify_resume+0xad/0x7d0
[<ffffffff8100c7b5>] int_signal+0x12/0x17

   $ addr2line -e vmlinux -i ffffffff8104f04e ffffffff81050bd8 \
           ffffffff8100b87d ffffffff8100c7b5
   kernel/signal.c:446
   kernel/signal.c:1806
   arch/x86/kernel/signal.c:805
   arch/x86/kernel/signal.c:871
   arch/x86/kernel/entry_64.S:694
```

Remember that since these addresses were found on the stack and not as the RIP value, they actually point to the _next_ instruction (they are return addresses). This becomes obvious when we look at the code for line 446:

```
422 int dequeue_signal(struct task_struct *tsk, sigset_t *mask, siginfo_t *info)
423 {
...
431                 signr = __dequeue_signal(&tsk->signal->shared_pending,
432                                          mask, info);
433                 /*
434                  * itimer signal ?
435                  *
436                  * itimers are process shared and we restart periodic
437                  * itimers in the signal delivery path to prevent DoS
438                  * attacks in the high resolution timer case. This is
439                  * compliant with the old way of self restarting
440                  * itimers, as the SIGALRM is a legacy signal and only
441                  * queued once. Changing the restart behaviour to
442                  * restart the timer in the signal dequeue path is
443                  * reducing the timer noise on heavy loaded !highres
444                  * systems too.
445                  */
```

```
446                 if (unlikely(signr == SIGALRM)) {
...
489 }
```

So instead of looking at 446, we should be looking at 431, which is the line that executes just before 446. Here we see that what we are looking for is `&tsk->signal->shared_pending`.

Our next task is now to figure out which function that puts items on this `shared_pending` list. A crude, but efficient tool, is `git grep`:

```
$ git grep -n 'shared_pending' kernel/
...
kernel/signal.c:828:    pending = group ? &t->signal->shared_pending : &t->pending;
kernel/signal.c:1339:   pending = group ? &t->signal->shared_pending : &t->pending;
...
```

There were more results, but none of them were related to list operations, and these were the only assignments. We inspect the line numbers more closely and find that this is indeed where items are being added to the list:

```
816 static int send_signal(int sig, struct siginfo *info, struct task_struct *t,
817                         int group)
818 {
...
828         pending = group ? &t->signal->shared_pending : &t->pending;
...
851         q = __sigqueue_alloc(t, GFP_ATOMIC, (sig < SIGRTMIN &&
852                                              (is_si_special(info) ||
853                                               info->si_code >= 0)));
854         if (q) {
855                 list_add_tail(&q->list, &pending->list);
...
890 }
```

and:

```
1309 int send_sigqueue(struct sigqueue *q, struct task_struct *t, int group)
1310 {
....
1339         pending = group ? &t->signal->shared_pending : &t->pending;
1340         list_add_tail(&q->list, &pending->list);
....
1347 }
```

In the first case, the list element we are looking for, `q`, is being returned from the function `__sigqueue_alloc()`, which looks like an allocation function. Let's take a look at it:

```
187 static struct sigqueue *__sigqueue_alloc(struct task_struct *t, gfp_t flags,
188                                           int override_rlimit)
189 {
190         struct sigqueue *q = NULL;
191         struct user_struct *user;
192
193         /*
194          * We won't get problems with the target's UID changing under us
195          * because changing it requires RCU be used, and if t != current, the
196          * caller must be holding the RCU readlock (by way of a spinlock) and
197          * we use RCU protection here
198          */
199         user = get_uid(__task_cred(t)->user);
200         atomic_inc(&user->sigpending);
201         if (override_rlimit ||
```

```
202             atomic_read(&user->sigpending) <=
203                     t->signal->rlim[RLIMIT_SIGPENDING].rlim_cur)
204                 q = kmem_cache_alloc(sigqueue_cachep, flags);
205         if (unlikely(q == NULL)) {
206                 atomic_dec(&user->sigpending);
207                 free_uid(user);
208         } else {
209                 INIT_LIST_HEAD(&q->list);
210                 q->flags = 0;
211                 q->user = user;
212         }
213
214         return q;
215 }
```

We see that this function initializes `q->list`, `q->flags`, and `q->user`. It seems that now is the time to look at the definition of `struct sigqueue`, e.g.:

```
14 struct sigqueue {
15         struct list_head list;
16         int flags;
17         siginfo_t info;
18         struct user_struct *user;
19 };
```

And, you might remember, it was a `memcpy()` on `&first->info` that caused the warning, so this makes perfect sense. It also seems reasonable to assume that it is the caller of `__sigqueue_alloc()` that has the responsibility of filling out (initializing) this member.

But just which fields of the struct were uninitialized? Let's look at kmemcheck's report again:

```
WARNING: kmemcheck: Caught 32-bit read from uninitialized memory (ffff88003e4a2024)
80000000000000000000000000000000000000088ffff0000000000000000
 i i i i u u u u i i i i i i i i u u u u u u u u u u u u u u u u
         ^
```

These first two lines are the memory dump of the memory object itself, and the shadow bytemap, respectively. The memory object itself is in this case `&first->info`. Just beware that the start of this dump is NOT the start of the object itself! The position of the caret (^) corresponds with the address of the read (ffff88003e4a2024).

The shadow bytemap dump legend is as follows:

- i: initialized
- u: uninitialized
- a: unallocated (memory has been allocated by the slab layer, but has not yet been handed off to anybody)
- f: freed (memory has been allocated by the slab layer, but has been freed by the previous owner)

In order to figure out where (relative to the start of the object) the uninitialized memory was located, we have to look at the disassembly. For that, we'll need the RIP address again:

```
RIP: 0010:[<ffffffff8104ede8>]  [<ffffffff8104ede8>] __dequeue_signal+0xc8/0x190

    $ objdump -d --no-show-raw-insn vmlinux | grep -C 8 ffffffff8104ede8:
    ffffffff8104edc8:       mov    %r8,0x8(%r8)
    ffffffff8104edcc:       test   %r10d,%r10d
    ffffffff8104edcf:       js     ffffffff8104ee88 <__dequeue_signal+0x168>
    ffffffff8104edd5:       mov    %rax,%rdx
    ffffffff8104edd8:       mov    $0xc,%ecx
    ffffffff8104eddd:       mov    %r13,%rdi
```

```
    ffffffff8104ede0:       mov    $0x30,%eax
    ffffffff8104ede5:       mov    %rdx,%rsi
    ffffffff8104ede8:       rep movsl %ds:(%rsi),%es:(%rdi)
    ffffffff8104edea:       test   $0x2,%al
    ffffffff8104edec:       je     ffffffff8104edf0 <__dequeue_signal+0xd0>
    ffffffff8104edee:       movsw  %ds:(%rsi),%es:(%rdi)
    ffffffff8104edf0:       test   $0x1,%al
    ffffffff8104edf2:       je     ffffffff8104edf5 <__dequeue_signal+0xd5>
    ffffffff8104edf4:       movsb  %ds:(%rsi),%es:(%rdi)
    ffffffff8104edf5:       mov    %r8,%rdi
    ffffffff8104edf8:       callq  ffffffff8104de60 <__sigqueue_free>
```

As expected, it's the "`rep movsl`" instruction from the `memcpy()` that causes the warning. We know about `REP MOVSL` that it uses the register RCX to count the number of remaining iterations. By taking a look at the register dump again (from the kmemcheck report), we can figure out how many bytes were left to copy:

```
RAX: 0000000000000030 RBX: ffff88003d4ea968 RCX: 0000000000000009
```

By looking at the disassembly, we also see that `%ecx` is being loaded with the value `$0xc` just before (ffffffff8104edd8), so we are very lucky. Keep in mind that this is the number of iterations, not bytes. And since this is a "long" operation, we need to multiply by 4 to get the number of bytes. So this means that the uninitialized value was encountered at 4 * (0xc - 0x9) = 12 bytes from the start of the object.

We can now try to figure out which field of the "`struct siginfo`" that was not initialized. This is the beginning of the struct:

```
40 typedef struct siginfo {
41         int si_signo;
42         int si_errno;
43         int si_code;
44
45         union {
..
92         } _sifields;
93 } siginfo_t;
```

On 64-bit, the int is 4 bytes long, so it must the union member that has not been initialized. We can verify this using gdb:

```
$ gdb vmlinux
...
(gdb) p &((struct siginfo *) 0)->_sifields
$1 = (union {...} *) 0x10
```

Actually, it seems that the union member is located at offset 0x10 – which means that gcc has inserted 4 bytes of padding between the members `si_code` and `_sifields`. We can now get a fuller picture of the memory dump:

```
         _----------------------------=> si_code
        /        _---------------------=> (padding)
       |        /        _------------=> _sifields(._kill._pid)
       |       |        /        _----=> _sifields(._kill._uid)
       |       |       |        /
-------|-------|-------|-------|
8000000000000000000000000000000000000088ffff0000000000000000
 i i i i u u u u i i i i i i i i u u u u u u u u u u u u u u u u
```

This allows us to realize another important fact: `si_code` contains the value 0x80. Remember that x86 is little endian, so the first 4 bytes "80000000" are really the number 0x00000080. With a bit of research, we find that this is actually the constant `SI_KERNEL` defined in `include/asm-generic/siginfo.h`:

```
144 #define SI_KERNEL       0x80            /* sent by the kernel from somewhere     */
```

This macro is used in exactly one place in the x86 kernel: In `send_signal()` in `kernel/signal.c`:

```
816 static int send_signal(int sig, struct siginfo *info, struct task_struct *t,
817                         int group)
818 {
...
828         pending = group ? &t->signal->shared_pending : &t->pending;
...
851         q = __sigqueue_alloc(t, GFP_ATOMIC, (sig < SIGRTMIN &&
852                                              (is_si_special(info) ||
853                                               info->si_code >= 0)));
854         if (q) {
855                 list_add_tail(&q->list, &pending->list);
856                 switch ((unsigned long) info) {
...
865                 case (unsigned long) SEND_SIG_PRIV:
866                         q->info.si_signo = sig;
867                         q->info.si_errno = 0;
868                         q->info.si_code = SI_KERNEL;
869                         q->info.si_pid = 0;
870                         q->info.si_uid = 0;
871                         break;
...
890 }
```

Not only does this match with the `.si_code` member, it also matches the place we found earlier when looking for where siginfo_t objects are enqueued on the `shared_pending` list.

So to sum up: It seems that it is the padding introduced by the compiler between two struct fields that is uninitialized, and this gets reported when we do a `memcpy()` on the struct. This means that we have identified a false positive warning.

Normally, kmemcheck will not report uninitialized accesses in `memcpy()` calls when both the source and destination addresses are tracked. (Instead, we copy the shadow bytemap as well). In this case, the destination address clearly was not tracked. We can dig a little deeper into the stack trace from above:

```
arch/x86/kernel/signal.c:805
arch/x86/kernel/signal.c:871
arch/x86/kernel/entry_64.S:694
```

And we clearly see that the destination siginfo object is located on the stack:

```
782 static void do_signal(struct pt_regs *regs)
783 {
784         struct k_sigaction ka;
785         siginfo_t info;
...
804         signr = get_signal_to_deliver(&info, &ka, regs, NULL);
...
854 }
```

And this `&info` is what eventually gets passed to `copy_siginfo()` as the destination argument.

Now, even though we didn't find an actual error here, the example is still a good one, because it shows how one would go about to find out what the report was all about.

Annotating false positives

There are a few different ways to make annotations in the source code that will keep kmemcheck from checking and reporting certain allocations. Here they are:

- **__GFP_NOTRACK_FALSE_POSITIVE** This flag can be passed to kmalloc() or kmem_cache_alloc() (therefore also to other functions that end up calling one of these) to indicate that the allocation should not be tracked because it would lead to a false positive report. This is a "big hammer" way of silencing kmemcheck; after all, even if the false positive pertains to particular field in a struct, for example, we will now lose the ability to find (real) errors in other parts of the same struct.

  Example:

  ```
  /* No warnings will ever trigger on accessing any part of x */
  x = kmalloc(sizeof *x, GFP_KERNEL | __GFP_NOTRACK_FALSE_POSITIVE);
  ```

- **kmemcheck_bitfield_begin(name)/kmemcheck_bitfield_end(name) and** kmemcheck_annotate_bitfield(ptr,name) The first two of these three macros can be used inside struct definitions to signal, respectively, the beginning and end of a bitfield. Additionally, this will assign the bitfield a name, which is given as an argument to the macros.

  Having used these markers, one can later use kmemcheck_annotate_bitfield() at the point of allocation, to indicate which parts of the allocation is part of a bitfield.

  Example:

  ```
  struct foo {
      int x;

      kmemcheck_bitfield_begin(flags);
      int flag_a:1;
      int flag_b:1;
      kmemcheck_bitfield_end(flags);

      int y;
  };

  struct foo *x = kmalloc(sizeof *x);

  /* No warnings will trigger on accessing the bitfield of x */
  kmemcheck_annotate_bitfield(x, flags);
  ```

  Note that kmemcheck_annotate_bitfield() can be used even before the return value of kmalloc() is checked – in other words, passing NULL as the first argument is legal (and will do nothing).

Reporting errors

As we have seen, kmemcheck will produce false positive reports. Therefore, it is not very wise to blindly post kmemcheck warnings to mailing lists and maintainers. Instead, I encourage maintainers and developers to find errors in their own code. If you get a warning, you can try to work around it, try to figure out if it's a real error or not, or simply ignore it. Most developers know their own code and will quickly and efficiently determine the root cause of a kmemcheck report. This is therefore also the most efficient way to work with kmemcheck.

That said, we (the kmemcheck maintainers) will always be on the lookout for false positives that we can annotate and silence. So whatever you find, please drop us a note privately! Kernel configs and steps to reproduce (if available) are of course a great help too.

Happy hacking!

Technical description

kmemcheck works by marking memory pages non-present. This means that whenever somebody attempts to access the page, a page fault is generated. The page fault handler notices that the page was in fact only hidden, and so it calls on the kmemcheck code to make further investigations.

When the investigations are completed, kmemcheck "shows" the page by marking it present (as it would be under normal circumstances). This way, the interrupted code can continue as usual.

But after the instruction has been executed, we should hide the page again, so that we can catch the next access too! Now kmemcheck makes use of a debugging feature of the processor, namely single-stepping. When the processor has finished the one instruction that generated the memory access, a debug exception is raised. From here, we simply hide the page again and continue execution, this time with the single-stepping feature turned off.

kmemcheck requires some assistance from the memory allocator in order to work. The memory allocator needs to

1. Tell kmemcheck about newly allocated pages and pages that are about to be freed. This allows kmemcheck to set up and tear down the shadow memory for the pages in question. The shadow memory stores the status of each byte in the allocation proper, e.g. whether it is initialized or uninitialized.
2. Tell kmemcheck which parts of memory should be marked uninitialized. There are actually a few more states, such as "not yet allocated" and "recently freed".

If a slab cache is set up using the SLAB_NOTRACK flag, it will never return memory that can take page faults because of kmemcheck.

If a slab cache is NOT set up using the SLAB_NOTRACK flag, callers can still request memory with the __GFP_NOTRACK or __GFP_NOTRACK_FALSE_POSITIVE flags. This does not prevent the page faults from occurring, however, but marks the object in question as being initialized so that no warnings will ever be produced for this object.

Currently, the SLAB and SLUB allocators are supported by kmemcheck.

CHAPTER

NINE

DEBUGGING KERNEL AND MODULES VIA GDB

The kernel debugger kgdb, hypervisors like QEMU or JTAG-based hardware interfaces allow to debug the Linux kernel and its modules during runtime using gdb. Gdb comes with a powerful scripting interface for python. The kernel provides a collection of helper scripts that can simplify typical kernel debugging steps. This is a short tutorial about how to enable and use them. It focuses on QEMU/KVM virtual machines as target, but the examples can be transferred to the other gdb stubs as well.

Requirements

- gdb 7.2+ (recommended: 7.4+) with python support enabled (typically true for distributions)

Setup

- Create a virtual Linux machine for QEMU/KVM (see www.linux-kvm.org and www.qemu.org for more details). For cross-development, http://landley.net/aboriginal/bin keeps a pool of machine images and toolchains that can be helpful to start from.
- Build the kernel with CONFIG_GDB_SCRIPTS enabled, but leave CONFIG_DEBUG_INFO_REDUCED off. If your architecture supports CONFIG_FRAME_POINTER, keep it enabled.
- Install that kernel on the guest, turn off KASLR if necessary by adding “nokaslr” to the kernel command line. Alternatively, QEMU allows to boot the kernel directly using -kernel, -append, -initrd command line switches. This is generally only useful if you do not depend on modules. See QEMU documentation for more details on this mode. In this case, you should build the kernel with CONFIG_RANDOMIZE_BASE disabled if the architecture supports KASLR.
- Enable the gdb stub of QEMU/KVM, either
  - at VM startup time by appending “-s” to the QEMU command line

  or
  - during runtime by issuing “gdbserver” from the QEMU monitor console
- cd /path/to/linux-build
- Start gdb: gdb vmlinux

  Note: Some distros may restrict auto-loading of gdb scripts to known safe directories. In case gdb reports to refuse loading vmlinux-gdb.py, add:

```
add-auto-load-safe-path /path/to/linux-build
```

  to ~/.gdbinit. See gdb help for more details.
- Attach to the booted guest:

```
(gdb) target remote :1234
```

Examples of using the Linux-provided gdb helpers

- Load module (and main kernel) symbols:

```
(gdb) lx-symbols
loading vmlinux
scanning for modules in /home/user/linux/build
loading @0xffffffffa0020000: /home/user/linux/build/net/netfilter/xt_tcpudp.ko
loading @0xffffffffa0016000: /home/user/linux/build/net/netfilter/xt_pkttype.ko
loading @0xffffffffa0002000: /home/user/linux/build/net/netfilter/xt_limit.ko
loading @0xffffffffa00ca000: /home/user/linux/build/net/packet/af_packet.ko
loading @0xffffffffa003c000: /home/user/linux/build/fs/fuse/fuse.ko
...
loading @0xffffffffa0000000: /home/user/linux/build/drivers/ata/ata_generic.ko
```

- Set a breakpoint on some not yet loaded module function, e.g.:

```
(gdb) b btrfs_init_sysfs
Function "btrfs_init_sysfs" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (btrfs_init_sysfs) pending.
```

- Continue the target:

```
(gdb) c
```

- Load the module on the target and watch the symbols being loaded as well as the breakpoint hit:

```
loading @0xffffffffa0034000: /home/user/linux/build/lib/libcrc32c.ko
loading @0xffffffffa0050000: /home/user/linux/build/lib/lzo/lzo_compress.ko
loading @0xffffffffa006e000: /home/user/linux/build/lib/zlib_deflate/zlib_deflate.ko
loading @0xffffffffa01b1000: /home/user/linux/build/fs/btrfs/btrfs.ko

Breakpoint 1, btrfs_init_sysfs () at /home/user/linux/fs/btrfs/sysfs.c:36
36              btrfs_kset = kset_create_and_add("btrfs", NULL, fs_kobj);
```

- Dump the log buffer of the target kernel:

```
(gdb) lx-dmesg
[     0.000000] Initializing cgroup subsys cpuset
[     0.000000] Initializing cgroup subsys cpu
[     0.000000] Linux version 3.8.0-rc4-dbg+ (...
[     0.000000] Command line: root=/dev/sda2 resume=/dev/sda1 vga=0x314
[     0.000000] e820: BIOS-provided physical RAM map:
[     0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff] usable
[     0.000000] BIOS-e820: [mem 0x000000000009fc00-0x000000000009ffff] reserved
....
```

- Examine fields of the current task struct:

```
(gdb) p $lx_current().pid
$1 = 4998
(gdb) p $lx_current().comm
$2 = "modprobe\000\000\000\000\000\000\000"
```

- Make use of the per-cpu function for the current or a specified CPU:

```
(gdb) p $lx_per_cpu("runqueues").nr_running
$3 = 1
(gdb) p $lx_per_cpu("runqueues", 2).nr_running
$4 = 0
```

- Dig into hrtimers using the container_of helper:

```
(gdb) set $next = $lx_per_cpu("hrtimer_bases").clock_base[0].active.next
(gdb) p *$container_of($next, "struct hrtimer", "node")
$5 = {
  node = {
    node = {
      __rb_parent_color = 18446612133355256072,
      rb_right = 0x0 <irq_stack_union>,
      rb_left = 0x0 <irq_stack_union>
    },
    expires = {
      tv64 = 1835268000000
    }
  },
  _softexpires = {
    tv64 = 1835268000000
  },
  function = 0xffffffff81078232 <tick_sched_timer>,
  base = 0xffff88003fd0d6f0,
  state = 1,
  start_pid = 0,
  start_site = 0xffffffff81055c1f <hrtimer_start_range_ns+20>,
  start_comm = "swapper/2\000\000\000\000\000\000"
}
```

List of commands and functions

The number of commands and convenience functions may evolve over the time, this is just a snapshot of the initial version:

```
(gdb) apropos lx
function lx_current -- Return current task
function lx_module -- Find module by name and return the module variable
function lx_per_cpu -- Return per-cpu variable
function lx_task_by_pid -- Find Linux task by PID and return the task_struct variable
function lx_thread_info -- Calculate Linux thread_info from task variable
lx-dmesg -- Print Linux kernel log buffer
lx-lsmod -- List currently loaded modules
lx-symbols -- (Re-)load symbols of Linux kernel and currently loaded modules
```

Detailed help can be obtained via “help <command-name>” for commands and “help function <function-name>” for convenience functions.

CHAPTER

TEN

USING KGDB, KDB AND THE KERNEL DEBUGGER INTERNALS

Author Jason Wessel

Introduction

The kernel has two different debugger front ends (kdb and kgdb) which interface to the debug core. It is possible to use either of the debugger front ends and dynamically transition between them if you configure the kernel properly at compile and runtime.

Kdb is simplistic shell-style interface which you can use on a system console with a keyboard or serial console. You can use it to inspect memory, registers, process lists, dmesg, and even set breakpoints to stop in a certain location. Kdb is not a source level debugger, although you can set breakpoints and execute some basic kernel run control. Kdb is mainly aimed at doing some analysis to aid in development or diagnosing kernel problems. You can access some symbols by name in kernel built-ins or in kernel modules if the code was built with `CONFIG_KALLSYMS`.

Kgdb is intended to be used as a source level debugger for the Linux kernel. It is used along with gdb to debug a Linux kernel. The expectation is that gdb can be used to "break in" to the kernel to inspect memory, variables and look through call stack information similar to the way an application developer would use gdb to debug an application. It is possible to place breakpoints in kernel code and perform some limited execution stepping.

Two machines are required for using kgdb. One of these machines is a development machine and the other is the target machine. The kernel to be debugged runs on the target machine. The development machine runs an instance of gdb against the vmlinux file which contains the symbols (not a boot image such as bzImage, zImage, uImage...). In gdb the developer specifies the connection parameters and connects to kgdb. The type of connection a developer makes with gdb depends on the availability of kgdb I/O modules compiled as built-ins or loadable kernel modules in the test machine's kernel.

Compiling a kernel

- In order to enable compilation of kdb, you must first enable kgdb.
- The kgdb test compile options are described in the kgdb test suite chapter.

Kernel config options for kgdb

To enable `CONFIG_KGDB` you should look under *Kernel hacking* → *Kernel debugging* and select *KGDB: kernel debugger*.

While it is not a hard requirement that you have symbols in your vmlinux file, gdb tends not to be very useful without the symbolic data, so you will want to turn on `CONFIG_DEBUG_INFO` which is called *Compile the kernel with debug info* in the config menu.

It is advised, but not required, that you turn on the CONFIG_FRAME_POINTER kernel option which is called *Compile the kernel with frame pointers* in the config menu. This option inserts code to into the compiled executable which saves the frame information in registers or on the stack at different points which allows a debugger such as gdb to more accurately construct stack back traces while debugging the kernel.

If the architecture that you are using supports the kernel option CONFIG_STRICT_KERNEL_RWX, you should consider turning it off. This option will prevent the use of software breakpoints because it marks certain regions of the kernel's memory space as read-only. If kgdb supports it for the architecture you are using, you can use hardware breakpoints if you desire to run with the CONFIG_STRICT_KERNEL_RWX option turned on, else you need to turn off this option.

Next you should choose one of more I/O drivers to interconnect debugging host and debugged target. Early boot debugging requires a KGDB I/O driver that supports early debugging and the driver must be built into the kernel directly. Kgdb I/O driver configuration takes place via kernel or module parameters which you can learn more about in the in the section that describes the parameter kgdboc.

Here is an example set of `.config` symbols to enable or disable for kgdb:

```
# CONFIG_STRICT_KERNEL_RWX is not set
CONFIG_FRAME_POINTER=y
CONFIG_KGDB=y
CONFIG_KGDB_SERIAL_CONSOLE=y
```

Kernel config options for kdb

Kdb is quite a bit more complex than the simple gdbstub sitting on top of the kernel's debug core. Kdb must implement a shell, and also adds some helper functions in other parts of the kernel, responsible for printing out interesting data such as what you would see if you ran `lsmod`, or `ps`. In order to build kdb into the kernel you follow the same steps as you would for kgdb.

The main config option for kdb is CONFIG_KGDB_KDB which is called *KGDB_KDB: include kdb frontend for kgdb* in the config menu. In theory you would have already also selected an I/O driver such as the CONFIG_KGDB_SERIAL_CONSOLE interface if you plan on using kdb on a serial port, when you were configuring kgdb.

If you want to use a PS/2-style keyboard with kdb, you would select CONFIG_KDB_KEYBOARD which is called *KGDB_KDB: keyboard as input device* in the config menu. The CONFIG_KDB_KEYBOARD option is not used for anything in the gdb interface to kgdb. The CONFIG_KDB_KEYBOARD option only works with kdb.

Here is an example set of `.config` symbols to enable/disable kdb:

```
# CONFIG_STRICT_KERNEL_RWX is not set
CONFIG_FRAME_POINTER=y
CONFIG_KGDB=y
CONFIG_KGDB_SERIAL_CONSOLE=y
CONFIG_KGDB_KDB=y
CONFIG_KDB_KEYBOARD=y
```

Kernel Debugger Boot Arguments

This section describes the various runtime kernel parameters that affect the configuration of the kernel debugger. The following chapter covers using kdb and kgdb as well as providing some examples of the configuration parameters.

Kernel parameter: kgdboc

The kgdboc driver was originally an abbreviation meant to stand for "kgdb over console". Today it is the primary mechanism to configure how to communicate from gdb to kgdb as well as the devices you want

to use to interact with the kdb shell.

For kgdb/gdb, kgdboc is designed to work with a single serial port. It is intended to cover the circumstance where you want to use a serial console as your primary console as well as using it to perform kernel debugging. It is also possible to use kgdb on a serial port which is not designated as a system console. Kgdboc may be configured as a kernel built-in or a kernel loadable module. You can only make use of `kgdbwait` and early debugging if you build kgdboc into the kernel as a built-in.

Optionally you can elect to activate kms (Kernel Mode Setting) integration. When you use kms with kgdboc and you have a video driver that has atomic mode setting hooks, it is possible to enter the debugger on the graphics console. When the kernel execution is resumed, the previous graphics mode will be restored. This integration can serve as a useful tool to aid in diagnosing crashes or doing analysis of memory with kdb while allowing the full graphics console applications to run.

kgdboc arguments

Usage:

```
kgdboc=[kms][[,]kbd][[,]serial_device][,baud]
```

The order listed above must be observed if you use any of the optional configurations together.

Abbreviations:

- kms = Kernel Mode Setting
- kbd = Keyboard

You can configure kgdboc to use the keyboard, and/or a serial device depending on if you are using kdb and/or kgdb, in one of the following scenarios. The order listed above must be observed if you use any of the optional configurations together. Using kms + only gdb is generally not a useful combination.

Using loadable module or built-in

1. As a kernel built-in:

   Use the kernel boot argument:

   ```
   kgdboc=<tty-device>,[baud]
   ```

2. As a kernel loadable module:

   Use the command:

   ```
   modprobe kgdboc kgdboc=<tty-device>,[baud]
   ```

   Here are two examples of how you might format the kgdboc string. The first is for an x86 target using the first serial port. The second example is for the ARM Versatile AB using the second serial port.

   (a) `kgdboc=ttyS0,115200`

   (b) `kgdboc=ttyAMA1,115200`

Configure kgdboc at runtime with sysfs

At run time you can enable or disable kgdboc by echoing a parameters into the sysfs. Here are two examples:

1. Enable kgdboc on ttyS0:

```
echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc
```

2. Disable kgdboc:

```
echo "" > /sys/module/kgdboc/parameters/kgdboc
```

Note:

You do not need to specify the baud if you are configuring the console on tty which is already configured or open.

More examples

You can configure kgdboc to use the keyboard, and/or a serial device depending on if you are using kdb and/or kgdb, in one of the following scenarios.

1. kdb and kgdb over only a serial port:

```
kgdboc=<serial_device>[,baud]
```

   Example:

```
kgdboc=ttyS0,115200
```

2. kdb and kgdb with keyboard and a serial port:

```
kgdboc=kbd,<serial_device>[,baud]
```

   Example:

```
kgdboc=kbd,ttyS0,115200
```

3. kdb with a keyboard:

```
kgdboc=kbd
```

4. kdb with kernel mode setting:

```
kgdboc=kms,kbd
```

5. kdb with kernel mode setting and kgdb over a serial port:

```
kgdboc=kms,kbd,ttyS0,115200
```

Note:

Kgdboc does not support interrupting the target via the gdb remote protocol. You must manually send a SysRq-G unless you have a proxy that splits console output to a terminal program. A console proxy has a separate TCP port for the debugger and a separate TCP port for the "human" console. The proxy can take care of sending the SysRq-G for you.

When using kgdboc with no debugger proxy, you can end up connecting the debugger at one of two entry points. If an exception occurs after you have loaded kgdboc, a message should print on the console stating it is waiting for the debugger. In this case you disconnect your terminal program and then connect the debugger in its place. If you want to interrupt the target system and forcibly enter a debug session you have to issue a `Sysrq` sequence and then type the letter `g`. Then you disconnect the terminal session

and connect gdb. Your options if you don't like this are to hack gdb to send the SysRq-G for you as well as on the initial connect, or to use a debugger proxy that allows an unmodified gdb to do the debugging.

Kernel parameter: kgdbwait

The Kernel command line option kgdbwait makes kgdb wait for a debugger connection during booting of a kernel. You can only use this option if you compiled a kgdb I/O driver into the kernel and you specified the I/O driver configuration as a kernel command line option. The kgdbwait parameter should always follow the configuration parameter for the kgdb I/O driver in the kernel command line else the I/O driver will not be configured prior to asking the kernel to use it to wait.

The kernel will stop and wait as early as the I/O driver and architecture allows when you use this option. If you build the kgdb I/O driver as a loadable kernel module kgdbwait will not do anything.

Kernel parameter: kgdbcon

The kgdbcon feature allows you to see printk() messages inside gdb while gdb is connected to the kernel. Kdb does not make use of the kgdbcon feature.

Kgdb supports using the gdb serial protocol to send console messages to the debugger when the debugger is connected and running. There are two ways to activate this feature.

1. Activate with the kernel command line option:

```
kgdbcon
```

2. Use sysfs before configuring an I/O driver:

```
echo 1 > /sys/module/kgdb/parameters/kgdb_use_con
```

Note:

If you do this after you configure the kgdb I/O driver, the setting will not take effect until the next point the I/O is reconfigured.

Important:

You cannot use kgdboc + kgdbcon on a tty that is an active system console. An example of incorrect usage is:

```
console=ttyS0,115200 kgdboc=ttyS0 kgdbcon
```

It is possible to use this option with kgdboc on a tty that is not a system console.

Run time parameter: kgdbreboot

The kgdbreboot feature allows you to change how the debugger deals with the reboot notification. You have 3 choices for the behavior. The default behavior is always set to 0.

1	`echo -1 > /sys/module/debug_core/parameters/kgdbreboot`	Ignore the reboot notification entirely.
2	`echo 0 > /sys/module/debug_core/parameters/kgdbreboot`	Send the detach message to any attached debugger client.
3	`echo 1 > /sys/module/debug_core/parameters/kgdbreboot`	Enter the debugger on reboot notify.

Kernel parameter: `nokaslr`

If the architecture that you are using enable KASLR by default, you should consider turning it off. KASLR randomizes the virtual address where the kernel image is mapped and confuse gdb which resolve kernel symbol address from symbol table of vmlinux.

Using kdb

Quick start for kdb on a serial port

This is a quick example of how to use kdb.

1. Configure kgdboc at boot using kernel parameters:

   ```
   console=ttyS0,115200 kgdboc=ttyS0,115200 nokaslr
   ```

   OR

   Configure kgdboc after the kernel has booted; assuming you are using a serial port console:

   ```
   echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc
   ```

2. Enter the kernel debugger manually or by waiting for an oops or fault. There are several ways you can enter the kernel debugger manually; all involve using the `SysRq-G`, which means you must have enabled `CONFIG_MAGIC_SysRq=y` in your kernel config.

   - When logged in as root or with a super user session you can run:

     ```
     echo g > /proc/sysrq-trigger
     ```

   - Example using minicom 2.2

     Press: `CTRL-A f g`

   - When you have telneted to a terminal server that supports sending a remote break

     Press: `CTRL-]`

     Type in: `send break`

     Press: `Enter g`

3. From the kdb prompt you can run the `help` command to see a complete list of the commands that are available.

   Some useful commands in kdb include:

`lsmod`	Shows where kernel modules are loaded
`ps`	Displays only the active processes
`ps A`	Shows all the processes
`summary`	Shows kernel version info and memory usage
`bt`	Get a backtrace of the current process using `dump_stack()`
`dmesg`	View the kernel syslog buffer
`go`	Continue the system

4. When you are done using kdb you need to consider rebooting the system or using the go command to resuming normal kernel execution. If you have paused the kernel for a lengthy period of time, applications that rely on timely networking or anything to do with real wall clock time could be adversely affected, so you should take this into consideration when using the kernel debugger.

Quick start for kdb using a keyboard connected console

This is a quick example of how to use kdb with a keyboard.

1. Configure kgdboc at boot using kernel parameters:

```
kgdboc=kbd
```

   OR

   Configure kgdboc after the kernel has booted:

```
echo kbd > /sys/module/kgdboc/parameters/kgdboc
```

2. Enter the kernel debugger manually or by waiting for an oops or fault. There are several ways you can enter the kernel debugger manually; all involve using the SysRq-G, which means you must have enabled CONFIG_MAGIC_SysRq=y in your kernel config.

   - When logged in as root or with a super user session you can run:

```
echo g > /proc/sysrq-trigger
```

   - Example using a laptop keyboard:

     Press and hold down: Alt

     Press and hold down: Fn

     Press and release the key with the label: SysRq

     Release: Fn

     Press and release: g

     Release: Alt

   - Example using a PS/2 101-key keyboard

     Press and hold down: Alt

     Press and release the key with the label: SysRq

     Press and release: g

     Release: Alt

3. Now type in a kdb command such as help, dmesg, bt or go to continue kernel execution.

Using kgdb / gdb

In order to use kgdb you must activate it by passing configuration information to one of the kgdb I/O drivers. If you do not pass any configuration information kgdb will not do anything at all. Kgdb will only actively hook up to the kernel trap hooks if a kgdb I/O driver is loaded and configured. If you unconfigure a kgdb I/O driver, kgdb will unregister all the kernel hook points.

All kgdb I/O drivers can be reconfigured at run time, if CONFIG_SYSFS and CONFIG_MODULES are enabled, by echo'ing a new config string to /sys/module/<driver>/parameter/<option>. The driver can be unconfigured by passing an empty string. You cannot change the configuration while the debugger is attached. Make sure to detach the debugger with the detach command prior to trying to unconfigure a kgdb I/O driver.

Connecting with gdb to a serial port

1. Configure kgdboc

   Configure kgdboc at boot using kernel parameters:

   ```
   kgdboc=ttyS0,115200
   ```

   OR

   Configure kgdboc after the kernel has booted:

   ```
   echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc
   ```

2. Stop kernel execution (break into the debugger)

   In order to connect to gdb via kgdboc, the kernel must first be stopped. There are several ways to stop the kernel which include using kgdbwait as a boot argument, via a SysRq-G, or running the kernel until it takes an exception where it waits for the debugger to attach.

   - When logged in as root or with a super user session you can run:

     ```
     echo g > /proc/sysrq-trigger
     ```

   - Example using minicom 2.2

     Press: CTRL-A f g

   - When you have telneted to a terminal server that supports sending a remote break

     Press: CTRL-]

     Type in: send break

     Press: Enter g

3. Connect from gdb

   Example (using a directly connected port):

   ```
   % gdb ./vmlinux
   (gdb) set remotebaud 115200
   (gdb) target remote /dev/ttyS0
   ```

   Example (kgdb to a terminal server on TCP port 2012):

   ```
   % gdb ./vmlinux
   (gdb) target remote 192.168.2.2:2012
   ```

   Once connected, you can debug a kernel the way you would debug an application program.

   If you are having problems connecting or something is going seriously wrong while debugging, it will most often be the case that you want to enable gdb to be verbose about its target communications. You do this prior to issuing the target remote command by typing in:

   ```
   set debug remote 1
   ```

Remember if you continue in gdb, and need to “break in” again, you need to issue an other SysRq-G. It is easy to create a simple entry point by putting a breakpoint at sys_sync and then you can run sync from a shell or script to break into the debugger.

kgdb and kdb interoperability

It is possible to transition between kdb and kgdb dynamically. The debug core will remember which you used the last time and automatically start in the same mode.

Switching between kdb and kgdb

Switching from kgdb to kdb

There are two ways to switch from kgdb to kdb: you can use gdb to issue a maintenance packet, or you can blindly type the command `$3#33`. Whenever the kernel debugger stops in kgdb mode it will print the message `KGDB or $3#33 for KDB`. It is important to note that you have to type the sequence correctly in one pass. You cannot type a backspace or delete because kgdb will interpret that as part of the debug stream.

1. Change from kgdb to kdb by blindly typing:

```
$3#33
```

2. Change from kgdb to kdb with gdb:

```
maintenance packet 3
```

> ***Note:***
>
> *Now you must kill gdb. Typically you press CTRL-Z and issue the command:*
>
> ```
> kill -9 %
> ```

Change from kdb to kgdb

There are two ways you can change from kdb to kgdb. You can manually enter kgdb mode by issuing the kgdb command from the kdb shell prompt, or you can connect gdb while the kdb shell prompt is active. The kdb shell looks for the typical first commands that gdb would issue with the gdb remote protocol and if it sees one of those commands it automatically changes into kgdb mode.

1. From kdb issue the command:

```
kgdb
```

   Now disconnect your terminal program and connect gdb in its place

2. At the kdb prompt, disconnect the terminal program and connect gdb in its place.

Running kdb commands from gdb

It is possible to run a limited set of kdb commands from gdb, using the gdb monitor command. You don't want to execute any of the run control or breakpoint operations, because it can disrupt the state of the kernel debugger. You should be using gdb for breakpoints and run control operations if you have gdb connected. The more useful commands to run are things like lsmod, dmesg, ps or possibly some of the memory information commands. To see all the kdb commands you can run `monitor help`.

Example:

```
(gdb) monitor ps
1 idle process (state I) and
27 sleeping system daemon (state M) processes suppressed,
use 'ps A' to see all.
Task Addr       Pid   Parent [*] cpu State Thread     Command

0xc78291d0        1        0  0    0   S  0xc7829404  init
0xc7954150      942        1  0    0   S  0xc7954384  dropbear
```

```
0xc78789c0      944        1  0    0   S  0xc7878bf4  sh
(gdb)
```

kgdb Test Suite

When kgdb is enabled in the kernel config you can also elect to enable the config parameter KGDB_TESTS. Turning this on will enable a special kgdb I/O module which is designed to test the kgdb internal functions.

The kgdb tests are mainly intended for developers to test the kgdb internals as well as a tool for developing a new kgdb architecture specific implementation. These tests are not really for end users of the Linux kernel. The primary source of documentation would be to look in the `drivers/misc/kgdbts.c` file.

The kgdb test suite can also be configured at compile time to run the core set of tests by setting the kernel config parameter KGDB_TESTS_ON_BOOT. This particular option is aimed at automated regression testing and does not require modifying the kernel boot config arguments. If this is turned on, the kgdb test suite can be disabled by specifying `kgdbts=` as a kernel boot argument.

Kernel Debugger Internals

Architecture Specifics

The kernel debugger is organized into a number of components:

1. The debug core

   The debug core is found in `kernel/debugger/debug_core.c`. It contains:

   - A generic OS exception handler which includes sync'ing the processors into a stopped state on an multi-CPU system.
   - The API to talk to the kgdb I/O drivers
   - The API to make calls to the arch-specific kgdb implementation
   - The logic to perform safe memory reads and writes to memory while using the debugger
   - A full implementation for software breakpoints unless overridden by the arch
   - The API to invoke either the kdb or kgdb frontend to the debug core.
   - The structures and callback API for atomic kernel mode setting.

     Note:

     kgdboc is where the kms callbacks are invoked.

2. kgdb arch-specific implementation

   This implementation is generally found in `arch/*/kernel/kgdb.c`. As an example, `arch/x86/kernel/kgdb.c` contains the specifics to implement HW breakpoint as well as the initialization to dynamically register and unregister for the trap handlers on this architecture. The arch-specific portion implements:

   - contains an arch-specific trap catcher which invokes `kgdb_handle_exception()` to start kgdb about doing its work
   - translation to and from gdb specific packet format to `pt_regs`
   - Registration and unregistration of architecture specific trap hooks

- Any special exception handling and cleanup
- NMI exception handling and cleanup
- (optional) HW breakpoints

3. gdbstub frontend (aka kgdb)

   The gdbstub is located in `kernel/debug/gdbstub.c`. It contains:

   - All the logic to implement the gdb serial protocol

4. kdb frontend

   The kdb debugger shell is broken down into a number of components. The kdb core is located in kernel/debug/kdb. There are a number of helper functions in some of the other kernel components to make it possible for kdb to examine and report information about the kernel without taking locks that could cause a kernel deadlock. The kdb core contains implements the following functionality.

   - A simple shell
   - The kdb core command set
   - A registration API to register additional kdb shell commands.
     - A good example of a self-contained kdb module is the `ftdump` command for dumping the ftrace buffer. See: `kernel/trace/trace_kdb.c`
     - For an example of how to dynamically register a new kdb command you can build the kdb_hello.ko kernel module from `samples/kdb/kdb_hello.c`. To build this example you can set `CONFIG_SAMPLES=y` and `CONFIG_SAMPLE_KDB=m` in your kernel config. Later run `modprobe kdb_hello` and the next time you enter the kdb shell, you can run the `hello` command.
   - The implementation for `kdb_printf()` which emits messages directly to I/O drivers, bypassing the kernel log.
   - SW / HW breakpoint management for the kdb shell

5. kgdb I/O driver

   Each kgdb I/O driver has to provide an implementation for the following:

   - configuration via built-in or module
   - dynamic configuration and kgdb hook registration calls
   - read and write character interface
   - A cleanup handler for unconfiguring from the kgdb core
   - (optional) Early debug methodology

   Any given kgdb I/O driver has to operate very closely with the hardware and must do it in such a way that does not enable interrupts or change other parts of the system context without completely restoring them. The kgdb core will repeatedly "poll" a kgdb I/O driver for characters when it needs input. The I/O driver is expected to return immediately if there is no data available. Doing so allows for the future possibility to touch watchdog hardware in such a way as to have a target system not reset when these are enabled.

If you are intent on adding kgdb architecture specific support for a new architecture, the architecture should define `HAVE_ARCH_KGDB` in the architecture specific Kconfig file. This will enable kgdb for the architecture, and at that point you must create an architecture specific kgdb implementation.

There are a few flags which must be set on every architecture in their `asm/kgdb.h` file. These are:

- **`NUMREGBYTES`:** The size in bytes of all of the registers, so that we can ensure they will all fit into a packet.
- **`BUFMAX`:** The size in bytes of the buffer GDB will read into. This must be larger than NUMREGBYTES.

- **CACHE_FLUSH_IS_SAFE:** Set to 1 if it is always safe to call flush_cache_range or flush_icache_range. On some architectures, these functions may not be safe to call on SMP since we keep other CPUs in a holding pattern.

There are also the following functions for the common backend, found in `kernel/kgdb.c`, that must be supplied by the architecture-specific backend unless marked as (optional), in which case a default function maybe used if the architecture does not need to provide a specific implementation.

int **kgdb_skipexception**(int *exception*, struct pt_regs * *regs*)
(optional) exit kgdb_handle_exception early

Parameters

int exception Exception vector number

struct pt_regs * regs Current `struct pt_regs`.

Description

On some architectures it is required to skip a breakpoint exception when it occurs after a breakpoint has been removed. This can be implemented in the architecture specific portion of kgdb.

void **kgdb_breakpoint**(void)
compiled in breakpoint

Parameters

void no arguments

Description

This will be implemented as a static inline per architecture. This function is called by the kgdb core to execute an architecture specific trap to cause kgdb to enter the exception processing.

int **kgdb_arch_init**(void)
Perform any architecture specific initalization.

Parameters

void no arguments

Description

This function will handle the initalization of any architecture specific callbacks.

void **kgdb_arch_exit**(void)
Perform any architecture specific uninitalization.

Parameters

void no arguments

Description

This function will handle the uninitalization of any architecture specific callbacks, for dynamic registration and unregistration.

void **pt_regs_to_gdb_regs**(unsigned long * *gdb_regs*, struct pt_regs * *regs*)
Convert ptrace regs to GDB regs

Parameters

unsigned long * gdb_regs A pointer to hold the registers in the order GDB wants.

struct pt_regs * regs The `struct pt_regs` of the current process.

Description

Convert the pt_regs in **regs** into the format for registers that GDB expects, stored in **gdb_regs**.

void **sleeping_thread_to_gdb_regs**(unsigned long * *gdb_regs*, struct task_struct * *p*)
Convert ptrace regs to GDB regs

Parameters

unsigned long * gdb_regs A pointer to hold the registers in the order GDB wants.

struct task_struct * p The `struct task_struct` of the desired process.

Description

Convert the register values of the sleeping process in **p** to the format that GDB expects. This function is called when kgdb does not have access to the `struct pt_regs` and therefore it should fill the gdb registers **gdb_regs** with what has been saved in `struct thread_struct` thread field during switch_to.

void **gdb_regs_to_pt_regs**(unsigned long * *gdb_regs*, struct pt_regs * *regs*)
Convert GDB regs to ptrace regs.

Parameters

unsigned long * gdb_regs A pointer to hold the registers we've received from GDB.

struct pt_regs * regs A pointer to a `struct pt_regs` to hold these values in.

Description

Convert the GDB regs in **gdb_regs** into the pt_regs, and store them in **regs**.

int **kgdb_arch_handle_exception**(int *vector*, int *signo*, int *err_code*, char * *remcom_in_buffer*, char * *remcom_out_buffer*, struct pt_regs * *regs*)
Handle architecture specific GDB packets.

Parameters

int vector The error vector of the exception that happened.

int signo The signal number of the exception that happened.

int err_code The error code of the exception that happened.

char * remcom_in_buffer The buffer of the packet we have read.

char * remcom_out_buffer The buffer of BUFMAX bytes to write a packet into.

struct pt_regs * regs The `struct pt_regs` of the current process.

Description

This function MUST handle the 'c' and 's' command packets, as well packets to set / remove a hardware breakpoint, if used. If there are additional packets which the hardware needs to handle, they are handled here. The code should return -1 if it wants to process more packets, and a `0` or `1` if it wants to exit from the kgdb callback.

void **kgdb_roundup_cpus**(unsigned long *flags*)
Get other CPUs into a holding pattern

Parameters

unsigned long flags Current IRQ state

Description

On SMP systems, we need to get the attention of the other CPUs and get them into a known state. This should do what is needed to get the other CPUs to call `kgdb_wait()`. Note that on some arches, the NMI approach is not used for rounding up all the CPUs. For example, in case of MIPS, `smp_call_function()` is used to roundup CPUs. In this case, we have to make sure that interrupts are enabled before calling `smp_call_function()`. The argument to this function is the flags that will be used when restoring the interrupts. There is `local_irq_save()` call before *kgdb_roundup_cpus()*.

On non-SMP systems, this is not called.

void **kgdb_arch_set_pc**(struct pt_regs * *regs*, unsigned long *pc*)
Generic call back to the program counter

Parameters

struct pt_regs * regs Current `struct pt_regs`.

unsigned long pc The new value for the program counter

Description

This function handles updating the program counter and requires an architecture specific implementation.

void **kgdb_arch_late**(void)
Perform any architecture specific initalization.

Parameters

void no arguments

Description

This function will handle the late initalization of any architecture specific callbacks. This is an optional function for handling things like late initialization of hw breakpoints. The default implementation does nothing.

struct **kgdb_arch**
Describe architecture specific values.

Definition

```
struct kgdb_arch {
  unsigned char gdb_bpt_instr;
  unsigned long flags;
  int (* set_breakpoint) (unsigned long, char *);
  int (* remove_breakpoint) (unsigned long, char *);
  int (* set_hw_breakpoint) (unsigned long, int, enum kgdb_bptype);
  int (* remove_hw_breakpoint) (unsigned long, int, enum kgdb_bptype);
  void (* disable_hw_break) (struct pt_regs *regs);
  void (* remove_all_hw_break) (void);
  void (* correct_hw_break) (void);
  void (* enable_nmi) (bool on);
};
```

Members

gdb_bpt_instr The instruction to trigger a breakpoint.

flags Flags for the breakpoint, currently just `KGDB_HW_BREAKPOINT`.

set_breakpoint Allow an architecture to specify how to set a software breakpoint.

remove_breakpoint Allow an architecture to specify how to remove a software breakpoint.

set_hw_breakpoint Allow an architecture to specify how to set a hardware breakpoint.

remove_hw_breakpoint Allow an architecture to specify how to remove a hardware breakpoint.

disable_hw_break Allow an architecture to specify how to disable hardware breakpoints for a single cpu.

remove_all_hw_break Allow an architecture to specify how to remove all hardware breakpoints.

correct_hw_break Allow an architecture to specify how to correct the hardware debug registers.

enable_nmi Manage NMI-triggered entry to KGDB

struct **kgdb_io**
Describe the interface for an I/O driver to talk with KGDB.

Definition

```
struct kgdb_io {
  const char * name;
  int (* read_char) (void);
  void (* write_char) (u8);
  void (* flush) (void);
  int (* init) (void);
  void (* pre_exception) (void);
  void (* post_exception) (void);
  int is_console;
};
```

Members

name Name of the I/O driver.

read_char Pointer to a function that will return one char.

write_char Pointer to a function that will write one char.

flush Pointer to a function that will flush any pending writes.

init Pointer to a function that will initialize the device.

pre_exception Pointer to a function that will do any prep work for the I/O driver.

post_exception Pointer to a function that will do any cleanup work for the I/O driver.

is_console 1 if the end device is a console 0 if the I/O device is not a console

kgdboc internals

kgdboc and uarts

The kgdboc driver is actually a very thin driver that relies on the underlying low level to the hardware driver having "polling hooks" to which the tty driver is attached. In the initial implementation of kgdboc the serial_core was changed to expose a low level UART hook for doing polled mode reading and writing of a single character while in an atomic context. When kgdb makes an I/O request to the debugger, kgdboc invokes a callback in the serial core which in turn uses the callback in the UART driver.

When using kgdboc with a UART, the UART driver must implement two callbacks in the `struct uart_ops`. Example from `drivers/8250.c`:

```
#ifdef CONFIG_CONSOLE_POLL
    .poll_get_char = serial8250_get_poll_char,
    .poll_put_char = serial8250_put_poll_char,
#endif
```

Any implementation specifics around creating a polling driver use the `#ifdef CONFIG_CONSOLE_POLL`, as shown above. Keep in mind that polling hooks have to be implemented in such a way that they can be called from an atomic context and have to restore the state of the UART chip on return such that the system can return to normal when the debugger detaches. You need to be very careful with any kind of lock you consider, because failing here is most likely going to mean pressing the reset button.

kgdboc and keyboards

The kgdboc driver contains logic to configure communications with an attached keyboard. The keyboard infrastructure is only compiled into the kernel when `CONFIG_KDB_KEYBOARD=y` is set in the kernel configuration.

The core polled keyboard driver driver for PS/2 type keyboards is in `drivers/char/kdb_keyboard.c`. This driver is hooked into the debug core when kgdboc populates the callback in the array called

kdb_poll_funcs[]. The kdb_get_kbd_char() is the top-level function which polls hardware for single character input.

kgdboc and kms

The kgdboc driver contains logic to request the graphics display to switch to a text context when you are using kgdboc=kms,kbd, provided that you have a video driver which has a frame buffer console and atomic kernel mode setting support.

Every time the kernel debugger is entered it calls kgdboc_pre_exp_handler() which in turn calls con_debug_enter() in the virtual console layer. On resuming kernel execution, the kernel debugger calls kgdboc_post_exp_handler() which in turn calls con_debug_leave().

Any video driver that wants to be compatible with the kernel debugger and the atomic kms callbacks must implement the mode_set_base_atomic, fb_debug_enter and fb_debug_leave operations. For the fb_debug_enter and fb_debug_leave the option exists to use the generic drm fb helper functions or implement something custom for the hardware. The following example shows the initialization of the .mode_set_base_atomic operation in drivers/gpu/drm/i915/intel_display.c:

```
static const struct drm_crtc_helper_funcs intel_helper_funcs = {
[...]
        .mode_set_base_atomic = intel_pipe_set_base_atomic,
[...]
};
```

Here is an example of how the i915 driver initializes the fb_debug_enter and fb_debug_leave functions to use the generic drm helpers in drivers/gpu/drm/i915/intel_fb.c:

```
static struct fb_ops intelfb_ops = {
[...]
       .fb_debug_enter = drm_fb_helper_debug_enter,
       .fb_debug_leave = drm_fb_helper_debug_leave,
[...]
};
```

Credits

The following people have contributed to this document:

1. Amit Kale <amitkale@linsyssoft.com>
2. Tom Rini <trini@kernel.crashing.org>

In March 2008 this document was completely rewritten by:

- Jason Wessel <jason.wessel@windriver.com>

In Jan 2010 this document was updated to include kdb.

- Jason Wessel <jason.wessel@windriver.com>

CHAPTER

ELEVEN

LINUX KERNEL SELFTESTS

The kernel contains a set of “self tests” under the tools/testing/selftests/ directory. These are intended to be small tests to exercise individual code paths in the kernel. Tests are intended to be run after building, installing and booting a kernel.

On some systems, hot-plug tests could hang forever waiting for cpu and memory to be ready to be offlined. A special hot-plug target is created to run full range of hot-plug tests. In default mode, hot-plug tests run in safe mode with a limited scope. In limited mode, cpu-hotplug test is run on a single cpu as opposed to all hotplug capable cpus, and memory hotplug test is run on 2% of hotplug capable memory instead of 10%.

Running the selftests (hotplug tests are run in limited mode)

To build the tests:

```
$ make -C tools/testing/selftests
```

To run the tests:

```
$ make -C tools/testing/selftests run_tests
```

To build and run the tests with a single command, use:

```
$ make kselftest
```

Note that some tests will require root privileges.

Running a subset of selftests

You can use the “TARGETS” variable on the make command line to specify single test to run, or a list of tests to run.

To run only tests targeted for a single subsystem:

```
$ make -C tools/testing/selftests TARGETS=ptrace run_tests
```

You can specify multiple tests to build and run:

```
$  make TARGETS="size timers" kselftest
```

See the top-level tools/testing/selftests/Makefile for the list of all possible targets.

Running the full range hotplug selftests

To build the hotplug tests:

```
$ make -C tools/testing/selftests hotplug
```

To run the hotplug tests:

```
$ make -C tools/testing/selftests run_hotplug
```

Note that some tests will require root privileges.

Install selftests

You can use kselftest_install.sh tool installs selftests in default location which is tools/testing/selftests/kselftest or a user specified location.

To install selftests in default location:

```
$ cd tools/testing/selftests
$ ./kselftest_install.sh
```

To install selftests in a user specified location:

```
$ cd tools/testing/selftests
$ ./kselftest_install.sh install_dir
```

Running installed selftests

Kselftest install as well as the Kselftest tarball provide a script named “run_kselftest.sh” to run the tests.

You can simply do the following to run the installed Kselftests. Please note some tests will require root privileges:

```
$ cd kselftest
$ ./run_kselftest.sh
```

Contributing new tests

In general, the rules for selftests are

- Do as much as you can if you’re not root;
- Don’t take too long;
- Don’t break the build on any architecture, and
- Don’t cause the top-level “make run_tests” to fail if your feature is unconfigured.

Contributing new tests (details)

- Use TEST_GEN_XXX if such binaries or files are generated during compiling.

  TEST_PROGS, TEST_GEN_PROGS mean it is the excutable tested by default.

TEST_PROGS_EXTENDED, TEST_GEN_PROGS_EXTENDED mean it is the executable which is not tested by default. TEST_FILES, TEST_GEN_FILES mean it is the file which is used by test.

Test Harness

The kselftest_harness.h file contains useful helpers to build tests. The tests from tools/testing/selftests/seccomp/seccomp_bpf.c can be used as example.

Example

```
#include "../kselftest_harness.h"

TEST(standalone_test) {
  do_some_stuff;
  EXPECT_GT(10, stuff) {
     stuff_state_t state;
     enumerate_stuff_state(:c:type:`state`);
     TH_LOG("expectation failed with state: ``s``", state.msg);
  }
  more_stuff;
  ASSERT_NE(some_stuff, NULL) TH_LOG("how did it happen?!");
  last_stuff;
  EXPECT_EQ(0, last_stuff);
}

FIXTURE(my_fixture) {
  mytype_t *data;
  int awesomeness_level;
};
FIXTURE_SETUP(my_fixture) {
  self->data = :c:func:`mytype_new()`;
  ASSERT_NE(NULL, self->data);
}
FIXTURE_TEARDOWN(my_fixture) {
  mytype_free(self->data);
}
TEST_F(my_fixture, data_is_good) {
  EXPECT_EQ(1, is_my_data_good(self->data));
}

TEST_HARNESS_MAIN
```

Helpers

TH_LOG(*fmt*, *...*)

Parameters

fmt format string

... optional arguments

Description

```
TH_LOG(format, ...)
```

Optional debug logging function available for use in tests. Logging may be enabled or disabled by defining TH_LOG_ENABLED. E.g., #define TH_LOG_ENABLED 1

If no definition is provided, logging is enabled by default.

If there is no way to print an error message for the process running the test (e.g. not allowed to write to stderr), it is still possible to get the ASSERT_* number for which the test failed. This behavior can be enabled by writing *_metadata->no_print = true;* before the check sequence that is unable to print. When an error occur, instead of printing an error message and calling *abort(3)*, the test process call *_exit(2)* with the assert number as argument, which is then printed by the parent process.

TEST(*test_name*)
Defines the test function and creates the registration stub

Parameters

test_name test name

Description

```
TEST(name) { implementation }
```

Defines a test by name. Names must be unique and tests must not be run in parallel. The implementation containing block is a function and scoping should be treated as such. Returning early may be performed with a bare “return;” statement.

EXPECT_* and ASSERT_* are valid in a *TEST()* { } context.

TEST_SIGNAL(*test_name*, *signal*)

Parameters

test_name test name

signal signal number

Description

```
TEST_SIGNAL(name, signal) { implementation }
```

Defines a test by name and the expected term signal. Names must be unique and tests must not be run in parallel. The implementation containing block is a function and scoping should be treated as such. Returning early may be performed with a bare “return;” statement.

EXPECT_* and ASSERT_* are valid in a *TEST()* { } context.

FIXTURE_DATA(*datatype_name*)
Wraps the struct name so we have one less argument to pass around

Parameters

datatype_name datatype name

Description

```
FIXTURE_DATA(datatype name)
```

This call may be used when the type of the fixture data is needed. In general, this should not be needed unless the *self* is being passed to a helper directly.

FIXTURE(*fixture_name*)
Called once per fixture to setup the data and register

Parameters

fixture_name fixture name

Description

```
FIXTURE(datatype name) {
  type property1;
```

```
  ...
};
```

Defines the data provided to *TEST_F()*-defined tests as *self*. It should be populated and cleaned up using *FIXTURE_SETUP()* and *FIXTURE_TEARDOWN()*.

FIXTURE_SETUP(*fixture_name*)
Prepares the setup function for the fixture. *_metadata* is included so that ASSERT_* work as a convenience

Parameters

fixture_name fixture name

Description

```
FIXTURE_SETUP(fixture name) { implementation }
```

Populates the required "setup" function for a fixture. An instance of the datatype defined with *FIXTURE_DATA()* will be exposed as *self* for the implementation.

ASSERT_* are valid for use in this context and will prempt the execution of any dependent fixture tests.

A bare "return;" statement may be used to return early.

FIXTURE_TEARDOWN(*fixture_name*)

Parameters

fixture_name fixture name

Description

```
FIXTURE_TEARDOWN(fixture name) { implementation }
```

Populates the required "teardown" function for a fixture. An instance of the datatype defined with *FIXTURE_DATA()* will be exposed as *self* for the implementation to clean up.

A bare "return;" statement may be used to return early.

TEST_F(*fixture_name*, *test_name*)
Emits test registration and helpers for fixture-based test cases

Parameters

fixture_name fixture name

test_name test name

Description

```
TEST_F(fixture, name) { implementation }
```

Defines a test that depends on a fixture (e.g., is part of a test case). Very similar to *TEST()* except that *self* is the setup instance of fixture's datatype exposed for use by the implementation.

TEST_HARNESS_MAIN()
Simple wrapper to run the test harness

Parameters

Description

```
TEST_HARNESS_MAIN
```

Use once to append a `main()` to the test file.

Operators

Operators for use in *TEST()* and *TEST_F()*. ASSERT_* calls will stop test execution immediately. EXPECT_* calls will emit a failure warning, note it, and continue.

ASSERT_EQ(*expected*, *seen*)

Parameters

expected expected value

seen measured value

Description

ASSERT_EQ(expected, measured): expected == measured

ASSERT_NE(*expected*, *seen*)

Parameters

expected expected value

seen measured value

Description

ASSERT_NE(expected, measured): expected != measured

ASSERT_LT(*expected*, *seen*)

Parameters

expected expected value

seen measured value

Description

ASSERT_LT(expected, measured): expected < measured

ASSERT_LE(*expected*, *seen*)

Parameters

expected expected value

seen measured value

Description

ASSERT_LE(expected, measured): expected <= measured

ASSERT_GT(*expected*, *seen*)

Parameters

expected expected value

seen measured value

Description

ASSERT_GT(expected, measured): expected > measured

ASSERT_GE(*expected*, *seen*)

Parameters

expected expected value

seen measured value

Description

ASSERT_GE(expected, measured): expected >= measured

ASSERT_NULL(*seen*)

Parameters

seen measured value

Description

ASSERT_NULL(measured): NULL == measured

ASSERT_TRUE(*seen*)

Parameters

seen measured value

Description

ASSERT_TRUE(measured): measured != 0

ASSERT_FALSE(*seen*)

Parameters

seen measured value

Description

ASSERT_FALSE(measured): measured == 0

ASSERT_STREQ(*expected*, *seen*)

Parameters

expected expected value

seen measured value

Description

ASSERT_STREQ(expected, measured): !strcmp(expected, measured)

ASSERT_STRNE(*expected*, *seen*)

Parameters

expected expected value

seen measured value

Description

ASSERT_STRNE(expected, measured): strcmp(expected, measured)

EXPECT_EQ(*expected*, *seen*)

Parameters

expected expected value

seen measured value

Description

EXPECT_EQ(expected, measured): expected == measured

EXPECT_NE(*expected*, *seen*)

Parameters

expected expected value

seen measured value

Description

EXPECT_NE(expected, measured): expected != measured

EXPECT_LT(*expected*, *seen*)

Parameters

expected expected value

seen measured value

Description

EXPECT_LT(expected, measured): expected < measured

EXPECT_LE(*expected*, *seen*)

Parameters

expected expected value

seen measured value

Description

EXPECT_LE(expected, measured): expected <= measured

EXPECT_GT(*expected*, *seen*)

Parameters

expected expected value

seen measured value

Description

EXPECT_GT(expected, measured): expected > measured

EXPECT_GE(*expected*, *seen*)

Parameters

expected expected value

seen measured value

Description

EXPECT_GE(expected, measured): expected >= measured

EXPECT_NULL(*seen*)

Parameters

seen measured value

Description

EXPECT_NULL(measured): NULL == measured

EXPECT_TRUE(*seen*)

Parameters

seen measured value

Description

EXPECT_TRUE(measured): 0 != measured

EXPECT_FALSE(*seen*)

Parameters

seen measured value

Description

EXPECT_FALSE(measured): 0 == measured

EXPECT_STREQ(*expected*, *seen*)

Parameters

expected expected value

seen measured value

Description

EXPECT_STREQ(expected, measured): !strcmp(expected, measured)

EXPECT_STRNE(*expected*, *seen*)

Parameters

expected expected value

seen measured value

Description

EXPECT_STRNE(expected, measured): strcmp(expected, measured)