Bug Hunting and Static Analysis

Red Hat
Ondřej Vašík <ovasik@redhat.com>
and Petr Müller <pmuller@redhat.com>
2011-02-11

Abstract

Basic overview of common error patterns in C/C++, few words about defensive programming and tools for dynamic analysis. In second part we will cover what static analysis is, which tools could be used.

Agenda

Common Errors in C/C++ Programs

- dereference of a dangling pointer or NULL pointer
- invalid or double `free()`
- buffer overflow
- resource leak (memory, file descriptor, etc.)
- use of uninitialized value
- ...
- dead code
- synchronisation problems

Agenda

How to Prevent a Failure in Production?

- defensive programming
- regression tests, valgrind, etc.
- Fedora users
- ABRT
- ...

- static analysis

Defensive Programming

- use compiler protection mechanisms
  - -D_FORTIFY_SOURCE=2
  - stack-protector, PIE/PIC, RELRO, ExecShield
  - don't ignore warnings (-Wall -Wextra)
- never trust anyone, never assume anything
  - memory boundaries
  - check return codes/error codes
  - use descriptors
  - respect uid/gids, don't over escalate privileges
  - asserts
- `www.akkadia.org/drepper/defprogramming.pdf`

redhat

Testing

Levels:

unit code piece testing, usually developer

integration testing interface between parts

system testing the whole stuff together

Purpose:

performance measurements of resource usage

regression finding new bugs after a change

load/stress reliability/robustness/scalability

The theory ain't clear and unified, so we could argue whole day...

How can you help testing

Of course

- Run test suites and investigate results

But also

- Submit testcases with your patches where applicable
- Use easy measures when you test, like MALLOC_PERTURB_
  - Drepper, Ulrich: MALLOC_PERTURB_: `http://udrepper.livejournal.com/11429.html`

Dynamic Analysis

- `valgrind`
- `systemtap`
- `oprofile`
- `strace`
- `ltrace`
- `gdb`

redhat

valgrind

- **very powerful suite for debugging/profiling, not just for checking resource leaks(memory, descriptors, ...)**
- Usage:
  - easy-to-use tool for searching for memory leaks with detailed log output
  - locates exact place of allocating memory which is not freed
- How?:
  - it is highly recommended to have debuginfo packages for libraries used by the binary you are checking
  - highest impact have resource leaks in applications running for long time (e.g. daemons) or in applications working with a lot of data (long time run)
  - simple usage to gather a lot of information is:
    valgrind -v --leak-check=full <binary> <arguments> 2>myvalgrind.log

redhat

strace, ltrace

- **monitoring utilities showing system/library calls executed by program and signals/exitcodes it received**
- Usage:
  - revealing a place of hanging (e.g. waiting for I/O, timeout)
  - revealing syscalls/library functions called by program, their parameters and exit codes
- Benefits:
  - for experienced maintainer a lot of information about the program run, he could analyze where the problem occured and fix the issue
  - no need for debuginfos, a lot of informations gathered in one log file
- How?:
  - just run program "under" strace(or ltrace):
    strace ./whatever 2>mystrace
  - or attach strace(or ltrace) to existing process:
    strace -p PID 2>mystrace

oprofile, gdb, systemtap

- useful dynamic tools covered by other talks on DevConf2011
- oprofile:
  - powerful tool for profiling, finding most called functions
  - covered by parallel workshop by Ivana Hutařová Vařeková
- gdb:
  - well known debugging tool
  - covered on Saturday 13:20 in D3 by Jan Kratochvíl
- SystemTap:
  - system-wide probe/trace tool
  - covered today 15:00 in B007 by Petr Müller

Agenda

redhat

Static Analysis

- Already done by the compiler (various warnings)
- Dynamic vs. static analysis
- Problem with boundaries: dependencies, libraries
- False positives

Static Analysis Techniques

- Error patterns (missing break, etc.)
- Enhanced type checking
- Attributes, annotations
- Model checking
- Abstract interpretation

Static Analysis Techniques – Examples (1/2)

FORWARD_NULL
: If a pointer is checked against NULL, it should be checked before the pointer is first dereferenced.

RESOURCE_LEAK
: The last handle of a resource (a piece of allocated memory, file descriptor, mutex, etc.) is definitely lost at some point in the program before the resource is released.

USE_AFTER_FREE
: The program logic allows to access (dereference, free, . . .) an already freed memory. Common mistake in error handling code of libraries.

redhat

Static Analysis Techniques – Examples (2/2)

CHECKED_RETURN
: If the return value of a particular function is checked in the vast majority of calls of the function, then the return value should likely be always checked.

DEAD_CODE
: If a certain part of source code can never be reached during execution of the program, it usually implies that the program does not do what the programmer intended to do.

http://www-2.cs.cmu.edu/ aldrich/courses/654-sp09/tools/cure-coverity-06.pdf

Example of a Fixed Code Defect

- a hidden bug in the cUrl project found by static analysis
- http://github.com/bagder/curl/compare/62ef465...7aea2d5

```
diff --git a/lib/rtsp.c b/lib/rtsp.c
--- a/lib/rtsp.c
+++ b/lib/rtsp.c
@@ -709,7 +709,7 @@
     while(*start && ISSPACE(*start))
       start++;

-    if(!start) {
+    if(!*start) {
       failf(data, "Got a blank Session ID");
     }
     else if(data->set.str[STRING_RTSP_SESSION_ID]) {
```

redhat

Static Analysis Tools

- sparse
- Clang Static Analyzer
- Coverity
- gcc plug-ins
- FindBugs (for our Java friends)
- Splint (and lints in general)
- gazillion more, but of various usefulness

redhat

sparse

- Tiny project, c.c.a. 30 000 lines of code
- Able to analyze the whole Linux kernel
- `make CC=cgcc`
- Provided also as a library, which is available on Fedora
- Checks mostly useful for kernel (mixing user and kernel space pointers)

clang Static Analyzer

- LLVM frontend

1. scan-build ./configure ...
2. scan-build make
3. scan-view

Coverity

- enterprise tool, not freely available
- often used to analyse free software
- static analysis + abstract interpretation
- modular, various checkers (including the examples above)
- advanced statistical methods for elimination of false positives

Coverity – Examples of Fixed Bugs

- abrt
  - missing check of a return value
  - buffer overflow
  - memory leak
  - use of uninitialized value
  - https://bugzilla.redhat.com/628716
- attr
  - memory leak
  - double free
  - logical error
  - http://lists.gnu.org/archive/html/acl-devel/2010-06/msg00000.html

Coverity – Examples of Fixed Bugs

- `libucil`
  - 3× resource leak
  - `http://launchpadlibrarian.net/57222837/0001-libucil-fix-some-memory-leaks.patch`
- `libunicap`
  - 8× memory error
  - OOM state handling
  - `http://launchpadlibrarian.net/58529876/0002-libunicap-fix-various-memory-errors.patch`
- `libunicapgtk`
  - invalid use of a local variable
  - `https://bugs.launchpad.net/unicap/+bug/656232`

redhat

gcc plug-ins

- Easy to use (just add a flag to CFLAGS)
- No parsing errors
- No unrecognized gcc options
- One intermediate code used for both analysis and building

- Universal gcc plug-ins (DragonEgg, Dehydra, Treehydra, ...)
- You can write custom gcc plug-ins

FindBugs

- Finding bugs in Java
- Works over bytecode
- Designed to avoid false positives

Splint

- Works with C programs
- Traditional lint-like tool
- Additionally, it works with annotations

There are more, but

Lots of tools are capable, as there is scientific community around.
The catches are:

- Scientists tend to use obscure languages (OCaml, F#)
- Scientists tend to publish the unfinished business and move on

But sometimes:

- Companies hire scientists for the job (Intel, MS)
- Scientists try to get rich (Coverity, Monoidics)

Agenda

Why?

In our opinion:

- OS projects: many changes, no or small control
- Unwanted side-effects quite likely
- Tests are fine, but slow and you need resources
- Tests may pass even if program is broken
- FV tools are usually quite fast
- False positives not much an issue: just changes can be watched

How to use the tools?

- Package them in Fedora
- Use them
- Tie them into build processes
- Improve them
- Any ideas?
- . . .