redhat

Common Errors in C/C++ Code and Static Analysis

Red Hat
Ondřej Vašík and Kamil Dudka

2011-02-17

Abstract

Overview of common programming mistakes in the C/C++ code, and comparison of a few available static analysis tools that can detect them.

Agenda

1. **Static Analysis – What Does It Mean?**
2. **Common Code Weaknesses in `C/C++` Programs**
3. **Available Tools for Static Analysis**
4. **Beyond Static Analysis**

redhat

Static Analysis

- generic definition: analysis of code without executing it
- various kinds of tools – generic + specialized
- already done by the compiler (optimization, warnings, . . .)
- we are interested in using static analysis to find bugs

redhat

Static Analysis – Finding Bugs

- usually requires code that we are able to compile
- usually fast (time of analysis close to time of compilation)
- high level of automation

- can't cover all bugs in code
- problem with false positives
- any code change = risk of regressions

Static Analysis Techniques

- error patterns – missing break, stray semicolon, ...
- enhanced type checking – may use attributes, such as `__attribute__((address_space(num)))` in sparse
- data-flow analysis – solving of data flow equations, usually works at the CFG level
- abstract interpretation – evaluates a program for all possible inputs at once over an abstract domain

Agenda

1. Static Analysis – What Does It Mean?
2. **Common Code Weaknesses in `C/C++` Programs**
3. Available Tools for Static Analysis
4. Beyond Static Analysis

Common Code Weaknesses in C/C++ Programs

- CWE from MITRE (Common Weakness Enumeration) (http://cwe.mitre.org/data/definitions/398.html)

- static analysis is especially good for:
  - boundary checks
  - resource leak checks
  - memory safety checks
  - dead code checks
  - uninitialized/unused variables checks
  - race conditions / synchronization checks
  - various "coded by humans" issues

Boundary Problems

- static/dynamic buffer overflows/underflows (CWE-125, CWE-120, CWE-170, CWE-124)
- types incompatibilities (signed/unsigned) (CWE-194)
- signedness overflow issue that caused segfault with multivolumes in star:

```
 diff -urNp star-1.5.1-orig/star/buffer.c star-1.5.1/star/buffer.c
--- star-1.5.1-orig/star/buffer.c
+++ star-1.5.1/star/buffer.c
@@ -799,7 +799,7 @@ initbuf(nblocks)

  bigptr = bigbuf = ___malloc((size_t) bufsize+10+pagesize,
  "buffer");
- bigptr = bigbuf = (char *)roundup((Intptr_t)bigptr, pagesize);
+ bigptr = bigbuf = (char *)roundup((UIntptr_t)bigptr, pagesize);
  fillbytes(bigbuf, bufsize, '\0');
  fillbytes(&bigbuf[bufsize], 10, 'U');
```

redhat

Resource Leaks

- memory leaks (CWE-404)
- descriptor leaks (CWE-404)
- recent util-linux resource leak fix in libmount/src/utils.c

```
@@ -427,6 +427,7 @@
 static int get_filesystems(const char *filename, char ***filesystems, const char *pattern)
 {
+       int rc = 0;
        FILE *f;
        char line[128];
@@ -436,7 +437,6 @@
        while (fgets(line, sizeof(line), f)) {
                char name[sizeof(line)];
-               int rc;

                if (*line == '#' || strncmp(line, "nodev", 5) == 0)
                        continue;
@@ -446,9 +446,11 @@
                rc = add_filesystem(filesystems, name);
                if (rc)
-                       return rc;
+                       break;
        }
-       return 0;
+       fclose(f);
+       return rc;
 }
```

redhat

Memory Safety

- dereference null (CWE-476), use after free (CWE-416)
- double free (CWE-415), bad free (CWE-590)
- dual doublefree due to missing exit in policycoreutils(sepolgen-ifgen-attr-helper.c):

```
@@ -212,6 +213,7 @@ int main(int argc, char **argv)
    /* Open the output policy. */
    fp = fopen(argv[2], "w");
    if (fp == NULL) {
        fprintf(stderr, "error opening output file\n");
        policydb_destroy(p);
        free(p);
+       return -1;
    }

    ...

    policydb_destroy(p);
    free(p);
    fclose(fp);
```

redhat

Dead Code Checking

- unnecessary code (CWE-561)
- wrong error check (CWE-252, CWE-665, CWE-569)

Uninitialized/Unused Variables

- unnecessary variable handling (CWE-563)
- using unitialized variable (CWE-457, CWE-456)

redhat

Race Conditions / Synchronization Checks

- TOCTOU (time of check / time of use) (CWE-367)
- unsynchronized access to shared data in multithread apps (CWE-362)
- issues with locks (CWE-362)
- potential deadlock in kernel on fail path (https://lkml.org/lkml/2010/9/4/73)

```
diff --git a/drivers/net/bna/bnad.c b/drivers/net/bna/bnad.c
 @@ -2706,7 +2706,7 @@ bnad_set_rx_mode(struct net_device *netdev)
             kzalloc((mc_count + 1) * ETH_ALEN,
                 GFP_ATOMIC);
         if (!mcaddr_list)
-            return;
+            goto unlock;

         memcpy(&mcaddr_list[0], &bnad_bcast_addr[0], ETH_ALEN);

 @@ -2719,6 +2719,7 @@ bnad_set_rx_mode(struct net_device *netdev)
         /* Should we enable BNAD_CF_ALLMULTI for err != 0 ? */
         kfree(mcaddr_list);
     }
+unlock:
     spin_unlock_irqrestore(&bnad->bna_lock, flags);
 }
```

Various "coded by humans" Issues

- various cut&paste issues, missing breaks (CWE-484)
- priority of operators issues(CWE-569)
- stray semicolon after if (CWE-398)
- missing asterisks in pointer operations (CWE-476)
- 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]) {
```

Defensive Programming

- not really static analysis technique, but good habbit
- use compiler protection mechanisms
  - `D_FORTIFY_SOURCE=2`
  - stack-protector, PIE/PIC, RELRO, ExecShield
  - don't ignore warnings (-Wall -Wextra)

- never trust anyone, never expect anything
  - memory boundaries
  - check return codes/error codes
  - use descriptors
  - respect uid/gids, don't over escalate privileges

- `http://www.akkadia.org/drepper/defprogramming.pdf`

Agenda

1. Static Analysis – What Does It Mean?
2. Common Code Weaknesses in `C/C++` Programs
3. **Available Tools for Static Analysis**
4. Beyond Static Analysis

redhat

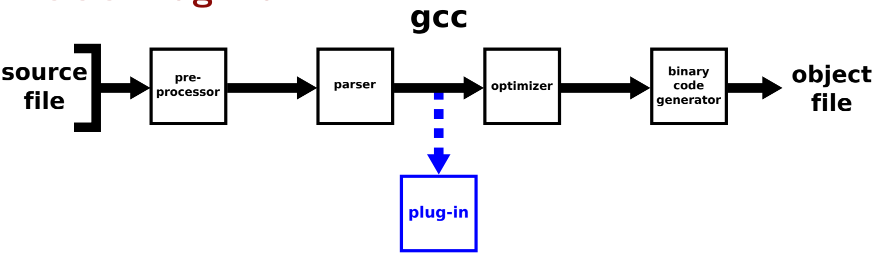
Available Tools for Static Analysis

- GCC – compile cleanly at high warning levels
- GCC plug-ins – suitable for projects natively compiled by GCC
- Clang Static Analyzer – uses LLVM Compiler Infrastructure
- sparse – developed and used by kernel maintainers (C only)
- cppcheck – easy to use, low rate of false positives

- commercial tools for static analysis (Coverity, . . .)
- research tools for static analysis (frama-c, . . .)
- some tools are not yet ready for industrial software (uno, splint, . . .)

redhat

GCC Plug-ins



- as easy to use as adding a flag to CFLAGS
- no parsing errors, no unrecognized compilation flags
- one intermediate code used for both analysis and building

- `repoquery --repoid=rawhide-source --arch=src --whatrequires gcc-plugin-devel`
- gcc-python-plugin – allows to write GCC plug-ins in python
- DragonEgg – allows to use LLVM as a GCC backend

Clang Static Analyzer

- based on LLVM Compiler Infrastructure
- code needs to compile with LLVM
- for an autoconf-based project, you can hook clang this way:
  1. `scan-build ./configure ...`
  2. `scan-build make`
  3. `scan-view ...`
- the steps above may fail on projects using some obscure build systems (e.g. ksh is known to cause problems)

redhat

sparse

- supports only C, not fully compatible with GCC
- able to analyze the whole Linux kernel
- provides a GCC wrapper called `cgcc` (`make CC=cgcc`)
- provides a library to build custom analyzers
- `sparse-llvm` – an LLVM front-end (still under development)

redhat

cppcheck

- uses its own parser and preprocessor
- reports only erorrs by default
- can be run directly on the sources (not always optimal)

- using the options `-D` and `-I` may help significantly
- `--template gcc` makes the output compatible with GCC
- `-jN` allows to run cppcheck in parallel

- [TODO: demo – libedit]

redhat

Coverity

- enterprise tool, not freely available
- often used to analyze free software – `http://scan.coverity.com`
- combination of all above mentioned static analysis techniques
- modular, various checkers
- advanced statistical methods for elimination of false positives

Coverity – How Do We Use It in Red Hat?

- scans of minor RHEL updates
  $\Longrightarrow$ prevent new defects introduced by backports and new features
- scans selected package set from Fedora Rawhide
  $\Longrightarrow$ packages with potential for next RHEL, working with upstream, to improve overall source code quality
  - 1500+ packages, 150M LoC, 170k pot. defects, 90% scan success rate
  - util-linux (70 bugs), ksh (50 bugs), e2fsprogs (40 bugs) and many other cleanups upstream based on scans
- scans of upstream projects developed by Red Hat
  $\Longrightarrow$ keeping upstream code quality at high level

Agenda

redhat

Beyond Static Analysis

- static analysis is a good bug-hunting technique, but what about false negatives?
- some properties are hard to check using static analysis only
  - absence of memory leaks
  - error label reachability
  - program termination
- software verification methods can guarantee zero false negatives for checking the properties above
- `http://sv-comp.sosy-lab.org/results/index.php`

redhat

Conclusion

- there is a lot of ready to use static analysis tools out there
- it is important not to rely on a single static analysis tool
- many of them are open source projects
- currently, the key problem of static analysis tools are false positives, which need to be filtered out manually