Static Analysis of a Linux Distribution

Kamil Dudka <kdudka@redhat.com>

March 22nd 2021

How to find programming mistakes efficiently?

0. users (preferably volunteers)
1. Automatic Bug Reporting Tool (ABRT)
2. code review, automated tests, fuzzing
3. static analysis

Why do we use static analysis at Red Hat?

- … to find programming mistakes soon enough – example:

```
Error: SHELLCHECK_WARNING:
/etc/rc.d/init.d/squid:136:10: warning: Use "${var:?}" to ensure this never expands to /* .
#  134|          RETVAL=$?
#  135|          if [ $RETVAL -eq 0 ] ; then
#  136|->                rm -rf $SQUID_PIDFILE_DIR/*
#  137|                  start
#  138|          else
```

https://bugzilla.redhat.com/1202858 – *[UNRELEASED] restarting testing build of squid results in deleting all files in hard-drive*

- Static analysis is required for Common Criteria certification.

Red Hat

Agenda

1. Code Review, Fuzzing
2. Linux Distribution, Reproducible Builds
3. Static Analysis of a Linux Distribution
4. Dynamic Analysis and Formal Verification

Red Hat

Code Review

- design (anti-)patterns
- error handling (OOM, permission denied, . . .)
- validation of input data (headers, length, encoding, . . .)
- sensitive data treatment (avoid exposing private keys, . . .)
- use of crypto algorithms
- resource management

Red Hat

Fuzzing

- Feeding programs with unusual input.
- Can be combined with valgrind, GCC sanitizers, etc.
- `radamsa` – general purpose data fuzzer

```
$ cat file | radamsa | program
```

- OSS-Fuzz – continuous fuzzing of open source software
  - service provided by Google
  - many security issues detected e.g. in curl

Agenda

1 Code Review, Fuzzing

2 **Linux Distribution, Reproducible Builds**

3 Static Analysis of a Linux Distribution

4 Dynamic Analysis and Formal Verification

Red Hat

Linux Distribution

- operating system (OS)
- based on the Linux kernel
- a lot of other programs running in user space
- usually open source

Red Hat

Upstream vs. Downstream

- Upstream SW projects – usually independent
- Downstream distribution of upstream SW projects
  - Red Hat uses the RPM package manager
  - Files on the file system owned by RPM packages:
    - Dependencies form an oriented graph over packages.
    - We can query package database.
    - We can verify installed packages.

Red Hat

Fedora vs. RHEL

- Fedora
  - new features available early
  - driven by the community (developers, users, ...)
- RHEL (Red Hat Enterprise Linux)
  - stability and security of existing deployments
  - driven by Red Hat (and its customers)

Red Hat

Where do RPM packages come from?

- Developers maintain source RPM packages (SRPMs).
- Binary RPMs can be built from SRPMs using `rpmbuild`:

```
rpmbuild --rebuild git-2.30.2-1.fc34.src.rpm
```

- Binary RPMs can be then installed on the system:

```
sudo dnf install git
```

Red Hat

Reproducible Builds

- Local builds are not reproducible.
- `mock` – chroot-based tool for building RPMs:

```
mock -r fedora-rawhide-x86_64 git-2.30.2-1.fc34.src.rpm
```

- `koji` – service for scheduling build tasks

```
koji build rawhide git-2.30.2-1.fc34.src.rpm
```

- Easy to hook static analyzers on the build process!

Red Hat

Agenda

1 Code Review, Fuzzing

2 Linux Distribution, Reproducible Builds

3 Static Analysis of a Linux Distribution

4 Dynamic Analysis and Formal Verification

Red Hat

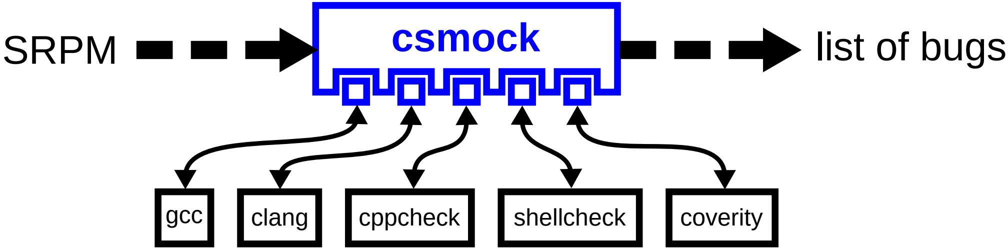
Static Analysis at Red Hat in Numbers

- Preliminary scan of all RHEL-9 packages in February 2021.
- Analyzed 480 million LoC (Lines of Code) in 3700 packages.
- 98.6 % packages scanned successfully.
- Approx. 680 000 potential bugs detected in total.
- Approx. one potential bug per each 750 LoC.

Red Hat

Analysis of RPM Packages

- Command-line tool to run static analyzers on RPM packages.
- One interface, one output format, plug-in API for (static) analyzers.
- Fully open-source, available in Fedora and CentOS.



Red Hat

csmock – Supported Static Analyzers

	C	C++	C#	Java	Go	JavaScript	PHP	Python	Ruby	Shell
gcc	✓	✓								
gcc -fanalyzer	✓									
clang --analyze	✓	✓								
cppcheck	✓	✓								
coverity	✓	✓	✓	✓	✓	✓	✓	✓	✓	
shellcheck										✓
pylint								✓		
bandit								✓		
smatch	✓									

Need more?

https://github.com/mre/awesome-static-analysis#user-content-programming-languages-1

Red Hat

What is important for developers?

The static analyzers need to:

- be fully automatic
- provide reasonable signal to noise ratio
- provide reproducible and consistent results
- be approximately as fast as compilation of the package
- support differential scans:
  - added/fixed bugs in an update?
  - https://github.com/kdudka/csdiff

Red Hat

csmock – Output Format

```
Error: RESOURCE_LEAK (CWE-772):
src/fptr.c:450: alloc_fn: Storage is returned from allocation function "calloc".
src/fptr.c:450: var_assign: Assigning: "e" = storage returned from "calloc(24UL, 1UL)".
src/fptr.c:450: overwrite_var: Overwriting "e" in "e = calloc(24UL, 1UL)" leaks the storage that "e" points to.
#  448|       if ((f = (struct opd_fptr *) l->u.refp[i]->ent)->ent == NULL)
#  449|         {
#  450|->       e = calloc (sizeof (struct opd_ent), 1);
#  451|         if (e == NULL)
#  452|           {

Error: CPPCHECK_WARNING (CWE-401):
src/fptr.c:464: error[memleak]: Memory leak: e
#  462|         }
#  463|
#  464|->   return ret;
#  465|   }

Error: RESOURCE_LEAK (CWE-772):
src/fptr.c:450: alloc_fn: Storage is returned from allocation function "calloc".
src/fptr.c:450: var_assign: Assigning: "e" = storage returned from "calloc(24UL, 1UL)".
src/fptr.c:464: leaked_storage: Variable "e" going out of scope leaks the storage it points to.
#  462|         }
#  463|
#  464|->   return ret;
#  465|   }
```

Red Hat

csmock – Output Format

```
Error: RESOURCE_LEAK (CWE-772):
src/fptr.c:450: alloc_fn: Storage is returned from allocation function "calloc".
src/fptr.c:450: var_assign: Assigning: "e" = storage returned from "calloc(24UL, 1UL)".
src/fptr.c:450: overwrite_var: Overwriting "e" in "e = calloc(24UL, 1UL)" leaks the storage that "e" points to.
#  448|       if ((f = (struct opd_fptr *) ]->u.refp[i]->ent)->ent == NULL)
#  449|         {
#  450|->        e = calloc (sizeof (struct opd_ent), 1);
#  451|         if (e == NULL)
#  452|           {

Error: CPPCHECK_WARNING (CWE-401):
src/fptr.c:464: error[memleak]: Memory leak: e
#  462|         }
#  463|
#  464|->   return ret;
#  465|   }

Error: RESOURCE_LEAK (CWE-772):
src/fptr.c:450: alloc_fn: Storage is returned from allocation function "calloc".
src/fptr.c:450: var_assign: Assigning: "e" = storage returned from "calloc(24UL, 1UL)".
src/fptr.c:464: leaked_storage: Variable "e" going out of scope leaks the storage it points to.
#  462|         }
#  463|
#  464|->   return ret;
#  465|   }
```

checker

key event

CWE ID

location info

other events

message associated with the key event

Red Hat

csmock – Output Format (Trace Events)

```
Error: RESOURCE_LEAK (CWE-772):
src/fptr.c:447: cond_true: Condition "i < l->nrefs", taking true branch.
src/fptr.c:448: cond_true: Condition "(f = (struct opd_fptr *)l->u.refp[i]->ent)->ent == NULL", taking true branch.
src/fptr.c:450: alloc_fn: Storage is returned from allocation function "calloc".
src/fptr.c:450: var_assign: Assigning: "e" = storage returned from "calloc(24UL, 1UL)".
src/fptr.c:451: cond_false: Condition "e == NULL", taking false branch.
src/fptr.c:456: if_end: End of if statement.
src/fptr.c:462: loop: Jumping back to the beginning of the loop.
src/fptr.c:447: loop_begin: Jumped back to beginning of loop.
src/fptr.c:447: cond_true: Condition "i < l->nrefs", taking true branch.
src/fptr.c:448: cond_true: Condition "(f = (struct opd_fptr *)l->u.refp[i]->ent)->ent == NULL", taking true branch.
src/fptr.c:450: overwrite_var: Overwriting "e" in "e = calloc(24UL, 1UL)" leaks the storage that "e" points to.
#  448|      if ((f = (struct opd_fptr *) l->u.refp[i]->ent)->ent == NULL)
#  449|        {
#  450|->        e = calloc (sizeof (struct opd_ent), 1);
#  451|        if (e == NULL)
#  452|          {
```

Red Hat

Example of a Fix

```
--- a/src/fptr.c
+++ b/src/fptr.c
@@ -438,28 +438,29 @@
 GElf_Addr
 opd_size (struct prelink_info *info, GElf_Word entsize)
 {
   struct opd_lib *l = info->ent->opd;
   int i;
   GElf_Addr ret = 0;
   struct opd_ent *e;
   struct opd_fptr *f;

   for (i = 0; i < l->nrefs; ++i)
     if ((f = (struct opd_fptr *) l->u.refp[i]->ent)->ent == NULL)
       {
        e = calloc (sizeof (struct opd_ent), 1);
        if (e == NULL)
          {
            error (0, ENOMEM, "%s: Could not create OPD table",
                   info->ent->filename);
            return -1;
          }

        e->val = f->val;
        e->gp = f->gp;
        e->opd = ret | OPD_ENT_NEW;
+       f->ent = e;
        ret += entsize;
       }

   return ret;
 }
```

Red Hat

Upstream vs. Enterprise

Different approaches to static analysis:

- Upstream
  - Fix as many bugs as possible.
  - False positive ratio increases over time!
- Enterprise
  - Run differential scans to verify code changes.
  - Up to 10% of bugs usually detected as new in an update.
  - Up to 10% of them usually confirmed as real by developers.

Red Hat

Agenda

1 Code Review, Fuzzing

2 Linux Distribution, Reproducible Builds

3 Static Analysis of a Linux Distribution

4 Dynamic Analysis and Formal Verification

Red Hat

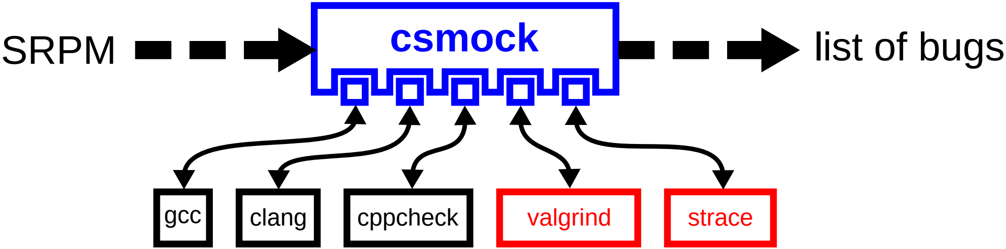
Dynamic Analysis

- Executes code in a modified run-time environment.
- Embedded in compilers: address sanitizer, thread sanitizer, UB sanitizer, ...
- Standalone tools: valgrind, strace, ...
- Not so easy to automate as static analysis.
- Good to have some test-suite to begin with.

Red Hat

Dynamic Analysis of RPM Packages

- Experimental csmock plug-ins for valgrind and strace:




```
$ sudo yum install csmock-plugin-valgrind
$ csmock -t valgrind -r fedora-rawhide-x86_64 *.src.rpm
```

Red Hat

Tests Embedded in RPM Packages

```
$ fedpkg clone -a logrotate
$ cd logrotate
$ grep -A8 '%build' logrotate.spec
%build
mkdir build && cd build
%global _configure ../configure
%configure --with-state-file-path=%{_localstatedir}/lib/logrotate/logrotate.status
%make_build

%check
%make_build -C build -s check

$ fedpkg srpm
$ rpmbuild --rebuild *.src.rpm
```

Red Hat

Dynamic Analysis of RPM Packages – Simple Approach

- Dynamic analyzers usually support tracing of child processes.
- Let's combine it together:
  - `valgrind --trace-children=yes rpmbuild --rebuild *.src.rpm`
  - `strace --follow-forks rpmbuild --rebuild *.src.rpm`
- But did we want to dynamically analyze rpmbuild, bash, make, etc.?
  - This makes the analysis extremely slow.
  - We get reports unrelated to `*.src.rpm`.

Red Hat

Dynamic Analysis of RPM Packages – Better Approach

- Produce binaries that will launch a dynamic analyzer for themselves.
- We can use a compiler wrapper to instrument the build of an RPM package:

```
$ export PATH=$(cswrap --print-path-to-wrap):$PATH
$ export CSWRAP_ADD_CFLAGS=-Wl,--dynamic-linker,/usr/bin/csexec-loader
$ export CSEXEC_WRAP_CMD=valgrind
$ rpmbuild --rebuild *.src.rpm
```

- Only binaries produced in `%build` will run through valgrind in `%check`.

Red Hat

Program Interpreter

- Program interpreter specified by shebang:

```
$ head -1 /usr/bin/yum
#!/usr/bin/python3
```

```
$ /usr/bin/yum [...]  ⟶  /usr/bin/python3 /usr/bin/yum [...]
```

- Program interpreter specified by ELF header:

```
$ file /sbin/logrotate
/sbin/logrotate: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=...
```

- ELF interpreter can be set to a custom value when linking the binary:

```
$ file ./logrotate
./logrotate: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /usr/bin/csexec-loader, BuildID[sha1]=...
```

Red Hat

Wrapper of Dynamic Linker – Implementation

- `csexec` works as a wrapper of the system dynamic linker:
  https://github.com/kdudka/cswrap/wiki/csexec
- `$CSEXEC_WRAP_CMD` can specify a dynamic analyzer to use.
- csexec runs the system dynamic linker explicitly (to eliminate self-loop):
  `./logrotate [...]` ⟶ `valgrind /lib64/ld-linux-x86-64.so.2 ./logrotate [...]`
- csexec uses the `--argv0` option of the system dynamic linker if available:
  https://sourceware.org/git/?p=glibc.git;a=commitdiff;h=c6702789
- csexec emulates the original target of the `/proc/self/exe` symlink.

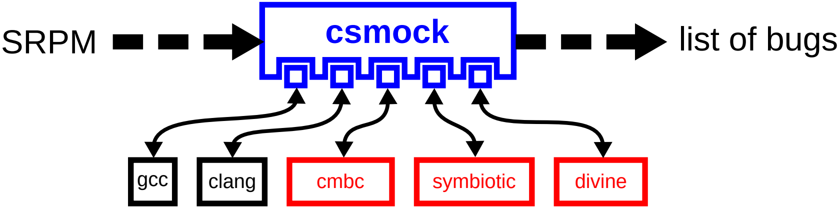
Red Hat

Wrapper of Dynamic Linker – Evaluation

- No completely unrelated bug reports.
- Minimal performance overhead.
- Minimal interference with commonly used testing frameworks.
- Able to successfully run upstream test-suite of GNU coreutils (without valgrind).
- Some tests fail if we wrap them by valgrind though:
  - a test that verifies the count open file descriptors
  - a test that intentionally sets non-existing `$TMPDIR`
  - ...
- [TODO: demo]

Red Hat

Formal Verification of RPM Packages



- AUFOVER (Automation of Formal Verification) project, supported by Technology Agency of the Czech Republic:
  https://starfos.tacr.cz/en/project/TH04010192
- SV-COMP (Competition on Software Verification):
  https://sv-comp.sosy-lab.org/2021/results/results-verified/

Red Hat

Slides Available Online

https://kdudka.fedorapeople.org/muni21.pdf