Static Analysis of a Linux Distribution

Kamil Dudka <kdudka@redhat.com>
Red Hat, Inc.
November 27th 2018

How to find programming mistakes efficiently?

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

Agenda

1. Code Review
2. Dynamic Analysis
3. Static Analysis
4. Linux Distribution
5. Static Analysis of a Linux Distribution

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

redhat

Dynamic Analysis

- good to have some test-suite to begin with
- memory error detectors, profilers, e.g. valgrind
- tools to measure test coverage, e.g. gcov/lcov
- compiler instrumentation, e.g. GCC built-in sanitizers (address sanitizer, thread sanitizer, UB sanitizer, . . .)
- not so easy to automate as static analysis

redhat

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

redhat

Static Analysis

- does not need to run the code
- does not need any test-suite
- can detect (potential) bugs fully automatically

redhat

Example – A Defect Found by ShellCheck

```
Error: SHELLCHECK_WARNING: [#def4]
/etc/rc.d/init.d/squid:136:10: warning: Use "${var:?}" to ensure this never expands to /* . [SC2115]
#  134|          RETVAL=$?
#  135|          if [ $RETVAL -eq 0 ] ; then
#  136|->                rm -rf $SQUID_PIDFILE_DIR/*
#  137|                  start
#  138|          else
```

https://github.com/koalaman/shellcheck/wiki/SC2115

Agenda

1 Code Review

2 Dynamic Analysis

3 Static Analysis

4 **Linux Distribution**

5 Static Analysis of a Linux Distribution

Linux Distribution

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

redhat

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 packages:
    - dependencies form an oriented graph over packages
    - we can query package database
    - we can verify installed packages

Fedora vs. RHEL

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

redhat

Where do RPM packages come from?

- developers maintain source RPM packages (SRPMs)
- binary RPMs can be built from SRPMs using `rpmbuild`:

```
rpmbuild --rebuild git-2.6.3-1.fc24.src.rpm
```

- binary RPMs can be then installed on the system:

```
sudo dnf install git
```

redhat

Reproducible Builds

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

```
mock -r fedora-rawhide-i386 git-2.6.3-1.fc24.src.rpm
```

- `koji` – service for scheduling build tasks

```
koji build rawhide git-2.6.3-1.fc24.src.rpm
```

- easy to hook static analyzers on the build process!

redhat

Reproducible Builds – Obstacles

- build env not 100% isolated from host env
- toolchain (compiler, linker, glibc, . . .) evolves
- parallel builds with missing dependencies (tricky to debug)
- installation of binary RPMs not (always) reproducible
- too many unexpected side effects – examples:
  - SMTP server fails to build on up2date kernel
  - one-line change of a man page doubles size of curl binary
  - cookies and certificates in curl upstream test-suite expire
  - autoconf tests: https://github.com/curl/curl/commit/curl-7_49_1-45-gb2dcf0347

redhat.

Reproducible Builds – Best Practices

- use `git archive` to create tarballs
  (does not work well with autotools)
- isolate build env from host env
  (chroot, mock, containers, VMs)
- do not use compiler flags like `-mtune=native`
- disable Internet acess during the build
- sign release tags and release tarballs

Agenda

1. Code Review
2. Dynamic Analysis
3. Static Analysis
4. Linux Distribution
5. **Static Analysis of a Linux Distribution**

redhat

Static Analysis of a Linux Distribution (1/2)

- RHEL-8 Beta released on November 14th 2018
- RHEL-8 Beta static analysis mass in July 2018
- analyzed 318 million LoC (Lines of Code) in 3390 packages
- 95.6% packages scanned successfully
- approx. 370 000 potential bugs reported in total
- approx. one potential bug per 1000 LoC

Static Analysis of a Linux Distribution (2/2)

- huge number of potential bugs, especially in some packages
- packages are developed independently of each other
- no control over programming languages and coding style
- code annotations (or even fixes) rejected by some upstreams
- ignored reports sometimes result in security issues later on

redhat.

Which static analyzers?

- some analyzers are tweaked for a particular project (e.g. sparse for kernel)
- Relying on a single static analyzer is insufficient!
- How to use multiple static analyzers easily?
- The csmock tool provides a common interface to GCC, Clang, Cppcheck, Shellcheck, Pylint, Bandit, Smatch, and Coverity.
- Coverity primarily analyzes C/C++, C#, and Java but also supports dynamic languages (JavaScript, PHP, Python, Ruby).

redhat

Example – Defects Found by Coverity Analysis

```
Error: NESTING_INDENT_MISMATCH: [#def1]
infinipath-psm-3.3-19_g67c0807_open/psm_diags.c:284: parent: This 'if'  statement is the parent, indented to column 5.
infinipath-psm-3.3-19_g67c0807_open/psm_diags.c:285: nephew: This 'if'  statement is nested within its parent, indented to column 7.
infinipath-psm-3.3-19_g67c0807_open/psm_diags.c:286: uncle: This 'if'  statement is indented to column 7, as if it were nested
within the preceding parent statement, but it is not.
#  284|       if (src == NULL || dst == NULL)
#  285|         if (src) psmi_free(src);
#  286|->       if (dst) psmi_free(dst);
#  287|         return -1;
#  288|     }

Error: COPY_PASTE_ERROR (CWE-398): [#def2]
gnome-shell-3.14.4/js/ui/boxpointer.js:517: original: "resX -= x2 - arrowOrigin" looks like the original copy.
gnome-shell-3.14.4/js/ui/boxpointer.js:536: copy_paste_error: "resX" in "resX -= y2 - arrowOrigin" looks like a copy-paste error.
gnome-shell-3.14.4/js/ui/boxpointer.js:536: remediation: Should it say "resY" instead?
#  534|               } else if (arrowOrigin >= (y2 - (borderRadius + halfBase))) {
#  535|                   if (arrowOrigin < y2)
#  536|->                      resX -= (y2 - arrowOrigin);
#  537|                   arrowOrigin = y2;
#  538|               }

Error: IDENTIFIER_TYPO: [#def3]
anaconda-21.48.22.90/pyanaconda/ui/gui/spokes/source.py:1388: identifier_typo: Using "mirorlist" appears to be a typo:
* Identifier "mirorlist" is only known to be referenced here, or in copies of this code.
* Identifier "mirrorlist" is referenced elsewhere at least 27 times.
anaconda-21.48.22.90/pyanaconda/packaging/__init__.py:1046: identifier_use: Example 1: Using identifier "mirrorlist".
anaconda-21.48.22.90/pyanaconda/packaging/yumpayload.py:732: identifier_use: Example 2: Using identifier "mirrorlist".
anaconda-21.48.22.90/pyanaconda/packaging/yumpayload.py:879: identifier_use: Example 3: Using identifier "mirrorlist".
anaconda-21.48.22.90/pyanaconda/packaging/yumpayload.py:726: identifier_use: Example 4: Using identifier "mirrorlist".
anaconda-21.48.22.90/pyanaconda/packaging/yumpayload.py:335: identifier_use: Example 5: Using identifier "mirrorlist".
anaconda-21.48.22.90/pyanaconda/ui/gui/spokes/source.py:1388: remediation: Should identifier "mirorlist" be replaced by "mirrorlist"?
# 1386|           url = self._repoUrlEntry.get_text().strip()
# 1387|           if self._repoMirrorlistCheckbox.get_active():
# 1388|->              repo.mirorlist = proto + url
# 1389|           else:
# 1390|              repo.baseurl = proto + url
```

What is important for developers?

The static analysis tools need to:

- be fully automatic
- provide reasonable signal to noise ratio
- results need to be reproducible and consistent
- be approximately as fast as compilation of the package

redhat

Priority Assessment Problem

- developers say:

  "I have 200+ already known bugs in my project waiting for a fix. Why should I care about additional bugs that users are not aware of yet?"

- not all bugs are equally important to be fixed!
- scoring systems like CWE (Common Weakness Enumeration)
- ... but none of them is universally applicable

Differential scans

- our packages contain a lot of potential bugs
- risk of creating new bugs while trying to fix existing bugs
- Which bugs were added/fixed in an update of something?

Example – Differential Scan of logrotate (1/2)

- On September 19 someone opened a pull request for logrotate (https://github.com/logrotate/logrotate/pull/146):

  ```
  logrotate.c:251:15: warning: Result of 'malloc' is converted
  to a pointer of type 'struct logStates', which is incompatible
  with sizeof operand type 'struct logState'
  ```

- On September 20 we agreed on a fix and pushed it (https://github.com/logrotate/logrotate/pull/149):

- Release of `logrotate-3.13.0` scheduled on October 13th. . .

redhat

Example – Differential Scan of logrotate (2/2)

- On October 12th (a day before the release) I ran a differential scan with the csbuild utility – demo:

```
git clone https://github.com/logrotate/logrotate.git
cd logrotate && git reset --hard eb322705^
autoreconf -fiv && ./configure
BUILD_CMD='make clean && make -j9'
csbuild -c $BUILD_CMD -g 3.12.3..master --git-bisect
```

- Luckily, I was able to fix it properly before the release (https://github.com/logrotate/logrotate/commit/eb322705):

```
csbuild -c $BUILD_CMD -g origin..master --print-fixed
```

Upstream vs. Enterprise

different approaches to static analysis:

upstream – fix as many bugs as possible

- false positive ratio increases over time!

enterprise – verify code changes in legacy SW

- up to 10% of bugs usually detected as new in an update
- up to 10% of them usually confirmed as real by developers

redhat.

Continuous Integration

- it is expensive to fix bugs detected late in the release cycle
- it is difficult and risky to fix bugs in already released products
- we would like to catch bugs at the time they are created
- an example using the csbuild utility:

```
csbuild --install 'automake libpopt-devel'          \
        --prep-cmd 'autoreconf -fiv && ./configure' \
        --build-cmd 'make clean && make -j9'        \
        --git-bisect --gen-travis-yml > .travis.yml

git add .travis.yml
git commit -m "notify me about newly introduced defects"
git push
```

Slides Available Online

https://kdudka.fedorapeople.org/muni18.pdf