Static Analysis of a Linux Distribution

Red Hat
Kamil Dudka
March 13th 2024

Red Hat

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.

Agenda

1. Linux Distribution, Reproducible Builds
2. Static Analysis of a Linux Distribution
3. Dynamic Analysis of a Linux Distribution
4. Static Analysis Results Interchange Format (SARIF)
5. OpenScanHub (OSH)

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-2.39.2-1.fc39.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-2.39.2-1.fc39.src.rpm
```

- koji – service for scheduling build tasks

```
koji build rawhide git-2.39.2-1.fc39.src.rpm
```

- Easy to hook static analyzers on the build process!
- Who cares about reproducible builds?
  https://reproducible-builds.org/who/projects/

Agenda

Red Hat

Static Analysis of a Linux Distribution

- Thousands of packages developed independently of each other.
- Huge number of (potential?) defects in certain projects.
- No control over technologies and programming languages.
- No control over upstream coding style.
- There is no person that would be familiar with all the code of a big project.

Red Hat

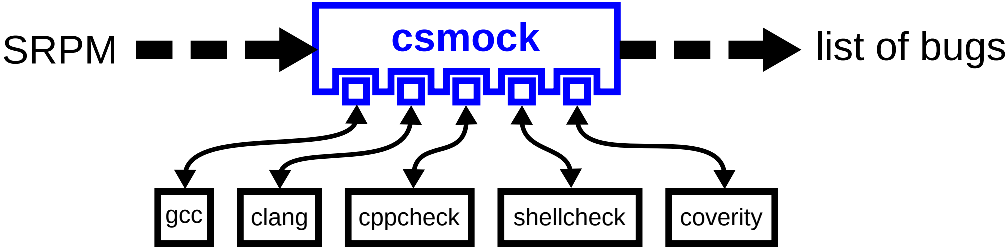
Static Analysis at Red Hat in Numbers

- 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 Stream.



csmock – Supported Static Analyzers

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

How could we fix all the 3 findings?

```
--- 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.

Agenda

1. Linux Distribution, Reproducible Builds
2. Static Analysis of a Linux Distribution
3. **Dynamic Analysis of a Linux Distribution**
4. Static Analysis Results Interchange Format (SARIF)
5. OpenScanHub (OSH)

Red Hat

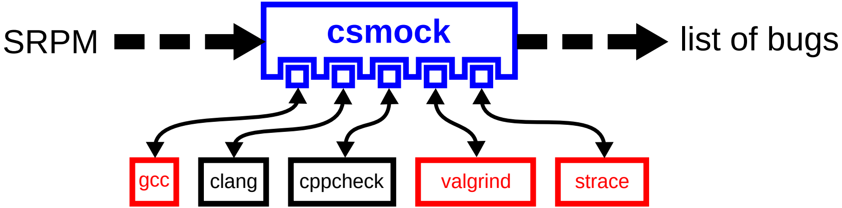
Dynamic Analysis

- 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 support for GCC sanitizers:
  https://github.com/csutils/csmock/pull/87
- csmock plug-ins for valgrind and strace:




```
$ sudo dnf 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 -A6 '%build' logrotate.spec
%build
%configure
%make_build

%check
%make_build 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/dnf
#!/usr/bin/python3
```

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

- 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/csutils/cswrap/wiki/csexec
- `$CSEXEC_WRAP_CMD` can specify a dynamic analyzer to use.
- If the variable is unset, the binaries are executed natively.
- csexec runs the system dynamic linker explicitly (to eliminate self-loop): `./logrotate [...]` ⟶ `valgrind /lib64/ld-linux-x86-64.so.2 ./logrotate [...]`

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`
  - ...

Red Hat

Agenda

1. Linux Distribution, Reproducible Builds
2. Static Analysis of a Linux Distribution
3. Dynamic Analysis of a Linux Distribution
4. **Static Analysis Results Interchange Format (SARIF)**
5. OpenScanHub (OSH)

Red Hat

Human-Readable Output Formats

- GCC's default output format is both human and machine-readable.

```
encode.c: In function 'th_set_path':
encode.c:91:17: warning: use of possibly-NULL '*t.th_buf.gnu_longname' where non-null expected [CWE-690] [-Wanalyzer-possible-null-argument]
encode.c:87:12: note: (1) following 'true' branch...
encode.c:90:42: note: (2) ...to here
encode.c:90:42: note: (3) this call could return NULL
encode.c:91:17: note: (4) argument 2 ('strdup(pathname)') from (3) could be NULL where non-null expected
```

- Supported by csdiff and IDEs (Integrated Development Environments).
- csdiff's parser needs to be tweaked for new versions of GCC (and other tools with GCC-compatible output format).
- Some tools produce human-redable output not suitable for parsing.

Red Hat

Machine-Readable Output Formats

- Usually based on JSON (GCC, ShellCheck) or XML (Cppcheck, Valgrind).
- Example – native JSON format supported by GCC-9 and newer:


```
[{"kind": "warning", "locations": [{"finish": {"byte-column": 60, "display-column": 74, "line": 91, "file": "encode.c", "column": 74},
"caret": {"byte-column": 3, "display-column": 17, "line": 91, "file": "encode.c", "column": 17}}], "path": [{"location":
{"byte-column": 5, "display-column": 12, "line": 87, "file": "encode.c", "column": 12}, "description": "following 'true' branch...",
"depth": 1, "function": "th_set_path"}, {"location": {"byte-column": 28, "display-column": 42, "line": 90, "file": "encode.c", "column": 42},
"description": "...to here", "depth": 1, "function": "th_set_path"}, {"location": {"byte-column": 28, "display-column": 42, "line": 90,
"file": "encode.c", "column": 42}, "description": "this call could return NULL", "depth": 1, "function": "th_set_path"},
{"location": {"byte-column": 3, "display-column": 17, "line": 91, "file": "encode.c", "column": 17},
"description": "argument 2 ('strdup(pathname)') from (3) could be NULL where non-null expected", "depth": 1, "function": "th_set_path"}],
"column-origin": 1, "option": "-Wanalyzer-possible-null-argument", "escape-source": false, "children": [{"kind": "note", "escape-source": false,
"locations": [{"finish": {"byte-column": 20, "display-column": 20, "line": 144, "file": "/usr/include/string.h", "column": 20}, "caret":
{"byte-column": 14, "display-column": 14, "line": 144, "file": "/usr/include/string.h", "column": 14}}],
"message": "argument 2 of 'strncpy' must be non-null"}],
"option_url": "https://gcc.gnu.org/onlinedocs/gcc/Static-Analyzer-Options.html#index-Wanalyzer-possible-null-argument",
"message": "use of possibly-NULL '*t.th_buf.gnu_longname' where non-null expected", "metadata": {"cwe": 690}}]
```


- These formats are not human-readable.
- Each tool uses its own JSON/XML scheme.

Red Hat

Static Analysis Results Interchange Format (SARIF)

- JSON-based data format standardized by OASIS:
  https://docs.oasis-open.org/sarif/sarif/v2.1.0/os/sarif-v2.1.0-os.html
- Extremely complex:
  - Tree structure with excessive nesting and cross-references.
  - Wastes bandwidth and memory.
  - Multiple ways to express the same thing.
  - Different tools/services implement it differently.
- Supported by csdiff as both input and output data format.
- Supported by GitHub and used by various GitHub Actions, e.g.:
  https://github.com/marketplace/actions/differential-shellcheck

Agenda

Red Hat

OpenScanHub (OSH)

- A service for fully automated static analysis based on csdiff and csmock.
- Developed and successfully used by developers at Red Hat since 2011.
- Transitioned into a fully open-source community project in 2023: https://openscanhub.dev/
- A publicly available instance is going to be deployed for Fedora: https://lists.fedoraproject.org/archives/list/devel@lists.fedoraproject.org/message/OMKLJFW4VC242QSA7R4KMGI6IGBT3YLM/

Slides Available Online

https://kdudka.fedorapeople.org/muni24.pdf