Coverity White Paper

coverity®

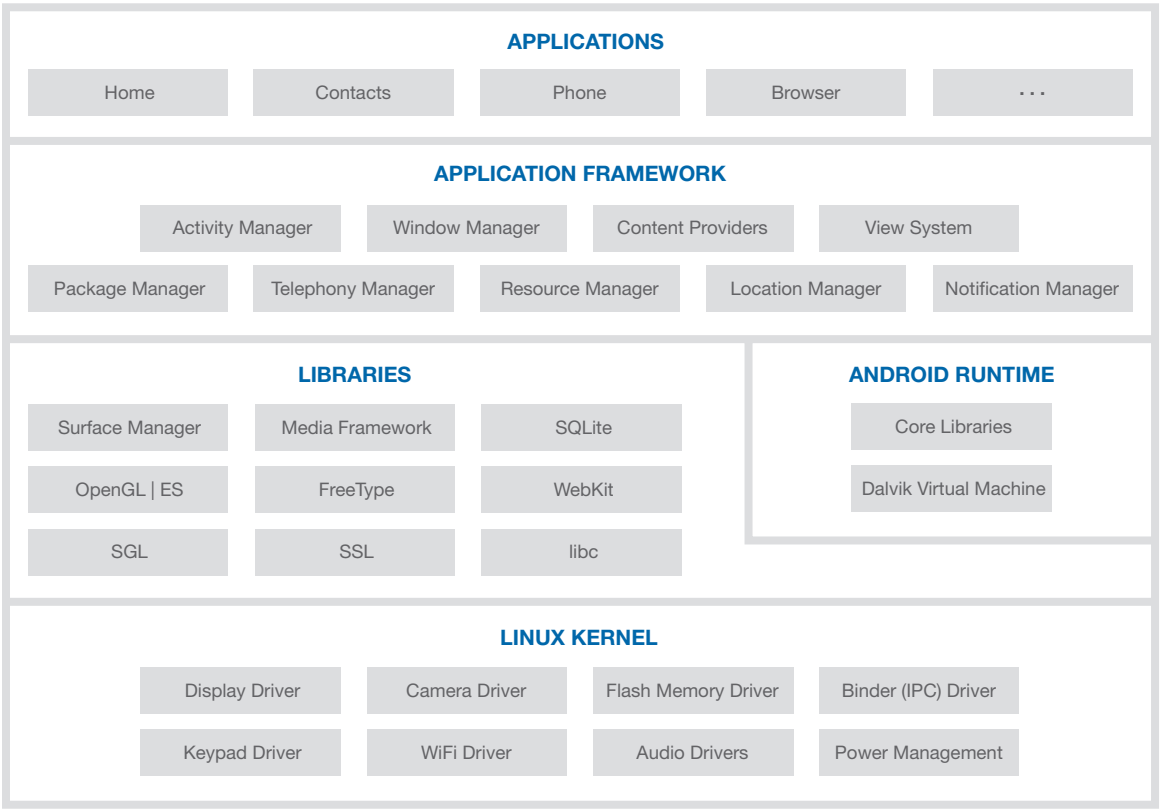
A Look Inside the Android Kernel with Automated Code Testing

March 2011

The Android Architecture – Bringing Innovative Devices to Market Faster

Android is the fastest growing mobile platform currently. In January 2011, comScore Inc, a market research company, reported that the number of Android users now exceeds the number of iPhone users. The success of Android has a lot to do with the philosophy with which it has been built; create a platform that is open and portable on a variety of devices. The Android stack (Figure 1), with the open source kernel and application framework makes it possible for a large number of OEMs (Original Equipment Manufacturers, such as phone manufacturers) to port it on their devices and allow application developers to write applications.

Figure 1: Android stack



Most major smart phone OEMs including HTC, Samsung, Motorola have taken advantage of this open architecture to bring new and innovative Smartphone devices to market. With the rapid adoption of Android also comes additional responsibility to ensure the code that OEMs are integrating into their development projects is held to the same quality and security standards as in-house code. A critical defect gone undetected in development and discovered in the field could have catastrophic consequences to time to market, revenues, profitability, and brand. The potential risks have led OEM development teams to turn to technology solutions, such as automated code testing via static analysis, to get visibility into the code they are shipping in their product, and under their brand.

Android Kernel Code Testing

In November 2010, Coverity tested the Android kernel (2.6.32 "Froyo") shipping in the HTC Droid Incredible phone as a part of the Coverity Scan 2010 Open Source Integrity Report (http://scan.coverity.com). Coverity's Scan service tests code for over 290 of the most popular open source projects, including Linux, Apache and Firefox. Since its inception in 2006, Coverity Scan has identified almost 50,000 defects in open source and has worked with the open source development community to fix over 15,000 defects. With this year's report, we decided to provide project level visibility into the Android kernel to help OEMs understand what they might be shipping in their product.

Specific to our test of the Android kernel 2.6.32, the open source kernel available at http://developer.htc.com was compiled and analyzed with the latest release of Coverity® Static Analysis. We then reviewed the results internally and worked with the Google Android developers to validate the existence of the defects. Because of the expansive Android development community— a supply chain in itself— what started as a discussion with the OEM and Google extended all the way out to the Linux developer community and other third-party contributors.

Figure 2: The Android kernel code testing process

The results revealed high risk defects such as memory corruptions, illegal memory accesses, resource leaks and uninitialized variables. The reason these types of errors are considered high risk is because of the impact they have on the Android device running this code. The worst offenders typically crash the program or the system, or result in unpredictable behavior. The Common Weakness Enumeration (CWE) list at http://cwe.mitre.org maps these defects to the potential security threats. We also noticed that some defects were unique to the different branches, and a large number of them were common and existed in all of the branches.

Defect Examples

This section looks at two defects found in the analysis.

Defect 1: Use of an uninitialized variable.
The following code is in one of the Android-specific driver file:

```
At conditional (28): "fp->cookie != node->cookie" taking the true branch.
1607                    if (fp->cookie != node->cookie) {
1608                            binder_user_error("binder: %d:%d sending u%p "
1609                                    "node %d, cookie mismatch %p != %p\n",
1610                                    proc->pid, thread->pid,
1611                                    fp->binder, node->debug_id,
1612                                    fp->cookie, node->cookie);
1613                            goto err_binder_get_ref_for_node_failed;
1614                    }
```

At the end of the function, the code is:

```
Using uninitialized value "return_error".
1783            thread->return_error = return_error;
1784
```

Coverity's analysis reported a defect of use of an uninitialized variable. One of the strengths of automated code testing via static analysis is assessing the possible paths that program execution can take without having to run the code. The analysis engine evaluates that on line 1607, if taking the true path in the 'if' conditional statement, the code fails to assign a value to the 'return_error' variable. Eventually when the code jumps to line 1783, the variable is used to assign a value to 'thread->return_error'.

Because an uninitialized variable will most likely contain an arbitrary value from an earlier computation, the system will not behave as expected. Coverity's analysis also tells you that this defect maps to CWE-457: Use of Uninitialized variable. CWE describes the likely hood of a security exploit because of this error high because this can lead to denial of service conditions, or modify control in unexpected ways.

This is one of the defects that Coverity found in the Android kernel used in the HTC Droid Incredible phone, and also in a subsequent test of the Google Android MSM Development Branch.

Defect 2: Dereference before a null check.
Another interesting defect is in the yaffs file system code. This defect is shared with the Linux kernel as well since the Android code is derived from the Linux kernel.

Directly dereferencing pointer "obj".

```
1449        dev = obj->myDev;
1450
1451        yaffs_GrossLock(dev);
1452
1453        inode = f->f_dentry->d_inode;
1454
1455        if (!S_ISBLK(inode->i_mode) && f->f_flags & O_APPEND)
1456                ipos = inode->i_size;
1457        else
1458                ipos = *pos;
1459
```

Dereferencing "obj" before a null check.

```
1460        if (!obj)
1461                T(YAFFS_TRACE_OS,
1462                        (TSTR("yaffs_file_write: hey obj is null!\n")));
```

The analysis provides information to help understand why this is a valid error. On line 1460, the code checks if 'obj' pointer is null. However, prior to that on line 1449, the pointer is dereferenced to assign a value to the 'dev' variable. Since this is a defect in the Linux kernel, Android developers pointed this to Linux kernel developers who then fixed it in the Linux development branch upstream and this change was later pulled in to the Android development branch.

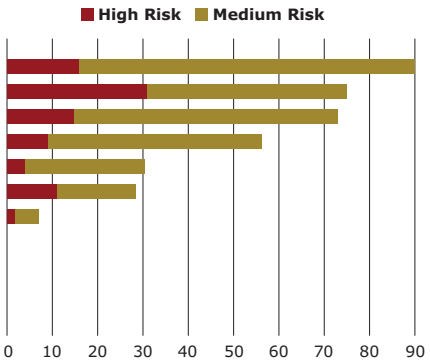
Conclusion

To meet the challenge of bringing new and innovative devices to the market faster, OEMs are increasingly dependent on their software supply chain and require visibility at various levels. Coverity is helping OEMs get visibility into the integrity of software across their supply chain by 1) automatically testing in-house, open source, and third party supplier code; and 2) generating reports which break down defects (quantity and risk level) by component.

Figure 3: Coverity Software Integrity Report showing Defect Risk by Component

Defect Risk by Component

Component	Owner	Defect Density
Fs		0.68
AndroidSpecific		0.78
Net		0.42
Drivers		0.63
Other		0.14
Kernal		0.55
Arch		0.41
-	-	0.00
-	-	0.00
-	-	0.00
-	-	0.00



We learned some interesting insights about the importance of open source and the modern software supply chain:

- **Even code with above-average level of quality still has high risk defects, and the problems don't disappear from one version to next.** The Coverity Software Integrity Report on the Android kernel, which compares code to industry standards, reported that the Android code is above-average. As Figure 3 shows, because the Android-specific part of the code base is newer, it has a higher concentration of defects when using a metric such as number of defects per thousand lines of code. As we saw with the first defect we discussed, the same defect from a kernel in the HTC Droid Incredible phone existed in the MSM development branch. With branching and code reuse, it becomes critical that all locations of a defect are identified and addressed.

- **Responsibility and standards for quality are fragmented due to the complexity of the supply chain.** When there are a number of different contributors to a development project or in a specific open source package, this many times means different standards for code quality and security. Because of a "you-build-it-you-fix-it" model, OEM vendors are responsible for maintaining anything they change, creating a reliance on the original developers of the code to fix defects. This also applies to the open source community. For example, Google Android developers are happy to fix code that is theirs, but rely on the upstream Linux kernel developers for fixing Linux-only code which has to be then pulled downstream into the Android branches. This complexity and fragmentation introduces risk.

- **OEMs should hold their suppliers accountable to the same standards they have in place for in-house developed code.** Because of a heavy reliance on open source and third-party code in building devices, OEMs face the difficult challenge of delivering a high quality product unless they have a consistent and reliable way to access the quality and security of all the code in their devices. Having a consistent way to measure the integrity of code from a variety of sources--and include only code that meets a certain level of quality--is one way to do it.

In summary, the analysis of the Android kernel is an interesting example of the modern software supply chain and some of the challenges for OEMs who use software from various third-party sources. Automated code testing via static analysis offers a way for vendors to get visibility into the integrity of software they include in their devices, and gain confidence in what is shipping in their product and under their brand.

About Coverity

Coverity, Inc. is the trusted standard for companies that have a zero tolerance policy for software failure, problems, and security breaches.

Coverity's award-winning portfolio of software integrity products enables customers to prevent software problems throughout the application lifecycle.

For More Information:
sales@coverity.com

Coverity Inc. Headquarters
185 Berry Street, Suite 1600
San Francisco, CA 94107 USA

U.S. Sales: (800) 873-8193
International Sales: +1 (415) 321-5237
www.coverity.com

coverity®