Back to developer.android.com

SEARCH

Search

ARCHIVE

► 2015 (114)
► 2014 (73)
▼ 2013 (48)
► December (3)
► November (2)
► October (7)
► September (2)
▼ August (5)
RenderScript Intrinsics
Respecting Audio Focus
Google Play Services 3.2

14 AUGUST 2013

Some SecureRandom Thoughts

Posted by Alex Klyubin, Android Security Engineer

The Android security team has been investigating the root cause of the compromise of a bitcoin transaction that led to the update of multiple Bitcoin applications on August 11.

We have now determined that applications which use the Java Cryptography Architecture (JCA) for key generation, signing, or random number generation may not receive cryptographically strong values on Android devices due to improper initialization of the underlying PRNG. Applications that directly invoke the system-provided OpenSSL PRNG without explicit initialization on Android are also affected. Applications that establish TLS/SSL connections using the `HttpClient` and `java.net` classes are not affected as those classes do seed the OpenSSL PRNG with values from `/dev/urandom`.

Developers who use JCA for key generation, signing or random number generation should update their applications to explicitly initialize the PRNG with entropy from `/dev/urandom` or `/dev/random`. A suggested implementation is provided at the end of this blog post. Also, developers should evaluate whether to regenerate cryptographic keys or other random values previously generated using JCA APIs such as `SecureRandom`, `KeyGenerator`, `KeyPairGenerator`, `KeyAgreement`, and `Signature`.

In addition to this developer recommendation, Android has developed patches that ensure that Android's OpenSSL PRNG is initialized correctly. Those patches have been provided to OHA partners.

We would like to thank Soo Hyeon Kim, Daewan Han of ETRI and Dong Hoon Lee of Korea University who notified Google about the improper initialization of OpenSSL PRNG.

Some SecureRandom Thoughts

ActionBarCompat and I/O 2013 App Source

► July (5)
► June (4)
► May (9)
► April (3)
► March (2)
► February (3)
► January (3)

► 2012 (41)
► 2011 (68)
► 2010 (72)
► 2009 (63)
► 2008 (40)
► 2007 (8)

Update: the original code sample below crashed on a small fraction of Android devices due to `/dev/urandom` not being writable. We have now updated the code sample to handle this case gracefully.

```
/*
 * This software is provided 'as-is', without any express or implied
 * warranty.  In no event will Google be held liable for any damages
 * arising from the use of this software.
 *
 * Permission is granted to anyone to use this software for any purpose,
 * including commercial applications, and to alter it and redistribute it
 * freely, as long as the origin is not misrepresented.
 */

import android.os.Build;
import android.os.Process;
import android.util.Log;

import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.UnsupportedEncodingException;
import java.security.NoSuchAlgorithmException;
import java.security.Provider;
import java.security.SecureRandom;
import java.security.SecureRandomSpi;
import java.security.Security;

/**
 * Fixes for the output of the default PRNG having low entropy.
 *
 * The fixes need to be applied via {@link #apply()} before any use of Java
 * Cryptography Architecture primitives. A good place to invoke them is in the
 * application's {@code onCreate}.
 */
public final class PRNGFixes {
```

```
    private static final int VERSION_CODE_JELLY_BEAN = 16;
    private static final int VERSION_CODE_JELLY_BEAN_MR2 = 18;
    private static final byte[] BUILD_FINGERPRINT_AND_DEVICE_SERIAL =
        getBuildFingerprintAndDeviceSerial();

    /** Hidden constructor to prevent instantiation. */
    private PRNGFixes() {}

    /**
     * Applies all fixes.
     *
     * @throws SecurityException if a fix is needed but could not be applied.
     */
    public static void apply() {
        applyOpenSSLFix();
        installLinuxPRNGSecureRandom();
    }

    /**
     * Applies the fix for OpenSSL PRNG having low entropy. Does nothing if the
     * fix is not needed.
     *
     * @throws SecurityException if the fix is needed but could not be applied.
     */
    private static void applyOpenSSLFix() throws SecurityException {
        if ((Build.VERSION.SDK_INT < VERSION_CODE_JELLY_BEAN)
                || (Build.VERSION.SDK_INT > VERSION_CODE_JELLY_BEAN_MR2)) {
            // No need to apply the fix
            return;
        }

        try {
            // Mix in the device- and invocation-specific seed.
            Class.forName("org.apache.harmony.xnet.provider.jsse.NativeCrypto")
                    .getMethod("RAND_seed", byte[].class)
                    .invoke(null, generateSeed());

            // Mix output of Linux PRNG into OpenSSL's PRNG
            int bytesRead = (Integer) Class.forName(
```

```
                    "org.apache.harmony.xnet.provider.jsse.NativeCrypto")
                    .getMethod("RAND_load_file", String.class, long.class)
                    .invoke(null, "/dev/urandom", 1024);
            if (bytesRead != 1024) {
                throw new IOException(
                        "Unexpected number of bytes read from Linux PRNG: "
                                + bytesRead);
            }
        } catch (Exception e) {
            throw new SecurityException("Failed to seed OpenSSL PRNG", e);
        }
    }

    /**
     * Installs a Linux PRNG-backed {@code SecureRandom} implementation as the
     * default. Does nothing if the implementation is already the default or if
     * there is not need to install the implementation.
     *
     * @throws SecurityException if the fix is needed but could not be applied.
     */
    private static void installLinuxPRNGSecureRandom()
            throws SecurityException {
        if (Build.VERSION.SDK_INT > VERSION_CODE_JELLY_BEAN_MR2) {
            // No need to apply the fix
            return;
        }

        // Install a Linux PRNG-based SecureRandom implementation as the
        // default, if not yet installed.
        Provider[] secureRandomProviders =
                Security.getProviders("SecureRandom.SHA1PRNG");
        if ((secureRandomProviders == null)
                || (secureRandomProviders.length < 1)
                || (!LinuxPRNGSecureRandomProvider.class.equals(
                        secureRandomProviders[0].getClass()))) {
            Security.insertProviderAt(new LinuxPRNGSecureRandomProvider(), 1);
        }

        // Assert that new SecureRandom() and
        // SecureRandom.getInstance("SHA1PRNG") return a SecureRandom backed
```

```
        // by the Linux PRNG-based SecureRandom implementation.
        SecureRandom rng1 = new SecureRandom();
        if (!LinuxPRNGSecureRandomProvider.class.equals(
                rng1.getProvider().getClass())) {
            throw new SecurityException(
                    "new SecureRandom() backed by wrong Provider: "
                            + rng1.getProvider().getClass());
        }

        SecureRandom rng2;
        try {
            rng2 = SecureRandom.getInstance("SHA1PRNG");
        } catch (NoSuchAlgorithmException e) {
            throw new SecurityException("SHA1PRNG not available", e);
        }
        if (!LinuxPRNGSecureRandomProvider.class.equals(
                rng2.getProvider().getClass())) {
            throw new SecurityException(
                    "SecureRandom.getInstance(\"SHA1PRNG\") backed by wrong"
                    + " Provider: " + rng2.getProvider().getClass());
        }
    }

    /**
     * {@code Provider} of {@code SecureRandom} engines which pass through
     * all requests to the Linux PRNG.
     */
    private static class LinuxPRNGSecureRandomProvider extends Provider {

        public LinuxPRNGSecureRandomProvider() {
            super("LinuxPRNG",
                    1.0,
                    "A Linux-specific random number provider that uses"
                        + " /dev/urandom");
            // Although /dev/urandom is not a SHA-1 PRNG, some apps
            // explicitly request a SHA1PRNG SecureRandom and we thus need to
            // prevent them from getting the default implementation whose output
            // may have low entropy.
            put("SecureRandom.SHA1PRNG", LinuxPRNGSecureRandom.class.getName());
            put("SecureRandom.SHA1PRNG ImplementedIn", "Software");
```

```
        }
    }

    /**
     * {@link SecureRandomSpi} which passes all requests to the Linux PRNG
     * ({@code /dev/urandom}).
     */
    public static class LinuxPRNGSecureRandom extends SecureRandomSpi {

        /*
         * IMPLEMENTATION NOTE: Requests to generate bytes and to mix in a seed
         * are passed through to the Linux PRNG (/dev/urandom). Instances of
         * this class seed themselves by mixing in the current time, PID, UID,
         * build fingerprint, and hardware serial number (where available) into
         * Linux PRNG.
         *
         * Concurrency: Read requests to the underlying Linux PRNG are
         * serialized (on sLock) to ensure that multiple threads do not get
         * duplicated PRNG output.
         */

        private static final File URANDOM_FILE = new File("/dev/urandom");

        private static final Object sLock = new Object();

        /**
         * Input stream for reading from Linux PRNG or {@code null} if not yet
         * opened.
         *
         * @GuardedBy("sLock")
         */
        private static DataInputStream sUrandomIn;

        /**
         * Output stream for writing to Linux PRNG or {@code null} if not yet
         * opened.
         *
         * @GuardedBy("sLock")
         */
        private static OutputStream sUrandomOut;
```

```
        /**
         * Whether this engine instance has been seeded. This is needed because
         * each instance needs to seed itself if the client does not explicitly
         * seed it.
         */
        private boolean mSeeded;

        @Override
        protected void engineSetSeed(byte[] bytes) {
            try {
                OutputStream out;
                synchronized (sLock) {
                    out = getUrandomOutputStream();
                }
                out.write(bytes);
                out.flush();
            } catch (IOException e) {
                // On a small fraction of devices /dev/urandom is not writable.
                // Log and ignore.
                Log.w(PRNGFixes.class.getSimpleName(),
                        "Failed to mix seed into " + URANDOM_FILE);
            } finally {
                mSeeded = true;
            }
        }

        @Override
        protected void engineNextBytes(byte[] bytes) {
            if (!mSeeded) {
                // Mix in the device- and invocation-specific seed.
                engineSetSeed(generateSeed());
            }

            try {
                DataInputStream in;
                synchronized (sLock) {
                    in = getUrandomInputStream();
                }
                synchronized (in) {
```

```
                in.readFully(bytes);
            }
        } catch (IOException e) {
            throw new SecurityException(
                    "Failed to read from " + URANDOM_FILE, e);
        }
    }

    @Override
    protected byte[] engineGenerateSeed(int size) {
        byte[] seed = new byte[size];
        engineNextBytes(seed);
        return seed;
    }

    private DataInputStream getUrandomInputStream() {
        synchronized (sLock) {
            if (sUrandomIn == null) {
                // NOTE: Consider inserting a BufferedInputStream between
                // DataInputStream and FileInputStream if you need higher
                // PRNG output performance and can live with future PRNG
                // output being pulled into this process prematurely.
                try {
                    sUrandomIn = new DataInputStream(
                            new FileInputStream(URANDOM_FILE));
                } catch (IOException e) {
                    throw new SecurityException("Failed to open "
                            + URANDOM_FILE + " for reading", e);
                }
            }
            return sUrandomIn;
        }
    }

    private OutputStream getUrandomOutputStream() throws IOException {
        synchronized (sLock) {
            if (sUrandomOut == null) {
                sUrandomOut = new FileOutputStream(URANDOM_FILE);
            }
            return sUrandomOut;
```

```
            }
        }
    }

    /**
     * Generates a device- and invocation-specific seed to be mixed into the
     * Linux PRNG.
     */
    private static byte[] generateSeed() {
        try {
            ByteArrayOutputStream seedBuffer = new ByteArrayOutputStream();
            DataOutputStream seedBufferOut =
                    new DataOutputStream(seedBuffer);
            seedBufferOut.writeLong(System.currentTimeMillis());
            seedBufferOut.writeLong(System.nanoTime());
            seedBufferOut.writeInt(Process.myPid());
            seedBufferOut.writeInt(Process.myUid());
            seedBufferOut.write(BUILD_FINGERPRINT_AND_DEVICE_SERIAL);
            seedBufferOut.close();
            return seedBuffer.toByteArray();
        } catch (IOException e) {
            throw new SecurityException("Failed to generate seed", e);
        }
    }

    /**
     * Gets the hardware serial number of this device.
     *
     * @return serial number or {@code null} if not available.
     */
    private static String getDeviceSerialNumber() {
        // We're using the Reflection API because Build.SERIAL is only available
        // since API Level 9 (Gingerbread, Android 2.3).
        try {
            return (String) Build.class.getField("SERIAL").get(null);
        } catch (Exception ignored) {
            return null;
        }
    }
```

```
    private static byte[] getBuildFingerprintAndDeviceSerial() {
        StringBuilder result = new StringBuilder();
        String fingerprint = Build.FINGERPRINT;
        if (fingerprint != null) {
            result.append(fingerprint);
        }
        String serial = getDeviceSerialNumber();
        if (serial != null) {
            result.append(serial);
        }
        try {
            return result.toString().getBytes("UTF-8");
        } catch (UnsupportedEncodingException e) {
            throw new RuntimeException("UTF-8 encoding not supported");
        }
    }
}
```

Join the discussion on
+Android Developers

Posted by Android Developers at 2:32 PM
Labels: Android, Security

Links to this post

Create a Link

Newer Post Home Older Post