Weak randomness in Android’s DNS resolver
CVE-2012-2808

Roee Hay & Roi Saltzman
<roeeh,roisa@il.ibm.com>

IBM Application Security Research Group

July 24, 2012

Android’s stub resolver is vulnerable to DNS poisoning due to weak randomness in its implementation. We show how an attacker can successfully guess the nonce of the DNS request with a probability that is sufficient for a feasible attack. We begin by defining the problem of DNS poisoning, then we explain some internals of the DNS resolver in Android, describe the vulnerability, and finally dive into a probability analysis.

1 The problem of DNS poisoning

A DNS request holds a unique identifier (‘nonce’) which consists of two 16 attributes: the TXID and the UDP source port. Ideally the two attributes are hard to predict, thus providing 32 bits of random data.

If a DNS response is received, and its nonce does not match the request’s, it is dropped by the resolver. Let \( R_{txid} \) and \( R_{port} \) be the random values which the TXID and UDP source port are generated from, and let \( R^*_{txid} \) and \( R^*_{port} \) be the attacker’s guesses, then ideally

\[
p = P_r(R^*_{txid} = R_{txid}, R^*_{port} = R_{port}) = 2^{-32}
\]

Considering the geometric properties of the attack, the expected time, \( T \), for a successful attack is

\[
\mathbb{E}(T) = \frac{X}{pN}
\]

where \( X \) is the time per try, and \( N \) is the number of spoofed responses the attacker can inject before the legitimate response has arrived from the DNS server. \( N \) depends on various parameters, such as the latency between the DNS resolver and the server.

The system is vulnerable to DNS poisoning if \( \mathbb{E}(T) \) is low. Ideally, when \( p = 2^{-32} \) (and \( N, X \) have reasonable values), the expected attack time is in magnitude of years. Notice that each random bit the that nonce loses, reduces \( \mathbb{E}(T) \) with a factor of 2.
2 DNS resolution in Android

Android provides its own libc implementation under codename 'Bionic'[1].
The DNS resolver implementation is located at libc/netbsd/resolv.
A bind wrapper is used in order to enhance Linux’s source port randomization:

```c
static int random_bind ( int s, int family )
{
    ...
    /* first try to bind to a random source port a few times */
    for ( j = 0; j < 10; j++ ) {
        /* find a random port between 1025 .. 65534 */
        int port = 1025 + (res_randomid() % (65535-1025));
        if (family == AF_INET)
            u.sin.sin_port = htons(port);
        else
            u.sin6.sin_port = htons(port);
        if ( !bind( s, &u.sa, slen ) )
            return 0;
    }
    /* nothing after 10 tries, our network table is probably busy */
    /* let the system decide which port is best */
    if (family == AF_INET)
        u.sin.sin_port = 0;
    else
        u.sin6.sin_port = 0;
    return bind( s, &u.sa, slen );
}
```

Listing 1: res_send.c!random_bind

The DNS query TXID is set in the file res_mkquery.c under the function res_mkquery, as can be seen below:

```c
int res_mkquery ( res_state statp,
    int op,           /* opcode of query */
    const char *dname, /* domain name */
    int class, int type, /* class and type of query */
    const u_char *data, /* resource record data */
    int datalen,      /* length of data */
    const u_char *newrr_in, /* new rr for modify or append */
    u_char *buf,      /* buffer to put query */
    int buflen )      /* size of buffer */
{
    ...
    hp = (HEADER *)(void *)buf;
    hp->id = htons(res_randomid());
```
...}

Listing 2: res_mkquery.c!res_mkquery

Afterwards, the wrapper tries to randomly acquire a port with its own implementation and if it fails 10
times in doing so, it delegates it to Linux, which chooses a port in the range 32768-61000.

As it can be seen, both functions make use of res_init.c!res_randomid, which is listed below:

```
int res_randomid(
    void
) {
    struct timeval now;
    gettimeofday(&now, NULL);
    return (0xffff & (now.tv_sec ^ now.tv_usec ^ getpid()));
}
```

Listing 3: res_init.c!res_randomid

Hence the TXID and source port are chosen using the following formula:

\[
R_{ID} = \text{WORD}(\text{time}_{sec} \oplus \text{time}_{\mu\text{sec}} \oplus \text{pid})
\]

where

\[
txid = R_{txid}
\]

\[
port = 1025 + (R_{port} \% (65535 - 1025))
\]

3 Vulnerability

Let \( t_{txid} \) be the time of which the victim generates \( R_{txid} \) by calling res_randomid, and let \( t_{port} \) be the time
that \( R_{port} \) is generated. Both time values are in \( \mu \text{sec} \) precision, and both random values are generated by
calling res_randomid. Since that function is used twice, in a very short time, \( t_{txid} \) and \( t_{port} \) become very
much correlated which has a direct impact on the correlation between \( R_{txid} \) and \( R_{port} \).

Section 3.1 exploits the weak randomness of

\[
\Delta \triangleq t_{port} - t_{txid}
\]

Section 3.2 exploits the weak randomness of

\[
\chi \triangleq t_{port} \oplus t_{txid}
\]

Our PoC [2] shows that the attack is feasible.
3.1 Scenario I: The attacker knows the PID of the target process

Notice the following relation between $R_{port}$ and $R_{txid}$:

$$R_{port} = (\Delta + R_{txid} \oplus t_{sec} \oplus pid) \oplus t_{sec} \oplus pid$$

We assume the attacker knows $t_{sec}$ since clocks are usually synchronized beyond the second precision. Therefore

$$P_r(R^*_{port} = R_{port} | R^*_{txid} = R_{txid}) \geq P_r(\Delta^* = \Delta) \equiv p_\Delta$$

where $\Delta^*$ is the attacker’s guess for $\Delta$. If the attacker has little knowledge of the distribution of $\Delta$ (only its max value), then the best he/she can do is a blind uniform guess.

Evidence (see the appendix) approves that the boundary for $\Delta$ is very small, thus $p_\Delta \gg 2^{-16}$. Moreover $\Delta$ is far from being uniform, therefore the attacker can choose $\Delta^*$ in a sophisticated manner.

Let $\Delta_{max}$ be the $x$ s.t. $P_r(\Delta = x)$ gets its maximum value.
Since

$$P_r(\Delta^* = \Delta) = \sum_x P_r(\Delta^* = x) \cdot P_r(\Delta = x) \leq P_r(\Delta = \Delta_{max})$$

we can conclude that the best strategy for the attacker would be to simply choose $\Delta^* = \Delta_{max}$.

Thus

$$p = P_r(R^*_{txid} = R_{txid}, R^*_{port} = R_{port}) = 2^{-16} \cdot P_r(\Delta = \Delta_{max})$$

See Appendix I: for the second environment, we have

$$p = 2^{-20.71137}$$

or 20.71137 random bits, which is much lower than the optimal value.

3.2 Scenario II: The attacker is unaware of the PID

Notice that we can lose the dependence on the PID, because

$$R_{port} \oplus R_{txid} = t_{txid} \oplus t_{port} = \chi$$

or

$$R_{\{txid, port\}} = R_{\{port, txid\}} \oplus \chi$$

Therefore, the attacker only needs to predict one of the random values and take the $\chi$ value with the highest probability (as explained in section [3.1]):

$$p = P_r(R^*_{txid} = R_{txid}, R^*_{port} = R_{port}) = 2^{-16} \cdot P_r(\chi = \chi_{max})$$

Again, evidence approves that $\chi$ is very small is far from the uniform distribution. In one environment, we can predict $\chi$ with probability $p = 0.022891$ hence that value has a total of 5.45 random bits.

Therefore the success probability is

$$p = 2^{-21.45}$$

or 21.45 random bits, which is much lower than the optimal value.
4 Impact

As usual, DNS poisoning attacks may endanger the integrity and confidentiality of the attacked system. For example, in Android, the Browser app can be attacked in order to steal the victim’s cookies of a domain of the attacker’s choice. If the attacker manages to lure the victim to browse to a web page controlled by him/her, the attacker can use JavaScript, to start resolving non-existing sub-domains. Upon success, a sub-domain points to the attacker’s IP, which enables the latter to steal wildcard cookies of the attacked domain, and even set cookies. In addition, a malicious app instantiate the Browser app on the attacker’s malicious web-page. If the attacker knows the PID (for example, a malicious app can access that information), the attack expected time can be reduced furthermore, as shown above.

5 Vendor Response

Android 4.1.1 has been released, and patches are available on AOSP. The random sample is now pulled from /dev/urandom, which should have adequate entropy by the time network activity occurs:

```c
#ifdef ANDROID_CHANGES
static int real_randomid(u_int *random_value) {
    /* open the nonblocking random device, returning -1 on failure */
    int random_device = open("/dev/urandom", O_RDONLY);
    if (random_device < 0) {
        return -1;
    }

    /* read from the random device, returning -1 on failure (or too many retries) */
    u_int retry = 5;
    for (retry; retry > 0; retry--) {
        int retval = read(random_device, random_value, sizeof(u_int));
        if (retval == sizeof(u_int)) {
            *random_value &= 0xffffffff;
            close(random_device);
            return 0;
        } else if ((retval < 0) && (errno != EINTR)) {
            break;
        }
    }

    close(random_device);
    return -1;
}
#endif /* ANDROID_CHANGES */

u_int res_randomid(void) {
    #ifdef ANDROID_CHANGES
    int status = 0;
    #endif
```
Listing 4: The patched function in AOSP

6 Timeline

07/24/2012 Public disclosure.
06/05/2012 Issue confirmed by Android Security Team and patch provided to partners.
05/21/2012 Disclosed to Android Security Team.

7 Acknowledgments

• We would like to thank the Android Security Team for the efficient way in which they handled this security issue.

• We are grateful to the following people for their contribution to this paper:
  – Lotem Guy
  – Omer Tripp
  – Omri Weisman
  – Jonathan Cohen

8 Appendix: Distributions of $\Delta$ and $\chi$ in various environments

8.1 Environment 1

Device Galaxy S I9000
Android version CyanogenMod 9 20120620 Nightly, Android 4.0.4
Network 802.11g, under attack.
Application Browser
Number of samples 15662
Histograms
### 8.2 Environment 2

**Device** Galaxy S I9000  
**Android version** CyanogenMod 7.2, Android 2.3.7
Network 802.11g, under attack.
Application Browser
Number of samples 16688
Histograms

<table>
<thead>
<tr>
<th>$\Delta$</th>
<th>$P_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>.038171</td>
</tr>
<tr>
<td>20</td>
<td>.037931</td>
</tr>
<tr>
<td>90</td>
<td>.026366</td>
</tr>
<tr>
<td>110</td>
<td>.024149</td>
</tr>
<tr>
<td>111</td>
<td>.022771</td>
</tr>
<tr>
<td>89</td>
<td>.022591</td>
</tr>
<tr>
<td>112</td>
<td>.022052</td>
</tr>
<tr>
<td>91</td>
<td>.019595</td>
</tr>
<tr>
<td>109</td>
<td>.018516</td>
</tr>
<tr>
<td>113</td>
<td>.017318</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\chi$</th>
<th>$P_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>.022891</td>
</tr>
<tr>
<td>100</td>
<td>.013363</td>
</tr>
<tr>
<td>37</td>
<td>.011865</td>
</tr>
<tr>
<td>44</td>
<td>.010606</td>
</tr>
<tr>
<td>28</td>
<td>.009767</td>
</tr>
<tr>
<td>35</td>
<td>.008150</td>
</tr>
<tr>
<td>64</td>
<td>.008150</td>
</tr>
<tr>
<td>108</td>
<td>.008090</td>
</tr>
<tr>
<td>66</td>
<td>.007610</td>
</tr>
<tr>
<td>65</td>
<td>.007550</td>
</tr>
</tbody>
</table>
8.3 Environment 3

Device Galaxy SIII I9300
Android version I93000BULF1, Android 4.0.4
Network 802.11g, under attack.
Application Browser
Number of samples 16335
Histograms

<table>
<thead>
<tr>
<th>$\Delta$</th>
<th>$P_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>0.026752</td>
</tr>
<tr>
<td>42</td>
<td>0.026018</td>
</tr>
<tr>
<td>40</td>
<td>0.024120</td>
</tr>
<tr>
<td>41</td>
<td>0.024059</td>
</tr>
<tr>
<td>43</td>
<td>0.022834</td>
</tr>
<tr>
<td>39</td>
<td>0.020875</td>
</tr>
<tr>
<td>60</td>
<td>0.020753</td>
</tr>
<tr>
<td>59</td>
<td>0.020631</td>
</tr>
<tr>
<td>61</td>
<td>0.020569</td>
</tr>
<tr>
<td>45</td>
<td>0.019712</td>
</tr>
</tbody>
</table>
9 References
