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 $\mathcal{R}_{txid}$ and $\mathcal{R}_{port}$ be the random values which the TXID and UDP source port are generated from, and let $\mathcal{R}^{\star}_{txid}$ and $\mathcal{R}^{\star}_{port}$ be the attacker's guesses, then ideally

$$p = P_r(\mathcal{R}^{\star}_{txid} = \mathcal{R}_{txid}, \mathcal{R}^{\star}_{port} = \mathcal{R}_{port}) = 2^{-32}$$

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:

```
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.sin6_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.sin6_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_nmkquery, as can be seen below:

```
int
res_nmkquery(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_nmkquery

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:

```
u_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:

$$\mathcal{R}_{ID} = WORD(time_{sec} \oplus time_{\mu frac} \oplus pid)$$

where

$$txid = \mathcal{R}_{txid}$$

$$port = 1025 + (\mathcal{R}_{port} \ \% \ (65535 - 1025))$$

3 Vulnerability

Let t_{txid} be the time of which the victim generates $\mathcal{R}_{txid}$ by calling res_randomid, and let t_{port} be the time that $\mathcal{R}_{port}$ is generated. Both time values are in μsec 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 $\mathcal{R}_{txid}$ and $\mathcal{R}_{port}$.

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 $\mathcal{R}_{port}$ and $\mathcal{R}_{txid}$:

$$\mathcal{R}_{port} = (\Delta + \mathcal{R}_{txid} \oplus time_{sec} \oplus pid) \oplus time_{sec} \oplus pid$$

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

$$P_r(\mathcal{R}^\star_{port} = \mathcal{R}_{port} | \mathcal{R}^\star_{txid} = \mathcal{R}_{txid}) \geq P_r(\Delta^\star = \Delta) \triangleq p_\Delta$$

where $\Delta^\star$ is the attacker's guess for Δ. If the attacker has little knowledge of the distribution of Δ (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 Δ is very small, thus $p_\Delta \gg 2^{-16}$. Moreover Δ is far from being uniform, therefore the attacker can choose $\Delta^\star$ in a sophisticated manner.

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

Since

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

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

Thus

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

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

$$\mathcal{R}_{port} \oplus \mathcal{R}_{txid} = t_{txid} \oplus t_{port} = \chi$$

or

$$\mathcal{R}_{\{txid,port\}} = \mathcal{R}_{\{port,txid\}} \oplus \chi$$

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

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

Again, evidence approves that χ is very small is far from the uniform distribution. In one environment, we can predict χ 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:

```
#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 &= 0xffff;
                        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;
        u_int output = 0;
        status = real_randomid(&output);
        if (status != -1) {
                return output;
        }
#endif /* ANDROID_CHANGES */
        struct timeval now;
        gettimeofday(&now, NULL);
        return (0xffff & (now.tv_sec ^ now.tv_usec ^ getpid()));
}
```

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.

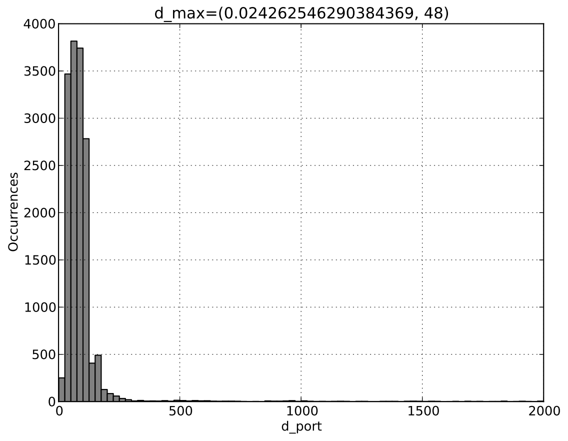
7 Acknowledgments

- 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

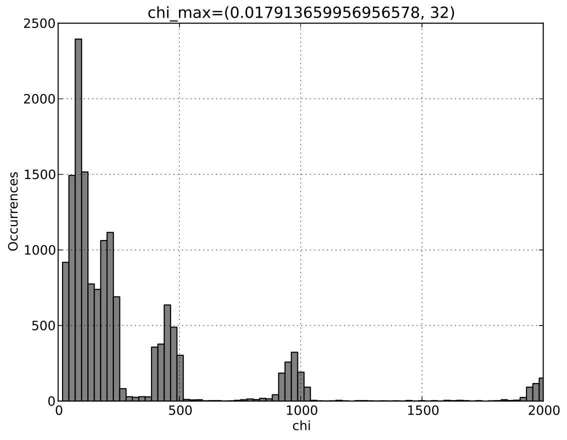
8 Appendix: Distributions of Δ and χ in various environments

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



Δ	P_r
48	.024263
88	.022411
49	.021326
50	.019729
32	.019218
96	.019155
72	.018644
40	.017686
87	.017622
108	.017558

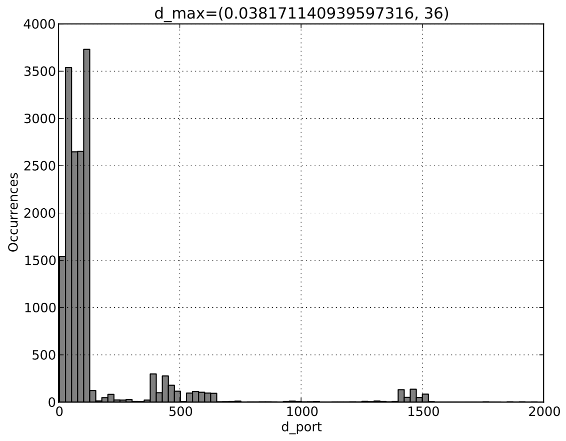


χ	P_r
32	.017914
33	.010508
72	.010444
88	.009811
96	.008735
48	.008735
80	.008419
73	.008229
97	.007279
35	.007216

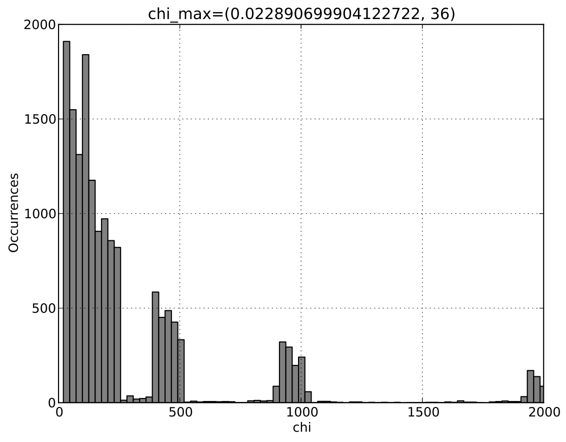
8.2 Environment 2

Device Galaxy S I9000
Android version CyanogenMod 7.2, Android 2.3.7

Network 802.11g, under attack.59646
Application Browser
Number of samples 16688
Histograms



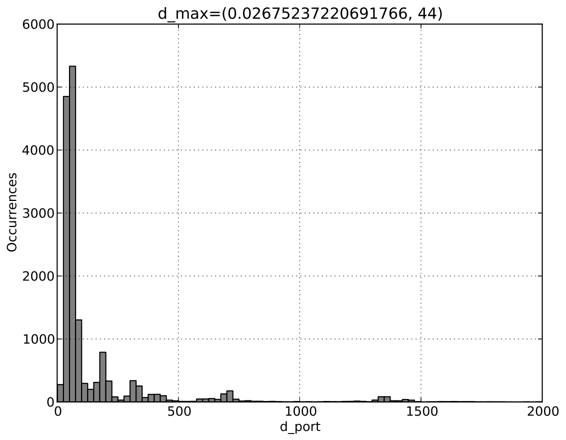
Δ	P_r
36	.038171
20	.037931
90	.026366
110	.024149
111	.022771
89	.022591
112	.022052
91	.019595
109	.018516
113	.017318



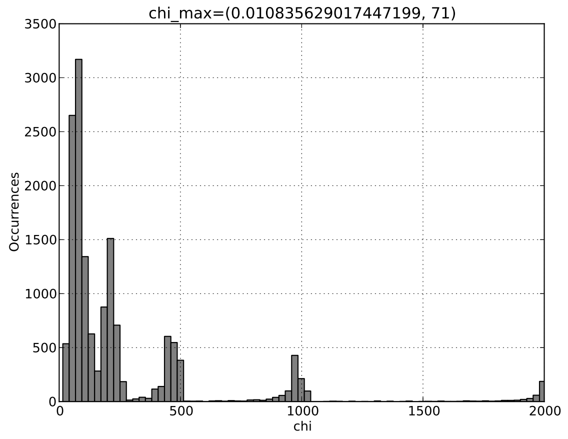
χ	P_r
36	.022891
100	.013363
37	.011865
44	.010606
28	.009767
35	.008150
64	.008150
108	.008090
66	.007610
65	.007550

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



Δ	P_r
44	.026752
42	.026018
40	.024120
41	.024059
43	.022834
39	.020875
60	.020753
59	.020631
61	.020569
45	.019712



χ	P_r
71	.010836
68	.010774
67	.010285
69	.010101
70	.009795
66	.009672
65	.009305
42	.008815
90	.008754
95	.008020

9 References

[1] Bionic (Software). `http://en.wikipedia.org/wiki/Bionic_(software)`.

[2] Roee Hay and Roi Saltzman. Video demo: Android dns poisoning: Randomness gone bad (cve-2012-2808), July 2012. `http://youtu.be/ffnF7Jej7l0`.