Bypassing ASLR by predicting a process' randomization

Hagen Fritsch <fritsch+stacksmashing@in.tum.de> January 23nd, 2009

Abstract

A flaw in the random number generator calling function can be used to calculate randomization values used by ASLR for up to two minutes after a target process has been launched. The attack will only work locally, with a theoretical likelihood of success of $\frac{1}{2.5}$, practical implications however lead to a lesser likelihood. If the attacker launches the target process himself, the likelihood becomes one. Since ASLR's purpose is to harden systems against buffer overflow exploits, several supposed-unexploitable bugs may be considered harmful now.

1 ASLR and the randomization bug

Address Space Layout Randomization is used to randomize parts of the address space to prevent attackers from knowing exact addresses which are of use for exploitations.

For randomization there are two functions in use: `randomize_range` and `arch_align_stack` which both call `get_random_int` to obtain their randomness. Additionally if re-keying is assumed to happen every second, the re-key interval should be set upropriately.

```
unsigned int get_random_int(void)
{
        /*
         * Use IP's RNG. It suits our purpose perfectly: it re-keys itself
         * every second, from the entropy pool (and thus creates a limited
         * drain on it), and uses halfMD4Transform within the second. We
         * also mix it with jiffies and the PID:
         */
        return secure_ip_id((__force __be32)(current->pid + jiffies));
}
__u32 secure_ip_id(__be32 daddr)
{
        struct keydata *keyptr;
        __u32 hash[4];

        keyptr = get_keyptr();
```

```
        /*
         *  Pick a unique starting offset for each IP destination.
         *  The dest ip address is placed in the starting vector,
         *  which is then hashed with random data.
         *
        hash[0] = (__force __u32)daddr;
        hash[1] = keyptr->secret[9];
        hash[2] = keyptr->secret[10];
        hash[3] = keyptr->secret[11];

        return half_md4_transform(hash, keyptr->secret);
}
```

As we can see, the output of `get_random_int` depends on the `pid`, `jiffies` and the secret in `keyptr`. Now reading the comment we see the assumption: "it re-keys itself every second". This means that the `secure_ip_id` will output the same value if called with the same input within one second.

But that's not all to the bug. A look at `rekey_seq_generator` shows the following:

```
schedule_delayed_work(&rekey_work, REKEY_INTERVAL);
```

Where `REKEY_INTERVAL` is #defined as `(300 * HZ)`, which is five minutes instead of one second.

2 Exploiting the bug

As we saw, the random number function has three parameters and can be seen as follows: $prf(s, j + p)$ where s is the secret from the IP stack being supposed to change every second, j is the jiffies which tick at a clock rate of `HZ` (usually each 4ms) and p is the `pid` of the current process.

If an attacker manages to call `execve(2)` with his desired target process within one jiffy (i.e. 4ms) after his process was launched, then the output of the *prf* function will be the same since neither `pid` p nor `jiffies` j changed. The secret s did not change either and would most likely also not change if re-keying was in fact every second. Thus the target process will use the very same randomization.

2.1 Predicting the randomization of a foreign process

Since within each timeframe of approximately five minutes the output of *prf* only depends on $j + p$, the question arises if such output can be pre- or post-produced. An

attacker is assumed to be a local user without root privileges. Thus he cannot call *prf* directly, and he also does not know the value of j as j is an internal kernel variable.

In the following we assume a target process is launched at t_0. This is known to the attacker, who thus also knows the time difference $\Delta t = t_{now} - t_0$ which implies $\Delta j = \Delta t / HZ$. Simple math shows, that the `pid` the attacker needs to get is $p_0 + j_0 = p_{attack} + j_{attack} \Rightarrow p_{attack} = p_0 - \Delta j = p_0 - \frac{t_{attack} - t_0}{HZ}$.

Thus an attacker has a time-frame of up to two minutes (depending on the pid of the target process) to get a process with a desired process id to read out its randomization. Process ids are completely deterministic, and a process can spawn child processes until it reaches a `pid` close to the required attack pid. The missing offset can be added by waiting for `jiffies` to increase.

The attacker does not know the exact time when the target process was started. It can read out the /proc/$pid/stat file and gets a time-stamp with a granularity of `USER_HZ` which is usually 10ms. Thus when calculating the attack pid, the attacker only has an accuracy of $\frac{2}{5}$.

The exploiting process can save randomization values for a range of outputs of *prf*. Within a process he or she can call `execve` to restart and output another randomization, thus an attacker is likely to save the output of $prf(s, j + p + i)\ \forall i \in [-x, x]$. If the attacker has multiple shots for exploiting the target, he can increase the likelihood of having acquired the correct randomization values up to absolute certainty with just a few guesses.

A not-yet-perfectly-optimized proof-of-concept demo program can be found at `http://itooktheredpill.dyndns.org/`.

2.2 Further impact

There has not yet been any research whether the output of the `secure_ip_id` function with IP packets can be used to leak randomization bits too. If there were a leakage that the attacker could read over the network, other attack scenarios need to be considered too.

3 Mitigation

The random number function should be fixed, so that calling it also modifies the seed and makes the output entirely unpredictable. Additionally if re-keying is assumed to happen every second, the re-key interval should be set up appropriately. There is no other easy work-around.