RASSLE: Return Address Stack based Side-channel LEakage

Artifact Evaluation Submission to TCHES 2021, Issue 2

Repository: `https://github.com/anirbanc3/RASSLE`
Demo video:
`https://www.youtube.com/watch?v=0bT1OAFX4NI`

This repository contains the source codes and scripts to demonstrate RASSLE in action.

1 System Specifications

The attack utilizes the deadline scheduler to achieve synchronization between the spy and victim. The demo have been tested on a system which has the following specifications:

- Processor - Intel Xeon CPU E5-2609 v4 (Broadwell)
- Operating System - Red Hat Enterprise Linux Server 7.7 ($kernel3.10.0 - 1062.9.1.el7.x86\_64$)
- Deadline scheduler parameters –
  - sched-runtime: 3600
  - sched-deadline: 3700
  - sched-period: 7200
- Return Address Stack (RAS) size - 16
- OpenSSL version – 1.1.1g
- linux-util(for `chrt`) - version 2.31

2 Getting Synchronization to work with the Deadline Scheduler

Most Linux-based operating systems offer a number of scheduling policies, which are crucial artifacts for controlling two asynchronous processes' execution. Among the available policies, the *deadline scheduler* is particularly interesting because it imposes a "deadline" on operations to prevent starvation of processes. In the deadline scheduler, each request by a process to access a system resource has an expiration time. A process holding a system resource does not need to be forcefully preempted, as the deadline scheduler automatically preempts it from the CPU after its request expiration time.

The operation of deadline scheduler depends on three parameters, namely 'runtime', 'period', and 'deadline'. These parameters can be adjusted using `chrt` command, which can be executed from user-level privilege by acquiring `CAP_SYS_NICE` permission. The permission can be provided to a user using `setcap cap_sys_nice+ep /usr/bin/chrt`.

The command to run an `<executable>` using deadline scheduler is as follows:

`chrt -d --sched-runtime` t_1 `--sched-deadline` t_2 `--sched-period` t_3 `0 <executable>`

where, t_1, t_2, and t_3 are the parameter values for 'runtime', 'deadline', and 'period' respectively. The kernel only allows scheduling with $t_1 \leq t_2 \leq t_3$. The usual practice is to set `sched-runtime` to some value greater than the average execution time of a task. We estimate the average execution time of each iteration of the target executable (ECC Montgomery ladder in this case) in terms of CPU clock cycles and convert the values into nanoseconds using CPU clock frequency. We set t_1 with the obtained value in nanoseconds. Further, we set the parameter `sched-deadline` to a value $t_2 = t_1 + \delta$ such that the ECC process leaves the CPU after execution of a single Montgomery ladder iteration. We set the parameter `sched-period` to a value $t_3 = 2 \times t_1$.

It should be noted that the procedure described here considers no change in the victim code and requires no use of `sched_yield()` *or* `nop` *or* `nanosleep()` *in the victim executable to preempt it from CPU.*

3 How to run the demo

The attack works in two phases - template building and template matching

3.1 Template Building

1. Run the shell script 'script_template.sh' to build the templates. In our experiments, we consider the most significant bit (msb) to be 1 and build templates for 6 msbs. Therefore total number of templates built is 32 (keeping msb as 1).

2. Once the above script ends, run 'generate_template.py' to create the template dataset.

3.2 Template Matching

1. Open 'ecc_encrypt.c' in a text editor. Comment out line 66. During template building phase, we varied the 6 msbs while keeping the remaining bits same. But, during the matching phase, the nonces are generated at random. So all the 256 bits of the nonce are used as input in this case.

2. Run the shell script 'script_nonce.sh' to generate the datasets containing timing values obtained through RASSLE. he script reads from a file containing random nonces and performs EC scalar multiplication using those nonces.

3. Once the above script ends, run 'template_matching.py' to retrieve the candidate "partial nonces" using Least Square Error method. The python script will also print the number of nonces correctly predicted.

4. Using these "partial nonces", the original secret signing key can be revealed using the well-known Lattice Attack.

We advise the users to follow the demonstration video to run the experiments.

4 Using a timestamp trigger signal

```
    FILE *fp = fopen("file_mont_ladder.txt", "a");
    for (i = cardinality_bits - 1; i >= 0; i--) {
        uint64_t start = rdtsc_begin();
        kbit = BN_is_bit_set(k, i) ^ pbit;

        EC_POINT_CSWAP(kbit, r, s, group_top, Z_is_one);

        /* Perform a single step of the Montgomery ladder */
        if (!ec_point_ladder_step(group, r, s, p, ctx)) {
            ECerr(EC_F_EC_SCALAR_MUL_LADDER, EC_R_LADDER_STEP_FAILURE);
            goto err;
        }
        /*
         * pbit logic merges this cswap with that of the
         * next iteration
         */
        pbit ^= kbit;
        uint64_t end = rdtsc_end();

        if (i == cardinality_bits - 1)
            fprintf(fp, "%lu\n", start);
        printf("Inside mont ladder\n");
    }

    fclose(fp);

    /* one final cswap to move the right value into r */
    EC_POINT_CSWAP(pbit, r, s, group_top, Z_is_one);
#undef EC_POINT_CSWAP

    /* Finalize ladder (and recover full point coordinates) */
    if (!ec_point_ladder_post(group, r, s, p, ctx)) {
        ECerr(EC_F_EC_SCALAR_MUL_LADDER, EC_R_LADDER_POST_FAILURE);
        goto err;
    }
```

Figure 1: Montgomery Ladder step with the timestamp trigger

As indicated in the snapshot above, we mark the start of the Montgomery Ladder operation by printing the timestamp value (from `rdtsc` counter) in a separate file (file_mont_ladder.txt). This is not mandatory for the success of the experiment, as the number of clock cycles can also be calculated from the start of the execution. We use this timestamp value as a trigger signal to align our traces for template building and matching phases.

In order to avoid confusion, we have added a sample OpenSSL repository to our GitHub repo. To install

1. Untar or extract openssl-1.1.1g-RASSLE.tar.xz and configure it using `./config`.
2. Execute `make` to build all the necessary files. After the build is complete, complete the installation by executing `sudo make install`. This version of OpenSSL will be installed in /usr/local/lib.

The OpenSSL repository contains the trigger in `ec_mult.c` file to indicate the start of the Montgomery Ladder process (as shown in Fig 1).

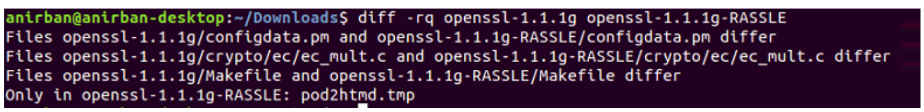

Figure 2: Result from `diff` tool showing the difference between the modified and the original repositories

We have compared the above mentioned OpenSSL repository with the original one using `diff` tool. The result is shown in Fig 2 which indicates that only the file `ec_mult.c` has been modified to include the timestamp trigger signals. We have also provided a comparison of the same file with the original one which shows the exact lines appended or modified in the file (Fig 3, 4).

```
 * For multiplication with precomputation, we use wNAF splitting, formerly at:
 *   http://www.informatik.tu-darmstadt.de/TI/Mitarbeiter/moeller.html#fastexp
 */

/* added by anirban     */

uint64_t rdtsc_begin() {
    uint64_t a, d;
    asm volatile ("mfence\n\t"
        "CPUID\n\t"
        "RDTSCP\n\t"
        "mov %%rdx, %0\n\t"
        "mov %%rax, %1\n\t"
        "mfence\n\t"
        : "=r" (d), "=r" (a)
        :
        : "%rax", "%rbx", "%rcx", "%rdx"
    );
    a = (d<<32) | a;
    return a;
}

uint64_t rdtsc_end() {
    uint64_t a, d;
    asm volatile("mfence\n\t"
        "RDTSCP\n\t"
        "mov %%rdx, %0\n\t"
        "mov %%rax, %1\n\t"
        "CPUID\n\t"
        "mfence\n\t"
        : "=r" (d), "=r" (a)
        :
        : "%rax", "%rbx", "%rcx", "%rdx"
    );
    a = (d<<32) | a;
    return a;
}

/*------------------------*/

/* structure for precomputed multiples of the generator */
struct ec_pre_comp_st {
    const EC_GROUP *group;      /* parent EC_GROUP object */
    size_t blocksize;           /* block size for wNAF splitting */
```

Figure 3: Custom functions included in `ec_mult.c` to read timestamp counter (`rdtscp`) values

```
     * This is XOR. pbit tracks the previous bit of k.
     */

        FILE *fp = fopen("file_mont_ladder.txt", "a");          // added by anirban
    for (i = cardinality_bits - 1; i >= 0; i--) {
                uint64_t start = rdtsc_begin();                         // added by anirban
        kbit = BN_is_bit_set(k, i) ^ pbit;

        EC_POINT_CSWAP(kbit, r, s, group_top, Z_is_one);

        /* Perform a single step of the Montgomery ladder */
        if (!ec_point_ladder_step(group, r, s, p, ctx)) {
            ECerr(EC_F_EC_SCALAR_MUL_LADDER, EC_R_LADDER_STEP_FAILURE);
            goto err;
        }
        /*
         * pbit logic merges this cswap with that of the
         * next iteration
         */
        pbit ^= kbit;
                uint64_t end = rdtsc_end();                             // added by anirban

                if (i == cardinality_bits - 1)                  // added by anirban
                        fprintf(fp, "%lu\n", start);            // added by anirban
                printf("Inside mont ladder\n");                 // added by anirban
    }

        fclose(fp);             // added by anirban

    /* one final cswap to move the right value into r */
    EC_POINT_CSWAP(pbit, r, s, group_top, Z_is_one);
#undef EC_POINT_CSWAP

    /* Finalize ladder (and recover full point coordinates) */
    if (!ec_point_ladder_post(group, r, s, p, ctx)) {
```

Figure 4: Added lines to read timestamp counter value at the start of Montgomery ladder step and store in a file