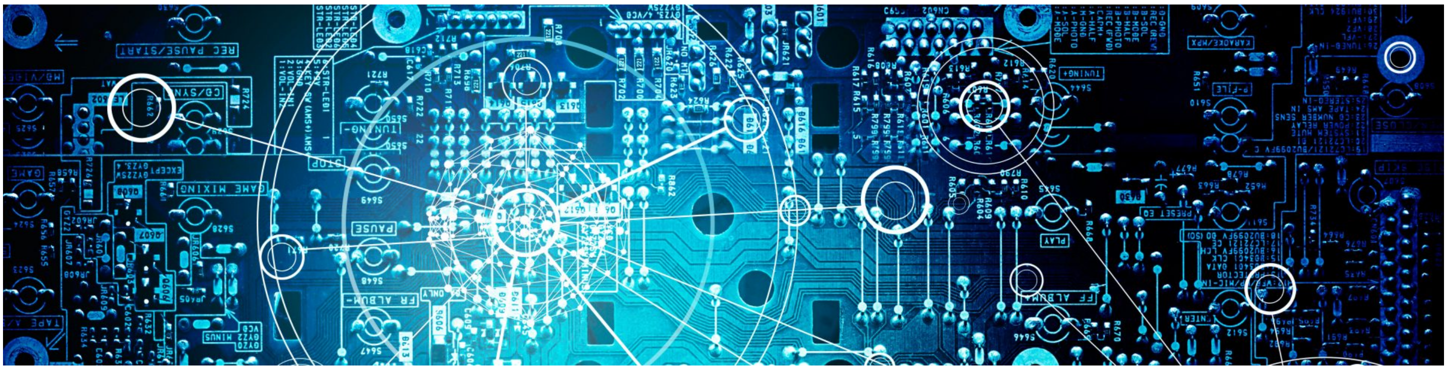
KUDELSKI SECURITY RESEARCH

The Latest News from Research at Kudelski Security

HOME CATEGORIES
HOME CATEGORIES



# HUNTING FOR VULNERABILITIES IN SIGNAL – PART 3

September 21, 2016 JP Aumasson Crypto, Research Leave a comment

Previous posts (part1 and part2) by Markus Vervier (@marver) and myself (@veorq) were about the Java code base and the Android client, now we'll discuss two bugs potentially affecting users of libsignal-protocol-c, the C implementation of the Signal protocol. More precisely, we identified bugs in the example callback functions used in the *unit tests* of the library. However, users of the library will need to define their own callbacks and will likely take the code from the unit tests as an example, as the library documentation suggests.

SEARCH

Se

CATEGORIES

Select

ARCHIVES

Select

One of the bugs will occur on 64-bit systems, the other bug will occur on 32-bit systems. Both will trigger a SIGSEV, and the second potentially leads to an heap overflow.

Both bugs have been rapidly patched by Open Whisper Systems, as well as a few benign potential null dereferences in serialization library functions.

IMPORTANT: the Signal mobile applications don't actually use the C implementation of the Signal protocol, and therefore *cannot be affected* by the bugs discussed in this post. WhatsApp does use this C lib, but allegedly does not use the same callbacks as the example ones where we found the bugs. Therefore WhatsApp doesn't seem to be affected either.

## Libsignal's crypto callback functions

When using the C implementation of the Signal protocol, the first step is to instantiate a global context (type `signal_context`), and in particular to provide pointers to functions handling cryptographic operations. Perhaps the most important one is `*encrypt_func` the callback for an AES encryption implementation provided by the user of the library. `*encrypt_func` is the function called by the library function `signal_encrypt()`:

```
int signal_encrypt(signal_context *context,
        signal_buffer **output,
        int cipher,
```

```
        const uint8_t *key, size_t key_len,
        const uint8_t *iv, size_t iv_len,
        const uint8_t *plaintext, size_t plaintext_len)
{
    assert(context);
    assert(context->crypto_provider.encrypt_func);
    return context->crypto_provider.encrypt_func(
            output, cipher, key, key_len, iv, iv_len,
            plaintext, plaintext_len,
            context->crypto_provider.user_data);
}
```

The function `test_encrypt()` in the file `tests/test_common.c` shows how to use OpenSSL to instantiate `*encrypt_func`. Although `test_encrypt()` isn't a library function, it's likely to be taken as an example or even copied altogether by users of the lib. So we believe bugs therein are relevant and worth disclosing.

In libsignal-protocol-c, the example code uses OpenSSL's EVP API:

```
int test_encrypt(signal_buffer **output,
        int cipher,
        const uint8_t *key, size_t key_len,
        const uint8_t *iv, size_t iv_len,
        const uint8_t *plaintext, size_t plaintext_len,
        void *user_data)
{
    int result = 0;
    uint8_t *out_buf = 0;

    const EVP_CIPHER *evp_cipher = aes_cipher(cipher,

    (...)

    EVP_CIPHER_CTX ctx;
    EVP_CIPHER_CTX_init(&ctx);
```

```
result = EVP_EncryptInit_ex(&ctx, evp_cipher, 0, key,

(...)

out_buf = malloc(sizeof(uint8_t) * (plaintext_len + EV

(...)

result = EVP_EncryptUpdate(&ctx, out_buf, &out_len

(...)

result = EVP_EncryptFinal_ex(&ctx, out_buf + out_len

(...)

*output = signal_buffer_create(out_buf, out_len + fir
```

The bugs, however, have nothing to do with the crypto, but are in the lines calling `malloc()` and `signal_buffer_create()` in the above code snippet.

## Integer type confusion and missing null check (64-bit systems)

In `test_encrypt()`, after finalizing encryption the following line implicitly casts `out_len + final_len` (both `int` types) to `size_t` (type of `signal_buffer_create()`'s second argument:

```
*output = signal_buffer_create(out_buf, out_len + fina
```

`signal_buffer_create()` is defined in `signal_protocol.c`, and calls the custom allocator of libsignal:

```
signal_buffer *signal_buffer_create(const uint8_t *dat
{
    signal_buffer *buffer = signal_buffer_alloc(len);
    if(!buffer) {
        return 0;
    }

    memcpy(buffer->data, data, len);
    return buffer;
}
```

Because of the type confusion, `signal_buffer_create()` may then call `signal_buffer_alloc(len)` (from `signal_protocol.c`) with an incorrect `len` argument:

```
signal_buffer *signal_buffer_alloc(size_t len)
{
    signal_buffer *buffer;
    if(len > (SIZE_MAX - sizeof(struct signal_buffer)) / siz
        return 0;
    }

    buffer = malloc(sizeof(struct signal_buffer) + (sizeof
    if(buffer) {
        buffer->len = len;
    }
    return buffer;
}
```

Due to the incorrect `len` value, a 64-bit unsigned integer, the `malloc()` fails when `len` is too large and then as a result `signal_buffer_alloc()` returns null and `signal_buffer_create` in turn returns 0.
However, `test_encrypt()` doesn't notice the failure and continues, because it doesn't test whether `*output` is zero.

When test_encrypt() is called by group_cipher_encrypt() or session_cipher_get_ciphertext(), for example, the unchecked pointer is later dereferenced by a signal_buffer_len() call that returns buffer->len which crashes the program.

For example, if the library function group_cipher_encrypt is called with a plaintext_len equal to 0x7ffffff2 (1024^3 * 2 – 14) then after encrypting in test_encrypt(), signal_buffer_create is called with as second argument out_len + final_len (both int types), with out_len = 0x7ffffff0 (2147483632) and final_len = 16, thus the sum is 0x80000000 (-2147483648, as an int). When calling signal_buffer_create() this value is cast to size_t value 18446744071562067968 (0xffffffff80000000L), for which malloc() obviously fails and eventually crashes the program.

The bug doesn't seem to be exploitable for code execution.

## Int overflow and potential heap overflow (32-bit systems)

In test_encrypt(), the following malloc() will overflow on 32-bit systems if plaintext_len is greater than SIZE_MAX - EVP_MAX_BLOCK_LENGTH:

```
out_buf = malloc(sizeof(uint8_t) * (plaintext_len + EVP_
```

The reason is that when size_t is 32-bit, the value plaintext_len + EVP_MAX_BLOCK_LENGTH will overflow.

Consequently, not enough memory is allocated to the output buffer where the ciphertext is written.

For example, if plaintext_len is equal to 2^32 – 1, with EVP_MAX_BLOCK_LENGTH equal to 32, then only 31 bytes will be allocated where 4GB are expected.

So now the risk is that that the encryption function will write encrypted data to unallocated heap memory, potentially allowing control flow hijacking and code execution. In the example encryption function test_encrypt(), OpenSSL's EVP API is used, which calls EVP_EncryptUpdate() to process plaintext of a given length:

```
int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx, unsigne
                      const unsigned char *in, int inl)
{
    int i, j, bl;

    if (ctx->cipher->flags & EVP_CIPH_FLAG_CUSTOM_CI
        if (is_partially_overlapping(out, in, inl)) {
            EVPerr(EVP_F_EVP_ENCRYPTUPDATE, EVP_R_PAI
            return 0;
        }

        i = ctx->cipher->do_cipher(ctx, out, in, inl);
        if (i < 0)
            return 0;
        else
            *outl = i;
        return 1;
    }

    if (inl <= 0) {
        *outl = 0;
        return inl == 0;
    }
```

```
(...)
```

Note that here the plaintext length is no longer a size_t type (unsigned), but a signed int. EVP_EncryptUpdate() will therefore see the large 32-bit size_t argument as a negative integer, and will abort. Had it used a size_t instead, the function would have written the ciphertext to unallocated heap memory. Thus the program will only crash, and won't heap overflow.

Like the previous bug, this shows that users of the library should be careful when reusing the example crypto callbacks.

« Data Science for Doofuses: What Toolbox to Use

BLAKE2X: Unlimited Hashing »

## LEAVE A REPLY

Enter your comment here...