Simple Public Key Encryption with RSA and OpenSSL

Source: https://shanetully.com/2012/04/simple-public-key-encryption-with-rsa-and-openssl/

Hey you! This post is outdated!

Take a look at a more correct, detailed, and useful one → /2012/06/openssl-rsa-aes-and-c/. What's the advantage? The EVP functions do implicit symmetric encryption for you so you don't get hung up on the max length limitations of RSA. Plus, it has an AES implementation.

Disclaimer: I am NOT a crypto expert. Don't take the information here as 100% correct; you should verify it yourself. You are dangerously bad at crypto. → http://happybearsoftware.com/you-are-dangerously-bad-at-cryptography.html

---

Last month I wrapped up my Alsa Volume Control server project → https://github.com/shanet/Alsa-Channel-Control. To test it, I exposed the server to my public Internet connection and within a few hours, my friend was using the lack of authentication to change the volume on my computer from his apartment. It may not be a serious security hole, and funny as it may be, it would certainly be annoying if someone had malicious intentions in mind. The simple solution is just disable the port forward so the server is only accessible via my LAN, but what fun is that? What if I feel like changing my volume from anywhere for whatever stupid reason I may have?! Thus, I needed to add authentication to the server, which means I also a needed a way to encrypt credentials as they went over the network. And so I opened up the OpenSSL documentation to figure out how to encrypt and decrypt simple messages with RSA in C. Here's a quick summary…

First up, to do anything with RSA we need a public/private key pair. I assume the reader knows the basic theory behind RSA so I won't go into the math inside a key pair. If you're interested, here's a good write-up → http://www.muppetlabs.com/~breadbox/txt/rsa.html on the math behind RSA.

```
RSA *keypair = RSA_generate_key(2048, 3, NULL, NULL);
```

Here we're using the RSA_generate_key → http://www.openssl.org/docs/crypto/RSA_generate_key.html function to generate an RSA public and private key which is stored in an RSA struct. The key length is the first parameter; in this case, a pretty secure 2048 bit key (don't go lower than 1024, or 4096 for the paranoid), and the public exponent (again, not I'm not going into the math here), is the second parameter.

So we have our key pair. Cool. So how do we encrypt something with it?

```
char *msg[2048/8]
printf("Message to encrypt: ");
fgets(msg, 2048/8, stdin);
msg[strlen(msg)-1] = '\0';    // Get rid of the newline

// Encrypt the message
char *encrypt = malloc(RSA_size(keypair));
int encrypt_len;
err = malloc(130);
if((encrypt_len = RSA_public_encrypt(strlen(msg)+1, (unsigned
char*)msg,
   (unsigned char*)encrypt, keypair, RSA_PKCS1_OAEP_PADDING))
== -1) {
    ERR_load_crypto_strings();
    ERR_error_string(ERR_get_error(), err);
    fprintf(stderr, "Error encrypting message: %s\n", err);
}
```

The first thing you'll notice is that the message length is limited to 2048 bits or 256 bytes, which is also our key size. A limitation of RSA is that you cannot encrypt anything longer than the key size, which is 2048 bits in this case. Since we're reading in chars, which are 1 byte and 2048bits translates to 256 bytes, the theoretical max length of our message is 256 characters long including the null terminator. In practice, this number is going to be slightly less because of the padding the encrypt function tacks on at the end. Through trial and error, I found this number to be around 214 characters for a 2048 bit key.

So we have the message. Let's encrypt it! We allocate memory for a buffer to store our encrypted message in (encrypt). We can determine the max length of the encrypted message via the `RSA_size` → http://www.openssl.org/docs/crypto/RSA_size.html function. We also allocate some memory for an error buffer, in case there's a problem encrypting the message like if the message is over the practical max length of a message (~214 bytes).

From here, all we have to do is call the `RSA_public_encrypt` → http://www.openssl.org/docs/crypto/RSA_public_encrypt.html function and let it do it's magic. We supply the number of bytes to encrypt, the message to encrypt, the buffer to put the encrypted message, they keypair to encrypt with, and finally, the type of padding to use for the message. The padding is where the discrepancy between the theoretical length and practical length comes from. The different types can be found on the documentation page for the `RSA_public_encrypt` function, but the one used above is the one that should be used for new implementations of RSA.

`RSA_public_encrypt` will return the number of bytes encrypted, or -1 on failure. If -1 we use the OpenSSL error functions to get a more descriptive error, and print it. The error functions are pretty self-explanatory if you read their documentation, so I won't go into them here. Another sanity check that I didn't check for would be to ensure that the number of bytes encrypted returned by `RSA_public_encrypt` is the key size divided by 8, or 256 in this case. If it isn't, something isn't right.

Now let's decrypt the message! Good news is that if you understood the encryption, decryption is very similar.

```
char *decrypt = malloc(RSA_size(keypair));
if(RSA_private_decrypt(encrypt_len, (unsigned char*)encrypt,
(unsigned char*)decrypt,
                       keypair, RSA_PKCS1_OAEP_PADDING) == -1)
{
   ERR_load_crypto_strings();
   ERR_error_string(ERR_get_error(), err);
   fprintf(stderr, "Error decrypting message: %s\n", err);
} else {
   printf("Decrypted message: %s\n", decrypt);
}
```

We allocate the length of our encrypted message to store the decrypted message in. The decrypted message may only be a few characters long, but we don't know how it's exact length prior to decryption, so we allocate the upper bound of its length to avoid any length issues. From here, decryption is a simple call to `RSA_private_decrypt` → http://www.openssl.org/docs/crypto/RSA_public_encrypt.html with the encrypted length, the encrypted message, the buffer to store the decrypted message in, the key to perform decryption with, and the padding type–all very similar to the encrypt function. `RSA_public_decrypt` returns -1 on error and we check for errors the same way as the encrypt function.

And that's it! You can now encrypt and decrypt messages with RSA!

But let's get a little closer to having something that's actually useful. Let's see if we can write our encrypted message to a file, read it back, and then decrypt it.

```
FILE *out = fopen("out.bin", "w");
fwrite(encrypt, sizeof(*encrypt),  RSA_size(keypair), out);
fclose(out);
printf("Encrypted message written to file.\n");
free(encrypt);
encrypt = NULL;
```

Writing to a file is actually pretty easy. The one caveat to remember is that we aren't dealing with plain text anymore–we're working with binary data now so the usual ways to write to a file like `fputs` aren't going to work here. Instead, we utilize `fwrite` which is going to write the encrypted message buffer to the file verbatim. We should check for errors here, but this is just a quick proof-of-concept.

Reading it back is also just as trivial.

```
printf("Reading back encrypted message and attempting
decryption...\n");
encrypt = malloc(RSA_size(keypair));
out = fopen("out.bin", "r");
fread(encrypt, sizeof(*encrypt), RSA_size(keypair), out);
fclose(out);
```

We free'd our encrypted message buffer after writing it to the file above as a proof-of-concept above so we need to allocate memory for it again. After that, remember that this data isn't plain text so the usual `fgets` isn't going to work. We need to use `fread` which will put the encrypted message back into the encrypt buffer which we can then use to send to the decrypt function above.

Let’s also make sure that the data we wrote the file is really there by firing up a terminal and looking at an od dump of the file we wrote.

```
$ od -c out.bin
0000000 P # 6 271 315 236 _ 344 267 % \v U 306 237 l 230
0000020 240 311 210 / ? 221 355 313 c 356 O * F 264 355 316
0000040 G 216 \t # G 1 [ 225 4 371 * 244 304 5 ) 211
0000060 213 365 236 240 367 025 256 _ \a 231 + 360 W 177 274
321
0000100 301 263 I 4 240 6 < ' 4 s z 4 236 360 w 244
0000120 006 261 203 214 \b 004 t 004 024 270 363 352 ` 340 207
321
0000140 317 o 211 D 222 363 017 372 k 244 353 003 237 v 275
241
0000160 W N 225 311 002 L 340 272 U 4 252 257 326 023 037 c
0000200 5 332 004 314 253 { * 032 Q - 330 213 374 247 301 256
0000220 R + 030 $ ? O 214 343 213 9 233 7 \a 033 \a \r
0000240 w $ 376 025 A " 027 316 277 265 004 227 n q 344 b
0000260 v 266 223 306 363 334 370 035 031 245 344 216 250 367
277 246
0000300 B 272 9 n 6 6 Y 356 G , 203 034 333 371 ( 177
0000320 p c 313 271 323 342 033 360 210 340 203 352 352 305 \r
w
0000340 244 274 343 351 { 217 342 345 036 326 | 032 261 002
206 004
0000360 & Z 335 341 x 231 376 X 203 \v 225 353 210 204 324 g
0000400
```

Here we can see why the file can’t be read as a regular text file. Some of the values are outside of the range of regular characters! Compare this to the plain text of the message that’s encrypted above (hint: it’s “hello”):

```
$od -c out.txt
0000000 h e l l o
0000006
```

Another thing we can do is separate the key pair into a public key and a private key, because what good does sending both the private and public key to decrypt a message to someone do? Let's revisit the original code we used to generate the key pair.

```
RSA *keypair = RSA_generate_key(KEY_LENGTH, PUB_EXP, NULL,
NULL);

BIO *pri = BIO_new(BIO_s_mem());
BIO *pub = BIO_new(BIO_s_mem());

PEM_write_bio_RSAPrivateKey(pri, keypair, NULL, NULL, 0, NULL,
NULL);
PEM_write_bio_RSAPublicKey(pub, keypair);

size_t pri_len = BIO_pending(pri);
size_t pub_len = BIO_pending(pub);

char *pri_key = malloc(pri_len + 1);
char *pub_key = malloc(pub_len + 1);

BIO_read(pri, pri_key, pri_len);
BIO_read(pub, pub_key, pub_len);

pri_key[pri_len] = '\0';
pub_key[pub_len] = '\0';

printf("\n%s\n%s\n", pri_key, pub_key);
```

We generate the key pair as before (this time with a generalized key length and public exponent), but now we used `BIO structs` to separate the public and private key. BIO's → http://www.openssl.org/docs/crypto/bio.html are just an OpenSSL abstraction to make our lives easier. We use the PEM_write_bio_RSAPrivateKey → http://www.openssl.org/docs/crypto/pem.html function and it's public key counterpart to copy the private and public keys into the newly created `BIO structs`. We then use the `BIO_pending` function to get how long our plain text character strings need to be to store the keys and allocate that amount of memory. From there, `BIO_read` copies the keys from the `BIO structs` into the character strings. Finally, let's print them out for fun. Here's an example of a key pair I generated via this method:

```
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAx5WRSyfFVe/JbPYnswghuMj5Nzo9YG82Z7ehyI/dbjkcdCIz
TlKdQcMvSUZafAnM9p3xnBrgbKaNltaVNrZNyN6A2ou0PQgms7ykJ67G9Hbbs/uo
0rPSGS4pYw0QiOvoYXjGqbOzQjXbAV7ez05XRb43nRdZUFO0LLvEp2VfaTL7WWza
an6rVe6p8t8JIwpWSn7njBYH2XPNJj1NccpvD+kT1kGn6kWZfmFBzR7Bw2+rW+rp
t02F4arxXfvzDYhZdxLKb7m2KqwZTiug2HoD5AY9l3GzRIdNvXIDP87XTl4960lp
g8cI8XuTSLFjSx0fvlXXFwTcgMLv7Q0+ISSXQwIBJQKCAQBbs2xFIBxlwTMIjMYh
003DmpwEnjfgMxj/OLId4Tw5j9ykD7a1SI1xPgD4Jz5Uqo6a0vJ4KAY/wiVg+s7v
n96MuUIfOUT5cnKlm9y4SWJUpVAwGa5upaW4itS+zqa7c08YBw8rYGcea8V9K5bN
84/hxhmNXcFAlSlE/FNvgZqKRwrdwuG5z/0g7AsoqlyE/VcaOjKrYQZFWZY77fGu
pwBvaymGGHxBu4ftxdBYJVCweOEiPA4PDX2cvEs2kyCIbKBe7iBLhA8p8llWWg0+
Rm+G4Y7t2gE7mfaq96XcOQHVRcjU7EdP3yN5S7clb2p807CzX6YHUlEokV4xa1tk
HsD9AoGBAPU+Sw33ewG2856ynXcDpj5hVd7cUAG26xBeYVm/5lPMUFEZ8znUvSkP
cqWPk/DRKUBE55soL5rBOofmGYaoAf21tvfaec0qrKbK3M6GjZMnqoqQWRnGjjxe
5xTsqctRFOlOmI7PKGGy87GNAybH7f4VSeeZhgIW4x8+jXNDt18pAoGBANBWl8ML
xoKxrgIF3dfQxDw3m08cp48TIBtymdafLVaNVSwf+wVKO5Ph5biq3z9yMETbj6hu
WujLTxEQKbwtiuCYZJtaTang7/td9D0Gnm7xJXvyXDx0uQeQFZRkxxcO/L2gGAND
BEr6TIBMFhiCj5IEhw2py7FN0JpC8Hxs7gyLAoGBALL2GxgHnvNp1F8MuBiT9dp+
YUHDXPpVDGXkAdm1jGap2b6kO94XyE5lN/xGLa+7OccdhmpNwd+hwu2MPCP+D0pv
2ItaPTTZ85xO2HsIPcw/iklwQQT4rPufMwFubhDoJAQyb1N0kxbciFEhtjEOb2Zi
j+BbRh0zS8qxGxzCtj6FAoGBAMUTpFC4HKUkna7i9HI0Lz/hkun4gtNy9NuxmHET
HQyvNOSM9F7zMXA2jTIlGF6cctlaEkVheJcFgiTlxp0/1mXAloUeEh08j/ueEIzB
Emjx8waLUFTdHbsLwWLb3uxMconcoRfXnEbsxOgQn0eeGRtsEQzsucNlSMlGPW7H
6BmzAoGBAIvsBQW/yxpJuWkSu2YvBOvxMe40MHc5L/Oe4b/LG6YHGavIdmIrxvXN
C0ZlE7kHr7csmGChxPopULKygywpX2+SBQ5am8UXM/rqiQj1fagpHvseOIv9BmO9
LKCldG8eOeGqaBnCG6GKXpzJ0kk4xey6Kj7+bdVlaBvVz0KGofVE
-----END RSA PRIVATE KEY-----
```

```
-----BEGIN RSA PUBLIC KEY-----
MIIBCAKCAQEAx5WRSyfFVe/JbPYnswghuMj5Nzo9YG82Z7ehyI/dbjkcdCIzTlKd
QcMvSUZafAnM9p3xnBrgbKaNltaVNrZNyN6A2ou0PQgms7ykJ67G9Hbbs/uo0rPS
GS4pYw0QiOvoYXjGqbOzQjXbAV7ez05XRb43nRdZUFO0LLvEp2VfaTL7WWzaan6r
Ve6p8t8JIwpWSn7njBYH2XPNJj1NccpvD+kT1kGn6kWZfmFBzR7Bw2+rW+rpt02F
4arxXfvzDYhZdxLKb7m2KqwZTiug2HoD5AY9l3GzRIdNvXIDP87XTl4960lpg8cI
8XuTSLFjSx0fvlXXFwTcgMLv7Q0+ISSXQwIBJQ==
-----END RSA PUBLIC KEY-----
```

So that's a lot of code! Let's put it all together into one complete example:

8

```c
#include <openssl/rsa.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <stdio.h>
#include <string.h>

#define KEY_LENGTH  2048
#define PUB_EXP     3
#define PRINT_KEYS
#define WRITE_TO_FILE

int main(void) {
    size_t pri_len;            // Length of private key
    size_t pub_len;            // Length of public key
    char   *pri_key;           // Private key
    char   *pub_key;           // Public key
    char   msg[KEY_LENGTH/8];  // Message to encrypt
    char   *encrypt = NULL;    // Encrypted message
    char   *decrypt = NULL;    // Decrypted message
    char   *err;               // Buffer for any error
messages

    // Generate key pair
    printf("Generating RSA (%d bits) keypair...",
KEY_LENGTH);
    fflush(stdout);
    RSA *keypair = RSA_generate_key(KEY_LENGTH, PUB_EXP,
NULL, NULL);

    // To get the C-string PEM form:
    BIO *pri = BIO_new(BIO_s_mem());
    BIO *pub = BIO_new(BIO_s_mem());

    PEM_write_bio_RSAPrivateKey(pri, keypair, NULL, NULL, 0,
NULL, NULL);
    PEM_write_bio_RSAPublicKey(pub, keypair);

    pri_len = BIO_pending(pri);
    pub_len = BIO_pending(pub);

    pri_key = malloc(pri_len + 1);
    pub_key = malloc(pub_len + 1);

    BIO_read(pri, pri_key, pri_len);
    BIO_read(pub, pub_key, pub_len);

    pri_key[pri_len] = '\0';
    pub_key[pub_len] = '\0';

    #ifdef PRINT_KEYS
        printf("\n%s\n%s\n", pri_key, pub_key);
    #endif
    printf("done.\n");

    // Get the message to encrypt
    printf("Message to encrypt: ");
    fgets(msg, KEY_LENGTH-1, stdin);
    msg[strlen(msg)-1] = '\0';

    // Encrypt the message
    encrypt = malloc(RSA_size(keypair));
    int encrypt_len;
    err = malloc(130);
    if((encrypt_len = RSA_public_encrypt(strlen(msg)+1,
 (unsigned char*)msg, (unsigned char*)encrypt,
                                         keypair,
RSA_PKCS1_OAEP_PADDING)) == -1) {
        ERR_load_crypto_strings();
        ERR_error_string(ERR_get_error(), err);
        fprintf(stderr, "Error encrypting message: %s\n",
err);
        goto free_stuff;
    }

    #ifdef WRITE_TO_FILE
    // Write the encrypted message to a file
        FILE *out = fopen("out.bin", "w");
        fwrite(encrypt, sizeof(*encrypt),  RSA_size(keypair),
out);
        fclose(out);
        printf("Encrypted message written to file.\n");
        free(encrypt);
        encrypt = NULL;

        // Read it back
        printf("Reading back encrypted message and attempting
decryption...\n");
        encrypt = malloc(RSA_size(keypair));
        out = fopen("out.bin", "r");
        fread(encrypt, sizeof(*encrypt), RSA_size(keypair),
out);
        fclose(out);
    #endif

    // Decrypt it
    decrypt = malloc(encrypt_len);
    if(RSA_private_decrypt(encrypt_len, (unsigned
char*)encrypt, (unsigned char*)decrypt,
                           keypair, RSA_PKCS1_OAEP_PADDING)
== -1) {
        ERR_load_crypto_strings();
        ERR_error_string(ERR_get_error(), err);
        fprintf(stderr, "Error decrypting message: %s\n",
err);
        goto free_stuff;
    }
    printf("Decrypted message: %s\n", decrypt);

    free_stuff:
    RSA_free(keypair);
    BIO_free_all(pub);
    BIO_free_all(pri);
    free(pri_key);
    free(pub_key);
    free(encrypt);
    free(decrypt);
    free(err);

    return 0;
}
```

To compile it (with debug symbols in case you want to debug it), make sure you have the OpenSSL library installed (libcrypto), and then run:

```
gcc -ggdb -Wall -Wextra -o rsa_test rsa_test.c -lcrypto
```

And there you have it, simple RSA encryption and decryption. I'll be writing more posts as I further implement this into my Alsa server project on the topics on sending the public key over the network, sending arbitrary size messages with the help of a symmetric cipher (probably AES), doing authentication with Unix users, and doing all this on Android.