Pentest-Report OpenPGP.js 02.2014

Cure53, Dr.-Ing. Mario Heiderich / Krzysztof Kotowicz / Jonas Magazinius / Franz Antesberger

Index

Intro

“This project aims to provide an Open Source OpenPGP library in JavaScript so it can be used on virtually every device. Instead of other implementations that are aimed at using native code, OpenPGP.js is meant to bypass this requirement (i.e. people will not have to install gpg on their machines in order to use the library).

The idea is to implement all the needed OpenPGP functionality in a JavaScript library that can be reused in other projects that provide browser extensions or server applications. It should allow you to sign, encrypt, decrypt, and verify any kind of text - in particular e-mails - as well as managing keys.”

Source: http://openpgpjs.org/

This penetration test was carried out and coordinated by four testers and yielded an overall of 26 issues. Among these findings, Cure53 has classified 12 as vulnerabilities, with 2 issues rated ‘critical’ in regards to their severity. This penetration test was conducted under a paid contract agreement sponsored by Radio Free Asia’s Open Technology Fund. The Cure53 team has spent 15 days on testing the library, documenting the identified issues and communicating the results to the OpenPGP.js Development Team.

Scope

- **OpenPGP JavaScript Implementation.**
  - http://openpgpjs.org/
  - https://github.com/openpgpjs/openpgpjs

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a chronological order rather than by their degree of severity and impact. The latter is simply given in brackets following the title heading for each vulnerability, which is also marked with a unique identifier for the purpose of facilitating future follow-up correspondence and referencing.

OP-01-005 Side-channel leak in RSA decryption *(High)*

The RSA decryption routine implemented in the library (see *src/crypto/public_key/rsa.js*) introduces timing information leak that can be used to factor RSA private key, as it has been described by Paul C Kocher in "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems"[1].

The library uses Chinese Remainder Theorem for optimization. Timing attack for that algorithm was demonstrated for OpenSSL in "Brumley, Boneh. Remote Timing Attacks are Practical"[2].

To protect from side-channel leaks in RSA decryption, RSA blinding should be used:

> *"RSA blinding involves computing the blinding operation $E(x) = xr^e \bmod N$, where r is a random integer between 1 and N and relatively prime to N (i.e. gcd(r, N) = 1), x is the ciphertext, e is the public RSA exponent and N is the RSA modulus. As usual, the decryption function $f(z) = z^d \bmod N$ is applied thus giving $f(E(x)) = x^d r^{ed} \bmod N = x^d r \bmod N$. Finally it is unblinded using the function $D(z) = zr^{-1} \bmod N$. Since $D(f(E(x))) = x^d \bmod N$, this is indeed an RSA decryption."*
>
> From: http://en.wikipedia.org/wiki/Blinding_(cryptography)

It is worth noting that RSA blinding in RSA decryption is also recommended to mitigate other side-channel attacks, such as acoustic attack on GnuPG (2013-4576)[3].

We recognize that protecting JavaScript-based code from side-channel leaks and, for example, introducing constant-time operations in JavaScript is very problematic. Further, it becomes even more complex as the library might be used by different implementers, run in different JS engines, including also those environments that offer varying code isolation levels (website, node.js module, Chrome Packaged App provide just a few examples). Currently no guarantee can be given that even the JS code carefully designed so as to avoid side-channel leaks cannot be forced to run in conditions that allow for e.g. timing attacks.

---

[1] Timing Attacks http://www.cryptography.com/public/pdf/TimingAttacks.pdf
[2] SSL Timing Attacks http://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf
[3] CVE-2013-4576 http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4576

Nevertheless, when dealing with such a crucial cryptographic library we recommend to take the best-effort approach and implement countermeasures for all known side-channel attacks, just as other non JS-based OpenPGP libraries do.

OP-01-007 Algorithm Preferences ignored upon Encryption *(Low)*

Upon message encryption performed by OpenPGP.js, the library appears to ignore the given symmetric algorithm preference that is stored together with the provided key. Instead, a static configuration value is used, currently defaulting to *AES256* algorithm. This can be observed in the source code below:

```
// src/config/config.js
var enums = require('../enums.js');

module.exports = {
  prefer_hash_algorithm: enums.hash.sha256,
  encryption_cipher: enums.symmetric.aes256,
  compression: enums.compression.zip,
  show_version: true,
  show_comment: true,
  integrity_protect: true,
  keyserver: "keyserver.linux.it", // "pgp.mit.edu:11371"

  versionstring: "OpenPGP.js VERSION",
  commentstring: "http://openpgpjs.org",

  node_store: './openpgp.store',

  debug: false
};

// message.js
Message.prototype.encrypt = function(keys) {
  var packetlist = new packet.List();
  //TODO get preferred algo from signature
  var sessionKey = crypto.generateSessionKey(enums.read(enums.symmetric,
config.encryption_cipher));
  keys.forEach(function(key) {
    var encryptionKeyPacket = key.getEncryptionKeyPacket();
    if (encryptionKeyPacket) {
      var pkESKeyPacket = new packet.PublicKeyEncryptedSessionKey();
      pkESKeyPacket.publicKeyId = encryptionKeyPacket.getKeyId();
      pkESKeyPacket.publicKeyAlgorithm = encryptionKeyPacket.algorithm;
      pkESKeyPacket.sessionKey = sessionKey;
      //TODO get preferred algo from signature
      pkESKeyPacket.sessionKeyAlgorithm = enums.read(enums.symmetric,
config.encryption_cipher);
      pkESKeyPacket.encrypt(encryptionKeyPacket);
      packetlist.push(pkESKeyPacket);
    } else {
      throw new Error('Could not find valid key packet for encryption in key ' +
key.primaryKey.getKeyId().toHex());
    }
  });
```

```
  var symEncryptedPacket;
  if (config.integrity_protect) {
    symEncryptedPacket = new packet.SymEncryptedIntegrityProtected();
  } else {
    symEncryptedPacket = new packet.SymmetricallyEncrypted();
  }
  symEncryptedPacket.packets = this.packets;
  //TODO get preferred algo from signature
  symEncryptedPacket.encrypt(enums.read(enums.symmetric,
config.encryption_cipher), sessionKey);
  packetlist.push(symEncryptedPacket);
  // remove packets after encryption
  symEncryptedPacket.packets = new packet.List();
  return new Message(packetlist);
};
```

This creates interoperability problems and is in fact a violation of the rules defined in RFC 4880[4], which read that:

> **"An implementation MUST NOT use a symmetric algorithm that is not in the recipient's preference list.** When encrypting to more than one recipient, the implementation finds a suitable algorithm by taking the intersection of the preferences of the recipients. Note that the MUST-implement algorithm, TripleDES, ensures that the intersection is not null. The implementation may use any mechanism to pick an algorithm in the intersection."

During the processes of messages' encryption, an intersection of algorithms preferred by the recipients **must** be used and a static platform-default should not be employed. When no preference has been chosen in the key, it should default to Triple DES (and ***not*** AES-256!). Interoperability issues aside, this problem is also important in case that some weaknesses of the algorithms are discovered, ensuring that OpenPGP key holders are protected by removing them from their preferences lists.

OP-01-008 Algorithm Preferences ignored upon Decryption *(Medium)*

Similarly to the issue noted in OP-01-007, this vulnerability addresses the fact that upon decrypting a message encrypted to a certain key, the library ignores algorithm preferences specified in that key. Instead, it is blindly accepting any implemented algorithm which the message is encrypted with. This can be observed in the following source code:

```
// src/packet/public_key_encrypted_session_key.js
PublicKeyEncryptedSessionKey.prototype.decrypt = function (key) {
  var result = crypto.publicKeyDecrypt(
    this.publicKeyAlgorithm,
    key.mpi,
    this.encrypted).toBytes();

  var checksum = util.readNumber(result.substr(result.length - 2));
```

[4] https://tools.ietf.org/html/rfc4880#section-13.2

```
  var decoded = crypto.pkcs1.eme.decode(
    result,
    key.mpi[0].byteLength());

  key = decoded.substring(1, decoded.length - 2);

  if (checksum != util.calc_checksum(key)) {
    throw new Error('Checksum mismatch');
  } else {
    this.sessionKey = key;
    this.sessionKeyAlgorithm =
      enums.read(enums.symmetric, decoded.charCodeAt(0));
// read the algorithm from the PKES packet
  }
};

//src/message.js
Message.prototype.decrypt = function(privateKey) {
  var encryptionKeyIds = this.getEncryptionKeyIds();
  if (!encryptionKeyIds.length) {
    // nothing to decrypt return unmodified message
    return this;
  }
  var privateKeyPacket = privateKey.getPrivateKeyPacket(encryptionKeyIds);
  if (!privateKeyPacket.isDecrypted) throw new Error('Private key is not
decrypted.');
  var pkESKeyPacketlist =
this.packets.filterByTag(enums.packet.publicKeyEncryptedSessionKey);
  var pkESKeyPacket;
  for (var i = 0; i < pkESKeyPacketlist.length; i++) {
    if (pkESKeyPacketlist[i].publicKeyId.equals(privateKeyPacket.getKeyId())) {
      pkESKeyPacket = pkESKeyPacketlist[i];
      pkESKeyPacket.decrypt(privateKeyPacket);
      break;
    }
  }
  if (pkESKeyPacket) {
    var symEncryptedPacketlist =
this.packets.filterByTag(enums.packet.symmetricallyEncrypted,
enums.packet.symEncryptedIntegrityProtected);
    if (symEncryptedPacketlist.length !== 0) {
      var symEncryptedPacket = symEncryptedPacketlist[0];
      symEncryptedPacket.decrypt(pkESKeyPacket.sessionKeyAlgorithm,
pkESKeyPacket.sessionKey); // ← algorithm taken blindly from PKES packet
      return new Message(symEncryptedPacket.packets);
    }
  }
};
```

Once again, interoperability problems occur and clear violation of the RFC 4880[5] takes place, as:

[5] https://tools.ietf.org/html/rfc4880#section-13.2

> If an implementation can decrypt a message that a keyholder doesn't have in their preferences, **the implementation SHOULD decrypt the message anyway, but MUST warn the keyholder that the protocol has been violated.** For example, suppose that Alice, above, has software that implements all algorithms in this specification. Nonetheless, she prefers subsets for work or home. If she is sent a message encrypted with IDEA, which is not in her preferences, the software warns her that someone sent her an IDEA-encrypted message, but it would ideally decrypt it anyway.

With this vulnerability in place, messages encrypted using algorithms with known weaknesses (or shorter key lengths) can be transparently decrypted by the library, regardless of the key holder explicitly opting-out of such algorithm usage. This can be demonstrated by encrypting a message with AES-128 and sending it to a recipient that allows only AES-256 and Triple DES:

```
$ gpg --edit aes256 showpref
gpg (GnuPG/MacGPG2) 2.0.19; Copyright (C) 2012 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Secret key is available.

pub  1024R/2CB935E4  created: 2014-01-30  expires: never       usage: SC
                     trust: ultimate      validity: ultimate
sub  1024R/CD145F74  created: 2014-01-30  expires: never       usage: E
[ultimate] (1). aes256 <aes256@example.com>

[ultimate] (1). aes256 <aes256@example.com>
     Cipher: AES256, 3DES
     Digest: SHA1
     Compression: ZIP, Uncompressed
     Features: MDC, Keyserver no-modify

$ echo "hello" | gpg -e --cipher-algo AES-128 -a -r aes256 > message.pgp
gpg: WARNING: forcing symmetric cipher AES (7) violates recipient preferences
```

Decrypting such a message succeeds without any warning:

```
openpgp.decryptMessage(priv,  openpgp.message.readArmored(t.value))
>> "hello
"
```

To fix this vulnerability, key-specified algorithm preference list should be checked upon decryption. For each and every case when algorithm that is not from the list is used, the decryption should either throw an error or some kind of warning messages channel should be implemented for library clients to use.

OP-01-009 Cleartext Messages Spoofing by Lax Armor Headers parsing *(**Critical**)*

When validating cleartext signed messages, the library ignores all cleartext armor headers as defined in the RFC 4880[6]. The “Hash” armor header is employed to inform the users of hash algorithms used to sign the message. As the library ignores all header values, it does not warn the user of a potential conflict between algorithms specified in the cleartext and the ones actually used in signature packet.

This allows the attacker to spoof a header value. For example, the following message is signed with a weak (in terms of collision-resistance) MD5 hash. Nonetheless, it verifies, thus giving the user a false sense of security.

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1024
Any-other-hashes: yes please!

signthis
-----BEGIN PGP SIGNATURE-----
Version: GnuPG/MacGPG2 v2.0.19 (Darwin)
Comment: GPGTools - http://gpgtools.org

iJwEAQEBAAYFAlLqpjYACgkQBDvBliy5NeRMYAQAt2S5W5uBYMdNMNC78nT8MLrz
OF+QQl+RwhUpBVR3qZ/VRetl8nKnKAsyCbAz/foHT4dnM1RSJYk9ultdA0s/IdtZ
HnPL9YkqDyWOkoQBBVKcGWjUrlZ9c0NsfrzJTq0qbHXywF0WxEn4si+ftGVqIjrO
FJSG9/Pw81DjckiUh/4=
=zi+b
-----END PGP SIGNATURE-----

openpgp.verifyClearSignedMessage(key.keys,
  openpgp.cleartext.readArmored(message_text)).signatures[0].valid
>> true
```

The hash algorithm specified in the armor header should be compared to what is specified in the following signature packet to prevent a mismatch. Similar spoofing in GnuPG results in a signature verification error:

```
$ gpg --verify clearsign.asc
gpg: Signature made Thu Jan 30 20:21:26 2014 CET using RSA key ID 2CB935E4
gpg: WARNING: signature digest conflict in message
gpg: Can't check signature: General error
```

Further code review shows that when the message is being de-armored, it is split on the first whitespace-only (or empty) line, with the first part being interpreted as headers and second as the message body. In accordance with RFC 4880 only the tab character (*0x09*) and space (*0x20*) are considered whitespace.

```
// src/encoding/armor.js
function splitHeaders(text) {
  var reEmptyLine = /^[\t ]*\n/m;
  var headers = "";
```

[6] https://tools.ietf.org/html/rfc4880#section-7

```
  var body = text;

  var matchResult = reEmptyLine.exec(text);

  if (matchResult !== null) {
    headers = text.slice(0, matchResult.index);
    body = text.slice(matchResult.index + matchResult[0].length);
  }

  return { headers: headers, body: body };
}
```

However, when de-armoring of the cleartext message takes place, headers are ignored and never parsed, as illustrated below:

```
// src/encoding/armor.js
function dearmor(text) {
  // ...
    // Reverse dash-escaping for msg and remove trailing whitespace at end of
line
    msg = splitHeaders(splittext[indexBase].replace(/^- /mg, '').replace(/[\t ]
+\n/g, "\n"));
    var sig = splitHeaders(splittext[indexBase + 1].replace(/^- /mg, ''));
    var sig_sum = splitChecksum(sig.body);

    result = { // <- headers ignored
      text:  msg.body.replace(/\n$/, '').replace(/\n/g, "\r\n"),
      data: base64.decode(sig_sum.body),
      type: type
    };

    checksum = sig_sum.checksum;
```

This is inconsistent with RFC 4880, section 6.2[7] which reads that:

> “The format of an Armor Header is that of a key-value pair. A colon (':' 0x38) and a single space (0x20) separate the key and value. **OpenPGP should consider improperly formatted Armor Headers to be corruption of the ASCII Armor.**”

Completely ignoring header parsing creates a critical vulnerability. In a nutshell, it not only allows the attacker to add arbitrary properly formed headers (like ‘Hash’ header described above), but to also permits to prepend arbitrary text to cleartext message using various unicode-based whitespace characters, all that without invalidating the signature. For example, despite appearing benign from the user’s perspective, the following message has been tampered with:

[7] https://tools.ietf.org/html/rfc4880#section-6.2

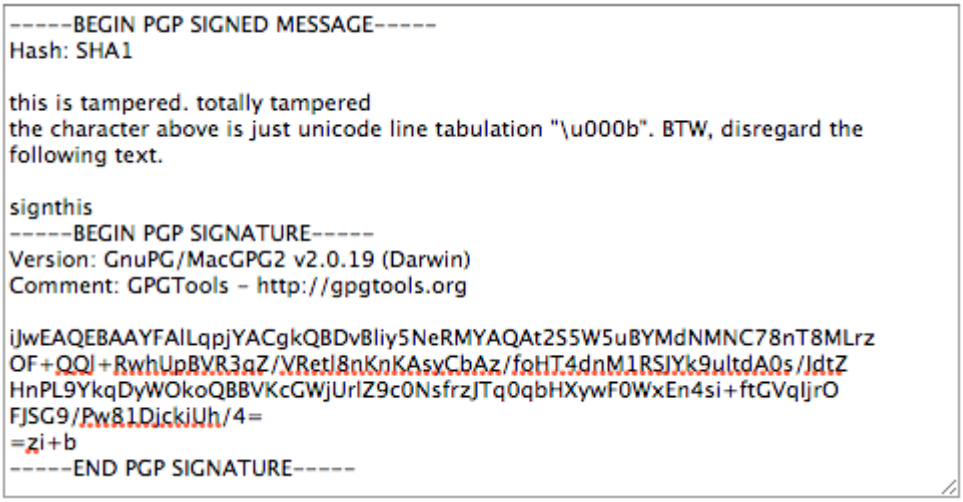

Fig.: Example view of a tampered message

```
c = "-----BEGIN PGP SIGNED MESSAGE-----\nHash: SHA1\n\u000b\nthis is tampered.
totally tampered\nthe character above is just unicode line
tabulation \"\\u000b\". BTW, disregard the following text.\n\nsignthis\n-----
BEGIN PGP SIGNATURE-----\nVersion: GnuPG/MacGPG2 v2.0.19 (Darwin)\nComment:
GPGTools -
http://gpgtools.org\n\niJwEAQEBAAYFAlLqpjYACgkQBDvBliy5NeRMYAQAt2S5W5uBYMdN…
0NsfrzJTq0qbHXywF0WxEn4si+ftGVqIjrO\nFJSG9/Pw81DjckiUh/4=\n=zi+b\n-----END PGP
SIGNATURE-----";

c.indexOf('\u000b')
>> 46

openpgp.verifyClearSignedMessage(key.keys,
openpgp.cleartext.readArmored(c)).signatures[0].valid
>> true
```

The library cuts off everything before *'signthis'* text and does not parse it even with the most rudimentary "*name: value\n*" rule. The user is here under the impression that the ignored headers actually form the real message. Other OpenPGP implementations correctly detect the invalid header part:

```
$ gpg --verify clearsign.asc
\v\n
gpg: invalid armor header:
gpg: Signature made Thu Jan 30 20:21:26 2014 CET using RSA key ID 2CB935E4
gpg: BAD signature from "aes256 <aes256@example.com>"
```

Conclusively, header values for cleartext messages must be parsed for correctness and the value used in the Hash header must be always compared to digest algorithm used in the signature packet.

OP-01-011 Error suppression in UTF-8 decoding function *(Medium)*

Upon trying to decode UTF-8 encoding, the *openpgp.util.decode_utf8()* function aborts on every error encountered and returns the original parameter passed. This might potentially be used to trigger a vulnerability in various locations of the library. As a rule, errors on any decoding procedures should always be thrown rather than silently ignored. Let us examine the following scenario:

```
// src/util.js
  /**
   * Convert a native JavaScript string to a string of utf8 bytes
   * @param {String} str The string to convert
   * @return {String} A valid squence of utf8 bytes
   */
  encode_utf8: function (str) {
    return unescape(encodeURIComponent(str));
  },

 /**
   * Convert a string of utf8 bytes to a native JavaScript string
   * @param {String} utf8 A valid squence of utf8 bytes
   * @return {String} A native JavaScript string
   */
  decode_utf8: function (utf8) {
    try {
      return decodeURIComponent(escape(utf8));
    } catch (e) {
      return utf8;
    }
  },
```

In fact, the following chameleon Object survives *decode_utf8* decoding, but presents itself as a string upon any utf-8 encoding:

```
> a = {chameleon: true, toString:function t() { if (t.caller.name == 'encodeURIComponent')
  {return 'innocent'} else return this }}
  ▶ Object {chameleon: true, toString: function}
> openpgp.util.decode_utf8(a)
  ▶ Object {chameleon: true, toString: function}
> openpgp.util.encode_utf8(a)
  "innocent"
```

Fig.: Chameleon object acting as a string upon being encoded

This technique therefore permits a creation of a keypair for some tricky user - trying to interfere with the affected software's logic:

```
> a = {chameleon: true, toString:function t() { if (t.caller.name ==
'encodeURIComponent') {return 'innocent'} else if (t.caller.name == 'escape')
{ return this } else return 'some-different-fellow' }}
```

```
// keypair will be generated for user 'innocent'
// (and this will be stored in armored message)
> kp = openpgp.generateKeyPair(1, 512, a, '1234')
> openpgp.key.readArmored(kp.publicKeyArmored).keys[0].users[0].userId.userid
"innocent"

// however an object is still present in KeyPair object
> kp.key.users[0].userId.userid
Object {chameleon: true, toString: function}
// and can report a different value when read as string
> ""+kp.key.users[0].userId.userid
"some-different-fellow"
```

Note: Beware of the impact of this vulnerability depending heavily upon applications which use the library and the type of validation that they perform. The library code should be changed to ensure that the decoding function always returns a String, throwing an error in case a different type was passed as an input parameter, or if UTF-8 decoding has failed.

OP-01-014 RSA Key Generation: Seeds are not destroyed *(Low)*

The FIPS documents outline a proper treatment of seeds upon key generation[8]:

> *"Prime number generation seeds SHALL be kept secret or destroyed when the modulus n is computed."*

The problem with the JavaScript cryptography implementation remains crucial and comes down to the fact that the random.js implementation uses the native *crypto.getRandomValues()*. This means that there is no access to the seeds and they cannot be destroyed. At the same time, OpenPGP.js is able to successfully run on a lot of different browsers, meaning that wherever a browser does not implement *crypto.getRandomValues()* correctly, all subsequent calls to *crypto.getRandomValues()* can be used to determine the previous calls to *crypto.getRandomValues()*. Same logic applies to browsers based on a system with weak crypto, like some older versions of Android. If a customized implementation of a PRNG was used, e.g. based on Fortuna or X9.61, the earlier potentially problem-causing seeds could be fully avoided.

[8] https://oag.ca.gov/sites/all/files/agweb/pdfs/erds1/fips_pub_07_2013.pdf

OP-01-015 EME-PKCS1-v1_5 padding uses Math.random() *(**Critical**)*

The EME-PKCS1-v1_5 padding used in RSA encryption scheme (as described in RFC 3447[9]) uses a pseudo-random number value. The library uses a weak number generation method - which is in fact the legendary *Math.random()*. *Math.random()* is not a cryptographically safe RNG and was found to be predictable (even cross-domain) in the past[10].

```
//crypto/pkcs1.js
module.exports = {
  eme: {
    /**
     * create a EME-PKCS1-v1_5 padding (See {@link
http://tools.ietf.org/html/rfc4880#section-13.1.1|RFC 4880 13.1.1})
     * @param {String} message message to be padded
     * @param {Integer} length Length to the resulting message
     * @return {String} EME-PKCS1 padded message
     */
    encode: function(message, length) {
      if (message.length > length - 11)
        return -1;
      var result = "";
      result += String.fromCharCode(0);
      result += String.fromCharCode(2);
      for (var i = 0; i < length - message.length - 3; i++) {
        result += String.fromCharCode(random.getPseudoRandom(1, 255));
      }
      result += String.fromCharCode(0);
      result += message;
      return result;
    },

// crypto/random.js
random.getPseudoRandom:  function(from, to) {
    return Math.round(Math.random() * (to - from)) + from;
}
```

The RSA encryption enjoys the same security level that it was initially assigned in case where no random padding is present. Some prerogatives are set when RFC 4880 requirements around random numbers' use is concerned:

> *"Certain operations in this specification involve the use of random numbers. An appropriate entropy source should be used to generate these numbers (see [RFC4086])."*

[9] http://tools.ietf.org/html/rfc3447#section-7.2.1
[10] http://ifsec.blogspot.com/2012/05/cross-domain-mathrandom-prediction.html

In comparison, libgcrypt[11] uses a CS-PRNG setup to produce random numbers of *GCRY_STRONG_RANDOM* quality in an equivalent function. GCRY_STRONG_-RANDOM quality should therefore be used for “session keys and similar purposes”[12].

```
// libgcrypt-1.6.0/cipher/rsa-common.c
gpg_err_code_t
_gcry_rsa_pkcs1_encode_for_enc (gcry_mpi_t *r_result, unsigned int nbits,
                                const unsigned char *value, size_t valuelen,
                                const unsigned char *random_override,
                                size_t random_override_len)
{
// ...
      p = _gcry_random_bytes_secure (i, GCRY_STRONG_RANDOM);
      /* Replace zero bytes by new values. */

// libgcrypt-1.6.0/random/random.c
/* The public function to return random data of the quality LEVEL;
   this version of the function returns the random in a buffer allocated
   in secure memory.  Caller must free the buffer. */
void *
_gcry_random_bytes_secure (size_t nbytes, enum gcry_random_level level)
{
  void *buffer;

  /* Historical note (1.3.0--1.4.1): The buffer was only allocated
     in secure memory if the pool in random-csprng.c was also set to
     use secure memory.  */
  buffer = xmalloc_secure (nbytes);
  do_randomize (buffer, nbytes, level);
  return buffer;
}

/* Helper function.  */
static void
do_randomize (void *buffer, size_t length, enum gcry_random_level level)
{
  if (fips_mode ())
    _gcry_rngfips_randomize (buffer, length, level);
  else if (rng_types.standard)
    _gcry_rngcsprng_randomize (buffer, length, level);
  else if (rng_types.fips)
    _gcry_rngfips_randomize (buffer, length, level);
  else if (rng_types.system)
    _gcry_rngsystem_randomize (buffer, length, level);
  else /* default */
    _gcry_rngcsprng_randomize (buffer, length, level);
}
```

A possible way of sufficiently mitigating this problem sufficiently is to consider following directives:

[11] http://www.gnu.org/software/libgcrypt/
[12] http://www.gnupg.org/documentation/manuals/gcrypt.pdf

- A removal of the method call to *random.getPseudoRandom()* - it appears to only be used in this very one place in library's codebase.
- An addition of the following code within pkcs1.js - it is to facilitate better randomness, as shown in the listing below:

```
function getPkcs1Padding(length) {
	var result = [];
	var randomByte;
	while (result.length < length) {
		randomByte = random.getSecureRandomOctet();
		if (randomByte.charCodeAt(0) !== 0) {
			result.push(randomByte);
		}
	}
	return result.join('');
}
```

OP-01-019 Cleartext Message Spoofing in Armor Headers *(**Critical**)*

In parsing the armor of a message, the *armor.getType()* method uses incorrect regular expressions that are largely too permissive. In essence, they are allowing an attacker to craft messages with malformed armor header-lines that will still be parsed as valid. An attacker can exploit this critical vulnerability to spoof cleartext signed messages.

According to RFC4880, header lines consist of a small number of key strings, e.g., "BEGIN PGP MESSAGE" surrounded by five dashes. The first line in *getType()* extracts all text surrounded by dashes and then matches it against the set of strings:

```
function getType(text) {
  var reHeader = /^-----([^-]+)-----$\n/m;

  var header = text.match(reHeader);
  // ...
  if (header[1].match(/BEGIN PGP MESSAGE, PART \d+\/\d+/)) {
      return enums.armor.multipart_section;
  }
```

The problem lies in the regular expression matching the extracted header against the strings. The understanding of a "match" equates it with any case where a string appears somewhere in the extracted string. Illustratively, it means that malformed headers like "DO NOT BEGIN PGP MESSAGE" also constitute a match.

Note that these types of malformed headers will be ignored or cause errors in other implementations, such as gnupg. However, here an attacker can create a message that will appear in one way to openpgp.js, but will come off differently to other clients. Analogically, since any character except "-" (\u002D) is matched by the first regular expression, even linebreaks and unicode characters will generate matches. In result, an attacker can craft a spoofed message that is visually indistinguishable from a valid message by disguising the spoofed part of the message inside the header line. The

Unicode character HYPHEN (*\u2010*) can be used to make the header line appear to end. Past that, any text (except "-") can be inserted, until a final set of five dashes ends the header line.

Here is an example of a spoofed message:

```
> c = '-----BEGIN PGP SIGNED MESSAGE\u2010\u2010\u2010\u2010\u2010\nHash:
SHA1\n\nIs this properly-----\n\nsigned?\n-----BEGIN PGP
SIGNATURE-----\nVersion: GnuPG v1.4.14
(GNU/Linux)\n\niJwEAQECAAYFAlLzAFYACgkQ4IT3RGwgLJdUpgQAiGXVoN7UfGOmyL+SbnlhzwcI\
nZJfprjF9ej3JkI1zFPucfcYJ0Y7dVokm0PY5UeIkEogyHfejC2vR8QVEZzM3Gc6T\nLEo5p3K3jgCJt
K36tdPeUk8k2l5Rs8q3Bnp2PNtb3NMIF4FwLg7RNUq2T8vI1j59\njl2Is6uC0OhMxSvjyEs=\n=Zgx8
\-----END PGP SIGNATURE-----'
-----BEGIN PGP SIGNED MESSAGE□□□□□
Hash: SHA1

Is this properly-----

signed?
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.14 (GNU/Linux)

iJwEAQECAAYFAlLzAFYACgkQ4IT3RGwgLJdUpgQAiGXVoN7UfGOmyL+SbnlhzwcI
ZJfprjF9ej3JkI1zFPucfcYJ0Y7dVokm0PY5UeIkEogyHfejC2vR8QVEZzM3Gc6T
LEo5p3K3jgCJtK36tdPeUk8k2l5Rs8q3Bnp2PNtb3NMIF4FwLg7RNUq2T8vI1j59
jl2Is6uC0OhMxSvjyEs=
=Zgx8
-----END PGP SIGNATURE-----'
> openpgp.verifyClearSignedMessage(key.keys,
openpgp.cleartext.readArmored(c)).signatures[0].valid
true
```

OP-01-020 Missing check in DSA signature generation *(Medium)*

The DSA signature generation algorithm is non-deterministic. Therefore when DSA signatures are being generated, the resulting parameters (derived from a random value) should not be equal to 0. This check is required by FIPS-186-4[13], which contains a proper clarification in Section 4.6:

> *"The values of r and s **shall** be checked to determine if r = 0 or s = 0. If either r = 0 or s = 0, a new value of k **shall** be generated, and the signature **shall** be recalculated. It is extremely unlikely that r = 0 or s = 0 if signatures are generated properly."*

The library does not perform said check. In the following code *s1* is *r*, and *s2* is *s* in respect to FIPS nomenclature:

```
// crypto/public_key/dsa.js
  function sign(hashalgo, m, g, p, q, x) {
```

[13] http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

```
    // If the output size of the chosen hash is larger than the number of
    // bits of q, the hash result is truncated to fit by taking the number
    // of leftmost bits equal to the number of bits of q.  This (possibly
    // truncated) hash function result is treated as a number and used
    // directly in the DSA signature algorithm.
    var hashed_data = util.getLeftNBits(hashModule.digest(hashalgo, m),
q.bitLength());
    var hash = new BigInteger(util.hexstrdump(hashed_data), 16);
    var k =
random.getRandomBigIntegerInRange(BigInteger.ONE.add(BigInteger.ONE),
q.subtract(BigInteger.ONE));
    var s1 = (g.modPow(k, p)).mod(q);
    var s2 = (k.modInverse(q).multiply(hash.add(x.multiply(s1)))).mod(q);
    var result = [];
    result[0] = s1.toMPI();
    result[1] = s2.toMPI(); // <- should assert that s1 !== 0 and s2 !== 0
    return result;
  }
```

The probability of *r* or *s* being 0 is extremely low, yet if this ever occurs everybody with access to this signature will see that the private key is probably not well formed, resulting in a rather high damage. Tackling this problem entails including the discussed check in the algorithm.

OP-01-024 Random Range Bias in DSA/Elgamal *(Low)*

When generating of a DSA signature takes place, a random and secret number *k* must be generated per each message. This number should be in the [**1**, q-1] (inclusive) range, as noted by e.g. FIPS 186-4, Appendix B.2.1[14]:

> *"k and −1 k are in the range [1, q–1]."*

However, in the code the parameter k is allowed within a different range, namely [**2**, q-1]:

```
// crypto/public_key/dsa.js
function DSA() {
  // s1 = ((g**s) mod p) mod q
  // s1 = ((s**-1)*(sha-1(m)+(s1*x) mod q)
  function sign(hashalgo, m, g, p, q, x) {
    var hashed_data = util.getLeftNBits(hashModule.digest(hashalgo, m),
q.bitLength());
    var hash = new BigInteger(util.hexstrdump(hashed_data), 16);
    var k =
random.getRandomBigIntegerInRange(BigInteger.ONE.add(BigInteger.ONE),
q.subtract(BigInteger.ONE));
    var s1 = (g.modPow(k, p)).mod(q);
    var s2 = (k.modInverse(q).multiply(hash.add(x.multiply(s1)))).mod(q);
    var result = [];
    result[0] = s1.toMPI();
    result[1] = s2.toMPI();
    return result;
```

[14] http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

```
}
```

The *random.getRandomBigIntegerInRange()* method's parameters define an inclusive range, so the resulting range reads as [1+1, q-1]. This gives a tiny bias to the generated k. A Similar issue was discovered with ElGamal encryption: Parameter k should be in the [1, p-2] range but is in the [2,p-2] range instead:

```
function Elgamal() {

  function encrypt(m, g, p, y) {
    //  choose k in {2,...,p-2}
    var two = BigInteger.ONE.add(BigInteger.ONE);
    var pMinus2 = p.subtract(two);
    var k = random.getRandomBigIntegerInRange(two, pMinus2);
    k = k.mod(pMinus2).add(BigInteger.ONE);
    var c = [];
    c[0] = g.modPow(k, p);
    c[1] = y.modPow(k, p).multiply(m).mod(p);
    return c;
  }
```

OP-01-025 EME-PKCS1-v1_5 Error Handling in RSA Decryption *(High)*

During our review of the EME-PKCS1-v1_5 implementation used in OpenPGP RSA encryption/decryption, several inconsistencies were spotted against the specification. RFC 3447 (*Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1)*, section 7.2.2[15] specifies the procedure that should be followed upon decryption:

> "3. EME-PKCS1-v1_5 decoding: Separate the encoded message EM into an octet string PS consisting of nonzero octets and a message M as
>
> EM = 0x00 || 0x02 || PS || 0x00 || M.
>
> **If the first octet of EM does not have hexadecimal value 0x00, if the second octet of EM does not have hexadecimal value 0x02, if there is no octet with hexadecimal value 0x00 to separate PS from M, or if the length of PS is less than 8 octets, output "decryption error" and stop.** (See the note below.)
>
> 4. Output M.
>
> **Note. Care shall be taken to ensure that an opponent cannot distinguish the different error conditions in Step 3, whether by error message or timing. Otherwise an opponent may be able to obtain useful information about the decryption of the ciphertext C**, leading to a strengthened version of Bleichenbacher's attack [6]; compare to Manger's attack [36].:"

The library implements the decoding as follows:

[15] http://www.ietf.org/rfc/rfc3447.txt

```
// src/crypto/pkcs1.js
/**
 * decodes a EME-PKCS1-v1_5 padding (See {@link
http://tools.ietf.org/html/rfc4880#section-13.1.2|RFC 4880 13.1.2})
 * @param {String} message EME-PKCS1 padded message
 * @return {String} decoded message
 */
decode: function(message, len) {
  if (message.length < len)
    message = String.fromCharCode(0) + message;
  if (message.length < 12 || message.charCodeAt(0) !== 0 ||
message.charCodeAt(1) != 2)
    return -1;
  var i = 2;
  while (message.charCodeAt(i) !== 0 && message.length > i)
    i++;
  return message.substring(i + 1, message.length);
}
```

Several problems have been spotted and must be noted here:

1. The library does not validate if a *0x00* byte separates the PS from M (in RFC nomenclature).
2. The library does not check the length of PS. The length of this padding is important and should be "*at least eight octets long, which is a security condition for public-key operations that makes it difficult for an attacker to recover data by trying all possible encryption blocks*" (RFC 3447).
3. Consideration should be given to making the decode process happen in constant-time. Due to short-circuit evaluation[16] of logical 'or' operator in JavaScript, the subsequent checks will not be performed if left-hand side conditions are met. This grants a tiny timing leak, which makes it possible for the attacker to distinguish the type of the padding error. These timing leaks are described in RFC 3447. To follow an example of constant-time EME-PKCS1-v1_5 decoding, please refer to PolarSSL source code[17].

OP-01-026 Errors in EMSA-PKCS1-v1_5 decoding routine *(High)*

The library uses RSASSA-PKCS-v.1.5 scheme (with EMSA-PKCS1-v1_5 padding) for verifying RSA signatures. As described in OP-01-018, the library's approach is to extract the hash value of padded message. As such, it implements the padding decoding routine, in which we have noticed several vulnerabilities that can potentially allow for a signature forgery.

```
// src/crypto/pkcs1.js
/**
 * extract the hash out of an EMSA-PKCS1-v1.5 padding (See {@link
http://tools.ietf.org/html/rfc4880#section-13.1.3|RFC 4880 13.1.3})
 * @param {String} data Hash in pkcs1 encoding
```

[16] http://en.wikipedia.org/wiki/Short-circuit_evaluation
[17] https://polarssl.org/rsa-source-code

```
 * @returns {String} The hash as string
 */
decode: function(algo, data) {
  var i = 0;
  if (data.charCodeAt(0) === 0) i++;
  else if (data.charCodeAt(0) != 1) return -1;
  else i++;

  while (data.charCodeAt(i) == 0xFF) i++;
  if (data.charCodeAt(i++) !== 0) return -1;
  var j = 0;
  for (j = 0; j < hash_headers[algo].length && j + i < data.length; j++) {
    if (data.charCodeAt(j + i) != hash_headers[algo][j]) return -1;
  }
  i += j;
  if (data.substring(i).length < hash.getHashByteLength(algo)) return -1;
  return data.substring(i);
}
```

A correctly padded message should be of the following form:

```
EM = 0x00 || 0x01 || PS || 0x00 || T.
```

where “PS” is at least 8 octets of “FF” value.

The following decoding errors were identified:

1. The function seems to be faulty when the first *if* statement is checked:
   - 00 FF 00 AA will pass but should fail
   - 01 FF 00 AA will pass but should fail
   - 00 01 FF FF 00 AA will fail, but should pass.
2. It seems that it only works when the leading zero is removed outside the procedure. The library also fails to detect the missing 01 value before the FF (first test case). There should be at least 8 octets of FF value. Library fails to check whether this requirement is met.
3. Another vulnerability comes from the missing check of the hash length. NIST FIPS 186-4, section 5.5, specifies additional requirements for handling of the padding during signature verification:

   “(f) For RSASSA-PKCS-v1.5, when the hash value is recovered from the encoded message EM during the verification of the digital signature, the extraction of the hash value ***shall*** *be accomplished by either:*
   • ***Selecting the rightmost (least significant) bits of EM****, based on the size of the hash function used, regardless of the length of the padding, or*
   • If the hash value is selected by its location with respect to the last byte of padding, including a check that the hash value is located in the ***rightmost*** *(least significant) bytes of EM (i.e.,* ***no other information follows the hash value in the encoded message****)”.*

In other words, special care needs to be given to ensure that the hash value encoded in the padding scheme is its last part (ie. no other values are appended). While no clear mention is given in FIPS 186-4 on reasoning for such a check, we suspect it is due to Bleichenbacher's RSA PKCS#1 signature forgery[18] and similar attacks[19]. This attack has been demonstrated and discussed earlier e.g. against OpenSSL[20].

The library only checks if there are enough bytes for a hash to fit and returns all bytes from a given offset. As a consequence, EMSA decoding routine will not detect arbitrary garbage being appended to an original hash and return it as well. Ultimately, the returned value will fail hash comparison in the caller.

```
// src/crypto/signature.js
var hash = pkcs1.emsa.decode(hash_algo, dopublic.toMPI().substring(2));
if (hash == -1) {
  throw new Error('PKCS1 padding in message or key incorrect.
Aborting...');
}
return hash == calc_hash;
```

As no reasoning is given in FIPS 186-4 specification, an exploit for such vulnerability cannot be demonstrated at this point. Still, we recommend to implement the length check nonetheless. At the end of the message there should be just enough bytes left to fit a hash, no surplus is desired.

The library implements EMSA-PKCS1-v1_5 decoding algorithm which, unlike encoding ones, is not clearly specified. The tests for this procedure are also missing from the library's codebase. We strongly recommend to change the approach to regenerate and compare the padding message instead of simply decoding it. Additional details on the motivations behind our advice are given in OP-01-018.

[18] http://www.imc.org/ietf-openpgp/mail-archive/msg06063.html
[19] http://www.cdc.informatik.tu-darmstadt.de/reports/reports/sigflaw.pdf
[20] http://www.openssl.org/news/secadv_20060905.txt

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of those findings are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while the vulnerability has been discovered, an exploit might not always be possible.

OP-01-001 Type confusion in crypto.random.RandomBuffer *(Low)*

For environments where secure *window.crypto.getRandomValues()* or *nodeCrypto* RNGs are unavailable, the library uses a buffer for externally generated random numbers. One notable usage of this logic is running the library in a HTML5 Web Worker[21]. In that case random numbers are being generated in main application window and are passed to Web Worker using HTML5 Web Messaging. There, Web Worker uses the random numbers provided for doing crypto operations, probing for new numbers when buffer is low.

The buffer used to hold the random numbers is believed to be of *Uint32Array* type (unsigned 32-bit integers), which can be seen in *src/crypto/random.js*:

```
/**
 * Initialize buffer
 * @param  {Integer} size size of buffer
 */
RandomBuffer.prototype.init = function(size) {
  this.buffer = new Uint32Array(size);
  this.size = 0;
};

/**
 * Concat array of secure random numbers to buffer
 * @param {Uint32Array} buf
 */
RandomBuffer.prototype.set = function(buf) {
...
};

/**
 * Take numbers out of buffer and copy to array
 * @param {Uint32Array} buf the destination array
 */
RandomBuffer.prototype.get = function(buf) {
...
};
```

Surprisingly, the actual buffers being used when *RandomBuffer.get()/set()* is called belong to *Uint8Array* type (8-bit unsigned numbers). For instance, when application generates random numbers and sends it to a Web Worker, the following code is used:

[21] http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html

```
// src/worker/async_proxy.js

/**
 * Send message to worker with random data
 * @param  {Integer} size Number of bytes to send
 */
AsyncProxy.prototype.seedRandom = function(size) {
  var buf = this.getRandomBuffer(size);
  this.worker.postMessage({event: 'seed-random', buf: buf});
};

/**
 * Get Uint8Array with random numbers
 * @param  {Integer} size Length of buffer
 * @return {Uint8Array}
 */
AsyncProxy.prototype.getRandomBuffer = function(size) {
  if (!size) return null;
  var buf = new Uint8Array(size);
  crypto.random.getRandomValues(buf);
  return buf;
};

// src/crypto/random.js
  /**
   * Helper routine which calls platform specific crypto random generator
   * @param {Uint8Array} buf
   */
  getRandomValues: function(buf) {
    if (typeof window !== 'undefined' && window.crypto) {
      window.crypto.getRandomValues(buf);
    } else if (nodeCrypto) {
      var bytes = nodeCrypto.randomBytes(buf.length);
      buf.set(bytes);
    } else if (this.randomBuffer.buffer) {
      this.randomBuffer.get(buf);
    } else {
      throw new Error('No secure random number generator available.');
    }
  },
```

This code will generate random numbers using native CSPRNG, put it in *Uint8Array* and send it to Worker, which then executes the following when retrieving the numbers:

```
// src/worker/worker.js
onmessage = function (event) {
  var data = null,
      err = null,
      msg = event.data,
      correct = false;
  switch (msg.event) {
    case 'seed-random':
      window.openpgp.crypto.random.randomBuffer.set(msg.buf);
      break;
```

This code will fill Worker's internal 32-bit-based random number buffer with 8-bits long integers.

Only three instances of using *crypto.random.getRandomValues()* in the library code were found and all pass newly created *Uint8Array* to this method. That confirms that in the current version this issue causes no vulnerability, as only 8-bit random numbers are expected anyway. Nevertheless, type confusions in weakly-typed JavaScript applications have been known to lead to catastrophic crypto vulnerabilities in the past (see e.g. DecryptoCat[22]). For this reason, we strongly recommend and advocate to:

1. Use *Uint8Array* in *RandomBuffer* implementation and documentation, as this seems consistent with other types used by the library;
2. Validate the type of array passed to *RandomBuffer* functions and to *crypto.random.getRandomValues()* method.

Alternatively, API could be simplified so that *getRandomValues()* creates *Uint8Array* itself, eliminating the possibility of type confusion in the caller altogether.

OP-01-002 Math.random() usage in dead Code Branch *(Low)*

During code review of the library we encountered *Math.random()* call in a dead code branch. While this presents no vulnerability at present, it should still be removed, as it can facilitate the prevention strategies implementations and assist in avoiding possible future mistakes of using *Math.random()* as CSPRNG.

```
// crypto/cipher/twofish.js
function randByte() {
  return Math.floor(Math.random() * 256);
}
```

OP-01-003 Suggested Code Enforcement of RandomBuffer *(Low)*

The library uses *RandomBuffer* functions for a feature described in OP-01-001. Every *RandomBuffer* instance holds a public variable *buffer* which contains random numbers currently in the buffer. New numbers are added with the use of *RandomBuffer.set()*, and they retrieved through *RandomBuffer.get()*.

Once a calling to *RandomBuffer.get()* takes place, random numbers are copied to a new buffer and returned to be called. Yet they are not cleared from internal buffer traces (i.e. respective buffer elements are not zeroed), only the index used for accessing the next number is changed:

```
// src/crypto/random.js
/**
 * Take numbers out of buffer and copy to array
 * @param {Uint32Array} buf the destination array
```

[22] http://tobtu.com/decryptocat.php

```
 */
RandomBuffer.prototype.get = function(buf) {
  if (!this.buffer) {
    throw new Error('RandomBuffer is not initialized');
  }
  if (this.size < buf.length) {
    throw new Error('Random number buffer depleted.')
  }
  for (var i = 0; i < buf.length; i++) {
    buf[i] = this.buffer[--this.size]; // number copied, and not cleared
  }
};
```

One can see that in action by executing the following JS code:

```
B = new RandomBuffer();
B.init(8);
// B.buffer == [0, 0, 0, 0, 0, 0, 0, 0]
// B.size == 0

random = new Uint32Array(2);
random[0] = 4;
random[1] = 5;
B.set(random);
// B.buffer == [4, 5, 0, 0, 0, 0, 0, 0]
// B.size == 2

get = new Uint32Array(1);
B.get(get);
// get == [5]
// B.buffer == [4, 5, 0, 0, 0, 0, 0, 0]
// B.size == 1
```

This signifies that there is a potential for retrieving past random numbers by a skilled attacker who injects the code in the scope of *RandomBuffer* being in operation. This is not a vulnerability in the tested library but may come handy when exploitation of said vulnerability is possible in applications using the library. For example, with XSS vulnerability attacker might not only snoop around or set random numbers to be used in the future, hence tampering with the whole application, but also gets access to numbers generated in the past.

We propose to enforce the design by clearing the number from internal buffer once it is used. Additionally, direct access to *RandomBuffer.buffer* may be prevented by turning it into a private variable[23]. This strategy would make *RandomBuffer* a more locked-down container for random numbers, capable of resisting some attacks even in the presence of code injection vulnerability.

[23] http://JavaScript.crockford.com/private.html

OP-01-004 Inconsistent Bit Length of RSA Keys *(Low)*

During code review for RSA key pair generation we encountered issue #67 - *Inconsistent bit length on key generation*[24]. We verified that RSA keys generated by the library will sometimes be one or more bits shorter than expected. That is because multiplying random numbers from a given range does not guarantee a product with a specific bit-length. This has little security impact, but marks an inconsistency with other OpenPGP libraries.

To narrow down this issue we reviewed the RSA key generation code of *libgcrypt*[25]. The fix applied there[26] simply repeats RSA *p* and *q* generation if bit length of n = *p*q* is smaller than expected. The same approach can be introduced in Openpgp.js. Code snippet from *generate*() function ensures the RSA key will be of expected length (though at a certain performance cost).

```
// src/crypto/public_key/rsa.js, function generate()
      if (key.p.compareTo(key.q) <= 0) {
        var t = key.p;
        key.p = key.q;
        key.q = t;
      }

      key.n = key.p.multiply(key.q);
      if (key.n.bitLength() < B) { // force n to be B bits long
        continue;
      }

      var p1 = key.p.subtract(BigInteger.ONE);
      var q1 = key.q.subtract(BigInteger.ONE);
      var phi = p1.multiply(q1);
      if (phi.gcd(key.ee).compareTo(BigInteger.ONE) === 0) {
        key.d = key.ee.modInverse(phi);
        key.dmp1 = key.d.mod(p1);
        key.dmq1 = key.d.mod(q1);
        key.u = key.p.modInverse(key.q);
        break;
      }
```

OP-01-006 Generated keys have no stored algorithm preference *(Medium)*

When keys are generated no symmetric algorithm preference is being saved in the self-signature.

```
// src/key.js
function generate(keyType, numBits, userId, passphrase) {
  var packetlist = new packet.List();

  //...
```

[24] https://github.com/openpgpjs/openpgpjs/issues/67
[25] http://www.gnu.org/software/libgcrypt/
[26] http://git.gnupg.org/cgi-bin/gitweb.cgi?p=libgcrypt.git;a=blob;f=cipher/rsa.c;h=e595e386eab8d2789e55cace2b50ebb5a6337544;hb=HEAD#l270

```
  var signaturePacket = new packet.Signature();
  // only a few properties are changed
  signaturePacket.signatureType = enums.signature.cert_generic;
  signaturePacket.publicKeyAlgorithm = keyType;
  //TODO we should load preferred hash from config, or as input to this function
  signaturePacket.hashAlgorithm = enums.hash.sha256;
  signaturePacket.keyFlags = [enums.keyFlags.certify_keys |
enums.keyFlags.sign_data];

  //...

  var subkeySignaturePacket = new packet.Signature();
  // only a few properties are changed for the subkey signature
  subkeySignaturePacket.signatureType = enums.signature.subkey_binding;
  subkeySignaturePacket.publicKeyAlgorithm = keyType;
  //TODO we should load preferred hash from config, or as input to this function
  subkeySignaturePacket.hashAlgorithm = enums.hash.sha256;
  subkeySignaturePacket.keyFlags = [enums.keyFlags.encrypt_communication |
enums.keyFlags.encrypt_storage];

  // ...
  packetlist.push(secretKeyPacket);
  packetlist.push(userIdPacket);
  packetlist.push(signaturePacket);
  packetlist.push(secretSubkeyPacket);
  packetlist.push(subkeySignaturePacket);

  return new Key(packetlist);
}

// src/packet/signature.js
function Signature() {
  // default values for properties
  //...
  this.preferredSymmetricAlgorithms = null;
  this.preferredHashAlgorithms = null;
  this.preferredCompressionAlgorithms = null;
}
```

As a result, the keypair generated with *openpgp.generateKeyPair()* has no algorithm preferences:

```
kp2 = openpgp.generateKeyPair(1, 1024, 'pq@example.com', '1234')
kp2.key.getPrimaryUser().selfCertificate.preferredSymmetricAlgorithms === null
```

All encryption and decryption procedures ignore algorithm preferences (which has been already described in separate issues above). As the library always defaults to AES-256 instead, interoperability problems emerge and violation of RFC 4880 [27] is undeniable:

[27] https://tools.ietf.org/html/rfc4880#section-13.2

“13.2 Symmetric algorithm preferences

The symmetric algorithm preference is an ordered list of algorithms that the keyholder accepts. Since it is found on a self-signature, it is possible that a keyholder may have multiple, different preferences. For example, Alice may have TripleDES only specified for "alice@work.com" but CAST5, Blowfish, and TripleDES specified for "alice@home.org". Note that it is also possible for preferences to be in a subkey's binding signature.

Since TripleDES is the MUST-implement algorithm, if it is not explicitly in the list, it is tacitly at the end. However, it is good form to place it there explicitly. Note also that **if an implementation does not implement the preference, then it is implicitly a TripleDES-only implementation.”**

To comply with the above, algorithm preference during keys’ generation needs to be saved either with the key explicitly or a platform default should be changed to Triple DES.

OP-01-010 Invalid Armor Checksum Validation *(Low)*

ASCII armored messages are accompanied by CRC-24 checksum to detect modifications (e.g. errors during message transport). Checksum itself is base64 encoded, thus always 4 bytes long (24 bits * 4/3 = 32 bits), amounting to **four** characters in ASCII encoding. However, during checksum verification, only **first three** characters are taken into account:

```
// src/encoding/armor.js
function getCheckSum(data) {
  var c = createcrc24(data);
  var str = "" + String.fromCharCode(c >> 16) +
    String.fromCharCode((c >> 8) & 0xFF) +
    String.fromCharCode(c & 0xFF); // <- 3 characters raw checksum
  return base64.encode(str); // <- 4 character encoding of the above
}

function verifyCheckSum(data, checksum) {
  var c = getCheckSum(data);
  var d = checksum;
  return c[0] == d[0] && c[1] == d[1] && c[2] == d[2];
}
```

It is possible to modify the checksum ‘s last character (or append arbitrary characters to it) without risking that the message will not be de-armored. For example, the following message de-armors correctly:

```
-----BEGIN PGP PRIVATE KEY BLOCK-----
Version: OpenPGP.js v0.3.0
Comment: http://openpgpjs.org

xbYEUubX7gEBANDWhzoP+Tr/IyRSv++vl5jBesQIPTYGQBdzF4YDnGEBABEB
```

```
AAH+CQMIfzdw4/PKNl5gVXdtfDFdSIN8yJT2rbeg3+SsWexXZNNdRaONWaiB
Z5cG9Q6+BoXKsEshIdcYOgwsAgRxlPpRA34Vvmg2QBk7PhdrkbK7aqENsJ1w
dIlLD6p9GmLE20yVff58/fMiUtPRgsD83SpKTAX6EM1ulpkuQQNjmrVc5qc8
7AMdF80JdW5kZWZpbmVkwj8EEAEIABMFAlLm1+4JEBD8MASZrpALAhsDAAAs
QgD8CUrwv7Hrp/INR0/UvAvzS52VztREQwQWTJMrgTNHBGjHtgRS5tfuAQEA
nys9SaSgR+l6iZc/M8hGIUmbuahE2/+mtw+/l0RO+WcAEQEAAf4JAwjr39Yi
FzjxImDN1IoYVsonA9M+BtIIJHafuQUHjyEr1paJJK5xS6KlyGgpMTXTD6y/
qxS3ZSPPzHGRrs2CmkVEiPmurn9Ed05tb0y9OnJkWtuh3z9VVq9d8zHzuENa
bUfli+P/v+dRaZ+1rSOxUFbFYbFB5XK/A9b/OPFrv+mb4KrtLxugwj8EGAEI
ABMFAlLm1+4JEBD8MASZrpALAhsMAAC3IgD8DnLGbMnpLtrX72RCkPW1ffLq
71vlXMJNXvoCeuejiRw=
=wJNM
-----END PGP PRIVATE KEY BLOCK-----
```

To reiterate, the following message checksum modification is not detected:

```
-----BEGIN PGP PRIVATE KEY BLOCK-----
Version: OpenPGP.js v0.3.0
Comment: http://openpgpjs.org

xbYEUubX7gEBANDWhzoP+Tr/IyRSv++vl5jBesQIPTYGQBdzF4YDnGEBABEB
AAH+CQMIfzdw4/PKNl5gVXdtfDFdSIN8yJT2rbeg3+SsWexXZNNdRaONWaiB
Z5cG9Q6+BoXKsEshIdcYOgwsAgRxlPpRA34Vvmg2QBk7PhdrkbK7aqENsJ1w
dIlLD6p9GmLE20yVff58/fMiUtPRgsD83SpKTAX6EM1ulpkuQQNjmrVc5qc8
7AMdF80JdW5kZWZpbmVkwj8EEAEIABMFAlLm1+4JEBD8MASZrpALAhsDAAAs
QgD8CUrwv7Hrp/INR0/UvAvzS52VztREQwQWTJMrgTNHBGjHtgRS5tfuAQEA
nys9SaSgR+l6iZc/M8hGIUmbuahE2/+mtw+/l0RO+WcAEQEAAf4JAwjr39Yi
FzjxImDN1IoYVsonA9M+BtIIJHafuQUHjyEr1paJJK5xS6KlyGgpMTXTD6y/
qxS3ZSPPzHGRrs2CmkVEiPmurn9Ed05tb0y9OnJkWtuh3z9VVq9d8zHzuENa
bUfli+P/v+dRaZ+1rSOxUFbFYbFB5XK/A9b/OPFrv+mb4KrtLxugwj8EGAEI
ABMFAlLm1+4JEBD8MASZrpALAhsMAAC3IgD8DnLGbMnpLtrX72RCkPW1ffLq
71vlXMJNXvoCeuejiRw=
=wJN@
-----END PGP PRIVATE KEY BLOCK-----

openpgp.message.readArmored(t.value)
>> Message {packets: Packetlist, getEncryptionKeyIds: function,
getSigningKeyIds: function, decrypt: function, getLiteralData: function…}
```

Additionally, a minor error was spotted in error message displayed when checksum fails. An object is passed to *getChecksum* (instead of a string), causing the message to always mention static *'twTO'* checksum instead of the actually expected value.

```
// function dearmor(text)  {
  // ...
  if (!verifyCheckSum(result.data, checksum)) {
    throw new Error("Ascii armor integrity check on message failed: '" +
      checksum +
      "' should be '" +
      getCheckSum(result) + "'");
  } else {
    return result;
  }
```

```
}
```

Another minor error was noted in the process of extracting a checksum from a message, when it was found to contain a newline character in the end. That part should be removed for checksum comparison, in hopes of avoiding parsing problems.

OP-01-012 RNG Bias in RSA Key Generation *(Low)*

When generating RSA keys, library picks up a large random number (prime candidate), and then applies the Miller-Rabin to assure that the generated number is prime with a certain probability. When number fails that test, the candidate is increased by two.

In *jsbn.fromInteger*:

```
// crypto/public_key/jsbn.js
while (!this.isProbablePrime(b)) {
    this.dAddOffset(2, 0);
    if (this.bitLength() > a) this.subTo(BigInteger.ONE.shiftLeft(a - 1), this);
}
```

This key generation does not conform to FIPS 186-4[28] “Generation of Random Primes that are Probably Prime” algorithm (described in B.3.3 section of the document):

```
4.2.    Obtain a string p of (nlen/2) bits from an RBG that supports
        the security_strength.
4.3.    If (p is not odd), then p = p + 1.
4.4.    If ((p<(√2)(2^(nlen/2)−1)), then go to step 4.2.
4.5.    If (GCD(p−1, e) = 1), then
        4.5.1 Test p for primality as specified in Appendix C.3,
        using an appropriate value from Table C-2 or C-3 in Appendix
        C.3 as the number of iterations.
        4.5.2 If p is PROBABLY PRIME, then go to step 5.
4.6.    i= i + 1.
4.7.    If (i ≥ 5(nlen/2)), then return (FAILURE, 0, 0) Else go to
        step 4.2.
```

When a prime candidate fails primality test, a new random odd number (prime candidate) should be picked. This is to prevent a RNG bias for primes with larger prime gaps[29] (see e.g. “Do Gaps Between Primes Affect RSA Keys?”[30]).

OP-01-013 RSA Key Gen.: Miller-Rabin-Test not conform with FIPS 186-4 *(Low)*

The generation of a RSA Private Key is designed to find two appropriate prime numbers. In order to test whether a given number is prime or not, the Miller-Rabin-Test, which is a part of Tom Wu’s jsbn.js library, is used. According to NIST 186-4 C.3.1, Miller-Rabin Test is specified as follows:

---

[28] http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf
[29] http://en.wikipedia.org/wiki/Prime_gap
[30] http://rjlipton.wordpress.com/2012/03/01/do-gaps-between-primes-affect-rsa-keys/

“for a given candidate w
4.1 Obtain a string b of wlen bits from an RBG.
Comment: Ensure that 1 < b < w-1.”

So b is a random number between 2 and w-1 inclusive. However, the implementation in jsbn.js performs the following operation:

```
j = lowprimes[Math.floor(Math.random() * lowprimes.length)];
```
(with “j” being the “b” mentioned by NIST).

This means, j is always chosen (nearly predictable!) within the lowest 168 primes. On a side note, the PRNG was passed as a parameter to the calling function, so it would not be a problem to pass it here too and use it here instead of *Math.random()*. The important difference, however, while it may have no influence on the generated keys, could make it possible to enforce a scenario where p and q are not prime! Luckily, it seems extremely unlikely to indeed get p or q other than prime.

OP-01-016 Comments on Javascript Code Quality *(Low)*

During the library source code audit we noticed a few issues concerning code quality that *per se* do not represent any vulnerabilities, but may be noted for future endeavors. This ticket therefore serves as a polite list that should be kept in mind, so that later changes do not lead to vulnerabilities based on these already known issues. Below we suggest specific improvements pertaining to our findings.

JSDoc should be validated with jshint / jscs

The JSDoc is being published, so it needs to be correct and therefore requires automatic tests to the actual code. Sometimes the JSDoc documentation differs from the code, e.g.

```
// packet/public_key.js
    /**
     * Internal Parser for public keys as specified in {@link http://tools.i ...
}
     * called by read_tag&lt;num&gt;
     * @param {String} input Input string to read the packet from
     * @return {Object} This object with attributes set by the parser
     */
    PublicKey.prototype.read = function (bytes) {
```

To underscore: the parameter referenced here is not “input” but in fact “bytes”.

Decrease size of API footprint

Some publicly exposed API functions are considered unnecessary and should be removed, e.g.

```
// packet/public_key.js
    /**
```

```
     * Alias of read()
     * @see module:packet/public_key~PublicKey#read
     */
    PublicKey.prototype.readPublicKey = PublicKey.prototype.read;
```

In case when a public function is being aliased, at least one of the aliased nodes should be deprecated to avoid developer confusion and faulty usage in the future.

Issues noticed in relation to JSHint

Several minor issues with JSHint were noticed and should be addressed:

- JSHint is referenced in the *Gruntfile.js* but there is no devDependency in the *package.json*;
- JSHint should be configured properly;
- In the Gruntfile.js, JSHint is just called with defaults and there is no *.jshint.rc* file present in the root;
- Consider an appropriate set of options, especially because the "latest" version dependency is used in *package.json*;
- JSHint is not integrated in the build chain, and a lot of JSHint warnings are emitted there once "*grunt jshint*" is being run. It is recommended to add the JSHint task as a task dependency;
- It is advised to not only “JSHint” the *./src* folder, but also *./test* and *./*js* folders in hopes of full coverage.

More scrutiny recommended with Bitwise operators

It is recommended to restrict Bitwise operators only to modules which specifically need them (such as SHA or AES). Bitwise should not be used when redundant (for example in *random.js*). Bitwise operators often work “in strange ways”, because they operate “signedly”. Take following snippet as an example of the “unpredictability”:

```
var x = 0x800000000;
x | 0 === x; // false!
```

Code Coverage

Code coverage is currently not being checked for. Code coverage checks can be of great help, when writing and completing test cases.

String vs Uint8Array

It’s recommended to consider a change from *String* to *Uint8Array*. Currently all interfaces use *String* as a byte buffer. For several years now, one can see Unit8Array vastly present across various browser engines. Perhaps a change from immutable Strings to Unit8Arrays should be considered as a brave step towards change?

OP-01-017 Consider to substitute the SHA module *(Low)*

Sha module should perhaps be substituted or at the very least updated to the new version on GitHub: https://github.com/Caligatio/jsSHA There is no SHA-3 present at the time of writing. The reasoning can be illustrated with the error example which shows aspects of signed versus unsigned Bitwise operators that should be avoided:

```
function rotl_32(x, n)
	{
		return (x << n) | (x >>> (32 - n));
	}
```

When the high bit is set, the function returns negative values. While this was manageable here, it is speculated to be a JavaScript engine issue. This creates an implementation flaw that may become a fragile point during any future changes

OP-01-018 Suggested improvement in RSA signature verification *(Low)*

When verifying RSA signatures the library tries to decode EMSA-PKCS1-v1_5 padding, extract its hash value and aims at comparing that with a calculated hash next.

```
//crypto/signature.js
verify: function(algo, hash_algo, msg_MPIs, publickey_MPIs, data) {
    var calc_hash = hashModule.digest(hash_algo, data);
    var dopublic;

    switch (algo) {
      case 1:
        // RSA (Encrypt or Sign) [HAC]
      case 2:
        // RSA Encrypt-Only [HAC]
      case 3:
        // RSA Sign-Only [HAC]
        var rsa = new publicKey.rsa();
        var n = publickey_MPIs[0].toBigInteger();
        var e = publickey_MPIs[1].toBigInteger();
        var x = msg_MPIs[0].toBigInteger();
        dopublic = rsa.verify(x, e, n);
        var hash = pkcs1.emsa.decode(hash_algo, dopublic.toMPI().substring(2));
        if (hash == -1) {
          throw new Error('PKCS1 padding in me [...] incorrect. Aborting...');
        }
        return hash == calc_hash;
```

While this approach is valid and allowed by RFC 3447[31], it unnecessarily introduces complexity to the codebase (decoding EMSA-PKCS-v1_5 requires e.g. parsing ASN.1). We recommend an alternative approach, as described in RFC 3447 Section 8.2.2, which is to generate the padding message from the calculated hash and compare the padded values:

[31] RFC 3447 http://www.ietf.org/rfc/rfc3447.txt

3. EMSA-PKCS1-v1_5 encoding: Apply the EMSA-PKCS1-v1_5 encoding operation (Section 9.2) to the message M to **produce a second encoded message EM'** of length k octets:

   EM' = EMSA-PKCS1-V1_5-ENCODE (M, k).

   If the encoding operation outputs "message too long," output "message too long" and stop. If the encoding operation outputs "intended encoded message length too short," output "RSA modulus too short" and stop.

4. **Compare the encoded message EM and the second encoded message EM'**. If they are the same, output "valid signature"; otherwise, output "invalid signature."

Under those conditions, the *pkcs1.emsa.decode()* function can be removed from the codebase. See also OP-01-026 for vulnerabilities spotted in this function.

OP-01-021 Silent error handling in various places *(Low)*

In a few places, the library functions validate input parameters but lacks proper reaction to a fault. In case those parameters do not pass the functions' validation, they are returned early with a bogus value instead of properly and promptly throwing an error.

While we could not find a place where this results in a vulnerability, it is advised to throw errors early and notify the caller explicitly, also for debugging reasons. The places in source code where we observed this questionable pattern are listed below:

```
//src/crypto/random.js
  getRandomBigIntegerInRange: function(min, max) {
    if (max.compareTo(min) <= 0) {
      return; // <--  result not type BigInt - will throw errors elsewhere
    }

    var range = max.subtract(min);
    var r = this.getRandomBigInteger(range.bitLength());
    while (r > range) {
      r = this.getRandomBigInteger(range.bitLength());
    }
    return min.add(r);
  },

  getRandomBigInteger: function(bits) {
    if (bits < 0) {
      return null;
    }

//crypto/public_key/dsa.js
  function verify(hashalgo, s1, s2, m, p, q, g, y) {
    var hashed_data = util.getLeftNBits(hashModule.digest(hashalgo, m),
q.bitLength());
    var hash = new BigInteger(util.hexstrdump(hashed_data), 16);
    if (BigInteger.ZERO.compareTo(s1) > 0 ||
```

```
      s1.compareTo(q) > 0 ||
      BigInteger.ZERO.compareTo(s2) > 0 ||
      s2.compareTo(q) > 0) {
      util.print_debug("invalid DSA Signature");
      return null; // <- dsa.verify normally returns BigInteger.
    }
    // ...
  }

//crypto/cipher/aes.js
function packBytes(octets) {
  var i, j;
  var len = octets.length;
  var b = new Array(len / 4);

  if (!octets || len % 4) return;

  for (i = 0, j = 0; j < len; j += 4)
    b[i++] = octets[j] | (octets[j + 1] << 8) | (octets[j + 2] << 16) |
(octets[j + 3] << 24);

  return b;
}
```

OP-01-022 Possible Optimization in RSA Supplemental Parameters *(Low)*

When generating RSA supplemental p and q parameters, which are later used for speeding up decryption, the following code can be seen:

```
// crypto/public_key/rsa.js
      if (key.p.compareTo(key.q) <= 0) { // <- asserts that p >= q
        var t = key.p;
        key.p = key.q;
        key.q = t;
      }
//…
        key.u = key.p.modInverse(key.q); // < u- = p^-1 mod q
```

This code is assuring that *p* >= *q.* In fact, if $u = p^{-1} mod\ q$, *p* should be **less** than *q* - as can be demonstrated e.g. in GnuPG documentation[32]. This is an optimization issue that does not affect security. We recommend to generate p that is lesser than q.

OP-01-023 Recommendation to avoid logging of Private Keys *(Low)*

There are some (deactivated) log message calls present in the library code. It is recommended to remove them; a developer could set “debug” to “true” by accident, causing a reveal of private keys and plaintext messages to the logging facility.

[32] http://www.gnupg.org/documentation/manuals/gcrypt/RSA-key-parameters.html

Spotted examples for log message calls can be seen below:

```
elgamal.js:46
util.print_debug("Elgamal Decrypt:\nc1:" + util.hexstrdump(c1.toMPI()) + "\n" +

rsa.js:58
util.print_debug("rsa.js decrypt\nxpn:" + util.hexstrdump(xp.toMPI()) + "\nxqn:"
+ util.hexstrdump(xq.toMPI()));

cfb.js:52
util.print_debug("prefixrandom:" + util.hexstrdump(prefixrandom));
```

OP-01-027 No check of armor type when de-armoring signed messages *(Low)*

The armor type heading describes what kind of information is contained inside and is to be used to delimit armored data. In some instances the type of armor is not checked, making malformed messages appear valid.

An example of a malformed but valid message:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

signed?
-----MALFORMED-----
Version: GnuPG v1.4.14 (GNU/Linux)

iJwEAQECAAYFAlLzAFYACgkQ4IT3RGwgLJdUpgQAiGXVoN7UfGOmyL+SbnlhzwcI
ZJfprjF9ej3JkI1zFPucfcYJ0Y7dVokm0PY5UeIkEogyHfejC2vR8QVEZzM3Gc6T
LEo5p3K3jgCJtK36tdPeUk8k2l5Rs8q3Bnp2PNtb3NMIF4FwLg7RNUq2T8vI1j59
jl2Is6uC0OhMxSvjyEs=
=Zgx8
```

In other implementations, such as gnupg, malformed messages as the one above produce a warning.

OP-01-028 Inconsistent documentation of PublicKey.read *(Low)*

PublicKey.read() is documented to return the key object but instead returns the length of the input +-1. The returned length is a miscalculation dependent on the version of the key (v.3 keys returns length -1, v.4 keys return length +1).

OP-01-029 Insufficient input validation in parsing packets *(Low)*

In parsing of packages, the parser, *packet.read()*, assumes that the input is well-formed and will accept malformed packets as valid.

In the following example, the input is read as a valid packet of type public key, which is then passed to the public key parser, *public_key.read()*, even though no further packet data exist. When the public key parser attempts to read from the empty data an exception is thrown (the ReferenceError thrown in the example is the result of a typo

when throwing the actual error). The packet-length header is only effective if the value is less than the remaining bytes of input, otherwise it is implicitly ignored.

```
> packetlist.read('\xc6'+ // Packet header, new format with type 6 (public key)
'\x00' // packet length header, 0 length
,0)
ReferenceError: version is not defined
       at PublicKey.read
(/home/jonas/Projects/Openpgp.js/openpgpjs/src/packet/public_key.js:105:34)
       at Packetlist.read
(/home/jonas/Projects/Openpgp.js/openpgpjs/src/packet/packetlist.js:42:12)
```

OP-01-030 Insufficient validation in parsing public keys *(Low)*

When parsing public key packages, the parser assumes that the input is well formed and will parse malformed data as valid. Input that has valid headers but is missing the mandatory multi-precision integers (MPIs) containing the key parameters is accepted since there is no check that the correct number of MPIs were parsed.

```
> (k = new pgp.packet.PublicKey()).read('\x04'+// version 4
'\x00\x00\x00\x00'+ // date
'\x01' // algorithm
),k
{ tag: 6,
  version: 4,
  created: Thu Jan 01 1970 01:00:00 GMT+0100 (CET),
  mpi: [],
  algorithm: 'rsa_encrypt_sign',
  expirationTimeV3: 0 }
```

OP-01-031 Insufficient Validation in parsing PK-encrypted Session Keys *(Low)*

When parsing public key-encrypted session key packages, the parser assumes that the input is-well formed and will parse malformed data as valid. No input validation is performed apart from the 10th byte describing the algorithm used. A packet that is missing the mandatory multi-precision integers (MPIs) containing the key parameters is accepted since there is no check against the correct number of MPIs being parsed.

```
> (k = new pgp.packet.PublicKeyEncryptedSessionKey()).read('BOGUSAAAA\x01'),k
{ tag: 1,
  version: 66,
  publicKeyId: { bytes: 'OGUSAAAA' },
  publicKeyAlgorithm: 'rsa_encrypt_sign',
  sessionKey: null,
  sessionKeyAlgorithm: 'aes256',
  encrypted: [ { data: [Object] } ] }
```

OP-01-032 Inconsistent documentation of SymEncryptedSessionKey.read *(Low)*

The documentation of *SymEncryptedSessionKey.read()* states that it takes three parameters but the implementation relies on only one parameter.

OP-01-033 Insufficient Validation for symmetrically Encrypted Session Keys *(Low)*

When parsing symmetrically encrypted session key packages, the parser assumes that the input is well-formed and will parse malformed data as valid. No input validation is performed apart from the 4th byte describing the hash algorithm used.

```
> (k = new pgp.packet.SymEncryptedSessionKey()).read('\x00\x00\x00\x01'),k
{ tag: 3,
  sessionKeyEncryptionAlgorithm: null,
  sessionKeyAlgorithm: 'plaintext',
  encrypted: null,
  s2k:
   { algorithm: 'md5',
       type: 'simple',
       c: 96,
       salt: '49Ô.JQM\u001f' },
  version: 0 }
```

OP-01-034 Insufficient Validation in Parsing One Pass Signatures *(Low)*

When parsing one pass signature packages, the parser assumes that the input is well-formed and will parse malformed data as valid. No input validation is performed apart from the 2nd, 3rd and 4th bytes respectively describing the signature type, hash algorithm, and public key algorithm used. A packet that is missing a signing key-id is accepted since there is no check of whether sufficiently many bytes of input are supplied.

```
> (k = new pgp.packet.OnePassSignature()).read('\x00\x00\x01\x01'),k
{ tag: 4,
  version: 0,
  type: 'binary',
  hashAlgorithm: 'md5',
  publicKeyAlgorithm: 'rsa_encrypt_sign',
  signingKeyId: { bytes: '' },
  flags: NaN }
```

OP-01-035 Insufficient Input Validation in Parsing Literals *(Low)*

When parsing literal packages, the parser assumes that the input is well-formed and will parse malformed data as valid. No input validation is performed apart from the 1st byte describing the format of the data. A packet that is missing a filename length, a filename, and associated data is accepted since there is no check that sufficiently many bytes of input are supplied.

```
> (k = new pgp.packet.Literal()).read('t'),k
{ tag: 11,
  format: 'text',
  data: 't',
  date: Thu Jan 01 1970 01:01:56 GMT+0100 (CET),
  filename: '' }
```

As a consequence, when the parser attempts to read the non-existent length of the filename, a NaN value will be returned. Due to arithmetics with NaN (X + NaN == 0), this has the effect on the 1st byte, which will now be read over and over again during further parsing of the input.

Affected code:

```
literal.js
Literal.prototype.read = function (bytes) {
  // ...
  var filename_len = bytes.charCodeAt(1);
  this.filename = util.decode_utf8(bytes.substr(2, filename_len));
  this.date = util.readDate(bytes.substr(2 + filename_len, 4)); // 2 + NaN == 0
  var data = bytes.substring(6 + filename_len); // 6 + NaN == 0
```

OP-01-036 Special “for your eyes only” Directive Ignored *(Low)*

According to RFC4880, literals containing the special filename “_CONSOLE” should be treated as particularly sensitive and measures should be taken to prevent disclosure of such data. RFC4880 5.9 reads as such:

> *“If the special name "_CONSOLE" is used, the message is considered to be "for your eyes only". This advises that the message data is unusually sensitive, and the receiving program should process it more carefully, perhaps avoiding storing the received data to disk, for example.”*

Conclusion

This penetration test took an entirety of 15 days and was carried out and coordinated by four testers of Cure53. Part of the test was to check the code quality and seek for common JavaScript implementation pitfalls. Other components included close study of documents, such as RFC 4880 for OpenPGP, RFC 3447 for RSA, or FIPS 186-4, all guided by the efforts to validating the implementation. We also compared code with other implementations of OpenPGP (e.g. GnuPG / libgcrypt, Bouncy Castle), as well as popular cryptographic libraries (openssl, polarssl). While interoperability issues and implementation completeness have been tested, focus has been put on possible vulnerabilities.

We identified several vulnerabilities in various layers of the library, both those concerning mathematical cryptographic weaknesses (including spotting misuse of the infamous *Math.random()* method), and those vulnerabilities resulting from lax decoding, Unicode issues and poor handling of user preferences stored with generated OpenPGP keys. Two out of twelve vulnerabilities were classified as critical, given their exploitability and possible impact. During the test-phase, timing leaks were successfully identified and demand to be underlined, as they are rarely thought of when JS crypto routines are being implemented but in fact pose a real life threat. We believe protecting libraries written in JS from such leaks is an important step for improving “JS crypto”. As such, we often opted to err on the safe side, reporting even issues that pose security threats only under certain very specific and often unlikely scenarios.

Several issues have been spotted that may be interpreted as general weaknesses and should be approached as solid code improvement suggestions. The library code quality differs, which is often the case when a mixed pool of authors has dealt with it in the past. It is visible that a more streamlined maintenance process has taken over recently, as newer parts of the code have been making a much better impression to the test team, particularly when it came to some of the underlying dependency libraries such as the SHA implementation or even JSBN.

We only checked the code “as is”, meaning that no analysis as to what tools like “browserify” or “uglify.js” had to do with the code during the build process. Specifically, we did not analyze the minified version of openpgp.js and the used shims. Furthermore, it was decided to wait with the test of some of the available features until other, more pressing issues with the library have been addressed. These mainly encompass compression, key revocation logic and string-to-key specifiers.

Cure53 would like to thank Dan Meredith and the RFA/OTF Team as well as the OpenPGP.js Team for their support and assistance during this assignment. We hope this report contributes substantial materials and plays a prominent role in enhancing and hardening OpenPGP.js, which is believed by many to be one of the foremostly and critically important JavaScript libraries operating on the front-lines of the future web.