Network Working Group
Internet-Draft
Intended status: Experimental
Expires: August 21, 2012

S. Farrell
Trinity College Dublin

February 18, 2012

Public Key Checking Protocol
draft-farrell-kc-01

Abstract

Some asymmetric key generation implementations do not use sufficient randomness giving rise to a number of bad public keys, for example with known factors, being used on the Internet. This memo specifies [[for now: just outlines]] an experimental protocol that could be used by a private key holder to talk to a responder that knows the values of (some of) those bad keys that have been seen in the wild. The protocol only allows a holder of the relevant private key to request information, as doing otherwise could weaken the overall security of the Internet and also considers confidentiality and privacy as important requirements, as information that a given bad public key is associated with a particular identifier could also weaken the security of the Internet.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 21, 2012.

Table of Contents

1. Introduction 3
2. Protocol Overview 5
3. Message Formats 6
3.1. ChalReq Message 6
3.2. ChalResp Message Format 6
3.3. CheckReq Message Format 7
3.4. CheckResp Message Format 7
3.5. Error Message Format 8
4. Cryptographic Operations 8
4.1. Signature Operation 8
4.2. Basic Proof Types 9
5. Sample Challenge Method 10
6. Responder Actions 10
7. Requestor Actions 11
8. Mandatory-to-Implement Things 11
9. Transport Considerations 11
10. Security Considerations 11
11. IANA Considerations 11
12. Acknowledgements 11
13. Changes 12
14. References 12
14.1. Normative References 12
14.2. Informative References 12
Author's Address 13

1. Introduction

[[Text in double square brackets (like this) is commentary. So far this is just an outline. I'll do more if there's interest. I'm also happy to get some help if someone wants to.]]

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Recent publications [blog][lens] have found yet again that some asymmetric key generation implementations do not use sufficient randomness giving rise to a number of bad public keys, estimated to be of the order of 0.2% of tested keys, being used on the Internet. Nonetheless, this small percentage maps to some tens of thousands of bad keys. And the distribution of bad keys is likely to be concentrated on specific devices or devices used in specific ways, so that their Pesudo Random Number Generators (PRNG) for one reason or another have not produced sufficient randomness at key generation time.

The publications referred to above involved acquiring large (in millions) sets of keys and then analysing those for example looking for common factors. While that is a computationally expensive process, once done, it should be much quicker to incrementally check if for example a single new RSA public key has one of the already known common factors or if any public key is an exact match for a known-bad key. Thus if a responder were to store and analyse many public keys it could assist key generators in knowing if they have inadvertently produced a bad key. Note that such a responder cannot, (especially in real-time), determine that a public key is good, but only whether the public key is known to be bad.

The entire set of known-bad keys cannot however be made available to all, as some of those keys are in real use and simply publishing their values could put Internet users at risk. However, if we have a responder with the bad keys and a protocol that only allows the relevant private key holder to make requests then we may be able to provide a useful service.

In addition to requiring that only private key holders can query the responder, we must also ensure that eavesdroppers cannot tell whether the answer to the query is that the key is known to be bad or not known to be bad. For example, response packet sizes could expose this information.

Servers implementing this protocol are REQUIRED to store the public keys presented to them for offline analysis. (Though they may also

acquire public keys for analysis in many other ways.) Thus, the answer that a requestor receives might change from not-known-bad to known-bad in a matter of minutes or hours. Some requestors could take advantage of this and not actually use a key until they have gotten not-known-bad answers for a configured period.

Note also that my public key may be good now but might become known to be bad after someone else has posted e.g. a public key with a common factor. In other words, every private key holder could benefit from periodically checking with a responder for this protocol.

While a responder may take hours to find new bad keys, once a responder has a set of e.g. factors of RSA moduli, then it can easily check if a supplied public key has one of those as a factor, and this is one of the bad-key patterns seen in the wild. This will not detect all bad keys however, a process that does reqiure more computation. Similarly, if there are blacklists of bad keys (e.g. as happened in the Debian case [deb]) then those can be spotted immediately. So the responder can in such cases give quick and accurate answers. Ultimately, the responder can do anything it wants for any algorithm - the specific checks are not a part of this protocol.

While a responder here could lie and say that a key is not-known-bad even if it is in fact known-bad, using more than one responder could mitigate that and reduce the level of trust required in the responder's honesty. Clients can also test any responder for this kind of dishonesty by occasionally generating and sending bad keys to check if the responder is honest. This is why we REQUIRE the responders to store and analyse the keys presented to it. [[Could be interesting games to play here.]]

If a responder considers a public key to be known bad, then the responder might know the corresponding private key in which case it can produce a proof that it know this, e.g by signing a hash of the request message or otherwise. Note that responders MUST ensure that the inclusion or omission of such proofs is done so as not to expose the responder's opinion of the status of the public key - in otherwords, keep all response messages the same length for a given public key length.

Note also that this protocol does not tell the requestor what to do with a known-bad public key. Presumably they'd stop using it but the best action to take will depend on the application using the public key so is out of scope here.

This is an experimental specification for at least two good reasons.

Firstly, it is not yet clear that it would be broadly adopted by private key holders. Secondly, it is not clear that responders with access to the data about known-bad keys will make that information available via this protocol, or at all. If responders for this protocol with significant data sets appear on the Internet and private key holders adopt this protocol then the experiment will have been successful and a future version of this could be considered for the IETF standards track.

2. Protocol Overview

[[The protocol below requires the private key be usable for signing. We could extend it to e.g. D-H public values if we put more structure into the ChalResp. Not sure if that's worthwhile for now since the overwhelming majority of keys will be ok for signing.]]

The abstract protocol is simple:

- ChalReq: the requestor sends a message asking for a challenge
- ChalResp: the responder replies with a challenge
- CheckReq: the requestor sends a signed query containing the public key and challenge
- CheckResp: the responder replies saying the public key is known to be bad, or not known to be bad

In addition there is an ErrorMsg defined.

This protocol MUST be run over a responder-authenticated TLS [RFC5246] session using a TLS ciphersuite that provides strong confidentiality.

In order to ensure confidentiality even in the face of traffic analysis, we ensure that all messages containing the responder's result are the same size (for a given public key algorithm and size). This can involve the responder sending random bits to the requestor, and those MUST be of sufficient quality to be useful as input to key generation.

For additional privacy, a requestor might choose to run this protocol over some onion routing network such as Tor. [tor] The protocol is designed to allow for such use-cases. [[not sure yet how to do that though, help appreciated]]

Note that the challenge has no strucure from the requestor

perspective but might have for the responder. For example, a responder could include encrypted values in order to ensure that the challenge is valid and or fresh. [[we might want to make that a MUST but its probably only useful if done so I could test it from outside.]]

3. Message Formats

In this section we describe the messages used in this protocol. [[The concrete encoding is TBD. Maybe JSON or just binary, dunno.]]

[[Would it be worthwhile REQUIRING that all messages be randomly padded out to some particular length that's longer than all real messages for the key length in question? Not sure.]]

All multi-octet values MUST be sent in network byte order.

3.1. ChalReq Message

This message is sent from a requestor to a responder asking for a fresh challenge.

Figure 1: ChalReq Message Format

This message has the following fields:

- Type: 0x01, 1-octet, meaning that this is a ChalReq
- Flags: TBD, 4-octets

3.2. ChalResp Message Format

This message is sent from a responder and contains a fresh challenge.

- Type: 0x02, 1-octet, meaning that this is a ChalResp
- Flags: TBD, 4-octets
- ChalId: 4-octets, a value chosen by the responder to index the challenge

- ChalLen: 4-octets, the length of the challenge in octets
- Chal: NN octets, the octets of the challenge

[[Maybe ChalId should be a transaction ID? figure out later. With the ChalId you could do all this over DTLS/UDP maybe. Probably not worth it though.]]

3.3. CheckReq Message Format

This message is sent from a requestor to a responder and contains a public key, challenge and signature over those.

- Type: 0x03, 1-octet, meaning that this is a CheckReq
- Flags: TBD, 4-octets
- ChalId: 4-octets, a value chosen by the responder to index the challenge
- ChalLen: 4-octets, the length of the challenge in octets
- Chal: NN octets, the octets of the challenge
- PKAlg: public key algorithm identifier and format (details TBD)
- PKLen: 4-octets, the length of the public key in octets
- PK: NN octets, the octets of the public key
- Sigalg: signature algorithm identifier and format (details TBD)
- Siglen: 4-octets, the length of the signature in octets
- Sig: NN octets, the octets of the signature

3.4. CheckResp Message Format

This message is sent from a responder to a requestor and contains the status of the public key according to the responder. .

- Type: 0x04, 1-octet, meaning that this is a CheckReq
- Flags: TBD, 4-octets
- ChalId: 4-octets, a value chosen by the responder to index the challenge

- Status: 1-octet, an even numbered value means the key is not known to be bad; an odd numbered value means the key is known to be bad (says the responder!)

- ProofType: 2-octets, proof algorithm identifier and format (details TBD)

- Prooflen: 4-octets, the length of the proof in octets

- Proof: NN octets, the octets of the proof

In order to keep response to the same length, a ProofType value of zero (0) means that the Proof field contains the relevant number of octets of random values.

3.5. Error Message Format

Error messages all have the following format.

- Type: 0x00, 1-octet, meaning that this is an ErrorMsg

- Flags: TBD, 4-octets

- ChalId: 4-octets, a value chosen by the responder to index the challenge or zero if no relevant ChalId is known

- ErrType: 2-octets, the specific error (values TBD)

- Errlen: 4-octets, the length of the error string in octets

- ErrorString: NN octets, the octets of the error string

[[There will be i18n silliness needed here, maybe.]]

4. Cryptographic Operations

In this section we define the signature operation and define proof types.

4.1. Signature Operation

The requestor has to sign some data for the CheckReq. The input to the signature is the following values, concatenated.

- A fixed string "CHECKING A PUBLIC KEY"

- Type: 0x03, 1-octet, meaning that this is a CheckReq
- Flags: TBD, 4-octets
- ChalId: 4-octets, a value chosen by the responder to index the challenge
- ChalLen: 4-octets, the length of the challenge in octets
- Chal: NN octets, the octets of the challenge
- PKAlg: public key algorithm identifier and format (details TBD)
- PKLen: 4-octets, the length of the public key in octets
- PK: NN octets, the octets of the public key
- Sigalg: signature algorithm identifier and format (details TBD)
- Siglen: 4-octets, the length of the signature in octets

This is essentially the fixed string (to prevent cross-protocol attacks) followed by the CheckReq message minus the Sig field but including the length.

4.2. Basic Proof Types

We define two types of proof here, a signature scheme (ProofType 1) and a random scheme (ProofType 0) to be used when the public key is not known to be bad.

For ProofType 0, the responder just includes the same number of random octets as it would have used for ProofType 1 had the status been known-bad. As with the challenge value those random octets MUST be good enough to use as a PRNG seed.

Note that a responder can use ProofType 0 even if it says that the public key is known-bad. Not all kinds of badness result in the responder being able to demonstrate that it knows the private key.

ProofType 1 involves the responder signing a sha-256 hash of the CheckReq message with the private key corresponding to the public key submitted in the CheckReq. This signature demonstrates to the private key holder that their key is toast.

[[At some stage think about this some more to see what's best to use as proof.]]

5. Sample Challenge Method

Regardless of the scheme used to generate the challenge, a responder's challenge MUST be a good random value, suitable for a requestor to use as an additional seed for a PRNG and the challenge MUST be at least 256 bits long.

A fresh challenge value MUST be used for all transactions. Attempts to re-use a challenge MUST result in an error (BadChal). [[Not sure this is needed but we should say something and this is the easiest to say, if not to implement;-)]]

If the responder wishes to remain stateless then it can emit challenge values that are a symmetically encrypted form (with integrity!) of the current responder time and/or a sequence number of some sort. This would allow the responder to detect attempts to use stale challenges.

6. Responder Actions

Responders need to ensure that simple timing attacks are not possible. The processing of CheckReq messages MUST take the same amount of time regardless of errors, known-bad status and of the ProofType used in the corresponding CheckResp.

For clarity: the time between receipt of a CheckReq and emission of a CheckResp or ErrroMsg corresponding to that CheckReq MUST be a constant for any given public key algorithm and key size.

When a responder recevives a message that does not decode properly it SHOULD return an ErrorMsg with an ErrType value of BadMessage (1). An example where an ErrorMsg might not be returned would be if the responder considered itself as being under a Denial-of-Service (DoS) attack.

When a responder receives a ChalReq message, it MUST generate a fresh ChalRep message.

When a responder receives a CheckReq message it MUST do all of the following:

1. Verify that the ChecReq challenge value meets whatever are the server's criteria. If it does not then the responder MUST return an ErrorMsg with ErrType BadChal (2).

2. Verify that the public key from the message verifies the signature from the message. If the signture check fails then the

responder SHOULD an ErrorMsg with ErrType BadSig (3). The responder might not send an ErrorMsg if it considered that some attack was under way, e.g. if many bad signatures were received for the same public key.

3. Store the public key for later analysis. The responder MUST NOT store any other information about the requestor, for example, its IP address.

4. Determine whether to answer that the key is known-bad or not-known-bad based on whatever local criteria are used.

5. Construct a corresponding CheckResp message and return that to the requestor.

7. Requestor Actions

[[Still TBD, but fairly obvious]]

8. Mandatory-to-Implement Things

[[You MUST be able to do RSA, prooftypes 0 (random bits) and 1 (signature), signatures MUST use rsa-sha256 with OAEP or pkcs1v1.5 maybe. What else?]]

9. Transport Considerations

[[Maybe just right over TLS over TCP, maybe via HTTP, with a .well-known URL, dunno.]]

10. Security Considerations

[[You'd have to imagine there are:-)]]

11. IANA Considerations

[[None yet, there will be a bunch with registries for many of the fields in the messages defined above.]]

12. Acknowledgements

Steve Bellovin proposed the main idea here independently [bell], I

only saw that after the -00 was out.

Thanks for Paul Hoffman for some off-list discussions that didn't quite convince him this is worth some effort;-)

13. Changes

This section describes the various versions of this draft and is to be removed later when/if this becomes an RFC.

-00: just a sketch of the protocol

-01: sketch -> fairly detailed outline

14. References

14.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

14.2. Informative References

[bell] Bellovin, S., "Duplicate primes in lots of RSA moduli", February 2012, <http://lists.randombit.net/pipermail/cryptography/2012-February/002310.html>.

[blog] Heniger, N., "New research: There's no need to panic over factorable keys--just mind your Ps and Qs", February 2012, <https://freedom-to-tinker.com/blog/nadiah/new-research-theres-no-need-panic-over-factorable-keys-just-mind-your-ps-and-qs>.

[deb] "Debian Security Advisory, DSA-1571-1: openssl -- predictable random number generator", May 2008, <http://www.debian.org/security/2008/dsa-1571>.

[lens] Lenstra, A., Hughes, J., Augier, M., Bos, J., Kleinjung, T., and C. Wachter, "Ron was wrong, Whit is right", Cryptology ePrint Archive Report 2012/064, February 2012, <http://eprint.iacr.org/2012/064>.

[tor] "The Tor Project", <http://www.torproject.org/>.

Author's Address

Stephen Farrell
Trinity College Dublin
Dublin, 2
Ireland

Phone: +353-1-896-2354
Email: stephen.farrell@cs.tcd.ie