Workgroup: WG Working Group
Internet-Draft:
draft-dew-cfrg-signature-key-blinding-00
Published: 7 March 2022
Intended Status: Informational
Expires: 8 September 2022
Authors: F. Denis
Fastly Inc.
E. Eaton
University of Waterloo
C. A. Wood
Cloudflare, Inc.

Key Blinding for Signature Schemes

Abstract

This document describes extensions to existing signature schemes for key blinding. This functionality guarantees that a blinded public key and all signatures produced using the blinded key pair are unlinkable to the unblinded key pair. Moreover, signatures produced using blinded key pairs are indistinguishable from signatures produced using unblinded key pairs.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://chris-wood.github.io/draft-dew-cfrg-signature-key-blinding/draft-dew-cfrg-signature-key-blinding.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-dew-cfrg-signature-key-blinding/.

Discussion of this document takes place on the CFRG Working Group mailing list (mailto:cfrg@irtf.org), which is archived at https://mailarchive.ietf.org/arch/browse/cfrg/.

Source for this draft and an issue tracker can be found at https://github.com/chris-wood/draft-dew-cfrg-signature-key-blinding.

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 https://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 8 September 2022.

Table of Contents

1. Introduction

EdDSA [EDDSA] is a type of Schnorr signature algorithm based on Edwards curves. The specification [RFC8032] describes several

variants of EdDSA with parameter sets for the edwards25519 and edwards448 curves as described in [RFC7748]. According to the specification, private keys are randomly generated seeds, which are then used to derive scalar elements and their corresponding public group element for signing and verifying messages, respectively.

Given an EdDSA private and public key pair (sk, pk), any message signed by sk is linkable to pk. One simply checks whether the message signature is valid under pk. In some settings, in is useful to produce signatures with a given key pair (sk, pk) such that the resulting signature is not linkable to pk without knowledge of a particular witness r. That is, given pk corresponding to sk, witness r, and a message signature, one can determine if the signature was indeed produced using sk. In effect, the witness "blinds" the key pair associated with a message signature.

This functionality is also possible with other signature schemes, including [ECDSA] and some post-quantum signature schemes [ESS21].

This document describes a modification to the EdDSA key generation and signing procedures in [RFC8032] to support this blinding operation, referred to as key blinding. It also specifies an extension to [ECDSA] that enables the same functionality.

1.1. DISCLAIMER

This document is a work in progress and is still undergoing security analysis. As such, it **MUST NOT** be used for real world applications. See Section 7 for additional information.

2. Conventions and Definitions

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The following terms are used throughout this document to describe the blinding modification.

*G: The standard base point.

*sk: A signature scheme private key. For EdDSA, this is a a randomly generated private seed of length 32 bytes or 57 bytes according to [RFC8032], Section 5.1.5 or [RFC8032], Section 5.2.5, respectively. For [ECDSA], sk is a random scalar in the prime-order elliptic curve group.

*pk(sk): The public key corresponding to the private key sk.

- concat(x0, ..., xN): Concatenation of byte strings. concat(0x01, 0x0203, 0x040506) = 0x010203040506.

- ScalarMult(pk, k): Multiply the public key pk by scalar k, producing a new public key as a result.

- ModInverse(x, L): Compute the multiplicative inverse of x modulo L.

In pseudocode descriptions below, integer multiplication of two scalar values is denoted by the * operator. For example, the product of two scalars x and y is denoted as x * y.

3. Key Blinding

At a high level, a signature scheme with key blinding allows signers to blind their signing key such that any signature produced under the blinded signing key is unlinkable from the unblinded signing key. Similar to the signing key, the blind is also a private key that remains secret. For example, the blind is a 32-byte or 57-byte random seed for Ed25519 or Ed448 variants, respectively, whereas the blind for ECDSA over P-256 is a random scalar in the P-256 group.

Key blinding introduces three new functionalities for the signature scheme:

- BlindPublicKey(pkS, skB): Blind the public key pkS using the private key skB.

- UnblindPublicKey(pkM, skB): Unblind the public key pkM using the private key skB.

- BlindKeySign(skS, skB, msg): Sign a message msg using the private key skS with the private blind skB.

Correctness requires the following equivalence to hold:

```
UnblindPublicKey(BlindPublicKey(pkS, skB), skB) = pkS
```

Security requires that signatures produced using BlindKeySign are unlinkable from signatures produced using the standard signature generation function with the same private key.

4. Ed25519ph, Ed25519ctx, and Ed25519

This section describes implementations of BlindPublicKey, UnblindPublicKey, and BlindKeySign as modifications of routines in [RFC8032], Section 5.1.

4.1. BlindPublicKey and UnblindPublicKey

BlindPublicKey transforms a private blind skB into a scalar for the edwards25519 group and then multiplies the target key by this scalar. UnblindPublicKey performs essentially the same steps except that it multiplies the target public key by the multiplicative inverse of the scalar, where the inverse is computed using the order of the group L, described in [RFC8032], Section 5.1.

More specifically, BlindPublicKey(pk, skB) works as follows.

1. Hash the 32-byte private key skB using SHA-512, storing the digest in a 64-octet large buffer, denoted h. Only the lower 32 bytes are used for generating the public key.

2. Interpret the buffer as a little-endian integer, forming a secret scalar s. Note that this explicitly skips the buffer pruning step in [RFC8032], Section 5.1. Perform a scalar multiplication ScalarMult(pk, s), and output the encoding of the resulting point as the public key.

UnblindPublicKey(pkM, skB) works as follows.

1. Compute the secret scalar s from skB as in BlindPublicKey.

2. Compute the sInv = ModInverse(s, L), where L is as defined in [RFC8032], Section 5.1.

3. Perform a scalar multiplication ScalarMult(pk, sInv), and output the encoding of the resulting point as the public key.

4.2. BlindKeySign

BlindKeySign transforms a private key skB into a scalar for the edwards25519 group and a message prefix to blind both the signing scalar and the prefix of the message used in the signature generation routine.

More specifically, BlindKeySign(skS, skB, msg) works as follows:

1. Hash the private key skS, 32 octets, using SHA-512. Let h denote the resulting digest. Construct the secret scalar s1 from the first half of the digest, and the corresponding public key A1, as described in [RFC8032], Section 5.1.5. Let prefix1 denote the second half of the hash digest, h[32],...,h[63].

2. Hash the 32-byte private key skB using SHA-512, storing the digest in a 64-octet large buffer, denoted b. Interpret the lower 32 bytes buffer as a little-endian integer, forming a

secret scalar s2. Let prefix2 denote the second half of the hash digest, b[32],...,b[63].

3. Compute the signing scalar s = s1 * s2 (mod L) and the signing public key A = ScalarMult(G, s).

4. Compute the signing prefix as concat(prefix1, prefix2).

5. Run the rest of the Sign procedure in [RFC8032], Section 5.1.6 from step (2) onwards using the modified scalar s, public key A, and string prefix.

5. Ed448ph and Ed448

This section describes implementations of BlindPublicKey, UnblindPublicKey, and BlindKeySign as modifications of routines in [RFC8032], Section 5.2.

5.1. BlindPublicKey and UnblindPublicKey

BlindPublicKey and UnblindPublicKey for Ed448ph and Ed448 are implemented just as these routines are for Ed25519ph, Ed25519ctx, and Ed25519, except that SHAKE256 is used instead of SHA-512 for hashing the secret blind to a 114-byte buffer and the order of the edwards448 group L is as defined in [RFC8032], Section 5.2.1.

5.2. BlindKeySign

BlindKeySign for Ed448ph and Ed448 is implemented just as this routine for Ed25519ph, Ed25519ctx, and Ed25519, except in how the scalars (s1, s2), public keys (A1, A2), and message strings (prefix1, prefix2) are computed. More specifically, BlindKeySign(skS, skB, msg) works as follows:

1. Hash the private key skS, 57 octets, using SHAKE256(skS, 117). Let h denote the resulting digest. Construct the secret scalar s1 from the first half of the digest, and the corresponding public key A1, as described in [RFC8032], Section 5.2.5. Let prefix1 denote the second half of the hash digest, h[57],...,h[113].

2. Perform the same routine to transform the secret blind skB into a secret scalar s2, public key A2, and prefix2.

3. Compute the signing scalar s = s1 * s2 (mod L) and the signing public key A = ScalarMult(A1, s2).

4. Compute the signing prefix as concat(prefix1, prefix2).

5. Run the rest of the Sign procedure in [RFC8032], Section 5.2.6 from step (2) onwards using the modified scalar s, public key A, and string prefix.

6. ECDSA

[[DISCLAIMNER: Multiplicative blinding for ECDSA is known to be NOT be SUF-CMA-secure in the presence of an adversary that controls the blinding value. [MSMHI15] describes this in the context of related-key attacks. This variant may likely be removed in followup versions of this document based on further analysis.]]

This section describes implementations of BlindPublicKey, UnblindPublicKey, and BlindKeySign as functions implemented on top of an existing [ECDSA] implementation. In the descriptions below, let p be the order of the corresponding elliptic curve group used for ECDSA. For example, for P-256, p = 115792089210356248762697446949407573529996955224135760342422259061068512044369.

6.1. BlindPublicKey and UnblindPublicKey

BlindPublicKey multiplies the public key pkS by an augmented private key skB yielding a new public key pkR. UnblindPublicKey inverts this process by multiplying the input public key by the multiplicative inverse of the augmented skB. Augmentation here maps the private key skB to another scalar using hash_to_field as defined in Section 5 of [H2C], with DST set to "ECDSA Key Blind", L set to the value corresponding to the target curve, e.g., 48 for P-256 and 72 for P-384, expand_message_xmd with a hash function matching that used for the corresponding digital signature algorithm, and prime modulus equal to the order p of the corresponding curve. Letting HashToScalar denote this augmentation process, BlindPublicKey and UnblindPublicKey are then implemented as follows: ~~~ BlindPublicKey(pk, skB) = ScalarMult(pk, HashToScalar(skB)) UnblindPublicKey(pk, skB) = ScalarMult(pk, ModInverse(HashToScalar(skB), p)) ~~~

6.2. BlindKeySign

BlindKeySign transforms the signing key skS by the private key skB into a new signing key, skR, and then invokes the existing ECDSA signing procedure. More specifically, skR = skS * HashToScalar(skB) (mod p).

7. Security Considerations

The signature scheme extensions in this document aim to achieve unforgeability and unlinkability. Informally, unforgeability means that one cannot produce a valid (message, signature) pair for any blinding key without access to the private signing key. Similarly,

unlinkability means that one cannot distinguish between two signatures produced from two separate key signing keys, and two signatures produced from the same signing key but with different blinds. Security analysis of the extensions in this document with respect to these two properties is currently underway.

Preliminary analysis has been done for a variant of these extensions used for identity key blinding routine used in Tor's Hidden Service feature [TORBLINDING]. For EdDSA, further analysis is needed to ensure this is compliant with the signature algorithm described in [RFC8032].

The constructions in this document assume that both the signing and blinding keys are private, and, as such, not controlled by an attacker. [MSMHI15] demonstrate that ECDSA with attacker-controlled multiplicative blinding for producing related keys can be abused to produce forgeries. In particular, if an attacker can control the private blinding key used in BlindKeySign, they can construct a forgery over a different message that validates under a different public key. Further analysis is needed to determine whether or not it is safe to keep this functionality in the specification given this problem.

8. IANA Considerations

This document has no IANA actions.

9. Test Vectors

This section contains test vectors for a subset of the signature schemes covered in this document.

9.1. Ed25519 Test Vectors

This section contains test vectors for Ed25519 as described in [RFC8032]. Each test vector lists the private key and blind seeds, denoted skS and skB and encoded as hexadecimal strings, along with their corresponding public keys pkS and pkB encoded has hexadecimal strings according to [RFC8032], Section 5.1.2. Each test vector also includes the blinded public key pkR computed from skS and skB, denoted pkR and encoded has a hexadecimal string. Finally, each vector includes the message and signature values, each encoded as hexadecimal strings.

```
// Randomly generated private key and blind seed
skS: 875532ab039b0a154161c284e19c74afa28d5bf5454e99284bbcffaa71eebf45
pkS: 3b5983605b277cd44918410eb246bb52d83adfc806ccaa91a60b5b2011bc5973
skB: c461e8595f0ac41d374f878613206704978115a226f60470ffd566e9e6ae73bf
pkB: 0de25ad2fc6c8d2fdacd2feb85d4f00cbe33a63a5b0939a608aeb5450990ccf6
pkR: e52bbb204e72a816854ac82c7e244e13a8fcc3217cfdeb90c8a5a927e741a20f
message: 68656c6c6f20776f726c64
signature: f35d2027f14250c07b3b353359362ec31e13076a547c749a981d0135fce06
7a361ad6522849e6ed9f61d93b0f76428129b9eb3f9c3cd0bfa1bc2a086a5eebd09

// Randomly generated private key seed and zero blind seed
skS: f3348942e77a83943a6330d372e7531bb52203c2163a728038388ea110d1c871
pkS: ada4f42be4b8fa93ddc7b41ca434239a940b4b18d314fe04d5be0b317a861ddf
skB: 0000000000000000000000000000000000000000000000000000000000000000
pkB: 3b6a27bcceb6a42d62a3a8d02a6f0d73653215771de243a63ac048a18b59da29
pkR: 7b8dcabbdfce4f8ad57f38f014abc4a51ac051a4b77b345da45ee2725d9327d0
message: 68656c6c6f20776f726c64
signature: b38b9d67cb4182e91a86b2eb0591e04c10471c1866202dd1b3b076fb86a61
c7c4ab5d626e5c5d547a584ca85d44839c13f6c976ece0dcba53d82601e6737a400
```

9.2. ECDSA(P-256, SHA-256) Test Vectors

This section contains test vectors for ECDSA with P-256 and SHA-256, as described in [ECDSA]. Each test vector lists the signing and blinding keys, denoted skS and skB, each serialized as a big-endian integers and encoded as hexadecimal strings. Each test vector also lists the unblinded and blinded public keys, denoted pkS and pkB and encoded as compressed elliptic curve points according to [ECDSA]. Finally, each vector lists message and signature values, where the message is encoded as a hexadecimal string, and the signature value is serialized as the concatenation of scalars (r, s) and encoded as a hexadecimal string.

```
// Randomly generated signing and blind private keys
skS: fb577883a7d5806392cc24433485716a663c3390050dd69f970340442ddfadf5f96
9f96119b9674d717231b3440ee790
pkS: 02c220ad8c512bb91ab8636c1d4ad8ad322d46786cb89c979335f871e017ced191c
67cb94a584d866e9e24cfca90dd4a45
skB: be45d7118e851486e201b720b1c101d4df4dc68a868a720397eb91428dcf2da4da4
c50f8ae6b0885af4d2e1801f9348a
pkB: 0313ae8964beca953c0db3c936a49de6c34ad198c2d5aaa7ada46b44a6742584587
1a9f87554dcbb2a75a7b7af7b324b08
pkR: 02f7741b9291001bd42f9aef9e99c010f1f69b1dfe115a95309fe81ca1f68e2ffaa
3dfc131e47752023537be2c3526d331
message: 68656c6c6f20776f726c64
signature: bd887d5e74742ce7e3ee42794b38f90afc8773bcdab84f8148f59a0b1006c
ab6bd6111052f6ddd2e3c9ed5db6e46c5e9fbb850091cb30bf70e5e11412556d7c1265f0
40ae1ff7c77a7196239058c51b311cc5a3038234c6bbba79bc9b53c148f
```

10. References

10.1. Normative References

[ECDSA] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry - The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI ANS X9.62-2005, November 2005.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/rfc/rfc8032>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

10.2. Informative References

[EDDSA] Bernstein, D., Duif, N., Lange, T., Schwabe, P., and B. Yang, "High-speed high-security signatures", Journal of Cryptographic Engineering Vol. 2, pp. 77-89, DOI 10.1007/s13389-012-0027-1, August 2012, <https://doi.org/10.1007/s13389-012-0027-1>.

[ESS21] Eaton, E., Stebila, D., and R. Stracovsky, "Post-Quantum Key-Blinding for Authentication in Anonymity Networks", 2021, <https://eprint.iacr.org/2021/963>.

[H2C] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-hash-to-curve-14, 18 February 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-14>.

[MSMHI15] Morita, H., Schuldt, J., Matsuda, T., Hanaoka, G., and T. Iwata, "On the Security of the Schnorr Signature Scheme and DSA Against Related-Key Attacks", Information Security and Cryptology - ICISC 2015 pp. 20-35, DOI

10.1007/978-3-319-30840-1_2, 2016, <https://doi.org/10.1007/978-3-319-30840-1_2>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/rfc/rfc7748>.

[TORBLINDING] Hopper, N., "Proving Security of Tor's Hidden Service Identity Blinding Protocol", 2013, <https://www-users.cse.umn.edu/~hoppernj/basic-proof.pdf>.

Acknowledgments

The authors would like to thank Dennis Jackson for helpful discussions that informed the development of this draft.

Authors' Addresses

Frank Denis
Fastly Inc.
475 Brannan St
San Francisco,
United States of America

Email: fde@00f.net

Edward Eaton
University of Waterloo
200 University Av West
Waterloo
Canada

Email: ted@eeaton.ca

Christopher A. Wood
Cloudflare, Inc.
101 Townsend St
San Francisco,
United States of America

Email: caw@heapingbits.net