Internet Research Task Force   Harkins
Internet-Draft   HP Enterprise
Intended status: Informational   November 28, 2016
Expires: June 1, 2017

PKEX
draft-harkins-pkex-02

Abstract

This memo describes a password-authenticated protocol to allow two devices to exchange "raw" (uncertified) public keys and establish trust that the keys belong to their respective identities.

Table of Contents

1. Introduction 2
1.1. Requirements Language 3
1.2. Notation 3
2. Properties 4
3. Assumptions 4
4. Cryptographic Primitives 5
5. Protocol Definition 6
5.1. Exchange Phase 6
5.2. Commit/Reveal Phase 7
6. IANA Considerations 8
7. Security Considerations 8
8. References 8
8.1. Normative References 8
8.2. Informative References 9
Appendix A. Generation of ECC Role-specific Elements 9
A.1. Role-specific Elements for NIST p256 10
A.2. Role-specific Elements for NIST p384 10
A.3. Role-specific Elements for NIST p521 11
A.4. Role-specific Elements for brainpool p256r1 13
A.5. Role-specific Elements for brainpool p384r1 13
A.6. Role-specific Elements for brainpool p512r1 14
Appendix B. Generation of FFC Role-Specific Elements 15
Author's Address 16

1. Introduction

Many authenticated key exchange protocols allow for authentication using uncertified, or "raw", public keys. Usually these specifications-- e.g. [RFC7250] for TLS and [RFC7670] for IKEv2-- assume keys are exchanged in some out-of-band mechanism.

[RFC7250] further states that "the main security challenge [to using 'raw' public keys] is how to associate the public key with a specific entity. Without a secure binding between identifier and key, the protocol will be vulnerable to man-in-the- middle attacks."

The Public Key Exchange (PKEX) is designed to fill that gap: it establishs a secure binding between exchanged public keys and identifiers, it provides proof-of-possession of the exchanged public keys to each peer, and it enables the establishment of trust in public keys that can subsequently be used to faccilitate authentication in other authentication and key exchange protocols.

1.1. Requirements Language

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].

1.2. Notation

This memo describes a cryptographic exchange using sets of elements called groups. Groups can be either traditional finite field or can be based on elliptic curves. The public keys exchanged by PKEX are elements in a group. Elements in groups are denoted in upper-case and scalar values are denoted with lower-case. The generator of the group is G.

When both the initator and responder use a similar, but unique, datum it is denoted by appending an "i" for initiator or "r" for responder, e.g. if each side needs an element C then the initiator's is Ci and the responder's is Cr.

During the exchange, one side will generate data and the other side will attempt to reconstruct it. The reconstructed data is "primed". That is, if the initiator generates C then when responder tries to reconstruct it, the responder will refer to it as C'. Data that is directly sent and received is not primed.

The following notation is used in this memo:

C = A + B
    The "group operation" on two elements, A and B, that produces a third element, C. For finite field cryptography this is the modular multiplication, for elliptic curve cryptography this is point addition.

C = A - B
    The "group operation" on element A and the inverse of element B to produce a third element, C. Inversion is defined such that the group operation on an element and its inverse results in the identity element, the value one (1) for finite field cryptography and the "point at infinity" for elliptic curve cryptography.

C = a * B
    This denotes repeated application of the group operation to B-- i.e. B + B-- (a - 1) times.

a = H(b)
    A cryptographic hash function that takes data b of indeterminate length and returns a fixed sized digest a.

a = F(B)
A mapping function that takes an element and returns a scalar. For elliptic curve cryptography, F() returns the x-coordinate of the point B. For finite field cryptography, F() is the identity function.

a = KDF-b(c, d)
A key derivation function that derives an output key a of length b from an input key c and context d.

c = a | b
Concatentation of data a with data b producing c.

{a}b
Authenticated-encryption of data a with key b.

2. Properties

Subversion of PKEX involves an adversary being able to insert its own public key into the exchange without the exchange failing, resulting in one of the parties to the exchange believing the adversary's public key actually belongs to the protocol peer.

PKEX has the following properties:

- o An adversary is unable to subvert the exchange without knowing the password.
- o An adversary is unable to discover the password through passive attack.
- o The only information exposed by an active attack is whether a single guess of the password is correct or not.
- o Proof-of-possession of the private key is provided.
- o At the end of the protocol, either trust is established in the peer's public key and the public key is bound to the peer's identity, or the exchange fails.

3. Assumptions

Due to the nature of the exchange, only DSA ([DSS]) and ECDSA ([X9.62]) keys can be exchanged with PKEX.

PKEX requires fixed elements that are unique to the particular role in the protocol, an initiator-specific element and a responder-specific element. They need not be secret. It is assumed that both

parties know the role-specific elements for the particular group in which their key pairs were derived. Techniques to generate role-specific elements, and generated elements for popular groups, are listed in Appendix A and Appendix B.

The authenticated-encryption algorithm provides deterministic "key wrapping". To achieve this the AE scheme used in PKEX is [RFC5297].

The KDF provides for the generation of a cryptographically strong secret key from an "imperfect" source of randomness. To achieve this the KDF used in PKEX is the unsalted version of [RFC5869].

The following assumptions are made on PKEX:

o Only the peers involved in the exchange know the password.

o The peers' public keys are from the same group.

o The discrete logarithms of the public role-specific elements are unknown, and determining them is computationally infeasible.

4. Cryptographic Primitives

HKDF requires an underlying hash function and AES-SIV requires a key length. To provide for consistent security the hash algorithm and key length depend on the group chosen to use with PKEX.

For ECC, the hash algorithm and key length depends on the size of the prime defining the curve, p:

o SHA-256 and 256 bits: when len(p) <= 256

o SHA-384 and 384 bits: when 256 < len(p) <= 384

o SHA-512 and 512 bits: when 384 < len(p)

For FFC, the hash algorithm depends on the prime, p, defining the finite field:

o SHA-256 and 256 bits: when len(p) <= 2048

o SHA-384 and 384 bits: when 2048 < len(p) <= 3072

o SHA-512 and 512 bits: when 3072 < len(p)

5. Protocol Definition

PKEX is a balanced PAKE. The identical version of the password is used by both parties.

PKEX consists of two phases: exchange and commit/reveal. It is described using the popular protocol participants, Alice (an initiator of PKEX), and Bob (a responder of PKEX).

We denote Alice's role-specific element a Pi and Bob's as Pr. The password is pw. For simplicity, Alice's identity is "Alice" and Bob's identity is "Bob". Alice's public key she wants to share with Bob is A and her private key is a, while Bob's public key he wants to share with Alice is B and his private key is b.

5.1. Exchange Phase

The Exchange phase is essentially the SPAKE2 key exchange. The peers derive ephemeral public keys, encrypt, and exchange them. Each party hashes a concatentation of his or her identity and the password and operates on the role-specific element to obtain a secret encrypting element. The group operation is then performed with the ephemeral key and the secret encrypting element to produce an encrypted ephmeral key.

```
      Alice:                             Bob:
      ------                             ----
  x, X = x*G                         y, Y = y*G
  Qi = H(Alice|pw)*Pi                Qr = H(Bob|pw)*Pr
  M = X + Qa
                          M ------>
                                     Qi = H(Alice|pw)*Pi
                                     X' = M - Qi
                                     N = Y + Qr
                          <------ N
  Qr = H(Bob|pw)*Pr
  Y' = N - Qr
```

Both M and N MUST be verified to be valid elements in the selected group. If either one is not valid the protocol fails.

At this point in time the peers have exchanged ephemeral elements that will be unknown except by someone with knowledge of the password. Given our assumptions that means only Alice and Bob can know the elements X and Y.

The secret encrypting elements are irretrievably deleted at this point.

5.2. Commit/Reveal Phase

In the Commit/Reveal phase the peers commit to the particular public key they wish to exchange and then reveal it to the peer.

```
      Alice:                                 Bob:
      ------                                 ----
  ka = KDF-n(F(a*Y'), F(M) | F(N) |
           F(A) | F(Y') | pw)
  u = HMAC(ka, F(X) | F(Y') |
           F(A) | Alice | 0)
  z = KDF-n(F(x*Y'), F(M) | F(N) |
           F(X) | F(Y') | pw)

                   {A, u}z ------>

                                    z = KDF-n(F(y*X'), F(M) | F(N) |
                                        F(X') | F(Y) | pw)
                                    if (SIV-decrypt returns fail) fail
                                    if (A not valid element) fail
                                    ka' = KDF-n(F(y*A), F(M) | F(N) |
                                              F(A) | F(Y) | pw)
                                    u' = HMAC(ka', F(X') | F(Y) |
                                              F(A) | Alice | 0)
                                    if (u' != u) fail
                                    kb = KDF-n(F(b*X'), F(N) | F(M) |
                                             F(B) | F(X') | pw)
                                    v = HMAC(kb, F(Y) | F(X') |
                                             F(B) | Bob | 1)

                         <------ {B, v}z

  if (SIV-decrypt returns fail) fail
  if (B not valid element) fail
  kb' = KDF-n(F(x*B'), F(N) | F(M) |
            F(B') | F(X) | pw)
  v' = HMAC(kb', F(Y') | F(X) |
            F(B') | Bob | 1)
  if (v'!= v) fail
```

where 0 and 1 are single octets of the value zero and one, respectively, n is the key length from Section 4, and both the KDF and HMAC use the hash algorithm from Section 4.

If the parties didn't fail they have each other's public key, knowledge that the peer possesses the corresponding private key, and trust that the public key belongs to the peer's stated identity.

6. IANA Considerations

This memo could create a registry of the fixed public elements for a nice cross section of popular groups. Or not. Once published this document will be a stable reference and a registry might not be needed.

7. Security Considerations

The encrypted shares exchanged in the Exchange phase MUST be ephemeral. Reuse of these keys, even with a different password, voids the security of the exchange.

The discrete logaritm of the fixed public elements MUST not be known. Knowledge of either of these values voids the security of the exchange.

The public keys exchanged in PKEX are never disclosed to an attacker, either passive or active. While they are, as the name implies, public, PKEX provides for secrecy of the exchanged keys for any protocol that might need such a capability.

PKEX has forward secrecy in the sense that exposure of the password used in a previous run of the protocol will not affect the security of that run. Also, once PKEX has finished, exposing the password to a third party would not change the fact that the public keys exchanged in that run of PKEX are trusted and bound to the entities that performed the exchange.

There is no proof of security of PKEX at this time but the Exchange phase is SPAKE2 and the security proof for that protocol can be used to help prove the security of PKEX.

8. References

8.1. Normative References

[DSS] U.S. Department of Commerce/National Institute of Standards and Technology, "Digital Signature Standard (DSS)", Federal Information Processing Standards FIPS PUB 186-4, July 2013.

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

[RFC5297] Harkins, D., "Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)", RFC 5297, DOI 10.17487/RFC5297, October 2008, <http://www.rfc-editor.org/info/rfc5297>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <http://www.rfc-editor.org/info/rfc5869>.

[X9.62] American National Standards Institute, "X9.62-2005", Public Key Cryptography for the Financial Services Industry (ECDSA), 2005.

8.2. Informative References

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <http://www.rfc-editor.org/info/rfc7250>.

[RFC7670] Kivinen, T., Wouters, P., and H. Tschofenig, "Generic Raw Public-Key Support for IKEv2", RFC 7670, DOI 10.17487/RFC7670, January 2016, <http://www.rfc-editor.org/info/rfc7670>.

Appendix A. Generation of ECC Role-specific Elements

A loop is performed to generate role-specific elements by generating a candidate point, testing the point, and exiting the loop once the test succeeds. A single octet counter is incremented each time through the loop (first time through the loop, the counter is one).

To find a candidate x-coordinate, a hash of the concatenation of the ASN.1 of the OID of the curve, a constant string, and the counter is produced. If the length of the hash's digest is less than the desired bits, the digest is pre-pended to the inputs and the result is fed back into the hash (this time it is a hash of a concatentation of the old digest, asn.1, constant string, counter) to produce the next length-of-digest bits. Excess octets are stripped off. The resulting string is interpreted as an integer with the first octet of (the first) hash being the low-order octet of the integer. Excess bits are masked to zero. If that number is larger than the prime defining the curve the counter is incremented and the loop continues. Once an x-candidate has been produced it is checked to see whether it can represent a point on the curve. If it does not, the counter is incremented and the whole loop is performed again. This process is

repeated until a point is found. The hash algorithm used to generate candidate x-coordinates is determined by Section 4.

The loop is performed twice for each curve to produce initiator- and responder-specific points. The string passed for the initiator-specific point is "PKEX Initiator", the string passed for the responder-specific point is "PKEX Responder".

Role-specific elements for popular elliptic curves are presented here.

A.1. Role-specific Elements for NIST p256

```
unsigned char nist_p256_initiator_x_coord[32] = {
    0x56, 0x26, 0x12, 0xcf, 0x36, 0x48, 0xfe, 0x0b,
    0x07, 0x04, 0xbb, 0x12, 0x22, 0x50, 0xb2, 0x54,
    0xb1, 0x94, 0x64, 0x7e, 0x54, 0xce, 0x08, 0x07,
    0x2e, 0xec, 0xca, 0x74, 0x5b, 0x61, 0x2d, 0x25
};
unsigned char nist_p256_initiator_y_coord[32] = {
    0x3e, 0x44, 0xc7, 0xc9, 0x8c, 0x1c, 0xa1, 0x0b,
    0x20, 0x09, 0x93, 0xb2, 0xfd, 0xe5, 0x69, 0xdc,
    0x75, 0xbc, 0xad, 0x33, 0xc1, 0xe7, 0xc6, 0x45,
    0x4d, 0x10, 0x1e, 0x6a, 0x3d, 0x84, 0x3c, 0xa4
};

unsigned char nist_p256_responder_x_coord[32] = {
    0x1e, 0xa4, 0x8a, 0xb1, 0xa4, 0xe8, 0x42, 0x39,
    0xad, 0x73, 0x07, 0xf2, 0x34, 0xdf, 0x57, 0x4f,
    0xc0, 0x9d, 0x54, 0xbe, 0x36, 0x1b, 0x31, 0x0f,
    0x59, 0x91, 0x52, 0x33, 0xac, 0x19, 0x9d, 0x76
};
unsigned char nist_p256_responder_y_coord[32] = {
    0x26, 0x04, 0x09, 0x45, 0x0a, 0x05, 0x20, 0xe7,
    0xa7, 0x27, 0xc1, 0x36, 0x76, 0x85, 0xca, 0x3e,
    0x42, 0x16, 0xf4, 0x89, 0x85, 0x34, 0x6e, 0xd5,
    0x17, 0xde, 0xc0, 0xb8, 0xad, 0xfd, 0xb2, 0x98
};
```

A.2. Role-specific Elements for NIST p384

```
unsigned char nist_p384_initiator_x_coord[48] = {
    0x95, 0x3f, 0x42, 0x9e, 0x50, 0x7f, 0xf9, 0xaa,
    0xac, 0x1a, 0xf2, 0x85, 0x2e, 0x64, 0x91, 0x68,
    0x64, 0xc4, 0x3c, 0xb7, 0x5c, 0xf8, 0xc9, 0x53,
    0x6e, 0x58, 0x4c, 0x7f, 0xc4, 0x64, 0x61, 0xac,
    0x51, 0x8a, 0x6f, 0xfe, 0xab, 0x74, 0xe6, 0x12,
    0x81, 0xac, 0x38, 0x5d, 0x41, 0xe6, 0xb9, 0xa3
};
unsigned char nist_p384_initiator_y_coord[48] = {
    0x89, 0xd0, 0x97, 0x7b, 0x59, 0x4f, 0xa6, 0xd6,
    0x7c, 0x5d, 0x93, 0x5b, 0x93, 0xc4, 0x07, 0xa9,
    0x89, 0xee, 0xd5, 0xcd, 0x6f, 0x42, 0xf8, 0x38,
    0xc8, 0xc6, 0x62, 0x24, 0x69, 0x0c, 0xd4, 0x48,
    0xd8, 0x44, 0xd6, 0xc2, 0xe8, 0xcc, 0x62, 0x6b,
    0x3c, 0x25, 0x53, 0xba, 0x4f, 0x71, 0xf8, 0xe7
};

unsigned char nist_p384_responder_x_coord[48] = {
    0xad, 0xbe, 0xd7, 0x1d, 0x3a, 0x71, 0x64, 0x98,
    0x5f, 0xb4, 0xd6, 0x4b, 0x50, 0xd0, 0x84, 0x97,
    0x4b, 0x7e, 0x57, 0x70, 0xd2, 0xd9, 0xf4, 0x92,
    0x2a, 0x3f, 0xce, 0x99, 0xc5, 0x77, 0x33, 0x44,
    0x14, 0x56, 0x92, 0xcb, 0xae, 0x46, 0x64, 0xdf,
    0xe0, 0xbb, 0xd7, 0xb1, 0x29, 0x20, 0x72, 0xdf
};
unsigned char nist_p384_responder_y_coord[48] = {
    0x54, 0x58, 0x20, 0xad, 0x55, 0x1d, 0xca, 0xf3,
    0x1c, 0x8a, 0xcd, 0x19, 0x40, 0xf9, 0x37, 0x83,
    0xc7, 0xd6, 0xb3, 0x13, 0x7d, 0x53, 0x28, 0x5c,
    0xf6, 0x2d, 0xf1, 0xdd, 0xa5, 0x8b, 0xad, 0x5d,
    0x81, 0xab, 0xb1, 0x00, 0x39, 0xd6, 0xcc, 0x9c,
    0xea, 0x1e, 0x84, 0x1d, 0xbf, 0xe3, 0x35, 0xf9
};
```

A.3. Role-specific Elements for NIST p521

```
unsigned char nist_p521_initiator_x_coord[66] = {
    0x00, 0x16, 0x20, 0x45, 0x19, 0x50, 0x95, 0x23,
    0x0d, 0x24, 0xbe, 0x00, 0x87, 0xdc, 0xfa, 0xf0,
    0x58, 0x9a, 0x01, 0x60, 0x07, 0x7a, 0xca, 0x76,
    0x01, 0xab, 0x2d, 0x5a, 0x46, 0xcd, 0x2c, 0xb5,
    0x11, 0x9a, 0xff, 0xaa, 0x48, 0x04, 0x91, 0x38,
    0xcf, 0x86, 0xfc, 0xa4, 0xa5, 0x0f, 0x47, 0x01,
    0x80, 0x1b, 0x30, 0xa3, 0xae, 0xe8, 0x1c, 0x2e,
    0xea, 0xcc, 0xf0, 0x03, 0x9f, 0x77, 0x4c, 0x8d,
   0x97, 0x76
};
unsigned char nist_p521_initiator_y_coord[66] = {
    0x01, 0x4c, 0x71, 0xfd, 0x1b, 0xd5, 0x9c, 0xa6,
    0xed, 0x39, 0xef, 0x45, 0xc5, 0x06, 0xfd, 0x66,
    0xc0, 0xeb, 0x0f, 0xbf, 0x21, 0xa3, 0x36, 0x74,
    0xfd, 0xaa, 0x05, 0x6e, 0x4e, 0x33, 0x95, 0x42,
    0x1a, 0x9d, 0x3f, 0x3a, 0x1c, 0x5e, 0xa8, 0x60,
    0xf7, 0xe5, 0x59, 0x1d, 0x07, 0xaa, 0x6f, 0x40,
    0x0a, 0x59, 0x3c, 0x27, 0xad, 0xe0, 0x48, 0xfd,
    0xd1, 0x83, 0x37, 0x4c, 0xdf, 0xe1, 0x86, 0x72,
    0xfc, 0x57
};

unsigned char nist_p521_responder_x_coord[66] = {
    0x00, 0x79, 0xe4, 0x4d, 0x6b, 0x5e, 0x12, 0x0a,
    0x18, 0x2c, 0xb3, 0x05, 0x77, 0x0f, 0xc3, 0x44,
    0x1a, 0xcd, 0x78, 0x46, 0x14, 0xee, 0x46, 0x3f,
    0xab, 0xc9, 0x59, 0x7c, 0x85, 0xa0, 0xc2, 0xfb,
    0x02, 0x32, 0x99, 0xde, 0x5d, 0xe1, 0x0d, 0x48,
    0x2d, 0x71, 0x7d, 0x8d, 0x3f, 0x61, 0x67, 0x9e,
    0x2b, 0x8b, 0x12, 0xde, 0x10, 0x21, 0x55, 0x0a,
    0x5b, 0x2d, 0xe8, 0x05, 0x09, 0xf6, 0x20, 0x97,
    0x84, 0xb4
};
unsigned char nist_p521_responder_y_coord[66] = {
    0x01, 0xb9, 0x9c, 0xc6, 0x41, 0x32, 0x5b, 0xd2,
    0x35, 0xd8, 0x8b, 0x2b, 0xe4, 0x6e, 0xcc, 0xdf,
    0x7c, 0x38, 0xc4, 0x5b, 0xf6, 0x74, 0x71, 0x5c,
    0x77, 0x16, 0x8a, 0x80, 0xa9, 0x84, 0xc7, 0x7b,
    0x9d, 0xfd, 0x83, 0x6f, 0xae, 0xf8, 0x24, 0x16,
    0x2f, 0x21, 0x25, 0x65, 0xa2, 0x1a, 0x6b, 0x2d,
    0x30, 0x62, 0xb3, 0xcc, 0x6e, 0x59, 0x3c, 0x7f,
    0x58, 0x91, 0x81, 0x72, 0x07, 0x8c, 0x91, 0xac,
    0x31, 0x1e
};
```

A.4. Role-specific Elements for brainpool p256r1

```
unsigned char brainpool_p256r1_initiator_x_coord[32] = {
    0x46, 0x98, 0x18, 0x6c, 0x27, 0xcd, 0x4b, 0x10,
    0x7d, 0x55, 0xa3, 0xdd, 0x89, 0x1f, 0x9f, 0xca,
    0xc7, 0x42, 0x5b, 0x8a, 0x23, 0xed, 0xf8, 0x75,
    0xac, 0xc7, 0xe9, 0x8d, 0xc2, 0x6f, 0xec, 0xd8
};
unsigned char brainpool_p256r1_initiator_y_coord[32] = {
    0x16, 0x30, 0x68, 0x32, 0x3b, 0xb0, 0x21, 0xee,
    0xeb, 0xf7, 0xb6, 0x7c, 0xae, 0x52, 0x26, 0x42,
    0x59, 0x28, 0x58, 0xb6, 0x14, 0x90, 0xed, 0x69,
    0xd0, 0x67, 0xea, 0x25, 0x60, 0x0f, 0xa9, 0x6c
};

unsigned char brainpool_p256r1_responder_x_coord[32] = {
    0x90, 0x18, 0x84, 0xc9, 0xdc, 0xcc, 0xb5, 0x2f,
    0x4a, 0x3f, 0x4f, 0x18, 0x0a, 0x22, 0x56, 0x6a,
    0xa9, 0xef, 0xd4, 0xe6, 0xc3, 0x53, 0xc2, 0x1a,
    0x23, 0x54, 0xdd, 0x08, 0x7e, 0x10, 0xd8, 0xe3
};
unsigned char brainpool_p256r1_responder_y_coord[32] = {
    0x2a, 0xfa, 0x98, 0x9b, 0xe3, 0xda, 0x30, 0xfd,
    0x32, 0x28, 0xcb, 0x66, 0xfb, 0x40, 0x7f, 0xf2,
    0xb2, 0x25, 0x80, 0x82, 0x44, 0x85, 0x13, 0x7e,
    0x4b, 0xb5, 0x06, 0xc0, 0x03, 0x69, 0x23, 0x64
};
```

A.5. Role-specific Elements for brainpool p384r1

```
unsigned char brainpool_p384r1_initiator_x_coord[48] = {
    0x0a, 0x2c, 0xeb, 0x49, 0x5e, 0xb7, 0x23, 0xbd,
    0x20, 0x5b, 0xe0, 0x49, 0xdf, 0xcf, 0xcf, 0x19,
    0x37, 0x36, 0xe1, 0x2f, 0x59, 0xdb, 0x07, 0x06,
    0xb5, 0xeb, 0x2d, 0xae, 0xc2, 0xb2, 0x38, 0x62,
    0xa6, 0x73, 0x09, 0xa0, 0x6c, 0x0a, 0xa2, 0x30,
    0x99, 0xeb, 0xf7, 0x1e, 0x47, 0xb9, 0x5e, 0xbe
};
unsigned char brainpool_p384r1_initiator_y_coord[48] = {
    0x54, 0x76, 0x61, 0x65, 0x75, 0x5a, 0x2f, 0x99,
    0x39, 0x73, 0xca, 0x6c, 0xf9, 0xf7, 0x12, 0x86,
    0x54, 0xd5, 0xd4, 0xad, 0x45, 0x7b, 0xbf, 0x32,
    0xee, 0x62, 0x8b, 0x9f, 0x52, 0xe8, 0xa0, 0xc9,
    0xb7, 0x9d, 0xd1, 0x09, 0xb4, 0x79, 0x1c, 0x3e,
    0x1a, 0xbf, 0x21, 0x45, 0x66, 0x6b, 0x02, 0x52
};

unsigned char brainpool_p384r1_responder_x_coord[48] = {
    0x03, 0xa2, 0x57, 0xef, 0xe8, 0x51, 0x21, 0xa0,
    0xc8, 0x9e, 0x21, 0x02, 0xb5, 0x9a, 0x36, 0x25,
    0x74, 0x22, 0xd1, 0xf2, 0x1b, 0xa8, 0x9a, 0x9b,
    0x97, 0xbc, 0x5a, 0xeb, 0x26, 0x15, 0x09, 0x71,
    0x77, 0x59, 0xec, 0x8b, 0xb7, 0xe1, 0xe8, 0xce,
    0x65, 0xb8, 0xaf, 0xf8, 0x80, 0xae, 0x74, 0x6c
};
unsigned char brainpool_p384r1_responder_y_coord[48] = {
    0x2f, 0xd9, 0x6a, 0xc7, 0x3e, 0xec, 0x76, 0x65,
    0x2d, 0x38, 0x7f, 0xec, 0x63, 0x26, 0x3f, 0x04,
    0xd8, 0x4e, 0xff, 0xe1, 0x0a, 0x51, 0x74, 0x70,
    0xe5, 0x46, 0x63, 0x7f, 0x5c, 0xc0, 0xd1, 0x7c,
    0xfb, 0x2f, 0xea, 0xe2, 0xd8, 0x0f, 0x84, 0xcb,
    0xe9, 0x39, 0x5c, 0x64, 0xfe, 0xcb, 0x2f, 0xf1
};
```

A.6. Role-specific Elements for brainpool p512r1

```
unsigned char brainpool_p512r1_initiator_x_coord[64] = {
    0x4c, 0xe9, 0xb6, 0x1c, 0xe2, 0x00, 0x3c, 0x9c,
    0xa9, 0xc8, 0x56, 0x52, 0xaf, 0x87, 0x3e, 0x51,
    0x9c, 0xbb, 0x15, 0x31, 0x1e, 0xc1, 0x05, 0xfc,
    0x7c, 0x77, 0xd7, 0x37, 0x61, 0x27, 0xd0, 0x95,
    0x98, 0xee, 0x5d, 0xa4, 0x3d, 0x09, 0xdb, 0x3d,
    0xfa, 0x89, 0x9e, 0x7f, 0xa6, 0xa6, 0x9c, 0xff,
    0x83, 0x5c, 0x21, 0x6c, 0x3e, 0xf2, 0xfe, 0xdc,
    0x63, 0xe4, 0xd1, 0x0e, 0x75, 0x45, 0x69, 0x0f
};
unsigned char brainpool_p512r1_initiator_y_coord[64] = {
    0x5a, 0x28, 0x01, 0xbe, 0x96, 0x82, 0x4e, 0xf6,
    0xfa, 0xed, 0x7d, 0xfd, 0x48, 0x8b, 0x48, 0x4e,
    0xd1, 0x97, 0x87, 0xc4, 0x05, 0x5d, 0x15, 0x2a,
    0xf4, 0x91, 0x4b, 0x75, 0x90, 0xd9, 0x34, 0x2c,
    0x3c, 0x12, 0xf2, 0xf5, 0x25, 0x94, 0x24, 0x34,
    0xa7, 0x6d, 0x66, 0xbc, 0x27, 0xa4, 0xa0, 0x8d,
    0xd5, 0xe1, 0x54, 0xa3, 0x55, 0x26, 0xd4, 0x14,
    0x17, 0x0f, 0xc1, 0xc7, 0x3d, 0x68, 0x7f, 0x5a
};

unsigned char brainpool_p512r1_responder_x_coord[64] = {
    0x2a, 0x60, 0x32, 0x27, 0xa1, 0xe6, 0x94, 0x72,
    0x1c, 0x48, 0xbe, 0xc5, 0x77, 0x14, 0x30, 0x76,
    0xe4, 0xbf, 0xf7, 0x7b, 0xc5, 0xfd, 0xdf, 0x19,
    0x1e, 0x0f, 0xdf, 0x1c, 0x40, 0xfa, 0x34, 0x9e,
    0x1f, 0x42, 0x24, 0xa3, 0x2c, 0xd5, 0xc7, 0xc9,
    0x7b, 0x47, 0x78, 0x96, 0xf1, 0x37, 0x0e, 0x88,
    0xcb, 0xa6, 0x52, 0x29, 0xd7, 0xa8, 0x38, 0x29,
    0x8e, 0x6e, 0x23, 0x47, 0xd4, 0x4b, 0x70, 0x3e
};
unsigned char brainpool_p512r1_responder_y_coord[64] = {
    0x2a, 0xbe, 0x59, 0xe6, 0xc4, 0xb3, 0xd8, 0x09,
    0x66, 0x89, 0x0a, 0x2d, 0x19, 0xf0, 0x9c, 0x9f,
    0xb4, 0xab, 0x8f, 0x50, 0x68, 0x3c, 0x74, 0x64,
    0x4e, 0x19, 0x55, 0x81, 0x9b, 0x48, 0x5c, 0xf4,
    0x12, 0x8d, 0xb9, 0xd8, 0x02, 0x5b, 0xe1, 0x26,
    0x7e, 0x19, 0x5c, 0xfd, 0x70, 0xf7, 0x4b, 0xdc,
    0xb5, 0x5d, 0xc1, 0x7a, 0xe9, 0xd1, 0x05, 0x2e,
    0xd1, 0xfd, 0x2f, 0xce, 0x63, 0x77, 0x48, 0x2c
};
```

Appendix B. Generation of FFC Role-Specific Elements

Haven't generated those yet. Use ECC.

Author's Address

Dan Harkins
HP Enterprise
1322 Crossman avenue
Sunnyvale, California 94089
USA

Phone: +1 415 997 9834
Email: dharkins@lounge.org