IEEE P802.11
Wireless LANs

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|

|  |
| --- |
| LB270 KCK clarification in 12.4 |
| Date: 2023-1-24 |
| Author(s): |
| Name | Affiliation | Address | Phone | email |
| Po-Kai Huang | Intel  |  |  | po-kai.huang@intel.com |
| Ido Ouzieli | Intel |  |  |  |
| Ilan Peer | Intel |  |  |  |
| Mohammad Alam | Microsoft |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

 |

Abstract

This submission proposes resolutions for the following comments from comment collection on P802.11-REVme D2.0:

3742

**Revision History:**

R0: Initial version.

# CID 3742

|  |  |  |
| --- | --- | --- |
| **CID****Clause****Page.Line** | **Comment** | **Proposed Change** |
| 3742 | KCK is used extensively in 12.4. However, for the context in 12.4, KCK should be SAE-KCK. (See page 2822 line 41). Givent that KCK is also used in 4-way. To avoid confusion, we should replace KCK in 12.4 as SAE KCK. | Go through all instances of KCK in 12.4 and change KCK to SAE-KCK after confirming the context is correct. Commenter is willing to submit contribution for the task. |

## Discussion:

KCK has been used in various context like 4-way handshake, group key handshake, SAE, TDLS, etc. This creates confusion on which KCK that the spec text in 12.4 is referred to. SAE KCK has been used in 12.4 and is a much better name for the context in 12.4. Propose to change KCK to SAE-KCK in 12.4. Note that TPK-KCK is used in the context of TDLS.

## Proposed Resolution: CID 3742

**REVISED**

**Instruction to TGme Editor:**

Implement the proposed text updates for CID 3742 in 11-23/0154r0

## Proposed Text Update: CID 3742

*Instruction to TGme Editor: Update REVme D2.0 12.4 as shown below (track change on).*

* **Authentication using a password**
* **SAE overview**

STAs, both AP STAs and non-AP STAs, may authenticate each other by proving possession of a password. Authentication protocols that employ passwords need to be resistant to off-line dictionary attacks.

Simultaneous authentication of equals (SAE) is a variant of *Dragonfly*, a password-authenticated key exchange based on a zero-knowledge proof. SAE is used by STAs to authenticate with a password; it has the following security properties:

* The successful termination of the protocol results in a PMK shared between the two STAs.
* An attacker is unable to determine either the password or the resulting PMK by passively observing an exchange or by interposing itself into the exchange by faithfully relaying messages between the two STAs.
* An attacker is unable to determine either the password or the resulting shared key by modifying, forging, or replaying frames to an honest, uncorrupted STA.
* An attacker is unable to make more than one guess at the password per attack. This implies that the attacker cannot make one attack and then go offline and make repeated guesses at the password until successful. In other words, SAE is resistant to dictionary attack.
* Compromise of a PMK from a previous run of the protocol does not provide any advantage to an adversary attempting to determine the password or the shared key from any other instance.
* Compromise of the password does not provide any advantage to an adversary in attempting to determine the PMK from the previous instance.

Unlike other authentication protocols SAE does not have a notion of an “Initiator” and “Responder” or of a “Supplicant” and “Authenticator.” The parties to the exchange are equals, with each side being able to initiate the protocol. Each side may initiate the protocol simultaneously such that each side views itself as the “initiator” for a particular run of the protocol. This is necessary to address the unique nature of MBSSs.

The parties involved are called *STA-A* and *STA-B*. They are identified by their MAC addresses, STA-A‑MAC and STA-B-MAC, respectively. STAs begin the protocol when they discover a peer by receiving Beacon or Probe Response frame(s), or when they receive an Authentication frame indicating SAE authentication from a peer.

SAE is an RSNA authentication protocol and is selected according to 12.6.2 (RSNA selection).

SAE shall be implemented on all mesh STAs to facilitate and promote interoperability.

* **Assumptions on SAE**

SAE uses two functions, H and CN, that are instantiated with hash functions in HMAC form. H takes a salt and an input key; CN takes a key, a counter, and a sequence of data. Each piece of data passed to CN is converted to an octet string and concatenated before being concatenated to the counter and passed, along with the key, to the hash function.

H(salt, ikm) = HMAC-Hash(salt, ikm)

CN(key, counter, X, Y, Z, …) = HMAC-Hash(key, counter || D2OS(X) || D2OS(Y) || D2OS(Z) || …)

where HMAC-Hash is a specific hash function in HMAC form and D2OS() represents the data to octet string conversion functions in 12.4.7.2 (Data type conversion). Each invocation of CN() specifies the format of the counter.

If used with the looping method(#344) described in 12.4.4.2.2 (Generation of the password element with ECC groups by looping) and 12.4.4.3.2 (Generation of the password element with FFC groups by looping), H and CN are instantiated with SHA‑256. If used with the hash-to-element method(#344) described in 12.4.4.2.3 (Hash-to-element(#331) generation of the password element with ECC groups) and 12.4.4.3.3 (Direct generation of the password element with FFC groups), H and CN are instantiated with a hash function from Table 12-1 (Hash algorithm based on length of prime) depending on the size of the prime defining the group being used with SAE.

|  |
| --- |
| * **Hash algorithm based on length of prime**
 |
| **ECC prime length** |  **FFC prime length** | **Hash algorithm** |
| p  256 |  p  2048 |  SHA-256 |
| 256 < p  384 | 2048 < p  3072 | SHA-384 |
| 384 < p | 3072 < p |  SHA-512 |

* **Representation of a password**

Passwords are used in SAE to deterministically compute a secret element in the negotiated group, called a password element. The input to this process needs to be in the form of a binary string. For the protocol to successfully terminate, it is necessary for each side to produce identical binary strings for a given password, even if that password is in character format. There is no canonical binary representation of a character and ambiguity exists when the password is a character string. To eliminate this ambiguity, a STA shall represent a character-based password as a UTF-8 string that is processed according to the OpaqueString profile of IETF RFC 8265, the output of which is an octet string. The octet string representation of the password, after being processed, is stored in the dot11RSNAConfigPasswordValueTable. When a “password” is called for in the description of SAE that follows the credential from the dot11RSNAConfigPasswordValueTable is used.

Similarly, to address ambiguity when identifying passwords, a STA shall represent a password identifier as a UTF-8 string that is processed according to the UsernameCasePreserved profile of IETF RFC 8265, the output of which is an octet string that is stored in the dot11RSNAConfigPasswordValueTable. When a “password identifier” is called for in the description of SAE that follows, the identifier from the dot11RSNAConfigPasswordValueTable is used.

In an infrastructure BSS for which an SAE AKM is indicated, the AP shall set the SAE Password Identifiers In Use subfield of the Extended Capabilities field of the Extended Capabilities element to 1 if any entry in the dot11RSNAConfigPasswordValueTable (#175)has a dot11RSNAConfigPasswordIdentifier that does not have a zero length, and shall set it to 0 otherwise. Similarly, an AP shall set the SAE Password Identifiers Used Exclusively subfield of the Extended Capabilities field of the Extended Capabilities element to 1 if every entry in the dot11RSNAConfigPasswordValueTable (#175)has a dot11RSNAConfigPasswordIdentifier that does not have a zero length and shall set it to 0 otherwise.

* **Finite cyclic groups**
* **General**

SAE uses discrete logarithm cryptography to achieve authentication and key agreement. Each party to the exchange derives ephemeral public and private keys with respect to a particular set of domain parameters that define a finite cyclic group. Groups may be based on either finite field cryptography (FFC) or on elliptic curve cryptography (ECC). Each component of a group is referred to as an *element*. Groups are negotiated using an identifying number from a repository maintained by IANA as “Group Description” attributes for IETF RFC 2409 (IKE) [B14][B29]. The repository maps an identifying number to a complete set of domain parameters for the particular group. Not all groups defined in this repository are suitable. Only FFC groups whose prime is at least 3072 bits and ECC groups defined over a prime field whose prime is at least 256 bits are suitable for use with SAE. ECC groups defined over a characteristic 2 finite field or ECC groups with a co‑factor greater than 1 shall not be used with SAE (see NIST Special Publication 800-57). For the purpose of interoperability, a STA shall implement support for group 19, an ECC group defined over a 256-bit prime order field.

More than one group may be configured on a STA for use with SAE by using the dot11RSNAConfigDLCGroupTable. Configured groups are prioritized in ascending order of preference. If only one group is configured, it is, by definition, the most preferred group.

NOTE—The preference of one group over another is a local policy issue.

SAE uses three arithmetic operators defined for both FFC and ECC groups, an operation that takes two elements to produce a third element (called the *element operation*), an operation that takes an integer (called *scalar*) and an element to produce a second element (called the *scalar operation*), and an operation that takes an element to produce a second element (called the *inverse operation*). The convention used here is to represent group elements in uppercase bold italic and scalar values in lowercase italic. The element operation takes two elements, ***X*** and ***Y***, to produce a third element, ***Z***, and is denoted ***Z*** = elem-op(***X***,***Y***); the scalar operation takes a scalar, *x*, and an element, ***Y***, to produce a second element ***Z***(#1409) and is denoted ***Z*** = scalar-op(*x*,***Y***); the inverse operation takes an element, ***X***, to produce a second element, ***Z***, and is denoted ***Z*** = inverse-op(***X***).

scalar-op(*x*,***Y***) is defined as successive iterations of elem-op(***Y***,***Y***). That is, it is possible to define scalar-op*(*1,***Y***) = ***Y*** and for *x* > 1, scalar-op(*x,* ***Y***) = elem-op(scalar-op(*x-*1,***Y***),***Y****)*. The specific definition of elem-op(**X***,****Y***) depends on the type of group, either ECC or FFC.

* **Elliptic curve cryptography (ECC) groups**
* **ECC group definition**

ECC groups used by SAE are defined by the sextuple (*p*, *a*, *b*, ***G***, *r*, *h*) where *p* is a prime number, *a* and *b* specify the elliptic curve defined by the equation, *y*2 = *x*3 + *ax* + *b* mod *p*, ***G*** is a generator (a base point on the elliptic curve), *r* is the prime order of ***G***, and *h* is the co-factor. Elements in ECC groups are the points on the elliptic curve defined by their coordinates—(*x*, *y*)—that satisfy the equation for the curve and the identity element, the so-called “point at infinity.”

The element operation in an ECC group is addition of two points on the curve resulting in a third point on the curve. For example, the point ***X*** is added to the point ***Y*** to produce the point ***Z***:

***Z*** = ***X*** + ***Y*** = elem-op(***X***,***Y***)

The scalar operation in an ECC group is multiplication of a point on the curve by a scalar resulting in a second point on the curve. For example, the point ***Y*** is multiplied by the scalar *x* to produce the point ***Z***:

***Z*** = *x****Y*** = scalar-op(*x*,***Y***)

The inverse operation in an ECC group is inversion of a point on a curve resulting in a second point on the curve. A point on an elliptic curve is the inverse of a different point if their sum is the “point at infinity.” In other words:

elem-op(***X***, inverse-op(***X***)) = “point at infinity”

ECC groups make use of a mapping function, F, that maps a point (*x*, *y*) that satisfies the curve equation to its x-coordinate—i.e., if ***P*** = (*x*, *y*) then F(***P***) = *x*. Function F is not defined with the identity element as input(#1075).

* **Generation of the password element with ECC groups by looping**

If the AP does not indicate support for the SAE hash-to-element method(#355) in its Extended RSN Capabilities field or the SAE initiator does not set the status code to SAE\_HASH\_TO\_ELEMENT in its SAE Commit message, the password element of an ECC group (***PWE***) shall be generated in the following random hunt-and-peck fashion.

NOTE 1—This method cannot be used with a password identifier.

The password and a counter, represented as a single octet and initially set to 1, are used with the peer identities to generate a password seed. The password seed shall then be stretched using the key derivation function (KDF) from 12.7.1.6.2 (Key derivation function (KDF)) to a length equal to the bit length of the prime number, *p*, from the elliptic curve domain parameters with the Label being the string “SAE Hunting and Pecking” and with the Context being the prime number. If the resulting password value is greater than or equal to the prime number, the counter shall be incremented, a new password seed shall be derived and the hunting-and-pecking shall continue. Otherwise, it shall be used as the x-coordinate of a candidate point (*x*, *y*) on the curve satisfying the curve equation, if such a point exists. If no solution exists, the counter shall be incremented, a new password-seed shall be derived and the hunting-and-pecking shall continue. Otherwise, there are two possible solutions: (*x*, *y*) and (*x*, *p* – *y*). The password seed shall be used to determine which one to use: if the least significant bit (LSB) of the password seed is equal to that of *y*, the ***PWE*** shall be set to (*x*, *y*); otherwise, it shall be set to (*x*, *p* – *y*).

In order to minimize the possibility of side-channel attacks that attempt to determine the number of interactions of the “hunting-and-pecking” loop required for a given <password, STA-A‑MAC, STA-B-MAC> tuple, implementations should perform at least *k* iterations regardless of whether ***PWE*** is discovered or not. The value *k* may be set to any non-negative value and should be set to a sufficiently large number to effectively guarantee the discovery of ***PWE*** in less than *k* iterations. If ***PWE*** is discovered in less than *k* iterations a random “password” can be used in subsequent iterations to further obfuscate the true cost of discovering ***PWE***.

NOTE 2—The probability that one requires more than *n* iterations of the “hunting and pecking” loop to find ***PWE*** is roughly (*r*/2*p*)*n*, which rapidly approaches 0 as *n* increases.

Algorithmically this process is described as follows:

 *found* = 0;

 *counter* = 1

 *Length* = len(*p*)

 *base = password*

 do {

 *pwd*-*seed* = H(MAX(STA-A-MAC, STA-B-MAC) || MIN(STA-A-MAC, STA-B-MAC),

 *base* || *counter*)

 (#478)*pwd*-*value* = KDF-*Hash*-*Length*(*pwd*-*seed*, “SAE Hunting and Pecking”, *p*)

 if (*pwd-value* < *p*)

 then

if (*pwd-value*3 + *a* × *pwd-value* + *b*) is a quadratic residue modulo *p*

 then

 if (*found*==0)

 then

 *x* = *pwd-value*

 *save* = *pwd-seed*

 *found* = 1

 *base* = a new random number

 fi

 fi

 fi

 *counter* = *counter* + 1

 } while ((*counter*  *k*) or (*found*==0))

 *y* = sqrt(*x*3 + *ax* + *b*) mod *p*

 if (LSB(*save*) == LSB(*y*))

 then

 ***PWE*** = (*x*, *y*)

 else

 ***PWE*** = (*x*, *p – y*)

 fi

where

(#478)KDF-*Hash*-*Length* is the key derivation function defined in 12.7.1.6.2 (Key derivation function (KDF)) using the hash algorithm identified by the AKM suite selector (see Table 9-188 (AKM suite selectors)); the context passed to (#478)KDF-*Hash*-*Length*, *p*, is the octet string representation of the prime per 12.4.7.2.2 (Integer to octet string conversion).

len() returns the length of its argument in bits

Checking whether a value is a quadratic residue modulo a prime can leak information that can be used in launching a side-channel attack. Therefore, a STA should use this blinding technique in determining a quadratic residue to address the possibility of a side-channel attack.

The blinding technique involves multiplication of the value with a random number so the value being checked for quadratic residue modulo a prime can take on all numbers between 1 and *p*–1 with equal probability. The blinded value is multiplied by a quadratic residue or quadratic nonresidue depending on the value of a coin flip and the result is checked whether the result is a quadratic residue or quadratic nonresidue, respectively.

This technique involves creation of a quadratic residue, *qr*, and quadratic nonresidue, *qnr*, prior to beginning of the hunting-and-pecking loop. These values can be chosen at random by checking their legendre symbol:

 do {

 *qr* = random() mod *p*

 } while ( LGR(*qr | p*) is not equal to 1)

 do {

 *qnr* = random() mod *p*

 } while ( LGR(*qnr | p*) is not equal to -1)

The blinding technique of determining whether a value, *v*, is a quadratic residue modulo a prime, *p*, is then:

 *r* = (random() mod (*p* – 1)) + 1

 *num* = (*v* × *r* × *r*) mod *p*

 if (LSB(*r*) == 1)

 then

 *num* = (*num* × *qr*) mod *p*

 if (LGR(*num | p*) == 1)

 then

 *v* is a quadratic residue modulo *p*

 fi

 else

 *num* = (*num* × *qnr*) mod *p*

 if (LGR(*num | p*) == –1)

 then

 *v* is a quadratic residue modulo *p*

 fi

 fi

 *v* is a quadratic nonresidue modulo *p*

The values *qr* and *qnr* may be used for all loops in the hunting-and-pecking process but a new value for *r* shall be generated each time a quadratic residue is checked.

* **Hash-to-element(#331) generation of the password element with ECC groups**

An SAE peer, e.g. a mesh STA or an AP, indicates support for the hash-to-element method(#344) to obtain an ECC password element by setting the SAE hash-to-element bit to 1 in the Extended RSN Capabilities field in all Beacon and Probe Response frames. A STA that uses a password identifier shall use the hash-to-element(#331) method. An SAE initiator that has identified a peer that supports this method(#344) (through receipt of Beacon or Probe Response frames) shall derive a secret element, PT, according to the following method(#344) and indicate this by setting the status code in the SAE Commit message to SAE\_HASH\_TO\_ELEMENT. An SAE initiator shall not indicate support for this form of element derivation unless its peer has already signalled support for this method. If an SAE Commit message is received with status code equal to SAE\_HASH\_TO\_ELEMENT the peer shall generate the PWE using the following method(#344) and reply with its own SAE Commit message with status code set to SAE\_HASH\_TO\_ELEMENT.

The hash-to-element method(#344) to derive an element of an ECC group is the Simplified Shallue-Woestijne-Ulas (SSWU) deterministic hash-to-element(#331) method. The SSWU method is called twice with two distinct functions to produce two points on the elliptic curve. The two points are summed to create a secret element PT.

This method works for all Weierstrass elliptic curves whose constants a and b are both not equal to zero. Other curves shall not be used with the hash-to-element(#331) method.

The hash-to-element(#331) method uses HKDF (IETF RFC 5869) with the hash algorithm taken from Table 12-1 (Hash algorithm based on length of prime) based on the length of the prime of the ECC group to perform both functions. First HKDF-Extract is passed a salt in the form of the SSID for which the password is to be used, the password, and optionally a password identifier to produce and intermediary password seed. The resulting seed is passed to HKDF-Expand to produce two distinct strings using different labels. Both values are reduced modulo p, the prime defining the curve, and then passed to SSWU to produce distinct points, *P1* and *P2*, whose sum is *PT*.

This secret *PT* is stored until needed to generate a session specific PWE (see 12.4.5.2 (PWE and secret generation)).

Algorithmically, this process is as follows:

 *len* = *olen(p)* + 

 *pwd-seed* = HKDF-Extract(*ssid*, *password* [|| *identifier*])

 *pwd-value* = HKDF-Expand(*pwd-seed*, “*SAE Hash to Element u1 P1*”, *len*)

 *u1* = *pwd-value* modulo *p*

 *P1* = SSWU(*u1*)

 *pwd-value* = HKDF-Expand(*pwd-seed*, “*SAE Hash to Element u2 P2*”, *len*)

 *u2* = *pwd-value* modulo *p*

 *P2* = SSWU(*u2*)

 *PT* = elem-op(*P1*, *P2*)

where

HKDF-Extract() and

HKDF-Expand() are the functions defined in IETF RFC 5869, instantiated with the hash algorithm from Table 12-1 (Hash algorithm based on length of prime)

*ssid* is an octet string that represents the SSID with which the password is to be used

*olen*() returns the length of its argument in octets

[|| *identifier*] indicates the optional inclusion of a password identifier, if present

SSWU(*u*) is a call to the Simple SWU routine passing in parameter *u*

The SSWU method produces two values, *x1*, and *x2*, at least one of which will represent an abscissa of a point on the curve. If *x1* is the abscissa, then *x1* becomes the x-coordinate otherwise *x2* becomes the x-coordinate. The equation of the curve with the x-coordinate produces the square of the y-coordinate which is recovered by taking the square root. The two possible results of the square root are discriminated by checking its least significant bit with the least significant bit of u. The result is a point on the curve.

The SSWU method takes a curve-specific parameter, *z*, which is determined from the following formula, given *p*, *a*, and *b*, from the curve’s domain parameter set:

 Assign a counter, *ctr*, the value 1. If the following conditions are true for *n* = *ctr* then *z* = *ctr*. Otherwise, if they are true for *n* = – *ctr* then *z* = – *ctr*. Otherwise increment *ctr* and repeat until a value for *z* is found.

* *n* is not a quadratic residue modulo *p*
* *n* is not –1
* the polynomial *x*3 + *a* × *x* + *b* – *n* is irreducible
* (*b*/(*n* × *a*))*3* + *a* × (*b*/(*n* × *a*)) + *b* is a quadratic residue modulo *p*

Values for some defined groups based on their IANA-assigned values are listed in Table 12-2 (Unique curve parameter).

|  |
| --- |
| * **Unique curve parameter**
 |
| **Curve name**  | **IANA value** | **z** |
| NIST p256 |  19 |  –10 |
| NIST p384 | 20 | –12 |
| NIST p521 | 21 | –4 |
| NIST p192 | 25 | –5 |
|  NIST p224 | 26 | 31 |
| Brainpool p256 | 28 | –2 |
| Brainpool p384 | 29 | –5 |
| Brainpool p512 | 30 | 7 |

Algorithmically, the Simplified SWU method is:

SSWU(*u*) {

  *m* = (*z2* × *u4* + *z* × *u2*) modulo *p*

  *l* = CEQ(*m*, *0*)

 *t* = *inv0*(*m*)

  *x1* = CSEL(*l*, (*b* / (*z* × *a*) modulo *p*), ((– *b*/*a*) × (*1* + *t*)) modulo *p*)

 *gx1* = (*x13* + *a* × *x1* + *b*) modulo *p*

 *x2* = (*z* × *u2* × *x1*) modulo *p*

 *gx2* = (*x23* + *a* × *x2* + *b*) modulo *p*

 *l* = *gx1* is a quadratic residue modulo *p*

 *v* = CSEL(*l*, *gx1*, *gx2*)

 *x* = CSEL(*l*, *x1*, *x2* )

  *y* = sqrt(*v*)

  *l* = CEQ(LSB(*u*), LSB(*y*))

  *P* = CSEL(*l*, (*x*,*y*), (*x*, *p* – *y*))

  *output P*

}

where

*p*, *a*, and *b* are all defined in the domain parameter set for the curve

*z* is a curve-specific parameter from Table 12-2 (Unique curve parameter)

*inv0*(*x*) is calculated as *x(p-2)* modulo *p*

*x* is a quadratic residue if *x((p-1)/2)* modulo *p* is zero or one

LSB(*x*) returns the least significant bit of *x*

CSEL(*x*,*y*,*z*) operates in constant time and returns *y* if *x* is true and *z* otherwise

CEQ(*x*,*y*) operates in constant time and returns true if *x* equals *y* and false otherwise

All operations in the SSWU algorithm shall be done in constant time.

NOTE—For curves based on a prime, *p*, such that *p* = 3 mod 4 the square root can be implemented with a single modular exponentiation of (*p*+1)/4, that is sqrt(w) = *w(p+1)/4* modulo *p*.

* **Finite field cryptography (FFC) groups**
* **FFC group definition**

FFC groups used by SAE are defined by the triple (*p*, ***G***, *r*), where *p* is a prime number, ***G*** is a generator, and *r* is the prime order of ***G*** mod *p*. An element, ***B***, in an FFC group satisfies ***B*** = ***G****i* mod *p* for some integer *i*. This special property differentiates elements from scalars, even though both elements and scalars can be represented as non-negative integers less than the prime modulus p. The notation convention of 12.4.4 (Finite cyclic groups) signifies this difference between an element and a scalar in an FFC group. The identity element for an FFC group is the value 1 mod *p*.

The element operation in an FFC group is modular multiplication of two elements of this group resulting in a third element of this group. For example, the element ***X*** is multiplied by the element ***Y*** to product the element ***Z***:

***Z*** = (***XY***) mod *p* = elem-op(***X***,***Y***)

The scalar operation in an FFC group is modular exponentiation of an element of this group by a scalar resulting in a second element of this group. For example, the point ***Y*** is raised to the power *x* to produce the element ***Z***:

***Z*** = ***Y****x* mod *p* = scalar-op(*x*,***Y***)

Some FFC groups in the IANA repository are based on *safe primes*, i.e., a prime, *p*, of the form *p* = 2*q* + 1, where *q* is also a prime number. For these FFC groups, the group generated by ***G*** always has order *r* = (*p* –1)/2 and thus is uniquely derived from context. For other FFC groups, the parameter *r* shall be explicitly stated as part of the domain parameters.

The inverse operation in an FFC group is modular inversion of an element of this group producing a second element in this group. An element ***Z*** is the inverse of a second element ***X*** of this group if their modular product is the identity element of the FFC group. In other words:

elem-op(***X***, inverse-op(***X***)) = 1 mod *p*

In contrast to ECC groups, FFC groups do not need a mapping function that maps an element of the FFC group to an integer (since those elements are already non-negative integers less than the prime number, *p*). However, for sake of uniform protocol definition, function F with FFC groups is defined as the identity function—i.e., if *x* is an element of the FFC group then F(*x*) = *x*.

* **Generation of the password element with FFC groups by looping**

If the AP does not indicate support for the SAE hash-to-element method(#355) in its Extended RSN Capabilities field or the SAE initiator does not set the status code to SAE\_HASH\_TO\_ELEMENT in its SAE Commit message, the password element of an FFC group (***PWE***) shall be generated in the following random hunt-and-peck fashion.

NOTE—This method cannot be used with a password identifier.

The password and a counter, represented as a single octet and initially set to 1, are used with the two peer identities to generate a password seed. The password seed shall then be stretched using the key derivation function (KDF) from 12.7.1.6.2 (Key derivation function (KDF)) to a length equal to the bit length of the prime number, *p*, from the group domain parameters with the Label being the string “SAE Hunting and Pecking” and the Content being the prime number. If the resulting password value is greater than or equal to the prime number, the counter shall be incremented, a new password seed shall be derived, and the hunting-and-pecking shall continue. Otherwise, it shall be raised to the power (*p –* 1) */ r* (where *p* is the prime number and *r* is the order) modulo the prime number to produce a candidate ***PWE***. If the candidate ***PWE*** is greater than 1, the candidate ***PWE*** becomes the ***PWE***; otherwise, the counter shall be incremented, a new password seed shall be derived, and the hunting-and-pecking shall continue.

Algorithmically this process is described as follows:

 *found* = 0;

 *counter* = 1

 *Length* = len(*p*)

 do {

 *pwd-seed* = H(MAX(STA-A-MAC, STA-B-MAC) || MIN(STA-A-MAC, STA-B-MAC),

 password || *counter*)

 (#478)*pwd-value* = KDF-*Hash*-*Length*(*pwd-seed*, “SAE Hunting and Pecking”, *p*)

 if (*pwd-value* < *p*)

 then

 ***PWE*** = *pwd-value(p-1)/r* mod *p*

 if (***PWE*** > 1)

 then

 *found* = 1

 fi

 fi

 *counter* = *counter* + 1

 } while (*found*==0)

where

(#478)KDF-*Hash*-*Length* is the key derivation function defined in 12.7.1.6.2 (Key derivation function (KDF)) using the hash algorithm identified by the AKM suite selector (see Table 9-188 (AKM suite selectors)); the context passed to (#478)KDF-*Hash*-Length, *p*, is the octet string representation of the prime per 12.4.7.2.2 (Integer to octet string conversion)

len() returns the length of its argument in bits

* **Direct generation of the password element with FFC groups**

An SAE peer indicates support for the hash-to-element method(#344) to obtain the FFC password element by setting the SAE hash-to-element bit to 1 in the Extended RSN Capabilities field in all Beacon and Probe Response frames. A STA that uses a password identifier shall use the hash-to-element method(#344). An SAE initiator that has identified a peer that supports the following technique (through receipt of Beacon or Probe Response frames) shall derive PT according to the following method(#344) and indicate this by setting the status code in the SAE Commit message to SAE\_HASH\_TO\_ELEMENT. An SAE initiator shall not indicate support for this form of PWE derivation unless its peer has already signalled support. If an SAE Commit message is received with status code equal to SAE\_HASH\_TO\_ELEMENT the peer shall generate the PWE using the following method(#344) and reply with its own SAE Commit message with status code set to SAE\_HASH\_TO\_ELEMENT.

The hash-to-element method(#344) uses HKDF (IETF RFC 5869) with the hash algorithm taken from Table 12-1 (Hash algorithm based on length of prime) based on the length of the prime of the FFC group.

To perform the hash-to-element method(#344), HKDF (IETF RFC 5869) is passed a salt in the form of the SSID for which the password is to be used, the password, optionally a password identifier, as an input key, a constant label “SAE Hash to Element”, and the length of the prime to produce a password value. The resulting password value shall be reduced into a range such that 1 < *pwd-value* < *p*. Then, it shall be raised to the power (*p*-1) / *q* and reduced modulo *p* (where *p* is the prime number and *q* is the order). This will ensure PT is a generator of order either 1 (if *PT* = 1) or *q* (for all other values). The probability of PT taking the value 1 is to be neglected.

This secret PT is stored until needed to generate a session specific PWE.

Algorithmically, this process is as follows:

 *len* = *olen*(*p*) + 

 *pwd-seed* = HKDF-Extract(*ssid*, *password* [|| *identifier*])

 *pwd-value* = HKDF-Expand(*pwd-seed*, “SAE Hash to Element”, *len*)

 *pwd-value* = (*pwd-value* modulo (*p* – 2)) + 2

 *PT* = *pwd-value(p-1)/q* modulo *p*

where

HKDF-Extract() and

HKDF-Expand() are the functions defined in IETF RFC 5869, instantiated with the hash algorithm from Table 12-1 (Hash algorithm based on length of prime)

*ssid* is an octet string that represents the SSID with which the password is to be used

*olen*() returns the length of its argument in octets

[|| *identifier*] indicates the optional inclusion of a password identifier, if present

*p* and *q* are defined in the domain parameter set for the group

This secret PT is stored until needed to generate a session specific PWE (see 12.4.5.2 (PWE and secret generation)).

* **SAE protocol**
* **Message exchanges**

The protocol consists of two message exchanges, a commitment exchange and a confirmation exchange. The commitment exchange is used to force each party to the exchange to commit to a single guess of the password. The confirmation exchange is used to prove that the password guess was correct. Authentication frames are used to perform these exchanges (see 9.3.3.11 (Authentication frame format), 12.4.7.3 (Encoding and decoding of SAE Commit messages) and 12.4.7.4 (Encoding and decoding of SAE Confirm messages))(Ed1). The rules for performing these exchanges are specified by the finite state machine in 12.4.8 (SAE finite state machine).

When a party has sent its message in the commit exchange it is said to have *committed* and when it has sent its message in the confirmation exchange it has *confirmed*. The following rules are ascribed to the protocol:

* A party may *commit* at any time
* A party *confirm*s after it has *committed* and its peer has *committed*
* A party *accept*s authentication after a peer has *confirmed*
* The protocol successfully *terminates* after each peer has *accepted*
* **PWE and secret generation**

Prior to beginning the protocol message exchange, the secret element ***PWE*** and two secret values are generated.

When a STA supports the hash-to-element method(#344) (according to 12.4.4.2.3 (Hash-to-element(#331) generation of the password element with ECC groups) or 12.4.4.3.3 (Direct generation of the password element with FFC groups)) it computes a secret element, PT, offline at provisioning time for all groups it wishes to support with that password. Prior to initiating SAE to a STA that also supports the direct form of hashing to a group element, or upon receipt of an SAE Commit message indicating it was generated using a direct form of hashing to a group element, it shall generate the PWE by hashing the two peer MAC addresses to produce a digest, reducing the digest modulo the order of the particular group, *r*, interpreting the reduced digest as an integer and using it with the secret element to generate the PWE:

*val = H(0n, MAX(STA-A-MAC, STA-B-MAC) || MIN(STA-A-MAC, STA-B-MAC))*

*val = val* modulo *(r – 1) + 1*

*PWE = scalar-op(val, PT)*

where 0n is a salt of all zeros whose length equals the length of the digest from the hash function used to instantiate H() (see Table 12-1 (Hash algorithm based on length of prime)).

If a STA does not support a direct form of hashing to a group element, it generates the PWE after selecting a group, either the most preferred group if the STA is initiating SAE to a peer, or the group from a received SAE Commit message if the STA is responding to a peer. The ***PWE*** shall be generated for that group (according to 12.4.4.2.2 (Generation of the password element with ECC groups by looping) or 12.4.4.3.2 (Generation of the password element with FFC groups by looping), depending on whether the group is ECC or FFC, respectively) using the identities of the two STAs and the configured password.

After generation of the ***PWE***, each STA shall generate a secret value, *rand*, and a temporary secret value, *mask*, each of which shall be chosen randomly such that 1 < *rand* < *r* and 1 < *mask* < *r* and (*rand + mask*)mod *r* is greater than 1, where *r* is the (prime) order of the group. If their sum modulo r is not greater than 1, they shall both be irretrievably deleted and new values shall be randomly generated. The values *rand* and *mask* shall be random numbers produced from a quality random number drawn from a uniform distribution generator. These values shall never be reused on distinct protocol runs.

* **Construction of an SAE Commit message**

The scalar and element in an SAE Commit message shall be produced using ***PWE*** and secrets generated in 12.4.5.2 (PWE and secret generation), as follows:

*commit*-*scalar* = (*rand* + *mask*) mod *r*

***COMMIT-ELEMENT*** = inverse-op(scalar-op(*mask*,***PWE***))

This message shall be transmitted to the peer as described in 12.4.7 (Framing of SAE). The temporary secret *mask* may be deleted at this point.

(M67)To derive keys for use with AKM 00-0F-AC:24 or AKM 00-0F-AC:25, an AKM Suite Selector element indicating 00-0F-AC:24 or 00-0F-AC:25 shall be included in an SAE Commit message transmitted to the peer.

(M67)If an SAE Commit message that includes an AKM Suite Selector element has been received, the AKM indicated in the AKM Suite Selector element is supported, and a SAE Commit message is constructed, then the SAE Commit message shall include an AKM Suite Selector element that indicates the same AKM.

* **Processing of a peer’s SAE Commit message**

If the peer’s SAE Commit message contains a (#2168)password identifier (PWE), the value of that identifier shall be used in construction of the (#2168)PWE for this exchange. If a password identifier is present in the peer’s SAE Commit message and there is no password with the given identifier a STA shall fail authentication.

If the peer’s SAE Commit message contains a Rejected Groups element, the list of rejected groups shall be checked to ensure that all of the groups in the list are groups that would be rejected. If any groups in the list would not be rejected then processing of the SAE Commit message terminates and the STA shall reject the peer’s authentication. While the rejected groups are appended to the Rejected Groups element as they are rejected (see 12.4.7.3 (Encoding and decoding of SAE Commit messages)) there is no inherent order to the groups in the list. The order in which they are sent and received shall be retained when deriving keys.

(M67)If the state of the SAE finite state machine is *Committed* (see 12.4.8.2.2 (Protocol instance states)) and the SAE Commit message that has been sent by the SAE finite state machine to transition into *Committed* state includes an AKM Suite Selector element, the authentication shall fail if either of the following conditions is true:

* the peer’s SAE Commit message does not contain an AKM Suite Selector element
* the peer’s SAE Commit message contains an AKM Suite Selector element and the AKM Suite Selector element does not indicate the same AKM

Upon receipt of a peer’s SAE Commit message both the scalar and element shall be verified.

If the scalar value is greater than 1 and less than the order, *r*, of the negotiated group, scalar validation succeeds; otherwise, it fails. Element validation depends on the type of group. For FFC groups, the element shall be an integer greater than 1 and less than the prime number *p* minus 1, (*p –*1), and the scalar operation of the element and the order of the group, *r*, shall equal 1 modulo the prime number *p*. If either of these conditions does not hold, element validation fails; otherwise, it succeeds. For ECC groups, both the x- and y-coordinates of the element shall be non-negative integers less than the prime number *p*, and the two coordinates shall produce a valid point on the curve satisfying the group’s curve definition, not being equal to the “point at the infinity.” If either of those conditions does not hold, element validation fails; otherwise, element validation succeeds.

If either scalar validation or element validation fails, the STA shall reject the peer’s authentication. If both the scalar and element from the peer’s SAE Commit message are successfully validated, a shared secret element, *K*, shall be derived using the scalar and element (*peer-commit-scalar* and ***PEER-COMMIT-ELEMENT***, respectively) from the peer’s SAE Commit message and the STA’s secret value.

***K***= scalar-op(*rand*, (elem-op(scalar-op(*peer-commit-scalar*, ***PWE***), ***PEER-COMMIT-ELEMENT***)))

If the shared secret element, ***K***, is the identity element for the negotiated group (the value one for an FFC group or the point-at-infinity for an ECC group) the STA shall reject the peer’s authentication. Otherwise, a secret value, *k*, shall be computed as:

*k* = F(***K***)

The entropy of *k* shall then be extracted using H to produce *keyseed*. The key derivation function from 12.7.1.6.2 (Key derivation function (KDF)) shall then be used with the hash algorithm identified for H() (see 12.4.2 (Assumptions on SAE)) to derive a SAE key confirmation key, SAE-KCK, and a pairwise master key, PMK, from *keyseed*.

(M67)The intended AKM for the purpose of PMK and SAE-KCK size determination (see below) is determined as follows:

* If an AKM Suite Selector element is not included in the SAE Commit message from the peer and the state of the SAE finite state machine is *Nothing* (see 12.4.8.2.2 (Protocol instance states)), then 00-0F-AC:8 or 00-0F-AC:9 shall be the intended AKM.
* If the state of the SAE finite state machine is *Committed* (see 12.4.8.2.2 (Protocol instance states)) and the SAE Commit message that has been sent by the SAE finite state machine to transition into *Committed* state does not include an AKM Suite Selector element, then 00-0F-AC:8 or 00-0F-AC:9 shall be the intended AKM.
* If an AKM Suite Selector element that indicates AKM 00-0F-AC:24 or AKM 00-0F-AC:25 is included in the SAE Commit message from the peer and the state of the SAE finite state machine is *Nothing* (see 12.4.8.2.2 (Protocol instance states)), then the indicated AKM shall be the intended AKM.
* If the state of the SAE finite state machine is *Committed* (see 12.4.8.2.2 (Protocol instance states)) and the SAE Commit message that has been sent by the SAE finite state machine to transition into *Committed* state includes an AKM Suite Selector element that indicates AKM 00-0F-AC:24 or AKM 00-0F-AC:25, then the indicated AKM shall be the intended AKM.

(M67)If the intended AKM is (M21)00-0F-AC:8 or 00-0F-AC:9 and the looping method of PWE generation (see 12.4.4.2.2 (Generation of the password element with ECC groups by looping) and 12.4.4.3.2 (Generation of the password element with FFC groups by looping)), both the SAE-KCK and PMK shall be 256 bits in length. (M67)If the intended AKM is 00-0F-AC:8 or 00-0F-AC:9 and the hash-to-element method(#344) of PWE generation (see 12.4.4.2.3 (Hash-to-element(#331) generation of the password element with ECC groups) and 12.4.4.3.3 (Direct generation of the password element with FFC groups)), the SAE-KCK shall have(M67) the length of the digest generated by H() and the PMK shall be 256 bits in length (M21)(see 12.7.1.3 (Pairwise key hierarchy)). (M67)If the intended AKM is 00-0F-AC:24 or 00-0F-AC:25, the hash-to-element method(#344) of PWE generation (see 12.4.4.2.3 (Hash-to-element(#331) generation of the password element with ECC groups) and 12.4.4.3.3 (Direct generation of the password element with FFC groups)) shall be used, the SAE-KCK and the PMK shall have the length of the digest generated by H().(M67) Use of other AKMs with the hash-to-element method(#344) will require definition of the length of the PMK. If both SAE Commit messages indicated a status code of SAE\_HASH\_TO\_ELEMENT, a salt consisting of the concatenation of the rejected groups from each peer’s Rejected Groups element shall be passed to the KDF; those of the peer with the highest MAC address go first (if only one sent a Rejected Groups element then the salt will consist of that list). If neither peer sent a Rejected Groups element or the status code was not SAE\_HASH\_TO\_ELEMENT, the salt shall consist of a series of octets of the value zero whose length equals the length of the digest of the hash function used to instantiate H().

*keyseed* = H(*salt*, *k*)

*context* = (*commit-scalar* + *peer-commit-scalar*) mod *r*

*Length* = *Q* + *PMK\_bits*(M21)

(#478)sae\_*kck\_and\_pmk* = KDF-*Hash*-*Length*(*keyseed*, “SAE KCK and PMK”, *context*)

*SAE-KCK* = L(sae\_*kck\_and\_pmk*, 0, *Q*)

*PMK* = L(sae\_*kck\_and\_pmk*, *Q*, *PMK\_bits*)(M21)

where

*salt* is either a series of 0 octets or a list of rejected groups (see 12.4.7.3 (Encoding and decoding of SAE Commit messages))

(#478)KDF-*Hash*-*Length* is the key derivation function defined in 12.7.1.6.2 (Key derivation function (KDF)) using the hash algorithm defined for H()

*Q* is the length of the digest of the H(), the hash function used

*context* is treated as an integer and converted into an octet string of length *m* such that 28m > *r* according to 12.4.7.2.2 (Integer to octet string conversion)

*PMK\_bits*  is the length of the PMK in bits, as defined in 12.7.1.3 (Pairwise key hierarchy)(M21)

The PMK identifier is defined as follows:

PMKID = L(*context*, 0, 128)

* **Construction of an SAE Confirm message**

A peer generates a confirmation, *confirm*, and inserts it into an SAE Confirm message by passing the SAE-KCK, the current value of the *send-confirm* counter (see 9.4.1.37 (Send-Confirm field)), the scalar and element from the sent SAE Commit message, and the scalar and element from the received SAE Commit message to the confirmation function CN.

*confirm* = CN(SAE-KCK*,* *send-confirm, commit*-*scalar,* ***COMMIT-ELEMENT***, *peer*-*commit*-*scalar,*

***PEER-COMMIT-ELEMENT***)

The *send-confirm* counter shall be encoded according to 9.2.2 (Conventions). The elements and scalars shall be in the format they were encoded in when transmitted in an SAE Commit message as described in 12.4.7.3 (Encoding and decoding of SAE Commit messages). The message shall be transmitted to the peer as described in 12.4.7 (Framing of SAE).

* **Processing of a peer’s SAE Confirm message**

Upon receipt of a peer’s SAE Confirm message a *verifier* is computed, which is the expected value of the peer’s confirmation, *peer-confirm*, extracted from the received an SAE Confirm message. The *verifier* is computed by passing the SAE-KCK, the peer’s send-confirm counter from the received an SAE Confirm message (see 9.4.1.37 (Send-Confirm field)), the scalar and element from the received SAE Commit message, and scalar and element from the sent SAE Commit message to the confirmation function CN.

*verifier* = CN(SAE-KCK*, peer-send*-*confirm,* *peer*-*commit*-*scalar,* ***PEER-COMMIT-ELEMENT***,
 *commit*-*scalar,* ***COMMIT-ELEMENT***)

The *peer-send-confirm* shall be encoded according to 9.2.2 (Conventions). The elements and scalars shall be in the format they were encoded in when transmitted in an SAE Commit message as described in 12.4.7.3 (Encoding and decoding of SAE Commit messages). If the *verifier* differs from the *peer*-*confirm,* verification of the peer’s SAE Confirm message shall fail.