Network Working Group
Internet Draft
draft-jablon-speke-02.txt

D. Jablon
Phoenix Technologies
October 22, 2003

The SPEKE Password-Based Key Agreement Methods

Status of this Memo

Abstract

This document describes the SPEKE, B-SPEKE, and W-SPEKE methods for password-based key agreement and authentication. In the same category of techniques as EKE and SRP, these methods provide a zero-knowledge password proof and authenticate session keys over an unprotected channel, with minimal dependency on infrastructure and proper user behavior. These methods are compatible with IEEE 1363 and ANSI X9 standards and provide important options for convenient and secure personal authentication.

Table of Contents

1. Introduction

This document describes SPEKE, B-SPEKE, and W-SPEKE, three methods that provide cryptographically strong password-based key agreement and network authentication. These methods are in the same category as the earlier EKE [BM92] and later SRP [RFC2945] methods, but use some different techniques. Some advantages of these methods over SRP are slightly increased security and compatibility with IEEE 1363 and ANSI X9 cryptography standards. An appendix highlights changes that can be made to an RFC 2945-compliant application or standard to use these as alternatives. Other documents may adapt these methods for specific applications.

Section 2 describes background and rationale for this class of method, where they were developed, what they do, and why. Section 3 describes the SPEKE methods in detail, and section 4 discusses method selection, parameter selectio, and other application issues. Two potentially relevant patents are listed in Section 6.

2. Background

Convenient and secure personal authentication is important for many Internet applications. People want their passwords to be easy to remember and type, and also secure. But most earlier password systems offer insufficient protection against passive and active network attack on passwords. While complex public key based infrastructures have evolved, they often still rely on a password as

the human link.

Some older systems fail by treating passwords as cryptographic keys, without any real ability to insure appropriate cryptographic quality for these values. As such, they provide the option for either security or convenience, but not both at the same time. Purely hash-based "challenge-response" techniques as described in [RFC2095] and [RFC1760], and various forms of Kerberos initial authentication [RFC1510], were designed to defeat simple sniffing attacks, but unfortunately use the password directly as a message authentication key, and can thus be compromised by exhaustive guessing or dictionary attack by one who obtains protocol messages [BM90]. Passwords often need to be simple, easy-to-remember, and easy-to-type, which means that they cannot safely be used like traditional encryption keys.

Still other systems try to protect transmitted passwords by relying on a separate security infrastructure. The password-thru-SSL approach as commonly used in web browsing is vulnerable to spoofing attacks. These are not the result of weak cryptography, but rather, due to the generally weak concept of giving away a password to prove knowledge of it. In so-called "tunnelled" protocols, the password may flow through a authenticated channel in which the sole act of server authentication is ignored or bypassed by a typical user. When people must authenticate the server, but don't, their passwords are at risk.

Old methods for proving knowledge of a secret force disclosure of the secret from the prover to the verifier during each act. Such practices should be discouraged where non-disclosing zero-knowledge methods are available.

Strong password methods, including SPEKE, were first discovered in the 1990's. They use ephemeral public key techniques to securely bind people to keys, using a small or not-so-random password, and perform a zero-knowledge password proof (ZKPP), where each party proves to the other that it knows the password without revealing it to any party that doesn't already know it. These methods also negotiate a mutually authenticated session key between the parties based on knowledge of the password. This can happen over an unprotected network, without revealing the password or keys to attackers, requiring no certificates, no long-term sensitive data at the client, and minimal demands on the user.

These benefits are attractive for providing secure and convenient personal authentication. While these methods work with passwords alone, they are also used to provide secure mobility and credential provisioning in systems that use additional secret key or public key infrastructure. Security is improved by reducing assumptions on proper user behavior and reducing dependencies on security infrastructure.

SPEKE and other zero-knowledge password methods have been deployed in both academic and commercial settings. SRP has been described in [RFC2944] and [RFC2945], and referenced in several Internet Drafts. These methods are also described in the proposed IEEE standard for password-based public key cryptography [P1363.2].

2.1 Summary of Features and Benefits

SPEKE, B-SPEKE and W-SPEKE all perform mutual authentication and key agreement across an untrusted network while protecting passwords and negotiated authenticated keys. These methods do not send any passwords or password-crackable data over the network; Instead they integrate the password into a standard Diffie-Hellman exchange in a way that negotiates a mutually authenticated key.

Even with a password that is too small to serve as a cryptographic key, these methods prevent passive and active network attacks (man-in-the-middle, replay, etc.), as well as password-sniffing and unconstrained brute force attack from the network. These methods surpass the requirements in [RFC1704] for non-disclosing authentication protocols. Because both acts of user and server authentication are based on a common credential, and a step that the user can't forget to do, these methods are superior to tunnelled methods in common applications. ZKPP techniques prevent unnecessary password disclosures whether or not the authentication succeeds, defending against both simple accidents (typing the wrong password)

and malicious attacks (interception by a spoofed or compromised server).

The methods are as efficient as a Diffie-Hellman key exchange (DH) computation, using any standard groups. The incorporation of DH provides the benefits of forward secrecy and so-called "backward secrecy", which are important for robust password changing protocols

[Jas96] and for strong key management in general.

These methods include both balanced and augmented password-based key agreement methods. SPEKE is a balanced method, in that both parties share identical password-derived data. B-SPEKE and W-SPEKE are augmented methods, typically used in client/server applications, in which a server stores password verification data derived from a one-way function of the client's password. A thief that steals augmented password verification data from a server cannot use it to pose as the user in the protocol, without first "cracking" the password. W-SPEKE is comparable to SRP, with differences in structure, benefits, and limitations as described in section 4.2 and Appendix A.

Two of these methods are compatible with standard Diffie-Hellman as described in [IEEE 1363] and [ANSI X9.42], and are also aligned with the emerging IEEE [P1363.2] standard for password-based cryptography (see section 4.5). Implementation requires little more than a hash function and big integer modular exponentiation. Use of alternative settings, such as elliptic curve groups [ANSI X9.63], is beyond the scope of this document.

Methods in this class are particularly valuable for bootstrapping applications, where an initial act of personal authentication authorizes the download, transmission, or remote verification of private keys, certificates, and other sensitive configuration data. Personal key retrieval, so-called roaming applications, and general provisioning of keys are all ideal applications in both wired and wireless settings. Continued proliferation of ad-hoc password systems still remains a large problem on the Internet -- SPEKE and other zero-knowledge password methods provide an appropriate foundation for future consolidation and replacement of such ad-hoc systems.

2.2 Origin

SPEKE stands for "Simple Password-authenticated Exponential Key Exchange", and was first published in [Jab96]. It has been reviewed, cited, and/or studied in many subsequent papers including [Jab97] and [MacK01b], and has been commercially deployed. SPEKE is a "balanced" method, where two parties share an identical password-derived value that is used to derive and mutually authenticate a Diffie-Hellman key.

B-SPEKE, first published in [Jab97], is an augmented method, where the server's password verification data cannot be used directly by the client.

W-SPEKE is an augmented method that has similarities to both B-SPEKE and SRP [Wu98]. W-SPEKE was described as "SRP-4" in an earlier Internet draft and a submission to IEEE P1363.

A brief comparison of some of these methods is in section 4.3, and lists of research papers are available in [P1363.2] and [ZKPPLinks].

3. Description

This section describes the SPEKE, B-SPEKE, and W-SPEKE password-authenticated key agreement methods. It begins with preliminary notation, conventions, and terminology in subsection 3.1, followed by a combined description of all three methods in 3.2.

3.1. Notation, Conventions, and Terminology

This section describes the system parameters, notation, and data format conversion functions used in these methods. The parameter notation is similar to RFC 2945, and the function names are aligned with IEEE 1363.

3.1.1 Parameters

The methods described here use multiplicative groups of integers, parameterized with the following values and functions:

N	field modulus, a suitable 1024-bit prime of the form kq + 1
q	prime order of the subgroup to be used, larger than 2^159
k	the "co-factor", k=N-1/q. It is recommended that all divisors of k/2 other than 1 be larger than q.
hash	SHA-1, or an alternate suitable hash function
kdf	KDF1-SHA1, as defined in [IEEE 1363], or an alternate suitable key derivation function

Selection of suitable values is discussed in section 4.4.

3.1.2 Notation

In these descriptions, the | symbol denotes concatenation of byte strings. All hash functions operate on and produce byte strings.

The * and + operators denote integer multiplication and addition. The ^ and % operators denote integer exponentiation and the integer modular remainder operation.

The / operator denotes integer division.

The := operator denotes either a definition of a term or the function of assigning a value to a variable.

Note also that variables may be bracketed with '<' and '>' in the text for readability, but the brackets are often omitted in equations; Unambiguous uses of x and <x> refer to the same thing.

"DH" stands for "Diffie-Hellman". "DL" refers to the so-called discrete logarithm setting that uses a multiplicative group of integers. ("EC" would refer to the elliptic curve setting.)

SHA1() refers to the SHA-1 Secure Hash Algorithm, as defined in [SHA1]. <hash> is a hash function used to process passwords. <kdf> is a key derivation function used to derive authenticated keys.

SHA1 may be used for both <hash> and <kdf>, although other suitable standard hash and key derivation functions are acceptable.

3.1.3 Data Formats and Conversion

The functions in this description are defined to operate on either byte strings or on unsigned big integers, with an implied standard conversion between these types of values. The necessary conversion functions are specified in this section.

The conversion functions used are OS2IP and FE2OSP as defined in [IEEE 1363]. OS2IP converts a byte string (or "octet string") into a non-negative integer, where the first byte represents the most significant 8 bits, presuming unsigned big-endian format. FE2OSP converts a field element into a big-endian byte string, where high zero bytes are added as needed to make the length equal to the number of significant bytes in the field modulus N. A field element is a big integer in the range [0,N-1], with no sign bit used in the encoding.

For example, the computation "hash(x)^k % N" implies that the big integer value x is converted to a byte string before being hashed, and the hash output is converted from a byte string to a big integer prior to the modular exponentiation. Thus, a long way to describe this step is "OS2FEP(hash(FE2OSP(x)))^k % N".

3.1.4 Bits and Bytes

Bit strings in this document (such as the input and output of hash functions) are always a multiple of 8 bits in length, and are treated as byte strings. The high bit of a bit string is treated as the high-bit of the first byte of the corresponding byte string. Thus, the input to SHA1 is a byte string of arbitrary length less than 2^56 bytes, and the 160-bit output is treated as a big-endian 20-byte string.

3.2 General Construction

This section describes the general construction of SPEKE, B-SPEKE and W-SPEKE.

When used in a client/server application, the authentication server must have access to a password repository (database or file) which associates a username or similar identifier with password verification data, and, optionally, a salt value.

For SPEKE, password verification data can be any arbitrary byte string (designated as <x>) that preserves the randomness of the password and is available to both client and server.

The parties must agree on which method to use, including domain parameters, hash, key derivation, and key confirmation functions. These values may be fixed in the implementation of the client and server. The parties must also initially agree on a password, and how they do that is beyond the scope of this document.

3.2.1 Enrollment

During enrollment of the user's credentials with a server, the password value x is hashed and converted to a value <g> that serves as password verification data. <g> is in the range [1,N-1]. For B-SPEKE and W-SPEKE, the server's password verification data is a specially constructed pair of values {<g>,<v>}.

```
SPEKE server stores:           { <username>, <g> [, <salt>] }
B-SPEKE/W-SPEKE server stores: { <username>, <g>, <v> [, <salt>] }
```

The value <g> is of order <q>. <v> is a standard DH public key corresponding to a base <g> raised to the power of password-derived DH private key <x>. These methods also require that <x> cannot be derivable from <g>, which is enforced using a hash function to compute <g> from <x>. These values are defined as follows:

```
p := the password or other potentially small authenticator string
x := any suitable derivation of <p> that preserves its randomness
g := REDP(x), "REDP" is a random element derivation function
v := g^x % N        (only used in B-SPEKE and W-SPEKE)
```

Two suitable REDP functions [P1363.2] are:

```
REDP-1(x) = hash(x)^k % N
REDP-2(x) = (ga * gb^hash(x)) % N
       ga = REDP-1("Alice")
       gb = REDP-2("Bob")
```

<ga> and <gb> are two arbitrary elements of order q that have no known exponential relationship to each other.

3.2.2 Identification

To start the login process, the client typically sends a user name or identifier to the server, after which the server retrieves the stored

value of <g>, and optionally <v> and/or <salt> values that have been associated with the user's password. The server sends back any optional <salt>.

```
      Client                                          Server

      <username>  ---------------->
                          get {g [,v] [,salt]} from storage
                             <--------------------  [<salt>]
```

3.2.3 Key Exchange

The client computes <x> and <g> from the user's password and (optionally) the server-supplied <salt> or other parameters. Alternately, in a peer-to-peer application using SPEKE, both parties may compute <g> directly from the shared password.

The client generates a secret random number <a> in the range [1,q-1], computes the remainder of raising <g> to the power <a> modulo the field prime <N>, and sends the result to the server.

The server generates a secret random number <b> in the range [1,q-1], computes the remainder of raising <g> to the power <b> modulo the field prime <N>, and sends the result to the client.

```
      Client                                          Server

      a := random integer
      A := g^a % N
      A  ------------------------>
                                          b := random integer
                                                 B := g^b % N
                              <--------------------------  B
```

In the DL setting where <k> equals 2 or 2r and where all divisors of r other than 1 are greater than q, the value of <A> or <B> is defined to be unacceptable if it is not in the range [2,<N>-2] inclusive. When <B> is an unacceptable value, the client must abort before revealing any information derived from <B>. When <A> is an unacceptable value, the server must abort before revealing any information derived from <A>.

In general, for any DH group, acceptable values of <A> and <B> are those that generate a suitably large set of possible values for <S>, to preclude enumeration of the possible results by an attacker.

3.2.4 Secret Value Derivation

When a received value is valid, each party computes a secret byte string <S> based on the appropriate computation as shown here:

```
      Client                                          Server
```



```
   For SPEKE:
         S := B^a % N                                S := A^b % N

   For BSPEKE:
         S := (B^a % N) | (B^x % N)      S := (A^b % N) | (v^b % N)
```

For WSPEKE:

```
      u := hash(B) / (2^128)              u := hash(B) / (2^128)
      S := B^(a + u*x) % N                  S := (A * v^u)^b % N
```

If the client has used the password that corresponds to the server's verification data, authentication is successful and both parties will share the same value for <S>.

3.2.5 Key Derivation

At this point, both parties can derive a set of one or more keys { K_1, K_2, ... } from the (hopefully) shared secret <S> using an appropriate key derivation function. Any set of prefix-free distinct derivation parameters can be used to derive a set of distinct derived keys. (See security note in 4.6.1.)

```
      K_i := kdf(S, <key derivation parameter i>)
```

Note that none of these keys <K_i> can be determined by any third-party observer of <B> and <A>, and no second party can negotiate a matching <S> without using the correct password data. Derived keys are useful for various purposes, such as for proving knowledge of <S> in key confirmation, and for use in generating secure session keys.

3.2.6 Key Confirmation

To complete the act of explicit mutual authentication, both parties prove to each other that their shared secret <S> values are identical. They can use a key confirmation function (KCF) to derive a unique parameterized key from <S>, send the KCF output key to the other party, after which each party verifies the other's correct computation.

One specific construction compatible with [P1363.2] also incorporates other shared secret and public values as shown here. The client and server agree that the client will send KCF(<kcfParam1>), and the server will send KCF(kcfParam2).

```
      Client                                          Server

      K_1 := kcf(<kcfParam1>)        K_1 := kcf(<kcfParam1>)
      K_1 ---------------------->                verify K_1

      K_2 := kcf(<kcfParam2>)        K_2 := kcf(<kcfParam2>)
      verify K_2              <---------------------- K_2
```

```
<kcfParam1> := hex byte 04
<kcfParam2> := hex byte 03
<kcf> := KCF1-SHA1

KCF1-SHA1(<kcfParam>) := SHA1(<kcfParam> | A | B | S | g)
```

The server computes its value for K_1 as a hash of its concatentated values for <kcfParam1>, A, B, S, and g, and then it compares its K_1 value to the value of K_1 received from the client. If they are not equal, the server must abort and signal an error. The client performs the analogous procedure for verifying the server's K_2.

3.3 Message Ordering

The message flows in these key agreement protocols do not necessarily have a one-to-one correspondence with application protocol messages. There are no special security constraints on message flows, so that they may be generally combined and arranged in any order that permits necessary computation and interoperability.

3.4 Compatibility with RFC 2945

In RFC 2945, <g> is of order (N-1), but here <g> is of order <q>.

For close alignment with RFC 2945, one can use the following specific definitions:

```
p        := the raw password byte string
username := byte string identifying the user
salt     := a random byte string stored on the server
x        := SHA1( <salt> | SHA1( <username> | ":" | <p> ) )
```

The value <u> is a 32-bit unsigned integer equal to the high-order 32 bits of SHA1(B), and is compatible with RFC 2945.

The "Interleaved SHA1" key derivation function in RFC 2945 is not compatible with KDF1-SHA1 or other KDFs defined in [IEEE 1363].

4. Method Selection and Application Notes

SPEKE, B-SPEKE, and W-SPEKE are three similar but distinct methods. The general benefits of these and other zero-knowledge password methods over earlier techniques is reviewed in section 2. This section compares these and some other methods in the same category, and discusses issues to consider in choosing a specific method,

selecting system parameters and primitive functions, and related application notes.

Flexibility may be desirable, as people have different opinions and relative priorities for efficiency, compatibility, security, and

licensing concerns in a variety of applications. This section discusses technical criteria for selecting one method over another. Business issues may also be relevant, but are beyond the scope of this document.

4.1 W-SPEKE, B-SPEKE, and SRP

W-SPEKE may be described as a variant of B-SPEKE that uses an optimized exponential computation from SRP. W-SPEKE may also be seen as a variant of SRP with the following differences:

* Derives generator <g> from password, instead of <g>=2.
* Removes the +v and -v steps associated with <B>.
* <g> is of order <q>, instead of order N-1.
* Host stores <g>, along with <username> and <v>.
* Modified test for unacceptable A and B.

The resulting limitations & benefits are:

SRP	W-SPEKE, B-SPEKE
Restrictions on message order	No restrictions on message order
Not aligned with IEEE/ANSI	May use IEEE/ANSI DH primitives
Two guesses per run [MacK01b]	One guess per run

B-SPEKE has somewhat increased computation over W-SPEKE and an added password verification value <g> stored on the server as compared to SRP.

See also Appendix A for a description of a form of W-SPEKE in the style of RFC 2945. Other changes between RFC 2945 and this document include additional background material and clarifications in presentation and safety recommendations.

SRP and W-SPEKE use an additional security parameter in the construction of <u>, that is not present B-SPEKE (or SPEKE), which may merit additional study for some applications.

4.2 Balanced vs. Augmented Methods

All of the SPEKE methods provide a strong basic level of protection against network attack for passwords and their authenticated keys. The augmented methods W-SPEKE and B-SPEKE provide the added benefit over SPEKE of permitting the host to store passwords in a form that is not directly useful to an attacker. In an augmented system, even if the host's password database is stolen, the thief still needs to perform a successful guessing attack to determine the password in order to login. In either case, however, servers are still required to maintain secure storage of password verification data. References that discuss relative value of augmented methods are [Wu98], [Jab97], and [PK99], the latter providing reasons why the augmented benefit may not be necessary for key-retrieval applications.

SPEKE is the simplest, most efficient, and by some measures the most widely studied of the three. These advantages warrant its use in applications that cannot obtain significant extra benefit from an augmented method. It may also be preferred for client/server applications where the server cannot control the format of password verification data -- for example, where the server has access to clear-text passwords or passwords transformed with a legacy one-way function.

4.3 B-SPEKE vs. W-SPEKE

W-SPEKE and B-SPEKE are both augmented methods that may provide an added benefit in client/server applications. W-SPEKE combines the benefits of B-SPEKE with the computational efficiency boost of an optimized computation similar to that used in SRP. A factor that might favor B-SPEKE over W-SPEKE is conformance with IEEE 1363 and ANSI DH standards.

4.4 Parameter Selection

This specification requires that the modulus <N> is, in general, any modulus that is suitable for a Diffie-Hellman key exchange.
A suitable <N> is prime, defines a suitably large subgroup of prime order q (for compliance with standard DH in [IEEE 1363] and [ANSI X9.42]), and allows group elements to be checked for membership in a small subgroup. If there are any divisors of <k>/2 other than 1, it

is recommended that they be larger than q.

The modulus must also be one in which the parties can trust has not been specially crafted to provide advantages for an attacker. Construction methods and examples of verifiable primes suitable for SPEKE are defined in [RFC2412] and in [FIPS 186-2] Appendix 2.

The random secret exponents <a> and <b> have length and quality constraints similar to any Diffie-Hellman exchange. A common acceleration trick for DH, that applies here too, is to choose <a> and <b> from a range [1,t], where <t> is much smaller than <q>, but still sufficiently large. The general rule is for t to have enough bits to avoid a brute-force attack of order 2^(t/2). A typical value is t = 2^160 to preclude known attacks using 2^80 operations. Such security issues are discussed in [IEEE 1363]. This document recommends that <a> and <b> each contain at least 160 random bits.

For the value <u> in W-SPEKE, implementors may choose to increase the length, as long as both client and server agree, but in general this document recommends that it not be shorter than 32 bits. This value limits a thief who steals <g> and <v> (but hasn't cracked the password) to having no more than a 1 in 2^32 chance of being successful in a single run with a lucky random guess.

4.5 Compatibility with Other Standards

Compatibility and alignment with other existing standards promotes re-use of implementation components. Even when strict compatibility cannot be achieved, re-use of standard settings, primitives, and related functions may still encourage implementation compatibility and help provide assurance of security.

These methods are aligned with the BPKAS-SPEKE, APKAS-BSPEKE, and APKAS-WSPEKE schemes as described in the IEEE proposed standard for password-based public key cryptography [P1363.2].

These methods are also compatible with the settings, schemes, and primitives defined for Diffie-Hellman in [IEEE 1363] and the related [ANSI X9.42] and [ANSI X9.63] equivalents. SPEKE, B-SPEKE and W-SPEKE in the DL and EC settings are compatible with the 1363 EC/DLSVDP-DH primitives, and SPEKE and B-SPEKE are furthermore

compatible with the EC/DLKAS-DH1 and EC/DLKAS-DH2 schemes.

This document further shows where W-SPEKE is aligned with RFC 2945 and where such alignment conflicts with other standards. It maximizes consistency with RFC 2945, and the various Internet Drafts and other documents that reference it, to facilitate "drop-in" replacement of these methods for SRP.

4.6 Other Key Derivation and Hash Functions

Standard DH1 and DH2 key agreement [IEEE 1363] specifies that the negotiated shared-secret field elements be converted to fixed-length byte strings using FE2OSP, and then processed through a standard key derivation function. Examples using KDF1-SHA1 are shown here:

```
SPEKE:  K_i = SHA1( FE2OSP(A^b % N) | <key derivation parameter> )
BSPEKE: K_i = SHA1( FE2OSP(A^b % N) | FE2OSP(v^b % N) |
                    <key derivation parameter> )
```

This document permits the key derivation function to be any suitable standard function, such as KDF1-SHA1, or other non-standard yet suitable functions, such as SHA_Interleave ([RFC2945] and Appendix A). Although SHA Interleave is not the preferred choice, we know of no security concern with it. However, although this function attempts to preserve up to 320 bits of entropy in <S>, we note that the effective entropy in <S> is also limited by the entropy in <a> and <b>.

Another well-studied alternative is the HMAC-SHA1 function [RFC2104] which is useful for deriving the keyed message authentication codes used in key confirmation.

Any of these methods can be used with suitable alternative hash functions other than SHA-1, such as SHA-256, RIPEMD, and HMAC constructions.

4.6.1 Security Note

To securely use any standard key derivation function, it is important that key derivation parameters be prefix-free, that is, no string should be a prefix substring of any other (as in "p1", "p2", ... "p10"). Using KDF1 with same-fixed-length strings or a length-

containing encoding such as ASN.1 BER or DER is suggested (see [IEEE 1363], D.5.1.4).

4.7 Key Confirmation Functions

Alternate key confirmation functions may be desired in different applications. For example RFC 2945 describes acceptable functions for key confirmation (also shown in Appendix A) that are not compatible with KCF1.

Note that the protocol messages prior to key confirmation, such as the user name, are not integrity-protected, and as such parties must not rely on these values for security-sensitive purposes. Including party identifiers in the key derivation parameter of a key confirmation message is one way to prevent identity confusion attacks.

Many application security protocols include all previously shared messages as key derivation parameters in a key confirmation messages, just to be sure.

4.8 Salt

The purpose of salt is to frustrate an attacker who may plan to build a generic dictionary of password verification data that corresponds to a set of likely passwords. Such a dictionary could assist a database thief who steals <g> or <v> in a mass-cracking attack. Salt forces such a dictionary to be built on a case-by-case basis.
To be effective, salt must be incorporated into both <g> and <v> in the augmented methods W-SPEKE and B-SPEKE.

Note that W-SPEKE incorporates salt in the computation of <A>, whereas in SRP3, salt is incorporated at a later stage. This may affect consolidated forms of a server-salted protocol (see Appendix A).

In applications where the client sends <A> before contacting the server, a server-supplied salt cannot be incorporated into <x>. In such applications, one might alternately use a self-salting technique, such as by incorporating the username or servername in <x>.

4.9 Iterated Hashing

Using an iterated hash function in combination with any of these protocols can increase the required cracking effort for stolen password verification data. For example, one may use:

```
<p> := hash(hash(... hash(<password>) ...))
     ... for perhaps thousands of iterations
```

However, in any case, security of the server password database remains a primary way to prevent unconstrained attack, whereas salt and iterated hashing are secondary backup defensive measures.

5. Security Considerations

Security considerations are discussed throughout this document, including parameter selection, relevant cryptographic research, and comparison of these to other methods.

6. Intellectual Property Notice

Phoenix Technologies Ltd. and Stanford University own patents that describe the SPEKE and SRP methods respectively. For more information, including contact information for resolving questions, readers are referred to the IPR statements available at http://www.ietf.org/ipr.html.

References

[ANSI X9.42] ANSI X9.42-2001, "Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography", ANSI, 2001.

[ANSI X9.63] ANSI X9.63-2001, "Public Key Cryptography for the Financial Services Industry, Key Agreement and Key Transport Using Elliptic Curve Cryptography", ANSI, 2001.

[BM90] Bellovin, S., and M. Merritt, "Limitations of the Kerberos Authentication System", ACM Computer Communications Review, October 1990.

[BM92] Bellovin, S., and M. Merritt, "Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks", Proceedings of the I.E.E.E. Symposium on Research in Security and Privacy, Oakland, May 1992.

[FIPS 186-2] FIPS 186-2, Digital Signature Standard, NIST.

[IEEE 1363] IEEE Std 1363-2000, "IEEE Standard Specifications for Public-Key Cryptography 2000", IEEE, 2000.

[Jab96] Jablon, D., "Strong Password-Only Authenticated Key Exchange", Computer Communication Review, ACM SIGCOMM, vol. 26, no. 5, pp. 5-26, October 1996.

[Jab97] Jablon, D., "Extended Password Key Exchange Protocols Immune to Dictionary Attacks", Proceedings of the Sixth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET-ICE '97), IEEE Computer Society, June 18-20, 1997, Cambridge, MA, pp. 248-255.

[Jas96] B. Jaspan, Dual-workfactor Encrypted Key Exchange: Efficiently Preventing Password Chaining and Dictionary

Attacks, Proceedings of the Sixth Annual USENIX Security Conference, July 1996, pp. 43-50.

[RFC1510] Kohl, J., and C. Neuman, "The Kerberos Network Authentication Service (V5)", RFC 1510, Digital Equipment Corporation, USC/Information Sciences Institute, September 1993.

[MacK01b] MacKenzie, P., "On the Security of the SPEKE Password-Authenticated Key Exchange Protocol", Cryptology ePrint Archive: Report 2001/057, <http://eprint.iacr.org/2001/057/>.

[P1363.2] IEEE P1363.2, "Standard Specifications for Public-Key Cryptography: Password-Based Techniques", (work in progress), IEEE, <http://grouper.ieee.org/groups/1363/>.

[PK99] Perlman, R. and C. Kaufman, "Secure Password-Based Protocol for Downloading a Private Key", Proceedings of the 1999 Network and Distributed System Security, February 3-5, 1999.

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[RFC1704] Haller, N. and R. Atkinson, "On Internet Authentication", RFC 1704, October 1994.

[RFC1760] Haller, N., "The S/Key One-Time Password System", RFC 1760, Feburary 1995.

[RFC2095] Klensin, J., Catoe, R. and P. Krumviede, "IMAP/POP AUTHorize Extension for Simple Challenge/Response", RFC 2095, January 1997.

[RFC2104] Krawczyk, H., Bellare, M. and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[RFC2944] T. Wu, "Telnet Authentication: SRP", RFC 2944, September

2000.

[RFC2945] T. Wu, "The SRP Authentication and Key Exchange System", RFC 2945, September 2000.

[SHA1] National Institute of Standards and Technology (NIST), "Announcing the Secure Hash Standard", FIPS 180-1, U.S. Department of Commerce, April 1995.

[Wu98] T. Wu, "The Secure Remote Password Protocol", In Proceedings of the 1998 Internet Society Symposium on Network and Distributed Systems Security, San Diego, CA, pp. 97-111.

[ZKPPLinks] "Research Papers on Strong Password Authentication", <http://www.integritysciences.com/links.html>.

Author's Address

David Jablon
Phoenix Technologies Ltd.
320 Norwood Park South
Norwood, MA 02062

EMail: david_jablon at phoenix.com

Full Copyright Statement

Acknowledgements

The author thanks Paul Funk, Nora Hanley, Jim Walker, and Tom Wu for their review and thoughtful comments on earlier drafts. This document includes significant material derived from RFC 2945. The author is also grateful for the encouragement and advice from Steve Bellovin and Michael Wiener during the initial refinement of these methods.

Appendix A: An APKAS-WSPEKE mechanism

This section describes a form of APKAS-WSPEKE employing SHA-1 to authenticate users and generate session keys. This text is revised from section 3 of RFC 2945 (SRP), and is suitable for "diff"ing.

The host stores user passwords as quartets of the form

```
   { <username>, <password generator>, <password verifier>, <salt> }
```

Password entries are generated as follows:

```
   <salt> = random()
   x = SHA(<salt> | SHA(<username> | ":" | <raw password>))
   <password generator> = g = SHA(x)^2 % N
   <password verifier> = v = g^x % N
```

The | symbol indicates string concatenation, the ^ operator is the exponentiation operation, and the % operator is the integer remainder operation. Most implementations perform the exponentiation and remainder in a single stage to avoid generating unwieldy intermediate results. Note that the 160-bit output of SHA is implicitly converted to an integer before it is operated upon.

Authentication is generally initiated by the client.

```
   Client                              Host
  --------                            ------
   U = <username>              -->
                               <--    s = <salt from passwd file>
```

Upon identifying himself to the host, the client will receive the salt stored on the host under his username.

```
   a = random()
   x = SHA(s | SHA(U | ":" | p))
   g = SHA(x)^2 % N
   A = g^a % N                 -->
                                      g = <stored password generator>
                                      v = <stored password verifier>
                                      b = random()
                               <--    B = g^b % N
   p = <raw password>

   S = B ^ (a + u * x) % N            S = (A * v^u) ^ b % N
   K = SHA_Interleave(S)              K = SHA_Interleave(S)
   (this function is described
    in section A.1)
```

The client generates a random number a, computes x and g from the password, raises g to the power of random number a and reduces it

modulo the field prime, and sends the result to the host. The host generates a random number b, raises g to the power of random number b and reduces it modulo the field prime, and sends the result to the client. Both sides then construct the shared session key based on the respective formulae.

The parameter u is a 32-bit unsigned integer which takes its value from the first 32 bits of the SHA1 hash of B, MSB first.

The client MUST abort authentication if B is less than 2 or greater than N-2.

The host MUST abort the authentication attempt if A is less than 2 or greater than N-2.

At this point, the client and server should have a common session key that is secure (i.e. not known to an outside party). To finish authentication, they must prove to each other that their keys are identical.

```
   M = H(H(N) XOR H(g) | H(U) | s | A | B | K)
                                -->
                                <--    H(A | M | K)
```

The server will calculate M using its own K and compare it against the client's response. If they do not match, the server MUST abort and signal an error before it attempts to answer the client's challenge.

If the server receives a correct response, it issues its own proof to the client. The client will compute the expected response using its own K to verify the authenticity of the server. If the client responded correctly, the server MUST respond with its hash value.

The transactions in this protocol description do not necessarily have a one-to-one correspondence with actual protocol messages. This description is only intended to illustrate the relationships between the different parameters and how they are computed. It is possible, for example, for an implementation of the APKAS-WSPEKE-SHA1 mechanism to consolidate some of the flows as follows:

```
   Client                              Host
  --------                            ------
   U                           -->
                               <--    s, B
   A, H(H(N) XOR H(g) | H(U) | s | A | B | K)
                               -->
```

```
                                <--    H(A | M | K)
```

(Note: In RFC 2945, A is sent along with U. This consolidated W-SPEKE protocol sends A after receiving s, as A is derived from s.)

The value of N used in this protocol must be agreed upon by the two parties in question. It can be set in advance, or the host can supply it to the client. In the latter case, the host should send N in the first message along with the salt. For maximum security, N should be a safe prime (i.e. a number of the form N = 2q + 1, where q is also prime), chosen in a random manner, and the client must have a means of assuring the suitability of N. Also, note that g is a generator of the group of order q, which means that for any element X in the group of order q, there exists a value x in the range [1,q] for which g^x % N == X.

A.1. Interleaved SHA

The SHA_Interleave function used in WSPEKE-SHA1 is used to generate a session key that is twice as long as the 160-bit output of SHA1. To compute this function, remove all leading zero bytes from the input. If the length of the resulting string is odd, also remove the first byte. Call the resulting string T. Extract the even-numbered bytes into a string E and the odd-numbered bytes into a string F, i.e.

```
E = T[0] | T[2] | T[4] | ...
F = T[1] | T[3] | T[5] | ...
```

Both E and F should be exactly half the length of T. Hash each one with regular SHA1, i.e.

```
G = SHA(E)
H = SHA(F)
```

Interleave the two hashes back together to form the output, i.e.

```
result = G[0] | H[0] | G[1] | H[1] | ... | G[19] | H[19]
```

The result will be 40 bytes (320 bits) long.

A.2. Other Hash Algorithms

W-SPEKE can be used with hash functions other than SHA. If the hash function produces an output of a different length than SHA (20 bytes), it may change the length of some of the messages in the protocol, but the fundamental operation will be unaffected.

Earlier versions of the SRP mechanism used the MD5 hash function, described in [RFC 1321]. Keyed hash transforms are also recommended for use with SRP; one possible construction uses HMAC [RFC 2104], using K to key the hash in each direction instead of concatenating it with the other parameters.

Any hash function used with SRP should produce an output of at least 16 bytes and have the property that small changes in the input cause significant nonlinear changes in the output. [Wu98] covers these issues in more depth.

Appendix B: Extended FIPS 186-2 Prime Generation and Verification

This section describes an extension of the method to generate "kosherized" primes for DSA, as described in [FIPS186-2]. The FIPS method could be used to generate primes of the form p=2qr+1, but it was limited to 1024 bit p with 160 bit subgroup order q. This extended method can generate larger p's and q's, and supports explicit options for requiring that r be prime, or r = 1, which are all useful for Diffie-Hellman, SPEKE, and related methods. The option for prime r is fully compatible with the standard in that all primes p,q of suitable size generated by this method can be verified by any compliant implementation of FIPS 186-2.

B.1 Extended Method for Generation of Primes

This method generates primes, p and q, satisfying the following three or four conditions:

a. 2^(M-1) < q < 2^M for a specified M (e.g. M = 224)

b. 2^(L-1) < p < 2^L for a specified L (e.g. L = 2048)

c. q divides p - 1.

d. (optionally) r is prime, where r = (p-1)/(2q).

This method is a compatible extension of the DSA prime generation method specified in FIPS 186-2 Appendix 2.2. It is extended to generate values for q that are larger than 160 bits, to generate primes where p = 2q+1, and with the option to require that r is prime.

This prime generation scheme starts by using the SHA-1 and a user supplied SEED to construct a prime, q, in the range 2^(M-1) < q< 2^M. Once this is accomplished, the same SEED value is used to construct an X in the range 2^(L-1) < X < 2^L. The prime, p, is then formed by rounding X to a number congruent to 1 mod 2q as described below.

An integer x in the range 0 <= x < 2g may be converted to a g-long sequence of bits by using its binary expansion as shown below:

```
x = x[1]*2^(g-1) + x[2]*2^(g-2) + ... + x[g-1]*2 + x[g]
     -> { x[1],...,x[g] }.
```

Conversely, a g-long sequence of bits { x1,...,xg } is converted to an integer by the rule

```
{ x[1],...,x[g] } -> x[1]*2^(g-1) + x[2]*2^(g-2) + ... +
    x[g-1]*2 + x[g].
```


Note that the first bit of a sequence corresponds to the most significant bit of the corresponding integer and the last bit to the least significant bit.

Let L - 1 = n*M + b, where both b and n are integers and 0 <= b < M.

Step 1. Choose an arbitrary sequence of at least 160 bits and call it SEED. Let g be the length of SEED in bits.

Step 2. Let z = (M+159) DIV 160. DIV is defined as integer division. For i = 0,...,z-1 compute

U[i] = SHA-1[SEED] XOR SHA-1[(SEED+1+i) mod 2^g].

Step 3. Let U be the integer

U = (U[0] + U[1]*2^160 + ... + U[z-1]*(z-1)2^160) mod 2^M.

Form q from U by setting the most significant bit(the 2^(M-1) bit) and the least significant bit to 1. In terms of boolean operations, q = U OR 2^(M-1) OR 1. Note that 2^(M-1) < q < 2^M.

Step 4. Use a robust primality testing algorithm to test whether q is prime [footnote 1]. If M = L-1, let p = 2q+1 and test whether p is prime.

Step 5. If q is not prime, go to step 1. If M = L-1, go to step 1 if p is not prime and go to step 15 if p is prime.

Step 6. Let counter = 0. Let offset = 1 + z.

Step 7. For k = 0,...,n let

V[k] = SHA-1[(SEED + offset + k) mod 2^g].

Step 8. Let W be the integer

W = V[0] + V[1]*2^160 + ... + V[n-1]*2^((n-1)*160) + (V[n] mod 2^b) * 2^(n*160)

and let X = W + 2^(L-1). Note that 0 <= W < 2^(L-1) and hence 2^(L-1) <= X < 2^L.

Step 9. Let c = X mod 2q and set p = X - (c - 1) and, if r must be prime, let r = X DIV 2q. Note that p is congruent to 1 mod 2q.

Step 10. If p < 2^(L-1), then go to step 13.

Step 11. Perform a robust primality test on p, and if r must be prime, perform a robust primality test on r.

Step 12. If p passes the test performed in step 11, and if r must

be prime and r passes the test, go to step 15. If p passes the test and r must be prime, but r fails the test, go to step 1.

Step 13. Let counter = counter + 1 and offset = offset + n + 1.

Step 14. If counter <= 2^12 = 4096 go to step 1, otherwise (i.e. if counter < 4096) go to step 7.

Step 15. Save the value of SEED and the value of counter for use in certifying the proper generation of p and q.

B.2 Method for Verification of Primes

FIPS 186-2 does not explicitly describe specific steps for verify that p and q have been generated properly. There are two somewhat obvious ways this might be done, with one being faster than the other. Both methods are described here, and both may be used to verify primes generated with either FIPS 186-2 or the extended method.

B.2.1 Fast method

Input: SEED, counter, p, q, and, if r must be prime, r.

Perform the generation method described in FIPS 186-2 Appendix 2.2 or the extended method in A.1 above as appropriate with the following inputs and changes:

Set L = the size of p, and (if extended) set M = the size of q and specify whether r must be prime.

In Step 1, set the "arbitrary sequence" to the SEED value to be verified.

Instead of Step 6, set counter equal to the counter value to be verified, and set offset = 1+z + counter*(n+1). (Let z = 1 when not using the extended method.)

After Step 11, stop.

If the Step 11 tests for the values of p, q, and (optionally) r have all passed, and these values are the same as their corresponding input values, the values are verified. Otherwise verification has failed.

B.2.2 Slow method

Note: This method may be very slow, by performing a lot of unnecessary searching and testing of irrelevant values, particularly in case of failure. It is described primarily to show how to perform verification using an implementation of the generation method that does not allow one to specify an initial counter.

Input: SEED, counter, p, q, and, if r must be prime, r

Perform the generation method described in FIPS 186-2 Appendix 2.2 or the extended method in A.1 above as appropriate with the following inputs:

Set L = the size of p, and (if extended) set M = the size of q and specify whether r must be prime.

In Step 1, set the "arbitrary sequence" to the SEED value to be verified.

Compare the resulting values of SEED, counter, p, q, and (optionally) r to the input values. If these values are the same, the values are verified. Otherwise verification has failed.