# Secure Password-Based Protocol for Downloading a Private Key

*Radia Perlman*
Sun Microsystems Laboratories
2 Elizabeth Drive
Chelmsford, MA 01824
radia.perlman@sun.com

*Charlie Kaufman*
Iris Associates
5 Technology Park Drive
Westford, MA 01886
charlie_kaufman@iris.com

## Abstract

We present protocols that allow a user Alice, knowing only her name and password, and not carrying a smart card, to "log in to the network" from a "generic" workstation, i.e., one that has all the necessary software installed, but none of the configuration information usually assumed to be known a priori in a security scheme, such as Alice's public and private keys, her certificate, and the public keys of one or more CAs. By "logging in", we mean the workstation retrieves this information on behalf of the user. This would be straightforward if Alice had a cryptographically strong password. We propose protocols that are secure even if Alice's password is guessable. We concentrate on the initial retrieval of Alice's private key from some server Bob on the network. We discuss various protocols for doing this that avoid off-line password guessing attacks by someone eavesdropping or impersonating Alice or Bob. We discuss auditable vs. unauditable on-line attacks, and present protocols that allow Bob to be stateless, avoid denial-of-service attacks, allow for salt, and are minimal in computation and number of messages.

## 1. Introduction

This paper addresses a very specific but common problem. If a group of users share a pool of workstations and want to be able to walk up to any one of them and "log in", there must be a way for the workstation to retrieve the user's environment from some repository on the network. In the timesharing world, this was easy because all of the user's state was held on the central system and the workstations were all effectively identical. Reproducing this simplicity in today's environment involves a number of challenges: making the workstations appear effectively identical by having enough commonality in installed software to make the environment at least familiar to the user; downloading the custom aspects of the user's environment from the network; finding the user's environment on the network; and doing all of this securely. This paper addresses only the last of these problems.

In particular, we assume that Alice comes to the workstation with only her user name and password (and possibly the name of a server holding more information about her). We assume that the workstation is installed with trustworthy software and without trojan horses that might steal her password or otherwise misbehave (a challenge not addressed by this paper), but that the workstation has no configuration information specific to Alice. We also assume the network to which the workstation is connected is completely untrustworthy, and Alice's password was chosen by her and could probably be guessed if an attacker could try a large number of values.

We assume that there is a server we'll call Bob out on the network somewhere with configuration information about Alice including strong (unguessable) cryptographic credentials. We look at protocols whereby the workstation knowing Alice's password can download that configuration information. We would like to attain the following security properties:

- An eavesdropper on the conversation between Alice and Bob can't learn Alice's password, Alice's configuration information, or be able to verify a guess of Alice's password from the eavesdropped information.
- Someone we'll call Trudy impersonating Alice to Bob will not get any useful information from Bob unless she successfully guesses Alice's password, and Bob will know if she makes a wrong guess and how many wrong guesses she made.
- Someone we'll call Ted impersonating Bob to Alice will not get any useful information from Alice unless he successfully guesses Alice's password, and Alice will know if he makes a wrong guess and how many wrong guesses he made.
- Even if Ted can read Bob's database and impersonate Bob to Alice, he can't learn Alice's password or trick Alice into accepting false configuration information.
- Someone who gets in the middle of a conversation between Alice and Bob cannot accomplish any more than Trudy and Ted.
- If Bob serves lots of users, someone who reads Bob's

database can't test a guessed password against multiple user accounts with less computational effort than testing it individually against each one.

There are some other security vulnerabilities that would be nice to avoid, but given the constraints are unavoidable. In particular:

- Someone who knows (or can guess) Alice's password can impersonate Alice to Bob and get Alice's configuration information, and can also impersonate Bob to Alice and trick Alice into accepting false configuration information.
- Someone who can read Bob's database can verify guesses of Alice's password in an unaudited fashion.
- Trudy can guess one password at a time, know whether she got it right, and Bob can't distinguish any one guess from the case where Alice mistyped.
- In each protocol, one side or the other discovers whether the other end is legitimate (i.e., agrees on the password) first. That end can be impersonated, and if the impersonator discovers the password guess they made is incorrect, cut off the connection so that the other party can't distinguish between an unsuccessful password guess and a network disconnect.

If Alice's password were cryptographically strong (i.e. we were not concerned about the possibility of an attacker doing a dictionary or exhaustive search attack), we could post Alice's security context in some public location encrypted using her password and allow it to be world readable. If Alice chooses a really easy to guess password, like her birthdate, then these protocols will not be secure because they cannot withstand on-line guessing. We are assuming moderately unguessable passwords. Some systems enforce policies such as password length. These policies continue to be a good idea, as annoying as they are to users, even with these protocols.

It is useful for Alice's security context, when decrypted with her password, to have some sort of checksum so that Alice can detect whether the information is correct. In some of our protocols if she were to mistype her password the only symptom would be that the decrypted "security context" would be garbage. By storing Alice's security context in such a fashion, she is immune from attacks even from someone who has compromised Bob, so long as her password is unguessable.

In section 2 ("Related Protocols") we discuss existing protocols that provide similar functionality, compare them to ours, and we review protocols that we need as a basis for our protocols.

As we will discuss in section 3.6 (Getting the Rest of the User's Security Context), if Alice can securely retrieve her public and private keys, it is straightforward for her to use this to download the rest of her security context. For most of this paper, we will talk only about protocols for downloading her private key. These protocols could be used, however, to download any sort of security context, including one using secret key based credentials.

## 2. Related Protocols

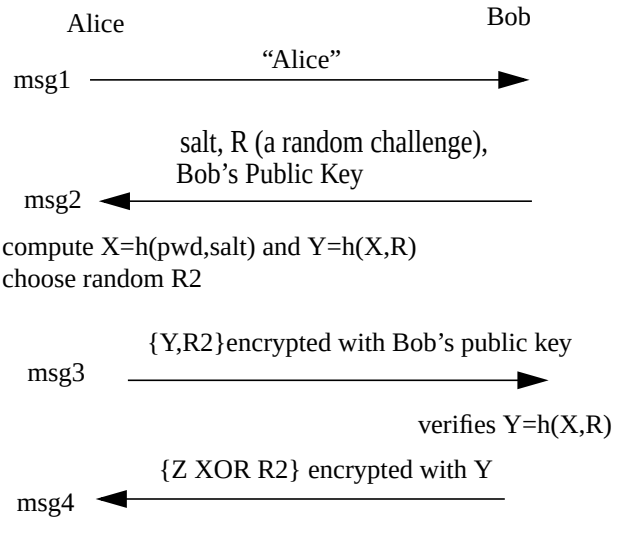
The protocol described in [LGSN89], a variant of which is implemented as part of NetWare Version 4 [LI94] satisfies some but not all of our security goals. Our protocol is based directly on EKE [BM92] and SPEKE [Jab96], with modifications to fit the needs of this specific problem and to improve performance.

### 2.1 NetWare Private Key Download Protocol

NetWare version 4 [LI94] has implemented a scheme for downloading an encrypted private key from a directory which is similar to a scheme published in [LGSN89]. We will describe a simplified version of those protocols that illustrates their strengths and weaknesses. We start with a client machine that has the necessary software installed, but no configured information for individual users, and a human that knows her name and password. We'll call the combination of the human and workstation "Alice" and the directory "Bob". Bob starts out with a database of users, and for each user it knows:

- user name
- X = h(pwd, salt); h is a cryptographic hash, "pwd" is the user's password
- salt; a user-specific value
- Z = user's private key encrypted with her password

Figure 1 Simplified NetWare 4 private key download scheme



At this point Alice can do the following:

- use Y to decrypt the returned value,
- XOR the result with R2 to obtain Z,
- use her password to decrypt Z, yielding her private key.

The main disadvantage of this protocol relative to the EKE/SPEKE family of protocols (see section 2.2), and the protocols we will present in this paper, is that it requires configuration of the workstation with Bob's public key. It isn't secure to just have Bob send it to Alice, and we'd like Alice to be able to log in from a completely generic workstation with no previous knowledge of the user or any specific server public keys.

If Alice just accepts the public key presented by Bob, the protocol is usable but less secure, because someone impersonating Bob could send their own public key to Alice, decrypt msg3, and then test any number of passwords by checking whether Y=h(h(pwd,salt),R). This threat may be acceptable in some situations.

In the protocol in Figure 1, an eavesdropper does not learn enough information to verify a guess of a password. Although Y is derived from Alice's password using information an eavesdropper can learn, and therefore an eavesdropper can guess a password and derive the resulting Y, the only thing the eavesdropper can do with Y is trial decryptions of the value transmitted in msg4 (Z XOR R2 encrypted with Y). But because R2 was randomly chosen and never exposed in the clear, the eavesdropper can't confirm or deny the guessed password because every value tried will produce random looking bytes. It is vital to the security of this protocol that when Z XOR R2 is encrypted, there is no recognizable padding or redundancy.

Since Bob sends no information based on Alice's password until Alice has proven knowledge of it in the third message, someone impersonating Alice must guess a single password for each execution of the protocol and Bob will detect failed attempts. All of these protocols assume Bob does something to limit guessing attempts, like alerting an administrator if there are large numbers of failures, locking out an account after some number of failures, or simply giving slow response time.

Someone impersonating Bob, say Ted, will learn no information that can be used to verify a guess of Alice's password because msg3 is encrypted under Bob's public key. (But as noted above, if Ted can trick Alice into using the wrong public key for Bob, and trick Alice into talking to Ted, then Ted will be able use what Alice sends in msg3 to do off-line, unauditable password guessing.)

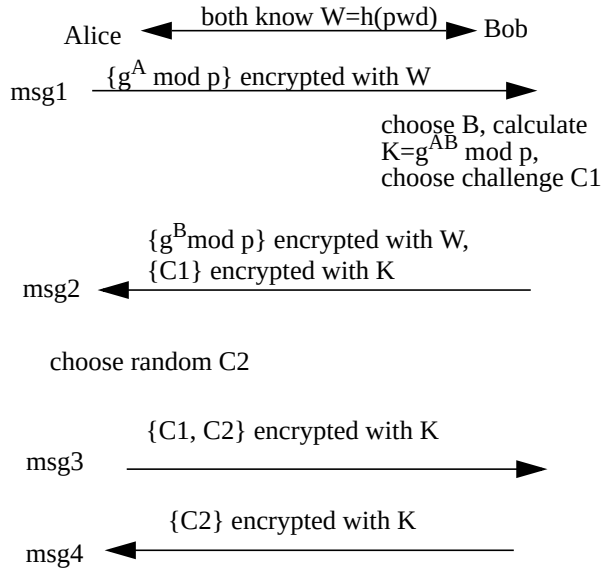
## 2.2 Review of EKE and SPEKE

Our protocols will use EKE or SPEKE, so we briefly describe them here.

EKE[BM92] and SPEKE[Jab96] are protocols in which Alice and Bob, who only share a weak, guessable secret (a user's password), can do mutual authentication and convert that weak secret into a strong secret that can be used for the remainder of the session without divulging information that would enable an eavesdropper to verify guesses of the weak secret.

[BM92] describes a family of protocols collectively known as EKE. In the variant we'll use, a weak secret is used to encrypt elements of a Diffie-Hellman exchange [DH76], winding up with the strong secret agreed upon by the Diffie-Hellman exchange. In detail, Alice and Bob share a weak secret W. Instead of transmitting $g^A$ mod p, Alice transmits {$g^A$ mod p} encrypted with W. In some variants of EKE, Bob transmits $g^B$ mod p unencrypted. In others he encrypts it with W (and similarly with Alice). The secret that Alice and Bob derive is $g^{AB}$ mod p.

Figure 2 EKE Protocol



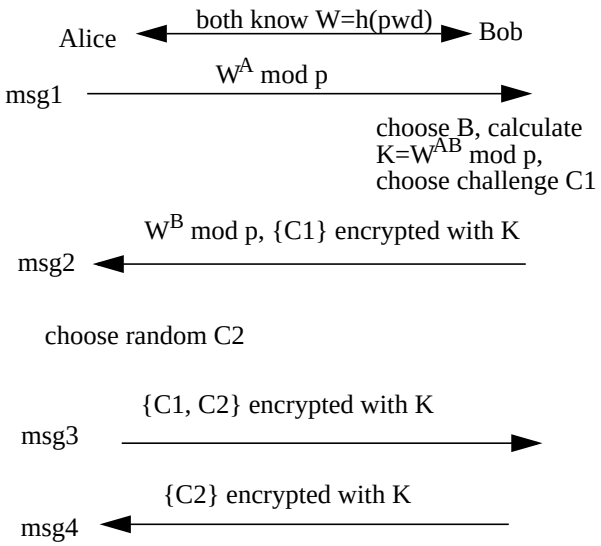

These protocols are designed for mutual authentication, and are described as 4-message protocols that agree on a strong session key K and do mutual authentication.

There are a number of ways for EKE to fail if not implemented carefully [Pat97]. What is particularly important is that when the Diffie-Hellman public numbers are encrypted, that someone can't use that to narrow down the password choices in an off-line attack. For instance, Dif-

fie-Hellman public numbers are always < p, so any password an attacker tried that decrypted to a value > p could be identified as wrong. Therefore, the quantity $\{g^A \bmod p\}$ encrypted with W could be used to reject a large number of candidate passwords, and the quantity $\{g^B \bmod p\}$ encrypted with W could be used to reject even more. Avoiding these attacks requires careful transformation of the Diffie-Hellman public numbers before encryption. Successful techniques are described in [Pat97] and [BM92].

SPEKE [Jab96] is similar to EKE, but instead of encrypting the Diffie-Hellman public numbers using W, it uses a secret generator derived as a function of W instead of a fixed g. The generator is still a function of the user's password, so we will call it W.

Figure 3 SPEKE Protocol




Either of these protocols could be enhanced to serve the function of downloading Alice's security context by including Alice's context encrypted under K in the last message. The resulting protocols would meet all our security goals except one. Anyone reading the server database could compute h(pwd) and then check it for a match against all user accounts. This could be avoided by "salting" the passwords, but this would add two more messages to the protocol. By salting, we mean having a different non-secret quantity associated with each user, having the user retrieve the salt from Bob, and using a secret key based on h(salt,pwd) instead of just h(pwd).

## 2.3 Augmented EKE, SPEKE, and Wu

The protocols described in [BM94], [Jab97], and [WU98] are related but more complex than EKE and SPEKE. They have the security advantage of not requiring storage at the server of a quantity that, if stolen, can be used to impersonate the user to that server. The quantity can be used for off-line password guessing, but it cannot directly be used to impersonate the user.

We will not use these protocols, because for our purposes there is no security advantage. The only thing someone gains by successfully impersonating Alice to Bob is the ability to read Alice's encrypted private key. Since someone who has read Bob's database already has that, there is no point in making it difficult for someone who has read Bob's database to impersonate Alice to Bob.

## 2.4 Choosing p and g

All of these protocols depend on Alice and Bob agreeing on a large prime p, and in the EKE derived protocols a generator g. We have assumed that workstations are not configured with any per-user or per-organization information, so how do we make this work? [BM92] suggests that the workstation could read the values of p and g from Bob and then evaluate their reasonableness. If someone impersonating Bob could trick Alice into using bad values of p and g, they could potentially get enough information to do off-line guesses of Alice's password. We believe this approach is a bad idea. First, it is computationally very expensive for the workstation to validate that p and g are "good". Further, if anyone - say through enormous computational effort or mathematical cleverness or both - could find a p and g over which they could efficiently compute discrete logarithms, they could impersonate any Bob to any workstation and gain the ability to guess passwords unaudited.

We believe that a better approach is to specify p and g as part of the protocol specification. Because this would be an attractive target for brute force attack, p should be chosen conservatively - at least 1024 bits and more likely 2048. If over time this value became weak, new values for p and g could be phased in by having multiple values configured in clients and servers, and by having the server be able to reject a connection based on an obsolete p and suggest new values that would be acceptable. The workstation would only accept new values that have been configured into the software.

# 3. Simple Password-Based Private Key Download Protocols

In this section we describe several new protocols for downloading the user's private key. These are variants on EKE and SPEKE trimmed down for better performance. We will give a series of protocols to show the evolution of

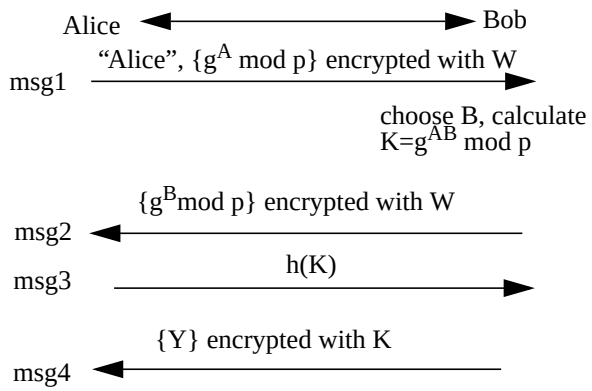
the design, and to examine the security and performance implications of each step in the evolution. The first protocol, in figure 4, is the most straightforward extension of EKE, but then in subsequent protocols we add performance enhancements, security against additional threats, and finally shorten it to two messages.

### 3.1 Four messages, no salt

The first protocol we present is closest in spirit to the EKE/SPEKE class of protocols. The workstation initially knows nothing. Alice types her password, and then the workstation can compute W=h(pwd). Bob has a database and knows, for each user:

- user name
- W=h(pwd)
- Y=user's private key encrypted with her password

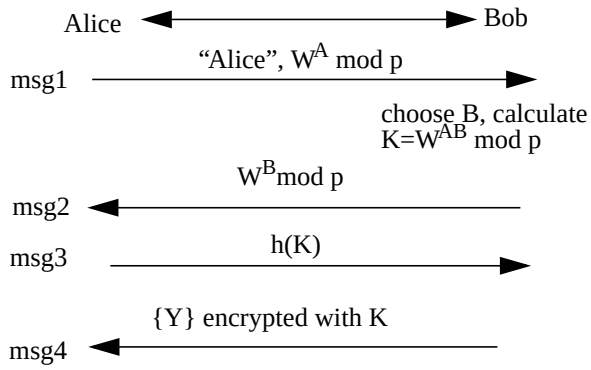
Figure 4 Basic 4-msg EKE-based




The protocol using SPEKE looks as follows. As before, Bob knows, for each user:

- user name
- W=h(pwd)
- Y=user's private key encrypted with her password

Figure 5 Basic 4-msg SPEKE-based




Note that one modification we have made is that we do not have Alice authenticate Bob. For our purposes (download of user's private key) it is unnecessary for Alice to authenticate Bob. All she cares about is that she receives a quantity that when decrypted with her password, yields her private key. It is true that someone who has stolen her password can trick her into using the wrong private key, but given that her password is her only means of authenticating Bob or her public key pair, this threat is unavoidable. So there is no security advantage to having Alice authenticate Bob. Therefore we can skip the fourth message of the EKE/SPEKE handshake and use the fourth message to download Y, encrypted with a strong session secret.

Another modification we've made is to dispense with Bob's challenge from the EKE and SPEKE protocols. It is sufficient to have Alice prove knowledge of K by sending h(K).

Note that in the protocols in this section, someone impersonating Bob can do a single unaudited on-line password guess. What that means is that if someone, say Ted, is impersonating Bob and guesses a single password for the user, he can verify his guess after msg3, by checking whether Alice returns the expected h(K). At this point Ted can stop responding and Alice cannot tell that she wasn't talking to the legitimate Bob. Ted has given no incorrect responses; he's just simply broken off communication, a situation unfortunately sufficiently common in the world of networks and servers that it might not raise suspicion. But Alice probably would become suspicious (or would at least give up on using the system) if it happened thousands of times.

Alice, on the other hand, cannot test even a single incorrect password without having Bob know she did not know the password, because in msg3 she will not have the cor-

rect response unless she chose the correct password. And she cannot break off communication earlier because until msg4 she has not obtained any useful information with which to verify even a single password guess.

Note that these protocols do not have the ability to use salt, because unless there is a message from Bob first to let Alice know what the salt is, she can't compute W (since the idea of salt is to require W to be a function of salt as well as the password). Therefore she can't compute what she needs to send in msg1 ($\{g^A \bmod p\}$ encrypted with W for the EKE-based protocol in Figure 4 and $W^A \bmod p$ for the SPEKE-based protocol in Figure 5).

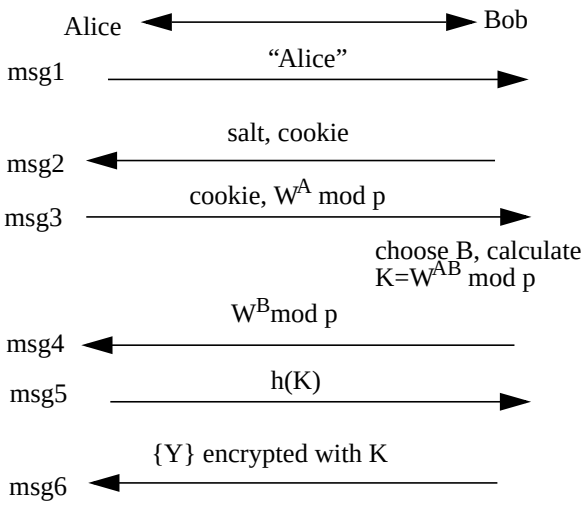
## 3.2 Adding Salt and a Cookie, 6 msgs

It is straightforward to add salt with an extra 2 messages prior to msg1 in which Alice requests, and Bob transmits, the salt.

Another feature we'd like to add is denial-of-service protection. In the protocols in figures 4 and 5, any requester can force Bob to perform an exponentiation. Since Bob is a server that can get arbitrary numbers of requests, there is a potential denial of service attack if anyone can force him to do something CPU-intensive. A cookie is a random number Bob sends and Alice returns, which proves that she can receive at the address she is claiming to come from. This is helpful against an attacker who sends lots of requests with forged source addresses to avoid capture.

The resulting protocol is:

Figure 6 Salt, cookie, SPEKE-based, 6 msgs




Obviously, to make it work for EKE, replace $W^x \bmod p$ with $\{g^x \bmod p\}$ encrypted with W.

## 3.3 Four messages, Saving Computation for Bob, Stateless Bob

We now make several improvements. We note that there is no security lost by having Bob always use the same B for a particular user, and precompute and store $\{g^B \bmod p\}$ encrypted with W for that user (in the case of EKE), or $W^B \bmod p$ (in the case of SPEKE). And it has the significant advantage of reducing computation for Bob.

Of course, it would be a disaster, in EKE, to use the same B with different users, because then someone knowing W1 for user 1 can decrypt $\{g^B \bmod p\}$ encrypted with W1 (sent to user 1) with W1 to obtain $g^B \bmod p$, and then test passwords against the quantity $\{g^B \bmod p\}$ encrypted with W2, (which will be transmitted on the wire when user 2 downloads his password).

If, in the case of EKE, we have Alice send $g^A \bmod p$ unencrypted, then Bob does not need to store W if he's storing $\{g^B \bmod p\}$ encrypted with W. Similarly in SPEKE, if we store $W^B \bmod p$ we no longer need to store W.

Note that in EKE, if Alice sent instead $\{g^A \bmod p\}$ encrypted with W, then Bob would need W in order to decrypt what Alice sends so that he can raise it to B. In the protocol in Figure 4, we had Alice send $\{g^A \bmod p\}$ encrypted with W, but in fact she could have sent $g^A \bmod p$ unencrypted. The general rule of thumb for these protocols is the person who proves knowledge of the password first does not need to encrypt the Diffie-Hellman value.

Now we note that since Bob is not storing W, but is rather storing a quantity dependent on B, which is unique for each user, we dispense with the need for salt! With what we're storing, someone that steals Bob's database and wants to check whether a particular password matches any user's account must do an exponentiation with each user's unique B in order to check that password for that user. So the modification we made for saving computation for Bob has the very nice side-effect of eliminating the need for salt.

The next modification is to make Bob stateless. In the protocol in figure 6, Bob had to remember what cookie/challenge he sent in msg2 in order to verify msg3. He also had to remember Alice's name. We can easily add Alice's name to msg3 in each case. We can also make it possible for Bob not to need to remember R.

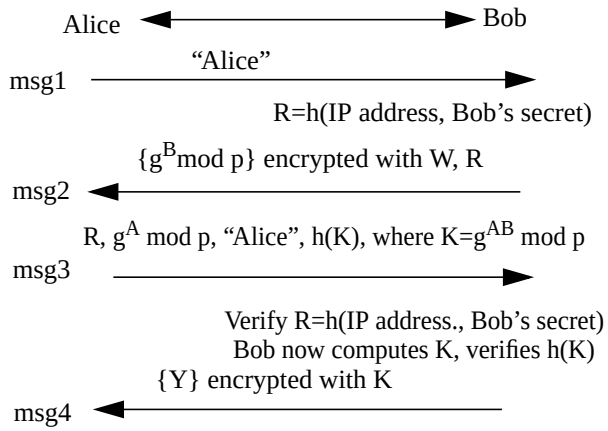
The way we do that is to have R be a function of Alice's IP address and a secret known only to Bob, e.g., R=h(IP address, Bob's server secret). Bob can change the secret fairly frequently (like every 5 minutes), and accept one of two values as a cookie from IP address x: the one based on Bob's current secret and the one from Bob's previous secret. That way Bob can be stateless, i.e., act in request/

response mode even though the protocols are 4 messages.

In the EKE-based protocol, Bob needs to store, for each user:

- user name
- Y=user's private key encrypted with her password
- B (the Alice-specific random number chosen by Bob when setting up Alice's account)
- $\{g^B \bmod p\}$ encrypted with W

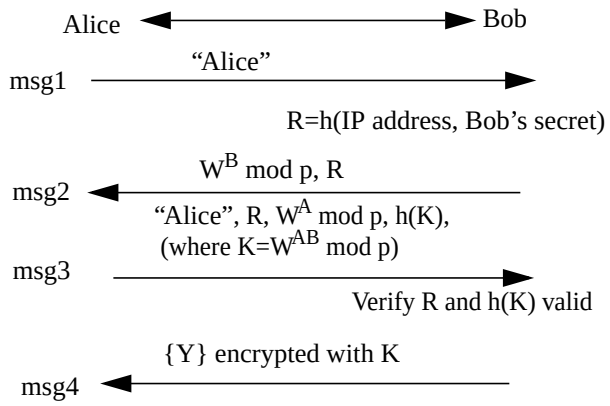
Figure 7 4-msgs, stateless Bob, precomputation, EKE-based




In the SPEKE-based protocol, Bob needs to store, for each user:

- user name
- Y=user's private key encrypted with her password
- B (the Alice-specific random number chosen by Bob when setting up Alice's account)
- $W^B \bmod p$

Figure 8 4-msgs, stateless Bob, precomputation, SPEKE-based




## 3.4 Two Messages, No Salt, EKE-based

Now we shrink the protocol down to two messages!

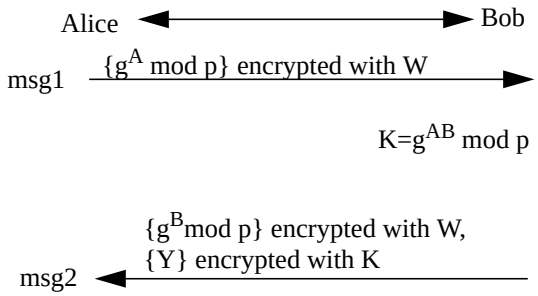
In the EKE-based protocol, Bob needs to store:

- user name
- W=h(pwd)
- Y=user's private key encrypted with her password

If we want to save computation for Bob, we can additionally have him store:

- B
- $\{g^B \bmod p\}$ encrypted with W

Figure 9 Two msgs, EKE-based




In this protocol, Bob simply makes a single response to Alice's request. We've eliminated all of the authentication from the EKE protocol in figure 2, but we don't need it. As before, Alice does not need to authenticate Bob. And in this case (as opposed to our 4-message protocols in the previous section), someone impersonating Bob does not even get a single password guess.

However, Alice does get a single chance to verify a password guess in an unaudited way. If she guesses the wrong password, she will have no information about K, and therefore no information about Y. However, she does get one piece of information on an incorrect guess: that the guess she chose was indeed incorrect. Bob cannot tell if someone requesting a private key download is legitimate or not. But, we stress, this is only a single *on-line* password guess. Although Bob cannot distinguish a legitimate download from a password guess, he ought to get suspicious if the same user requests thousands of password downloads within a short time.

Another disadvantage of this two-message EKE-based protocol in comparison to the protocol in figure 7 is that we no longer solve the problem generally solved by salt, which is to prevent a single guessed password from being

easily tested against many user accounts. Bob needs to store W so that he can decrypt ({$g^A$ mod p} encrypted with W) sent by Alice in order to raise it to B. In some cases, it might be possible to use the user's name as salt, but this can be problematic if users' names can change, or if a user has aliases.

But by using SPEKE instead of EKE we can get the benefit of salt, as we'll see in the next section.

And one more disadvantage of the two-message protocol is that we can no longer use a cookie. Any request will force Bob to do an exponentiation.

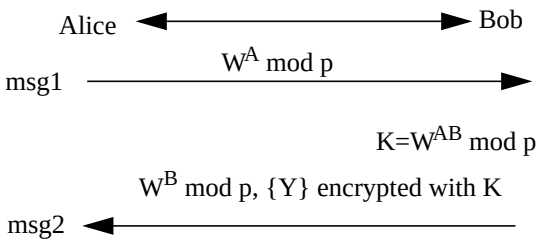
No protocol can guard against an adversary guessing and verifying a correct password, or testing an incorrect password. It would admittedly be better if each wrong guess could be audited. But as we said, the form factor of a 2-message protocol is so nice that in practice the very small security downsides might well be worth the price.

### 3.5 Two Messages, SPEKE-based

In the SPEKE-based two-message protocol, Bob needs to store:

- user name
- Y=user's private key encrypted with her password
- B (the Alice-specific random number chosen by Bob when setting up Alice's account)
- $W^B$ mod p

Figure 10 Two msgs, SPEKE-based




In this protocol, we get the benefit of salt since we don't store W, and $W^B$ mod p is different for each user (since B is unique for each user). We save computation for Bob, since he does not have to choose B each time and raise W to B. (He still has to compute K, of course). Although this protocol, unlike the EKE-based protocol in figure 9, has the advantage of salt, it has the same other disadvantages of the EKE-based protocol:

- Alice gets a single unaudited on-line password guess.
- We cannot use a cookie.

### 3.6 Getting the Rest of the User's Security Context

If the workstation is to competently act on behalf of Alice, it will need other information that makes up her security context. This is likely to include her certificate, the public keys of one or more CAs she trusts, and possibly some encoding of a trust policy - asserting, for example, the location of her mailbox and the certifiers trusted to authenticate it.

Whatever form this information takes, it can be downloaded securely by having it stored signed and (if necessary) encrypted with Alice's key. If the information changes from time to time, the user can sign and encrypt successive versions with a timestamp the workstation can display to prevent undetected reversions. Alternatively, Alice could trust some administrator to manage her security state, in which case her key would sign a statement of trust in the administrator's key, and other state would be stored signed with the administrator's key and encrypted (if necessary) with Alice's public key. In any event, any form of state can be securely loaded by the workstation and could not be forged by anyone without Alice's password or private key.

## 4. Summary

We provide several protocols for downloading a user's private key and other security context such as a CA public key from a directory. Deployed protocols such as NetWare 4 require foreknowledge (or insecure download) of the directory's public key. There are other protocols similar in spirit to our protocols, but they were designed only for mutual authentication and are unnecessarily strong in some cases for our purposes, and therefore require more messages or more computation.

We present protocols that require no preknowledge (such as server public keys), with the following new advantages:

- denial of service resistance
- minimizing server computation
- allowing the server to be stateless even in a four-message protocol
- salt
- two-message protocols with most of the benefits

## 5. Acknowledgements

We wish to thank Mary Ellen Zurko, Jonathan Trostle, and four anonymous reviewers for their helpful comments on this paper.

## 6. Bibliography


1. [BM92] S. Bellovin and M. Merritt, “Encrypted Key Exchange: Password-based protocols secure against dictionary attacks”, Proceedings of the IEEE Symposium on Research in Security and Privacy, May 1992.
2. [BM94] S. Bellovin and M. Merritt, “Augmented Encrypted Key Exchange: a Password-Based Protocol Secure Against Dictionary Attacks and Password File Compromise, ATT Labs Technical Report, 1994.
3. [DH76] W. Diffie and M. Hellman, “New Directions in Cryptography”, IEEE Transactions on Information Theory, November 1976.
4. [Jab96] D. Jablon, “Strong password-only authenticated key exchange”, ACM Computer Communications Review, October 1996.
5. [Jab97] D. Jablon, “Extended Password Protocols Immune to Dictionary Attack”, Proceedings of the WETICE ‘97 Enterprise Security Workshop, June 1997.
6. [KPS95] C. Kaufman, R. Perlman, and M. Speciner, “Network Security: Private Communication in a Public World”, Prentice Hall, 1995.
7. [LGSN89] T. Lomas, L. Gong, J. Saltzer, and R. Needham, “Reducing Risks from Poorly Chosen Keys”. Proceedings of the 12th ACM Symposium on Operating System Principles, December, 1989.
8. [LI94] R. Lee and J. Israel, “Understanding the Role of Identification and Authentication in NetWare 4”, Novell Application Notes, October 1994.
9. [Pat97] S. Patel, “Number Theoretic Attacks On Secure Password Schemes”, Proceedings of the IEEE Symposium on Security and Privacy, May 1997.
10. [SS88] G. Steiner and J. Schiller, “Kerberos: An authentication service for open network systems”, Proceedings of the USENIX Winter Conference, February 1988.
11. [WU98] T. WU, “The Secure Remote Password Protocol”, ISOC NDSS Symposium, 1998.