USENIX Association

Proceedings of the 10th USENIX Security Symposium

Washington, D.C., USA
August 13–17, 2001

PDM: A New Strong Password-Based Protocol

Charlie Kaufman
Iris Associates
ckaufman@iris.com

Radia Perlman
Sun Microsystems Laboratories
radia.perlman@sun.com

Abstract

In this paper we present PDM (Password Derived Moduli), a new approach to strong password-based protocols usable either for mutual authentication or for downloading security information such as the user's private key. We describe how the properties desirable for strong password mutual authentication differ from the properties desirable for credentials download. In particular, a protocol used solely for credentials download can be simpler and less expensive than one used for mutual authentication since some properties (such as authentication of the server) are not necessary for credentials download. The features necessary for mutual authentication can be easily added to a credentials download protocol, but many of the protocols designed for mutual authentication are not as desirable for use in credentials download as protocols like PDM and basic EKE and SPEKE because they are unnecessarily expensive when used for that purpose. PDM's performance is vastly more expensive at the client than any of the protocols in the literature, but it is more efficient at the server. We claim that performance at the server, since a server must handle a large and potentially unpredictable number of clients, is more important than performance at the client, assuming that client performance is "good enough". We describe PDM for credentials download, and then show how to enhance it to have the properties desirable for mutual authentication. In particular, the enhancement we advocate for allowing PDM to avoid storing a password-equivalent at the server is less expensive than existing schemes, and our approach can be used as a more efficient (at the server) variant of augmented EKE and SPEKE than the currently published schemes. PDM is important because it is a very different approach to the problem than any in the literature, we believe it to be unencumbered by patents, and because it can be a lot less expensive at the server than existing schemes.

1 Introduction

This paper presents a new mechanism for allowing a user, armed only with a name and password, to connect to a network from a "generic" client machine, one loaded with software, but not with any user-specific configuration information (such as a private key or the public key of a trusted CA). The most secure solution to the problem of a human attaching via a generic workstation is a smart card. But until smart cards and readers become ubiquitous, and to handle the case when the user has left his smart card at home, there will still be a need for authentication based solely on something humans can remember and type, i.e., a password. Unfortunately, passwords are subject to dictionary attacks because most people are not willing to type and remember a sufficiently long and hard-to-guess password. So it is important to design a protocol in which even though passwords are used as keys, an eavesdropper or someone impersonating either the client or the server will not obtain information with which to do a dictionary attack.

There are several protocols in the literature for solving this problem. EKE [BM92] uses a Diffie-Hellman exchange encrypted with the user's password. SPEKE [Jab96] uses a function of the user's password as the base in a Diffie-Hellman exchange. Later, enhancements to both EKE and SPEKE were added to avoid storing a password-equivalent at the server [BM94], [Jab97]. SRP [Wu98], has the same properties as the augmented EKE and SPEKE, but better performance. AMP [Kwon01] is similar to SRP, with similar properties. [GL00] presents a protocol with similar properties using linear polynomials over $GF(2^n)$, with a proof that the result is as secure as factorization. [BMP00] presents a 3-message variant of augmented EKE and proves it as secure as the Decision Diffie-Hellman (DDH) in the random oracle model. Similarly, [MS99] presents a protocol based on RSA, and proves the security of it based on the random oracle model. There is an unpublished protocol called S.N.A.K.E. by Peter Gunn that does Diffie-Hellman based not on a single strong prime, but a set of strong primes selected from a known set (of perhaps 2000), with the subset chosen being based on the password. [RCW98] presents a protocol (called S3P-RSA) in

which an exponent is generated deterministically from a user's password, and used to transmit a strong secret. But this protocol has been shown to be broken, since many passwords can be tested simultaneously in an on-line attack.

[FK00] presents an additional interesting property that a strong password scheme might have, along with an algorithm for accomplishing that property. This property is the ability to break a user's strong secret into multiple pieces, such that theft of multiple servers' databases are required in order to do a dictionary attack. The disadvantage of this approach is that it requires multiple servers to be available or else the user will not be able to obtain his credentials, and it has lower performance because the user must do a protocol with multiple servers. Our protocol (PDM) does not have this property, and instead requires interaction with only a single server, which in many situations would be more desirable.

PDM (password derived moduli) does Diffie-Hellman based on a safe prime p ("safe" means a prime for which (p-1)/2 is also prime), where p is deterministically generated from the user's password, salted with information such as the user's name. A new approach, even if it gave no new functionality over old approaches, is still potentially important. Sometimes an approach will be found to have flaws, so alternatives are useful. Sometimes two approaches that seem to provide identical functionality are later found to have different properties in subtle ways. For example, although EKE and SPEKE seemed to provide identical functionality, [PK99] demonstrated that a 2-message protocol with salt was possible with SPEKE (with a composite modulus), and was not possible with EKE.

But PDM's performance properties make it a potentially important approach. PDM is vastly more computation intensive for the client than previous approaches, but since a client machine only needs to do the computation intensive operation once (per user, assuming the user has typed her password correctly), what is important is whether the performance is "good enough", which we claim it is. Although lower performance at the client is obviously a disadvantage, it has the beneficial side-effect of making on-line password guessing much slower.

Computation at the server is far more important, since a server might have to simultaneously deal with multiple clients. An attacker impersonating a client forces the server to do as much computation as legitimate clients. PDM can be vastly more efficient at the server since, with secret moduli, Diffie-Hellman must be broken *per password guess*. In most protocols, a single p is used for all users, so it is worth considerable effort to break Diffie-Hellman for that p. For most uses a Diffie-Hellman prime of less than 1000 bits would not be considered secure, but for PDM, a 500-bit prime might be sufficiently secure since breaking 500-bit Diffie-Hellman is estimated to require 8000 MIP-years - a high price to pay to test a single password guess for a single user. If PDM is sufficiently secure with a prime half as big, it will require 1/4 as much computation at the server as the best of any of the other schemes.

One natural worry is low-performance clients, such as hand-held devices. But those devices are carried by the user, and owned by the user, and therefore can be configured with a user-specific high-quality secret. Therefore such devices do not need schemes such as the one in this paper, which are naturally suited to the environment where there is an adequately powered workstation that has no configured information for the user.

In this paper we first describe how the desirable properties for a credentials download protocol differ from those for a mutual authentication protocol. Then we present the simplest PDM scheme, the one suitable for credentials download, or a mutual authentication protocol with the properties of EKE or SPEKE. As described in [Pat97] and [BM92], it is tricky to design such schemes so that an eavesdropper gains no information. We give an example of a potential vulnerability of EKE in section 3.2. In this paper we analyze our scheme for such vulnerabilities and design the protocol so that it does not leak information which would enable an eavesdropper to eliminate passwords.

Then we show how to enhance PDM to create a mutual authentication protocol that has higher performance at the server than existing password-based mutual authentication protocols (although it will still be more expensive at the client). This involves a method of avoiding storing a password-equivalent at the server. The scheme presented in this paper for accomplishing this is more efficient (for the server) than any previous scheme, even without the savings of using a smaller Diffie-Hellman modulus. Another enhancement is to prevent two servers from impersonating each other to a client that uses the same password on each of them. This enhancement works for any of the protocols, and has been proposed in the protocols in [BPR00], [MS99] and [GL00].

2 Properties of Credentials Download vs. Mutual Authentication Schemes

In general, desirable properties of a strong password scheme include:

- User Alice need only know her name and password.
- The workstation need not be configured with any user-specific security information (such as the public key of the server to which Alice will authenticate or download her credentials).
- An eavesdropper on an authentication exchange between user Alice and server Bob cannot learn Alice's password or be able to capture any information that could be used in an off-line password-guessing attack.
- Someone impersonating Alice to Bob, or Bob to Alice, will not be able to gain any information with which to do an off-line password-guessing attack, though one of them will be able to verify a single on-line guess.
- Bob's database should be *salted*, so that a dictionary attack against a stolen copy of the database would have to be launched separately per user, rather than computing hashes of all passwords in the dictionary, and comparing it against all users' information.

Additional properties desirable for a mutual authentication protocol, that are not necessary for a credentials download protocol are:

- Alice authenticates Bob. This is not necessary in credentials download because all Alice cares about is whether she's getting authentic credentials, not whether she's getting it from an authentic source.
- Bob authenticates Alice. This is not necessary in credentials download if the credential is encrypted with a high quality secret such that the requester cannot do a dictionary attack.
- An attacker will not be able to authenticate using replayed messages. This is not an issue in credentials download since all an attacker can do by replaying Alice's message is to get Bob to replay what he previously transmitted.
- Someone that steals Bob's database will not be able to directly use the information to impersonate Alice to Bob or any other server (there is no *password-equivalent* stored at Bob), though they will be able to use it to do an off-line password-guessing attack. (In credentials download, this is not important because if someone has stolen Bob's database he already has Alice's encrypted credential, which was also in the database, so being able to do Alice's piece of the protocol is not an advantage).
- If Alice uses the same password on multiple servers, say Bob and Ted, the information Bob stores cannot be used to impersonate Ted to Alice. (Again, in credentials download, it does not matter who gives the credential to Alice).

As a result of not needing these properties, a credentials download protocol can be simpler. It can be stateless for the server, have better performance, and require fewer messages (e.g., two). Once credentials are securely downloaded, the client can engage in any authentication protocol that assumes strong secrets and/or configured CA public keys (e.g. SSL, IPSec, Kerberos, SSH, etc.).

3 PDM: Password Derived Moduli

The key to our protocol, just as with EKE, SPEKE, and many descendents of these protocols, is to modify a Diffie-Hellman exchange with a function of the user's password. For example, EKE encrypts the Diffie-Hellman public number with a function of the user's password. SPEKE uses the user's password to calculate a base for the Diffie-Hellman exchange. We calculate a prime p that is a function of the user's password. This is done by using the user's password as a seed for a pseudo-random number generator that will be used in the search for an appropriate prime.

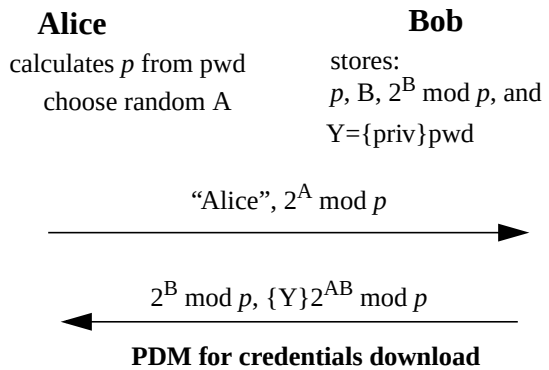
3.1 PDM for Credentials Download

We begin with just the simplest form of a strong password-based authentication protocol, that has only the functionality of the original EKE and SPEKE, and only the properties necessary for credentials download. On the surface, the protocol is extremely simple. The server Bob stores, for user Alice, p. We will always use 2 as the base. The reasons for using 2:

- it makes it easy to recognize small exponent cheating by someone impersonating the client (see section 3.2.2),
- to be different from SPEKE, to avoid potential patent infringement, and
- to use the law of quadratic reciprocity to choose candidate p's for which 2 is certain to be a generator (and thus avoid the performance cost of having to search for a generator, or the possible security loss as explained in section 3.4.2 of using a base that isn't a generator). If p is equal to 3 mod 8 and p is a *safe prime* ((p-1)/2 is also prime), 2 will be a generator.

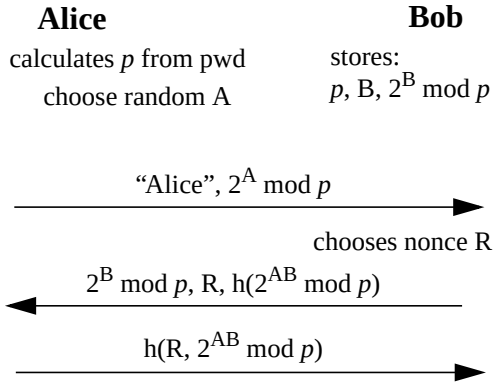
We chose a target of 10 seconds for a reasonable amount of computation time for the client to pick a *p*. Unfortunately, finding a safe *p* of a size considered secure for traditional Diffie-Hellman (say 1000 bits) would take longer (on today's typical client machines) than our target of 10 seconds for the user to log in. So as we discuss in section 3.4, there are various corners we can cut while maintaining good enough security for practical purposes. Indeed, on a 400 MHz processor, a 500-bit safe prime can be found within 10 seconds, and we argue that for our application, Diffie-Hellman with a 500-bit modulus would give adequate security because of the necessity for an eavesdropper doing a dictionary attack to break 500-bit Diffie-Hellman *for each guessed password*. And with the trick suggested in section 3.4.3, even a 1000-bit safe prime can be generated well within our budget of 10 seconds at the client. We give timing estimates for generating safe primes of various sizes in section 3.5.

For a simple 2-message credentials download protocol, the server Bob stores *p* and the credential Y, which is, for instance, the user's private RSA key encrypted with her password. The workstation calculates *p* from the user's password. As observed in [PK99], for credentials download it is possible to save Bob an exponentiation by having him always use the same B for Alice, and storing B and $2^B \bmod p$. (Note: the notation "{data}key" means "data" encrypted with key "key").



PDM for credentials download

For mutual authentication, especially if the rest of the session is not cryptographically protected by the resulting Diffie-Hellman key, then Bob can still save himself an exponentiation, but has to additionally furnish a nonce R in message 2, and Alice should return a function of both the nonce and $2^{AB} \bmod p$. Without the nonce an eavesdropper could replay Alice's messages and Bob would accept this as Alice having authenticated. So here is a mutual authentication PDM-based protocol in which Bob need only do one exponentiation. This scheme stores a password-equivalent at Bob, but in currently published schemes Bob requires more than 2 exponentiations.



Single exponentiation mutual authentication

This trick of saving Bob an exponentiation will not work if we want the additional feature of not storing a password-equivalent at Bob. Also, if the protocol is being used to establish a session key as well as just doing the initial authentication, perfect forward secrecy would be lost by having Bob always use the same B.

3.2 Avoiding Leaking Information

As discussed in both [BM92] and [Pat97] protocols such as these need to be implemented carefully or else information will be leaked. For instance, in the most straightforward implementation of EKE, one might encrypt $g^A \bmod p$ with a hash of the password. An eavesdropper that observed an encrypted $g^A \bmod p$ could do trial decryptions with various passwords, and eliminate any passwords in which the result was larger than *p*. If *p* was just a little more than a power of 2, then about half the passwords could be eliminated each time an eavesdropper observed a Diffie-Hellman value encrypted with a password. This might occur twice per authentication in variants of EKE that have both sides encrypting their Diffie-Hellman values.

3.2.1 Choosing *p* from a Small Range

In PDM, care must be taken to avoid allowing an eavesdropper to eliminate passwords based on seeing $2^A \bmod p$ and $2^B \bmod p$. If either transmitted Diffie-Hellman number was greater than the *p* derived from a candidate password, an eavesdropper could rule out that password.

We solve this problem by discarding A (or B) in the case where $2^A \bmod p$ is greater than the smallest possible p that could be derived from any password. To make the probability acceptably low that an A would have to be discarded (forcing an additional exponentiation) we choose p's from a very small range (e.g., if the smallest p and the largest p differ by less than 0.1%, then a Diffie-Hellman number will need to be rejected less than one time in 1000). We choose a p from a narrow range very close to a power of 2. We make it a narrow range by fixing the top 64 bits of the number at which our search will take place. Any constant will do, but to make maximal use out of the bits, the constant might as well be 63 1's followed by a 0. With a prime of, say, 700 bits, that gives a space of 700-64 bits, or 636 bits from which to choose p's, obviously large enough that there will be no shortage of p's, and yet the fraction of 700-bit space from which the p's are chosen is $1/2^{64}$. With this fraction, the probability of ever getting a $2^A \bmod p$ larger than the smallest possible p is $1/2^{64}$. And if it did occur, the only consequence is that authentication would take a little longer since another A would need to be chosen.

3.2.2 User Impersonator Picking Small A such that $2^A < p$

Another threat is that Trudy, impersonating Alice, could choose a very small A, such that 2^A would not be larger than p. Then Trudy could guess passwords based on what Bob sends, since Trudy has not committed to a value of p (because $2^A \bmod p$ has the same value for all possible p). In order to test a candidate password, Trudy needs to know the A corresponding to $2^A \bmod p$ for various values of p. If 2^A is less than p, then she knows such a pair for all p. If 2^A is even slightly more than p, and therefore needs to be reduced by p, she gets only a single pair.

Using the constant "2" for the base has the fortunate side-effect that it is very easy to detect if someone is cheating and sending a value that did not need to be reduced mod p. We require the sender to choose a Diffie-Hellman exponent larger than the log of p (i.e., if p is 700 bits long, then the exponent must be > 700) so that the result will need to be reduced by p. Since 2 is a generator, there cannot be two different exponents that yield the same value mod p. Therefore, if the number is of the form $10000...00000_2$, (i.e., the binary representation contains a single 1) then the sender has cheated by using an exponent sufficiently small that it did not need to be reduced by any modulus.

3.2.3 Timing Attacks

Because calculating p from a password involves searching for a prime at a pseudo-random value and testing until one is found, different passwords would take substantially different amounts of time to compute p. If an eavesdropper knew with some precision how long it took Alice's machine to compute p, this information could be used to eliminate many candidate passwords.

For example, a protocol which would give an eavesdropper timing information is one in which Alice's machine does not start computing p until it receives a message from Bob, perhaps because it needs to receive a salt value (see section 3.3) from Bob before it can compute p. The time until Alice's reply will be approximately the amount of time required for the machine to calculate p.

So it is best if Alice's workstation can compute p from the password before beginning the authentication protocol. This is possible if the salt is implicit, e.g. it is a canonical representation of the user name, since then the computation is done before messages are sent and an eavesdropper cannot time how long it took to compute p. A second choice, if implicit salt is not possible (too many variations on the name), would be to have Alice's typing of the password occur after she types the name of the server she wishes to contact. Since user typing times are highly variable, an eavesdropper will not be able to tell how much of the interval between Bob's message (e.g., sending salt), and Alice's machine's reply was due to computation of p and how much was due to Alice typing the password.

3.3 User Salt

It is highly desirable for user Alice's machine to be able to compute p before talking to the server, because:

- it will take the client machine a long time to compute p, so it would be good to be computing it while the user is doing other things, for instance, typing the name of the service she wishes to access.
- we don't want to allow an eavesdropper to tell how long it takes to compute p.
- If p is user/password dependent, but not server dependent, then a user can use the same p on multiple servers, ensuring that the expensive computation of p need only be done once per user, even if the user is using PDM for mutual authentication with multiple servers.
- it would take an extra message to send the salt.

In order to compute *p* before talking to the server, the salt value must be *intrinsic*, i.e., computable from information known locally about the user. Since this consists of the user's name and password, the logical choice for salt value is the user's name. It is important, however, to have a canonical version of the name. Capitalization or nicknames must not affect the computation of *p*.

3.4 Performance

The computation the server must perform to execute the basic PDM protocol (assuming equal sized moduli) is comparable to the best of the protocols with similar functionality even if the same size modulus is used. (This assumes that a protocol such as EKE or SPEKE is modified as suggested in [PK99] to have the server store B per user to save an exponentiation).

By using a different technique (as described in section 4) to achieve the goal of not storing a password equivalent at the server, PDM has better performance (even with the same sized modulus) than any of the previous schemes, though that technique could apply to EKE or SPEKE to make them equivalent in server performance (with the same size modulus). Although PDM is more expensive at the client than any of the prior protocols, we claim that since the client machine only needs to do the computation once, the only thing that matters in practice is for performance at the client to be "good enough". During the initial authentication, a human is waiting, and it is unacceptable for a user to wait for more than about 10 seconds to log in (and that's pushing it). Choosing *p* to be a 1000 bit safe prime would take more than a minute (see section 3.5) on today's typical desktop machine. Fortunately, there are some shortcuts we can take that raise performance dramatically. Note that as machines get faster we can drop more and more of the shortcuts.

3.4.1 Size of *p*

Today's conventional wisdom says that the size of a prime used in a Diffie-Hellman exchange should be on the order of 1000 bits. But given that this is not an ordinary Diffie-Hellman exchange, might a smaller prime be acceptable? Computation time falls dramatically with the size of the prime. What is the threat if our prime is smaller?

It is within the realm of possibility to break Diffie-Hellman with a size of, say, 500 bits, which at today's estimates would take on the order of 8,000 MIP-years. Eavesdropping on any authentication would yield a quantity with which password guesses could be verified, but it requires, for each guessed password, computing *p* and breaking Diffie-Hellman with that *p*. So an attacker would have to break 500-bit Diffie-Hellman *per password guess*.

Perfect forward secrecy would be endangered using Diffie-Hellman primes that are within the realm of possibility to crack, because if someone were to record conversations, and subsequently learn the user's password, then he'd be able to compute *p*, break 500-bit Diffie-Hellman, and then recover the session keys of the authentications that used that *p*. In practice, this is a sufficiently obscure threat that the size of the Diffie-Hellman prime is unlikely to be the weakest link in the chain (on-line password guessing, or using the learned password to directly impersonate the user in future conversations would probably be more fruitful), so in practice a 500 bit *p* might suffice. Alternately, and at some cost in complexity and server computation, the perfect forward secrecy attack could be circumvented by supplementing this protocol with a second anonymous Diffie-Hellman exchange with fixed adequate strength primes. If the result of that second Diffie-Hellman exchange contributes to the session key, perfect forward secrecy is preserved. And since computation with a small *p* is so efficient, the double Diffie-Hellman (with one large fixed *p* and one small, based-on-the-password *p*) would still be of comparable performance at the server to the best of existing schemes.

3.4.2 Non-Safe Prime

We can also save time in generating *p* by not requiring *p* to be a safe prime. The cost of breaking Diffie-Hellman is a function of both the size of *p* and the size of the largest prime factor of *p*-1. It is much faster to find a *p* with the property that (*p*-1)/2 isn't prime, but merely has a large prime factor. Although it is believed that Diffie-Hellman will be sufficiently secure with a *p* of this form, we run into a problem of finding a generator for *p* if *p* is not a safe prime, since without knowing the factorization of *p*-1 it is difficult (if not impossible) to determine whether a given *g* is a generator of the group. Traditional Diffie-Hellman does not need to assure that *g* is a generator. It only needs to assure that *g* generates a large subgroup of *p*. But for us, it is important that our *g* (which will be 2) is a generator. Otherwise, it might leak information to an eavesdropper. If the eavesdropper knew, for a particular password, that 2 was not a generator for the corresponding *p*, and then saw a value that 2 could not generate for that *p*, that password could be ruled out for that user.

3.4.3 User-Supplied Hint

Another method of increasing performance is to use a trick suggested by Jeff Schiller of giving the user a hint to tell the workstation, such as several of the bits of the selected *p*. If the user can't remember the hint, the workstation must test all candidate numbers. If the user misremembers the hint, then authentication will fail since the workstation will compute the wrong *p*. The user will recognize that it is probably the wrong hint since computing p will be as slow as without the hint.

How much will performance be improved? Assuming you've sieved for factors of *p* and (*p*-1)/2 up to 10,000, to get a safe prime of 512 bits you'd have to test, on average, 1600 numbers. On a 400 Mhz processor, a safe prime of 512 bits can be found within our budget of 10 seconds without the hint. Using the "hint" telling you, for instance, 6 bits of *p*, reduces computation by a factor of 64, making it under our target of 10 seconds even for 1024-bit safe primes. This hint could be in the form of a single character (using upper and lower case, numbers, and two more characters).

3.5 Measured Timing for Generating *p*

On a 400 MHz processor, using code that was not optimized for performance, the following table shows mean generation times with and without a six-bit user-supplied hint.

size of *p*	without hint	with hint
512	8.1 seconds	.11 seconds
768	34 seconds	.57 seconds
1024	111 seconds	1.8 seconds

Times for Generating *p*

Even more speedup could be attained with a larger hint, but of course this stretches the abilities of the human to remember the hint.

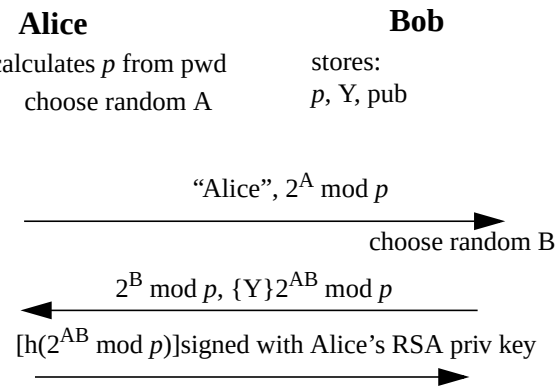
4 Avoiding a Password-Equivalent at the Server

In this section we discuss a different method of avoiding storing a password-equivalent at the server that is higher performance at the server than previous schemes. The approaches suggested in this section could be used for EKE and SPEKE as well, but not for AMP or SRP. The best previous method, SRP, involved doing two expensive exponentiations and one exponentiation with a 32-bit exponent. We present two new variants. The first is a little better in performance than SRP, assuming equal sized moduli, because our inexpensive exponentiation is an RSA verification (so the exponent could be as small as 3, rather than a 32-bit number as in SRP). The second involves only a single Diffie-Hellman exponentiation at the server and an RSA verification, so it is about half as much computation at the server as SRP, but it gives up perfect forward secrecy if someone steals Bob's database. (and again, this is assuming equal sized moduli).

In any of the schemes (ours as well as augmented EKE, SPEKE and SRP) it will be possible to do off-line password-guessing using a stolen copy of the server database, but without correctly guessing and verifying the password, the information in the server database would not be usable for impersonating the user to that (or any other) server.

The augmented versions of EKE and SPEKE, and protocols such as SRP and AMP do variants of having the server store $g^X \bmod p$ (where X is a function of the password), and require knowledge of X on the client. The augmented feature of these protocols requires an extra expensive exponentiation at the server.

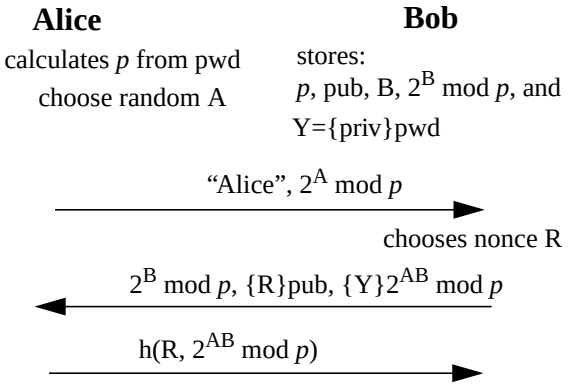
By using an RSA private key encrypted with Alice's password in place of $g^X \bmod p$ we can reduce the total computation for Bob to two expensive exponentiations and a single RSA public key verify, which can be very inexpensive (for example, if the public exponent is 3). Basing it on RSA is especially attractive because the same protocol works for download of an RSA private key as for mutual authentication. Bob stores *p*, Y (an RSA private key encrypted with the user's password), and pub (the associated public key). The protocol is:



No pwd-equivalent stored at server

In the protocol above, Bob has to compute two expensive exponentiations: raising 2 to B mod *p*, and raising (2^A mod *p*) to B mod *p*, and an inexpensive exponentiation (an RSA verify). This is slightly better in performance than the best previous scheme (SRP) because our inexpensive exponentiation, an RSA verify, is less expensive that SRP's inexpensive exponentation with a 32-bit exponent. It might also be the case with a secret modulus *p* (our scheme described in section 3) that the Diffie-Hellman exchange can be secure with a smaller *p*, which would further reduce the work for Bob. Note also that Alice must authenticate Bob. She does this by checking to see if Y, when decrypted, has the encoding of an RSA private key.

With the RSA-based scheme, we can reduce the work for Bob down to a single expensive exponentiation by allowing Bob to use the same B each time and adding a nonce as we did in section 3.1. If we make the session key be a function of the nonce as well as the Diffie-Hellman key, we can achieve "partial forward secrecy", a term we are using to mean someone would have to steal *both* Alice's private key and Bob's database in order to decrypt previous conversations.



Partial forward secrecy, single exponentiation

The session key should be some function of both the Diffie-Hellman key and R, such as h(1,R,2^{AB} mod *p*). We give up perfect forward secrecy because if someone steals *both* B and Alice's private key, they can decrypt a previously recorded conversation, since they will be able to compute 2^{AB} mod *p* (because they will have stolen B from Bob's database), and extract R (because of having stolen Alice's private key).

5 Preventing Servers from Impersonating Each Other to the User

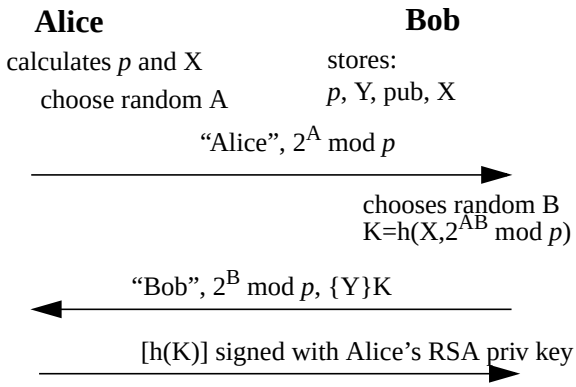
The third proposed enhancement is to prevent servers from impersonating each other to the user. If the information stored for user Alice is the same at server Bob as at server Carol, then Bob and Carol will be able to impersonate each other to Alice.

For this reason it is important to customize the information per server, so that even if Alice chooses the same password at multiple servers, the information at each will be different, and not usable to impersonate a different server to Alice.

The method of accomplishing this is to have some of the information stored for Alice be a function of the password and the server's name. It is desirable for Alice to have the same value for *p* at each server, since it is computation-intensive for Alice to compute *p*. So there should be some other quantity, X, that is a function of the server name. X will enable Bob to authenticate to Alice as "Bob" rather than as "any server on which user Alice has that password". Then even if the *p* is the same at Bob and Carol, they will not be able to impersonate one another to Alice because each only knows its own X.

So we suggest that *p* be computed using a seed which is solely a function of the user's name and user's password, and X be a function of the server's name, the user's name, and the user's password.

Bob stores *p* (generated from Alice's name and password, Y (Alice's private RSA key encrypted with her password), X (a hash of Alice's name, password, and Bob's name), and Alice's public key:



Prevent servers impersonating each other

6 Summary

In this paper we present PDM, a new method of doing strong password-based credentials download or mutual authentication. It has better performance at the server than any of the existing schemes, especially since it can use smaller moduli, because there is no single modulus on which the world could concentrate its Diffie-Hellman breaking efforts. Instead, Diffie-Hellman would have to be broken per user per password guess. We show that although performance at the client is far more expensive, that it is “good enough”, especially with an optional user-supplied hint. We also present a method for avoiding a password equivalent which is less expensive than existing schemes at the server. This scheme could be applied to EKE or SPEKE, but not to schemes such as SRP and AMP that depend on everything being based on Diffie-Hellman. And we present a scheme with “partial forward secrecy” that is half as expensive as SRP, even with the same sized modulus.

Acknowledgments

We wish to thank Eric Rescorla for writing the code for generating PDM primes, and timing it for various sizes of *p*. We also wish to thank David Jablon for offering helpful comments.

References

[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.

[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.

[BMP00] V. Boyko, P. MacKenzie, and S. Patel, “Provably Secure Password Authenticated Key Exchange Using Diffie-Hellman”, Advances in Cryptology - EUROCRYPT 2000.

[BPR00] M. Bellare, D. Pointcheval, and P. Rogaway, “Authenticated Key Exchange Secure Against Dictionary Attacks”, Advances in Cryptology - EUROCRYPT 2000.

[DH76] W. Diffie and M. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory, November 1976.

[FK00] W. Ford and B. Kaliski, “Server-Assisted Generation of a Strong Secret from a Password”, Proceedings of the IEEE 9th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2000.

[GL00] O. Goldreich and Y. Lindell, “Session-Key Generation using Human Passwords Only”, Cryptology ePrint Archive: Report 2000/057.

[Jab96] D. Jablon, "Strong password-only authenticated key exchange", ACM Computer Communications Review, October 1996.

[Jab97] D. Jablon, "Extended Password Protocols Immune to Dictionary Attack", Proceedings of the WETICE `97 Enterprise Security Workshop, June 1997.

[Kwon01] T. Kwon, “Authentication and Key Agreement via Memorable Password”, ISOC NDSS Symposium, 2001.

[KPS95] C. Kaufman, R. Perlman, and M. Speciner, "Network Security: Private Communication in a Public World", Prentice Hall, 1995.

[MS99] P. MacKenzie and R. Swaminathen, “Secure Network Authentication with Password Identification”, submission to IEEE P1363.

[Pat97] S. Patel, "Number Theoretic Attacks On Secure Password Schemes", Proceedings of the IEEE Symposium on Security and Privacy, May 1997.

[PK99] R. Perlman and C. Kaufman, “Secure Password-Based Protocol for Downloading a Private Key”, ISOC NDSS Symposium, 1999.

[RCW98] M. Roe, B. Christianson, D. Wheeler, “Secure Sessions from Weak Secrets, Technical report from University of Cambridge and University of Hertfordshire, 1998.

[SS88] G. Steiner and J. Schiller, "Kerberos: An authentication service for open network systems", Proceedings of the USENIX Winter Conference, February 1988.

[Wu98] T. Wu, "The Secure Remote Password Protocol", ISOC NDSS Symposium, 1998.