Optimal Authentication Protocols Resistant to Password Guessing Attacks

Li Gong

SRI International
Computer Science Laboratory
Menlo Park, California 94025, U.S.A.
(gong@csl.sri.com)

Abstract

Users are typically authenticated by their passwords. Because people are known to choose convenient passwords, which tend to be easy to guess, authentication protocols have been developed that protect user passwords from guessing attacks. These proposed protocols, however, use more messages and rounds than those protocols that are not resistant to guessing attacks. This paper gives new protocols that are resistant to guessing attacks and also optimal in both messages and rounds, thus refuting the previous belief that protection against guessing attacks makes an authentication protocol inherently more expensive.

1 Introduction

Identifying users is an indispensable element of computer security and, because auxiliary devices such as smart-card are not likely to be ubiquitous in the foreseeable future, users have to be authenticated through their passwords. (We do not discuss authentication methods based on physical or biological technologies.) People are known to use poorly chosen passwords that are vulnerable to dictionary attacks or guessing attacks [9], while all available evidence suggests that forcing people to choose and remember good passwords – those that tend to be long character strings including both Roman letters and digits – is unworkable because such well-chosen passwords are also quite unmemorable [3, 7].

Authentication protocols have been proposed that are resistant to password guessing attacks [8, 6, 1, 2], although they are more expensive in terms of the numbers of messages and rounds than those authentication protocols without the additional requirement to protect weak passwords [4, 5]. For example, it is proven that the optimal mutual authentication (with handshake and using nonces for challenge and response) uses five messages [4], while the Nonce Protocol uses seven messages [6]. It was thought that such increased cost is inherent, and in particular, is because the server must decide if a client request is fresh before giving a reply – otherwise, guessing attack can materialize [6].

In this paper, we show that this constraint is incidental to the techniques used in those protocols, and by a different design, we can develop authentication protocols that are resistant to password guessing attacks while at the same time being optimal both in the number of messages and in that of rounds.

In the rest of this paper, we first review the techniques and protocols for protecting passwords from guessing attacks. Readers familiar with existing literature on this subject can skip Section 2. Then we show how to design protocols that are optimal in messages and rounds. We also discuss extensions of the protocols to other scenarios, such as direct authentication. We finally conclude with a summary and a discussion as to why synchronized clocks may not help in further reducing the numbers of messages and rounds.

2 Defeating Guessing Attacks

We use the following notation throughout the paper. The notation $\{m\}_k$ denotes the result of encrypting message m using key k, "," denotes concatenation (m, n represents the concatenation of m and n), and $\oplus$ denotes the bit-wise exclusive-or operation. "$A \rightarrow B : m$" represents A sending a message m to B.

We now summarize the basic techniques for protecting passwords from guessing attacks. (More technical details can be found in an earlier paper [6].) Suppose

1063-6900/95 $4.00 © 1995 IEEE

A registers a password $k2$ at server S, and A knows the public key of S, $k1$. (The case of A not knowing S's public key can be handled easily by a simple extension [6].) Then, suppose that A asks S a question represented by the number n, and expects to get an answer in the form of $f(n)$, where $f()$ can be a well known function. The protocol is shown in Table 1.

1. A $\rightarrow$ S: $\{c1, c2, c3, n, k2\}_{k1}$
2. S $\rightarrow$ A: $\{c2, c3 \oplus f(n)\}_{k2}$

Table 1: An illustration of basic techniques

In step 1, A selects three sufficiently large random numbers $c1$, $c2$, and $c3$, encrypts them (together with the question n and the password $k2$) under S's public key $k1$, and sends the ciphertext to S. In step 2, S decrypts message 1 using his private key, checks the password, uses $c3$ to mask the answer $f(n)$, encrypts with A's password $k2$, and replies with message 2. Upon receiving message 2, A decrypts it using his password and checks the result. If the first part of message 2 is indeed $c2$, then the reply must be fresh (i.e., after receiving message 1) and has not been tampered en route (assuming encryption also provides integrity). Now A can use $c3$ to unmask the second part of the reply and obtain the answer to his question.

To mount a guessing attack, the attacker who has recorded all the exchanges over the network can guess $k2$ and try to decrypt message 2. But all he can see in the decrypted text is a random string, which gives no indication as to whether his guess of $k2$ is correct or not. Furthermore, $k1$ is a public key and thus its corresponding private key is commonly assumed too long to guess in a computationally feasible way. Therefore, the attacker cannot decrypt message 1, and can only hope to reconstruct it in order to verify if a guess is correct. This reconstruction is infeasible because he does not know $c1$. In other words, to attack password $k2$ by guessing, the attacker effectively has to guess both the password and $c1$ (or S's private key), but the latter is too long to guess. Therefore, the attacker cannot know whether a guess is correct or not, and guessing attack is rendered impotent. If the attacker attempts to use a guessed password in an online transaction, then a failed guess can be detected and logged.

Based on these basic techniques, a protocol using nonces has been developed [6], as shown in Table 2. Here, Ka and Kb are A's and B's passwords respectively. Ks is server's public key. This protocol is adapted from the Compact Protocol that uses timestamps [6]. The modification is to let A (and also B) obtain a freshness identifier ns (a nonce in this case) from the server S (messages 1 and 2). After that, the protocol is the same as the Compact Protocol except that the timestamp in messages 3 and 4 are substituted by the nonce ns.

1. $A \rightarrow S : A, B$
2. $S \rightarrow A : A, B, ns$
3. $A \rightarrow B : \{A, B, na1, na2, ca, \{ns\}_{Ka}\}_{Ks}, ns, ra$
4. $B \rightarrow S$: $\{A, B, na1, na2, ca, \{ns\}_{Ka}\}_{Ks}$, $\{B, A, nb1, nb2, cb, \{ns\}_{Kb}\}_{Ks}$
5. $S \rightarrow B : \{na1, k \oplus na2\}_{Ka}, \{nb1, k \oplus nb2\}_{Kb}$
6. $B \rightarrow A : \{na1, k \oplus na2\}_{Ka}, \{f1(ra), rb\}_{k}$
7. $A \rightarrow B : \{f2(rb)\}_{k}$

Table 2: The Nonce Protocol

This protocol works as follows. A first obtains a nonce ns from the server, composes a fresh request message, and sends it to S via B (and at the same time passes along S's nonce). B composes a similar request message. The server checks, by examining $\{ns\}_{Ka}$ and $\{ns\}_{Kb}$, that both parts of message 4 are fresh and they originate from A and B. S then selects a session key k and replies with message 5. B decrypts the second part of message 5 using his password, finds $nb1$, and thus is satisfied that the message is from S, is fresh, and has not been tampered with during transmission. A does a similar check, before they complete a handshake. Here $f1()$ and $f2()$ are predefined functions.

In this protocol, we protect the passwords not only from an outside attacker, but also from insiders, who can be either malicious or merely incompetent. For example, A cannot guess B's password, and vice versa, even with the aid of the residue of a successful authentication. A safeguard that works under these circumstances also ensures that neither party can cause the compromise of the other's password by compromising their own.

The reason for requiring the initial nonce acquisition is that, if message 3 (or 4) is not fresh, then the attacker can reuse it and obtain two different versions of message 6: $\{na1, k \oplus na2\}_{Ka}$ and $\{na1, k' \oplus na2\}_{Ka}$. The two session keys, k and k', are different because S chooses a new session key each time. However, because both messages contain the same value for $na1$, the attacker can guess Ka and decrypt both messages to see if the same value $na1$ emerges. A match indicates that the guess is correct with very high probability.

It is this constraint – that the server must respond only to fresh requests – that makes the nonce-based protocol use two messages more than the optimal case. In the next section, we show how to remove this constraint and thus to derive optimal protocols.

3 An Optimal Protocol

The central idea is to let A choose an additional random number $na3$ that is to be used by S as the encryption key in the reply message. The idea of user-generated encryption key is not new, but the typical objection is that the authentication server is much better at selecting high-quality keys. In our circumstances, however, the alternative key is a user-chosen password, which is unlikely to be cryptographically stronger than the random number $na3$. (Of course $na3$ should not be weak keys specific to the cryptosystems used.) Moreover, $na3$ is a one-time key such that cryptographically breaking its encryption does not endanger the password and subsequent authentication sessions. The optimal protocol is shown in Table 3.

1. $A \rightarrow B : \{A, B, na1, na2, ca, na3, \{na3\}_{Ka}\}_{Ks}, ra$
2. $B \rightarrow S$: $\{A, B, na1, na2, ca, na3, \{na3\}_{Ka}\}_{Ks}$, $\{B, A, nb1, nb2, cb, nb3, \{nb3\}_{Kb}\}_{Ks}$
3. $S \rightarrow B : \{na1, k \oplus na2\}_{na3}, \{nb1, k \oplus nb2\}_{nb3}$
4. $B \rightarrow A : \{na1, k \oplus na2\}_{na3}, \{f1(ra), rb\}_k$
5. $A \rightarrow B : \{f2(rb)\}_k$

Table 3: An optimal, five-message nonce protocol

This protocol works as follows. A selects random numbers $(na1, na2, ca, na3, ra)$ and composes and sends message 1. B does the same by sending message 2. The server S checks that the pair $na3$ and $\{na3\}_{Ka}$ matches with A's password Ka, which proves that the original sender is A. S does the same check for B. Then the server selects a session key k for A and B to share and replies with message 3.

In message 3, the inclusion of $na1$ and $nb1$ demonstrates that the reply is fresh, while $na1$ and $nb1$ hide the value of the session key. The message is encrypted, in two parts, under keys $na3$ and $nb3$. After that, A and B complete a handshake exchange, the same as is done in the earlier Nonce Protocol.

The security of the protocol can be argued similarly as that of the Nonce Protocol. Basically, because Ks is the server's public key, only the server can obtain $na3$ and $nb3$. Since message 3 is also fresh and its integrity is maintained, then it must have come from the server, and thus the key k must be the session key chosen by the server. Moreover, if the attacker attempts to mount a guessing attack on a password (say Ka), then he needs to reconstruct message 1 because a guessed value of Ka does not lead to any other information related to subsequent messages (i.e., no verifiable texts in later messages). However, to reconstruct message 1, he must also guess the value of ca (the confounder [6]), which is assumed to be chosen at random from a large space and thus infeasible to guess by exhaustive search.

Although the attacker can replay an old message 1 – because the server cannot decide its freshness – all the attacker can get is a pair (or more) messages in the form of $\{na1, k \oplus na2\}_{na3}$ and $\{na1, k' \oplus na2\}_{na3}$. These do not help him in compromising a future session key k'' or in guessing the password Ka.

The optimality of the protocol is easier to see – it uses five messages, reaching the proven lower bound [4]. (More detailed definitions and terminologies related to optimality can be found in our previous publications [4, 5].) It is also simple to re-arrange the messages so that it uses four rounds, again a proven lower bound, as shown in Table 4.

1. $A \rightarrow B : \{A, B, na1, na2, ca, na3, \{na3\}_{Ka}\}_{Ks}, ra$
2. $B \rightarrow S : \{A, B, na1, na2, ca, na3, \{na3\}_{Ka}\}_{Ks}$, $\{B, A, nb1, nb2, cb, nb3, \{nb3\}_{Kb}\}_{Ks}, rb$
3. $S \rightarrow A : \{na1, k \oplus na2\}_{na3}, rb$
4. $S \rightarrow B : \{nb1, k \oplus nb2\}_{nb3}$
5. $A \rightarrow B : \{f2(rb)\}_k$
6. $B \rightarrow A : \{f1(ra)\}_k$

Table 4: An optimal, four-round nonce protocol

Here, messages 3 and 4, and 5 and 6, can be sent in the same round. Note that the server relays B's nonce rb to A in message 3. An earlier proof (Case 8, NB+AH+SO [5]) applies, which shows that it is impossible to design a protocol with five messages and four rounds.

Note that a lot of replayed messages may overload the authentication server. This does not necessarily pose a security threat, unless we consider cryptanalysis (on the server's responses) a significant problem. Techniques are available to make such attacks more difficult.

4 Beware of Subtle Attacks

As we have shown, it is not important if the server cannot decide the freshness of the request messages, which is the main reason why we have been able to develop optimal protocols. It is vital, however, that the server can identify the senders of those request messages because otherwise S may be cheated into telling B that A is at the other end of the connection when in fact it is an attacker C. In our protocol, identification is done through two pairs of texts, $na3$ and $\{na3\}_{Ka}$, and $nb3$ and $\{nb3\}_{Kb}$, because only the holders of the passwords (Ka and Kb) can generate such pairs.

Such explicit identification may be necessary, as we now show how to break a variation of the protocol where identification is inexplicit. Suppose we remove $na3$ from the pair, as shown in Table 5 (the case for B is identical). Because $\{na3\}_{Ka}$ is still present, intuitively only S knows Ka and can obtain $na3$ to compose a reply message.

1. $A \rightarrow B : \{A, B, na1, na2, ca, \{na3\}_{Ka}\}_{Ks}, ra$
2. $B \rightarrow S : \{A, B, na1, na2, ca, \{na3\}_{Ka}\}_{Ks}, \{B, A, nb1, nb2, cb, \{nb3\}_{Kb}\}_{Ks}$
3. $S \rightarrow B : \{na1, k \oplus na2\}_{na3}, \{nb1, k \oplus nb2\}_{nb3}$
4. $B \rightarrow A : \{na1, k \oplus na2\}_{na3}, \{f1(ra), rb\}_{k}$
5. $A \rightarrow B : \{f2(rb)\}_{k}$

Table 5: A variation that is insecure

In this case, the attacker can take the following line of actions. He selects random numbers ($na1$, $na2$, ca, and x) and compose a message of the form $\{A, B, na1, na2, ca, x\}_{Ks}$. He sends this in place of message 1, claiming that it originates from A. The server then treats x as $\{na3\}_{Ka}$ for some value of $na3$, and in due course the attacker receives $y = \{na1, k \oplus na2\}_{na3}$. Now the attacker guesses a value of Ka, uses it to decrypt x to obtain $na3$, and uses that to further decrypt y. If $na1$ emerges from the decryption, the attacker knows that he has guessed Ka correctly with very high probability.

Note that even if message 3 is modified to be $\{k \oplus na2\}_{na3}$ so that the evidence $na1$ is removed, the same attack can still succeed. Now, the attacker himself also takes the role of B, through which he (quite legitimately) obtains the session key k. After decrypting y, he only need to see if the plaintext is identical to $k \oplus na2$ (he knows both k and $na2$). A match indicates a successful guess of Ka.

5 Two-Party Direct Authentication

A and B sometimes may already share a poorly chosen secret (say Kab) and wish to establish, in a secure way, a well-chosen session key. In the following direct authentication protocol, $k1$ is a public key chosen by A, and k is the session key chosen by B.

1. $A \rightarrow B : na, \{k1\}_{Kab}$
2. $B \rightarrow A : \{B, A, nb, cb, k, \{na\}_{Kab}\}_{k1}$
3. $A \rightarrow B : \{nb\}_{k}$

Table 6: An optimal direct authentication protocol

In this protocol, A selects public key $k1$, encrypts it with the shared password Kab, and sends it and a nonce na to B. B decrypts to get $k1$, selects three random numbers (nb, cb, k), and sends message 2. A then uses the private key corresponding to $k1$ to decrypt this message and obtain the session key k. The presence of $\{na\}_{Kab}$ proves to A that the message is sent by B and is fresh. Finally, A completes the handshake by sending message 3.

The security argument is similar to those for the three-party protocol in Section 3. This protocol is more efficient than those previously proposed that use five messages [1, 6]. Our protocol is in fact optimal because three messages and three rounds are lower bounds proven for nonce-based protocols that carry out only handshakes [4].

It is easy to modify the protocol so that both clients contribute to the selection of the session key. For example, A can propose another key k', and then they use $h(k, k')$ as the session key, where $h()$ is a one-way hash function, as follows.

1. $A \rightarrow B : na, \{k1, k'\}_{Kab}$
2. $B \rightarrow A : \{B, A, nb, cb, k, \{na\}_{Kab}\}_{k1}$
3. $A \rightarrow B : \{nb\}_{h(k,k')}$

6 Using "Secret Public Keys"

In the optimal protocol in Section 3, the clients A and B must know the server's public key before protocol invocation. If the clients cannot be assumed to have this knowledge, we can add an extra round of initial exchange to obtain a "secret public key" protocol [6].

1. $A \rightarrow S : A, B$
2. $S \rightarrow A : \{Ks\}_{Ka}$

Alternatively, if the clients can generate public keys in real time, then we can use the technique in Section 5 to obtain a more efficient "secret public key" protocol, as shown in Table 7.

1. $A \rightarrow B : na, \{k1\}_{Ka}$
2. $B \rightarrow S : na, \{k1\}_{Ka}, nb, \{k2\}_{Kb}$
3. $S \rightarrow B : \{A, B, cs1, k, \{na\}_{Ka}\}_{k1}$
   $\{B, A, cs2, k, \{nb\}_{Kb}\}_{k2}$
4. $B \rightarrow A : \{A, B, cs1, k, \{na\}_{Ka}\}_{k1}, \{na\}_k, nb$
5. $A \rightarrow B : \{nb\}_k$

Table 7: An optimal "secret public key" protocol

In this protocol, $k1$ and $k2$ are public keys chosen by A and B, and na and nb are their nonces. Numbers $cs1$ and $cs2$ are confounders chosen by S. They must be independently chosen because otherwise A and B may be able to guess each other's password if a secret public key is later revealed.

Rearranging the messages can yield a four-round protocol, as done in Section 3. These protocols are thus optimal because they meet the lower bounds of the numbers of messages and rounds [4].

7 Do Timestamps Improve Protocol Efficiency?

In this section, we discuss why the use of timestamps may not help to make the protocols more efficient. In general, the availability of synchronized clocks and the use of timestamps often can reduce the numbers of messages and rounds for authentication protocols. For example, an optimal mutual authentication protocol assuming synchronized clocks uses four messages or three rounds, cheaper than nonce-based protocols [4]. Thus, a question remains as to whether the efficiency of the protocols in Sections 3, 5, and 6 can be further improved if timestamps are used.

Clearly the use of timestamps will enable the server (or a client, in a situation of direct authentication) to check if an initial request message is fresh and thus to respond only to fresh requests. However, as we have shown, the security of the protocol does not depend on the server knowing whether the request messages are fresh, even with the additional requirement that protocols be resistant to password guessing attacks. To use timestamps in later stages of the protocol does not increase protocol efficiency either, because by then all parties will have the chance to exchange nonces (piggybacked on earlier messages).

Moreover, current techniques for protecting passwords [1, 6] all require that a client, before receiving the session key, must either generate and send a public key (to the other client) or send a message to the server encrypted with the server's public key. This is equivalent, in terms of efficiency, to requiring that each client send a nonce before receiving the session key, which is in fact a security requirement for nonce-based protocols. Therefore, we conjecture that protocols resistant to password guessing attacks cannot be more efficient than nonce-based protocols, even if synchronized clocks can be assumed.

We can further argue for this conjecture from another angle. Suppose timestamps do help, and because the difference in lower bounds between timestamp-based and nonce-based protocols is only one message or one round [4, 5], then we can deduce that three messages are sufficient for both A and B to receive the session key from the server. In this case, because A has to initiate the protocol, a little analysis will show that B must receive the key in his first contact with the server. We know of no technique that achieves this goal – note that guessing attacks must not become possible after the session key is later used for handshaking or for encrypting traffic – unless we assume that S knows about a trap-door function on B's side (e.g., B's public key). An impossibility proof along this line of reasoning will be very useful.

Nevertheless, in the case with a trusted third party, if either (or both) client, instead of the server, chooses the session key, then it is not difficult to check that using timestamps can reduce the numbers of messages and rounds to the theoretical lower bounds [4].

8 Related Work

Gong et al. [6, 8] and Bellovin and Merritt [1, 2] developed the first authentication protocols that are resistant to password guessing attacks. Gong [4, 5] and Yahalom [11], among others, have investigated the design of optimal authentication protocols.

Tsudik and Van Herreweghen suggested protocol modifications in order to reduce the amount of encryption [10]. Their techniques, including their use of user-generated encryption keys, potentially can also reduce the number of messages. However, in their protocols, a client cannot know if the session key he receives from the server is correct (i.e., has not been tampered with) until he later uses it. This deficiency is nevertheless compensated by their use of extremely short messages. This trade-off appears to be intrinsic.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.