pwdArmor: Protecting Conventional Password-based Authentications*

Timothy W. van der Horst and Kent E. Seamons
Internet Security Research Lab
Brigham Young University
{timv, seamons}@cs.byu.edu

Abstract

pwdArmor is a framework for fortifying conventional password-based authentications. Many password protocols are performed within an encrypted tunnel (e.g., TLS) to prevent the exposure of the password itself, or of material for an offline password guessing attack. Failure to establish, or to correctly verify, this tunnel completely invalidates its protections. The rampant success of phishing demonstrates the risk of relying solely on the user to ensure that a tunnel is established with the correct entity.

pwdArmor wraps around existing password protocols. It thwarts passive attacks and improves detection, by both users and servers, of man-in-the-middle attacks. If a user is tricked into authenticating to an attacker, instead of the real server, the user's password is never disclosed. Although pwdArmor does not require an encrypted tunnel, it gains added protection from active attack if one is employed; even if the tunnel is established with an attacker and not the real server. These assurances significantly reduce the effectiveness of password phishing. Wrapping a protocol with pwdArmor requires no modification to the underlying protocol or to its existing database of password verifiers.

1 Introduction

In the typical password-based login (e.g., HTML form, SSH "keyboard-interactive") the user's plaintext password is sent to the server, which uses it to compute a verifier. This verifier is then compared to the copy of the verifier stored by the server and, if they match, the authentication succeeds. It is customary to establish a server-authenticated, encrypted tunnel (e.g., TLS with a server certificate, SSH transport layer) to prevent passive observation of the plaintext password as it is sent to the server and to protect the resulting session from eavesdroppers and hijackers.

*This research was supported by funding from the National Science Foundation under grant no. CCR-0325951, prime cooperative agreement no. IIS-0331707, and The Regents of the University of California.

There are several methods that attackers use to circumvent this encrypted tunnel. The simplest is to not create it and hope that the user does not notice. This is the bread-and-butter of phishers around the globe[1]. A web page is made to appear exactly like the "real" server, but it is not sent over HTTPS so the server is never authenticated to the client (as it usually would have been). An attempt to "login" to the fake site discloses the user's password to the phisher.

A variation on this attack, hereafter referred to as the *certificate trick*, involves tricking a user into accepting the wrong certificate for the real server. There are two main avenues to mount this attack. First, the attacker sends the phishing page over HTTPS[2] using a valid certificate issued to the phishing site, not the real site. As the certificate is valid for this site, no warnings are issued by the browser. Second, the attacker creates a self-signed certificate and assumes the user will ignore all browser warnings and accept this certificate as if it were from the real server[3].

SSH demonstrates the useful practice of reusing existing password verifiers; it can rely on user account information that is created and managed externally. As SSH relies on a server-authenticated, encrypted tunnel, the attacker must trick the user into accepting her public key in lieu of the real server's key in order to perform the certificate trick. If successful, the attacker learns the user's password. SSH uses a key continuity approach that informs users if the server's public key has changed since their last login.

In the context of their verifiers, password authentication mechanisms fall into two categories: password-equivalent and password-dependent. In a password-equivalent protocol (e.g., HTTP Digest authentication [11], Kerberos, EKE [7]) the server's password verifier is, for all intents and purposes, equivalent to the user's password. In a password-dependent protocol (e.g., HTTP Basic authentication [11],

[1] The Anti-Phishing Working Group (APWG) reports that 99.23% of phishing occurs over plain HTTP [4].

[2] APWG estimates that 0.28% of phishing occurs over HTTPS [4].

[3] These attacks are not limited to phishers. Cain and Able [8] is a popular tool that uses ARP poisoning to force all local network traffic to flow through the tool. It also provides an automated TLS man-in-the-middle attack using a self-signed certificate.

S/Key [14], SRP [30]) the verifier cannot be used directly to impersonate the client, as it is dependent on the plaintext password, but it can be useful to an offline guessing attack. We believe that password-equivalent verifiers have an inherent risk (especially from malicious insiders) that can and should be avoided, particularly when many of these verifiers can readily be used in a password-dependent fashion.

Password-authenticated key exchange (PAKE) protocols (e.g., EKE, SRP), do not require encrypted channels to protect the password and have the added benefit of establishing a mutually-authenticated session key that can be used to protect a subsequent session. Unlike conventional protocols within encrypted tunnels, these protocols cannot readily use existing password verifiers in a password-dependent manner (see Section 2.3). Also, current PAKE protocols do not provide privacy protection to the user's identity.

Our Contributions pwdArmor is a framework for leveraging conventional password protocols, and existing password verifier databases, to create PAKE protocols. Unlike other PAKE protocols, pwdArmor is neither password-equivalent nor password-dependent, rather it preserves this characteristic from its underlying password protocol. Also, pwdArmor can provide privacy protection to the user's identity, with or without an external encrypted tunnel.

pwdArmor treats the server authentication of an encrypted tunnel as an added bonus rather than a critical hinge of its security. Even if a user is tricked into performing the protocol directly with an attacker, the user's password is never exposed.

As a proof of concept, we used pwdArmor to wrap HTTP Basic authentication and the One-Time Password (OTP) protocol [15] (a derivative of S/Key). The client-side is realized as an extension to Firefox and also as a signed applet. The server-side is implemented as Servlet Filter in Tomcat.

Although some verifier databases contain essentially a password-dependent verifier (e.g., hash of the password), the selected conventional password protocol may use it as a password-equivalent verifier (e.g., HTTP Digest authentication, MS-CHAPv2 [31]). Using pwdArmor, and a different conventional password protocol, these verifier databases can be leveraged in a more secure, password-dependent manner, potentially eliminating the need for the original password-equivalent protocol. We demonstrate this by replacing HTTP Digest authentication with pwdArmor and HTTP Basic authentication.

Paper Outline Section 2 lays the foundation for pwdArmor. Section 3 presents the pwdArmor framework. Section 4 analyzes its security. Section 5 considers deployment issues. Section 6 discusses the prototype implementation. Section 7 examines related work. Section 8 contains conclusions and future work.

2 Foundation

In this paper, user (U) and host (H) desire to mutually authenticate and optionally establish a key that will provide forward secrecy. We assume that U has a password pwd_{U} and that H stores a verifier pwd_{U}^{ver} and associated information α, such that $pwd_{\mathsf{U}}^{ver} = Verifier(pwd_{\mathsf{U}}, \alpha)$. α contains the additional information (e.g., salt, realm, index), if any, required to create the verifier from the password.

2.1 Threat Model

This section specifies the threat model used to compare existing conventional password protocols and pwdArmor. This model defines the likely deployment scenarios, the common methods of attack, and the attackers.

Target Scenarios We target two common scenarios, which, based on the properties of their communications channels, are categorized as follows:

$\mathcal{S}_{clear}$ An unsecured channel (e.g., HTTP) is used for all communications.

$\mathcal{S}_{tunnel}$ A server-authenticated, encrypted tunnel (e.g., HTTPS, SSH) is used for all communications.

A third scenario, which adds server authentication to $\mathcal{S}_{clear}$, is the least likely to be used in practice and will not be specifically addressed in this paper due to lack of space.

Attacks Password protocols are designed such that pwd_{U} is required to impersonate U to H. Obtaining pwd_{U} constitutes a $\mathcal{PWD}$ break. As the number of potential passwords is relatively small (especially when compared to the size of keys typically used in cryptographic protocols), an attacker's ability to correctly guess the password is of particular concern. There are two approaches to password guessing: online and offline.

Online guessing attacks repeatedly invoke the protocol with H while varying the password. The best protection an online password protocol can provide is to ensure that attackers cannot obtain an advantage against the protocol greater than online guessing.

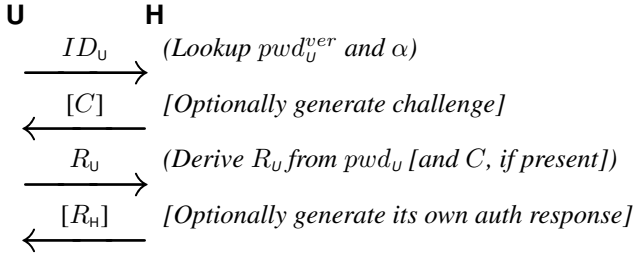
Offline guessing attacks are more efficient than online attacks, but they require verification material (i.e., the result of a known deterministic function of the password) to determine if a guess is correct. Obtaining this material constitutes a $\mathcal{LEAK}$ break, which is a step towards a $\mathcal{PWD}$ break. In practice, many organizations consider it acceptable that an attacker cannot compromise a protocol (e.g., Kerberos) with an advantage greater than offline-guessing. In these situations, services often dictate the minimum strength of user passwords.

If the attacker can insert herself between U and H she can become a *man-in-the-middle* (MITM). By simply relaying the authentication protocol's messages between the two unsuspecting parties, she can, after the authentication is complete, hijack U's session with H. This constitutes a $\mathcal{MITM}$ break, which enables the attacker to impersonate U without a $\mathcal{PWD}$ break. In practice, there are three common methods to perform this attack: 1) Routing-MITM, an attacker controls a router between U and H or has tricked traffic to route through her (e.g., ARP poisoning); 2) Pharming-MITM, DNS is poisoned so that lookups return a network address controlled by the attacker; and 3) Phishing-MITM, U is tricked into connecting to the attacker in lieu of the legitimate host (e.g., clicks a link in a phishing email).

Attackers Two attackers are considered in this model. Eve ($\mathcal{E}$) is a passive eavesdropper whose goals are $\mathcal{PWD}$ and $\mathcal{LEAK}$ breaks. Mallory ($\mathcal{M}$) is an active attacker whose goals are $\mathcal{PWD}$, $\mathcal{LEAK}$, and $\mathcal{MITM}$ breaks. $\mathcal{E}$ is limited to observing the normal interactions between U and H. $\mathcal{M}$ can observe, inject, modify, delay, destroy, and replay messages as well as create multiple concurrent sessions with any other party.

2.2 Conventional Password Protocols

Conventional password protocols consist of these logical message elements:



In the first round, U discloses her identifier ID_U and H responds with an optional challenge C, which may be dependent on ID_U. U then sends her authentication response R_U, to which H can optionally respond with its own authentication response R_H. If C and R_H are not required, then the protocol can be condensed into a single message with two logical message elements: $U \rightarrow H : ID_U, R_U$.

R_U enables U to demonstrate knowledge of pwd_U to H. Based on the characteristics of R_U (and assuming the absence of an encrypted tunnel to protect it) we classify password protocols as follows:

Type-0 R_U is always the same and therefore replayable. A challenge by the server is typically not required. Examples include responses that contain the password itself (e.g., HTML forms, SSH "keyboard-interactive").

Type-1 R_U is *not* replayable, but it can be used to mount an offline password guessing attack. A challenge by the server is required to construct R_U. Examples include conventional challenge/response protocols (e.g., HTTP Digest authentication) and some one-time password protocols (e.g., OTP [15]).

Type-2 R_U is *not* replayable and it does *not* contain material for an offline password guessing attack. Specifically, although R_U may contain some form of the password it also involves a large, unobservable, session-specific secret that significantly complicates an offline attack as password guesses must also correctly guess the value of the session secret. Examples include PAKE protocols like SRP and SPEKE [16].

An orthogonal characteristic to this classification is whether or not pwd_U^{ver} is equivalent to pwd_U (i.e., password-dependent vs. password-equivalent)[4]. Another orthogonal characteristic is whether or not ID_U is required to generate C. For example, in HTTP Digest authentication the challenge (a server-generated nonce) is independent of ID_U whereas the challenge in OTP is identity-dependent because it contains the user's seed and hash count.

In our target deployment scenarios both Type-0 and Type-1 protocols rely on the following assumption:

Assumption 1 *U and H assume that messages passed between them will not be observed, or modified, by an attacker.*

The strength of this assumption is dictated by the scenario. In $\mathcal{S}_{clear}$, this assumption equates to "security by obscurity." In practice, this is an acceptable risk for many low-value sites and services. In $\mathcal{S}_{tunnel}$, this assumption is only valid when the tunnel is correctly established with the legitimate host. Recall that phishers regularly trick users into negating this assumption.

Attacking Type-0/1 Protocols If Assumption 1 holds then $\mathcal{PWD}$, $\mathcal{LEAK}$, and $\mathcal{MITM}$ breaks cannot occur in Type-0/1 protocols. If Assumption 1 does not hold, then the following attacks are possible:

	Type-0	Type-1
$\mathcal{E}$	$\mathcal{PWD}$, $\mathcal{LEAK}$	$\mathcal{LEAK}$
$\mathcal{M}$	$\mathcal{PWD}$, $\mathcal{LEAK}$, $\mathcal{MITM}$	$\mathcal{LEAK}$, $\mathcal{MITM}$

2.3 Why not just use Type-2?

pwdArmor augments Type-0/1 protocols with the properties of a Type-2 protocol. As both pwdArmor and Type-2 protocols require new client-side software, what advantages does pwdArmor offer over a Type-2 protocol? The

[4]In PAKE protocols password-equivalent and password-dependent are called balanced and augmented, respectively.

primary advantage is that pwdArmor can reuse existing (legacy) password verifier databases while maintaining their password-dependence.

Current password-equivalent Type-2 protocols (e.g., EKE) can reuse existing verifier databases, however, doing so eliminates any password-dependent benefits these verifiers may have enjoyed with their original password protocol. Such a transition negates a significant benefit of the original protocol, since the verifier, if it is stolen, can now immediately be used to impersonate the user.

Current password-dependent Type-2 protocols (e.g., SRP) require databases of their own specialized password verifiers in order to preserve password-dependence. Relying on existing verifiers makes these protocols password-equivalent, thus negating their password-dependent benefits. Requiring new verifiers introduces a significant deployment overhead and potentially breaks compatibility with legacy systems that require the old verifier databases.

A second advantage is that pwdArmor provides optional privacy protection to the identity of the user.

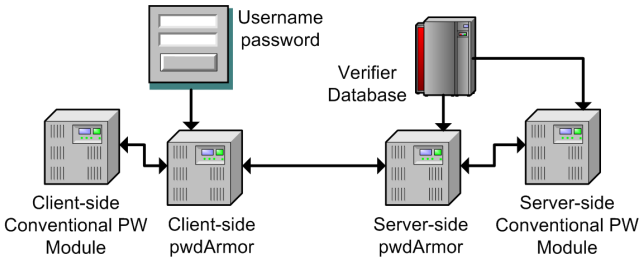


Figure 1. pwdArmor uses middleware to augment conventional password protocols. Both the server-side pwdArmor and conventional password modules require access to the verifier database. pwdArmor uses the verifier to secure the user's authentication response, while the conventional password protocol uses it to verify the password.

3 pwdArmor

Goals The primary mission of a password protocol is to ensure that pwd_{U} is required to impersonate U to H. In addition to this aim, pwdArmor is designed to meet the following goals:

1. Eliminate $\mathcal{PWD}$ breaks.
2. Limit the damage when Assumption 1 does not hold.
3. Detect MITM attacks.
4. Allow H to dictate the difficultly of $\mathcal{MITM}$ breaks.

The first goal is to limit the avenues for users to inadvertently disclose their passwords to phishers, eavesdroppers, and other attackers.

The second goal is motivated by the effectiveness of phishers in negating Assumption 1, even if H employs server-authenticated, encrypted tunnels to strengthen this assumption. As users can be easily tricked into negating a tunnel's benefits, pwdArmor uses a secure tunnel, if available, in the following capacities: 1) Improved identification of H to detect potential MITM attacks before they happen and to prevent $\mathcal{LEAK}$ breaks by $\mathcal{M}$; 2) Protect, after a successful authentication, the resulting session; and 3) Provide privacy protection to ID_{U} during the authentication.

The third goal is based in the observation that, in both $\mathcal{S}_{clear}$ and $\mathcal{S}_{tunnel}$, it is possible to detect a MITM attack by using information (e.g., domain names, network addresses, digital certificates used in a tunnel's creation) that is readily accessible to each party. Adding, or strengthening as the case may be, server authentication also helps assure users that they are communicating with the desired host and not just a phisher that accepts any password and then tries to elicit additional information from their victims.

The fourth goal captures the intuitive idea that H's choice of $\mathcal{S}_{tunnel}$ instead of $\mathcal{S}_{clear}$ should complicate $\mathcal{M}$'s ability to successfully achieve a $\mathcal{MITM}$ break. This idea is not reflected in existing tunnels (e.g., TLS, SSH) as they rely solely on users to detect MITM attacks and, as such, hosts have no say in the difficultly of $\mathcal{MITM}$ breaks.

For the purposes of evaluating pwdArmor, $\mathcal{E}$ is considered successful if she obtains $\mathcal{LEAK}$ or $\mathcal{PWD}$ breaks with a non-negligible advantage over online guessing. With respect to $\mathcal{M}$, $\mathcal{LEAK}$ breaks are acceptable and therefore she is considered successful only if she obtains a non-negligible advantage over offline guessing. Additionally, $\mathcal{MITM}$ breaks are unacceptable, with the exception of the routing-MITM attack when H uses $\mathcal{S}_{clear}$ as this attack in this scenario is virtually undetectable.

High Level Approach pwdArmor leverages middleware (see Figure 1) to wrap unmodified Type-0/1 protocols and bind them to an external secure tunnel, if present. pwdArmor encrypts U's authentication response R_{U} to ensure its confidentiality and integrity. This encrypted message also contains the identifying host information (as observed by U), hereafter referred to as ID_{H}, in order to facilitate the detection, by H, of MITM attacks. As this information is scenario-dependent, H can also detect if U is using pwdArmor in a different scenario than expected (e.g., H requires logins in $\mathcal{S}_{tunnel}$, but $\mathcal{M}$ has tricked U into using $\mathcal{S}_{clear}$).

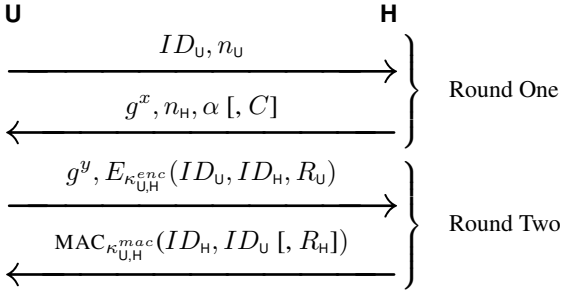
The key ($\kappa_{\mathsf{U,H}}^{enc}$) used to encrypt this message is composed of two components: 1) $\kappa_{\mathsf{U,H}}$, which is created using a Diffie-

Hellman (DH) key exchange [10]; and 2) pwd_{U}^{ver}. The use of $\kappa_{\mathsf{U,H}}$ ensures that $\mathcal{E}$ cannot be successful without compromising the assumptions of DH. As DH is subject to a MITM attack, $\mathcal{M}$ can force U and H to derive different values for $\kappa_{\mathsf{U,H}}$, each known to $\mathcal{M}$. In this event, $\mathcal{LEAK}$ breaks are possible, however, the pwd_{U}^{ver} component of $\kappa_{\mathsf{U,H}}^{enc}$ ensures that she cannot decrypt the message and obtain R_{U} or modify ID_{H}. Without modifying ID_{H}, it is unlikely a $\mathcal{MITM}$ break will be successful.

Once the authentication succeeds, then a separate key derived from $\kappa_{\mathsf{U,H}}$ is suitable for use as a mutually-authenticated key which provides forward secrecy.

3.1 Framework

The pwdArmor messages are illustrated below:



Round One U begins by submitting her identifier and a nonce n_{U}. H responds with its DH public parameter g^x and its own nonce n_{H}. H's response also includes α (how to derive pwd_{U}^{ver} from pwd_{U}) as well as the challenge, if any, supplied by the underlying password protocol.

Round Two Using the elements from H's message, U derives $\kappa_{\mathsf{U,H}}^{enc}$. Specifically:

$$\begin{aligned} \kappa_{\mathsf{U,H}} &= PRF_r(g^{xy}); r = n_{\mathsf{U}} \| n_{\mathsf{H}} \\ \kappa_{\mathsf{U,H}}^{enc} &= PRF_{\kappa_{\mathsf{U,H}}}(pwd_{\mathsf{U}}^{ver}, \text{“}enc\text{”}) \end{aligned}$$

where PRF is a family of pseudorandom functions (e.g., HMAC [25] is commonly used as a PRF). U then returns its encrypted response, along with its own DH public parameter g^y, to H. Including ID_{U} in $E_{\kappa_{\mathsf{U,H}}^{enc}}(\cdot)$ allows U to securely assert her identifier to H. As the original transmission of ID_{U} is sent without any cryptographic protection, $\mathcal{M}$ could modify it so that the user logs into a different account (this assumes that these accounts share the same password).

In the final message H demonstrates to U that it knows both pwd_{U}^{ver} and $\kappa_{\mathsf{U,H}}^{mac} = PRF_{\kappa_{\mathsf{U,H}}}(pwd_{\mathsf{U}}^{ver}, \text{“}mac\text{”})$ through the use of a message authentication code (MAC) function (e.g., HMAC). This message must be sent only if the user authentication succeeds, otherwise, it constitutes a $\mathcal{LEAK}$ break as the initiator of an authentication knows $\kappa_{\mathsf{U,H}}$. After U verifies H's message, the key $\kappa_{\mathsf{U,H}}^{other} = PRF_{\kappa_{\mathsf{U,H}}}(\text{“}other\text{”})$ can be used by other applications as a mutually authenticated key that provides forward secrecy.

Identifying H ($\mathbf{ID_H}$) The scenario determines ID_{H}:

$\mathcal{S}_{clear}$ ID_{H} = Network identifiers (e.g., domain name, network address).

$\mathcal{S}_{tunnel}$ ID_{H} = H's certificate and network identifiers.

Privacy Protection If $\mathcal{S}_{tunnel}$ cannot be used by H, but ID_{U} must be kept secret from $\mathcal{E}$, then ID_{U} can be optionally encrypted using $\kappa_{\mathsf{U,H}}^{priv} = PRF_{\kappa_{\mathsf{U,H}}}(\text{“}priv\text{”})$ before it is sent to H (this is a well known approach to providing privacy when DH is involved). There are two different situations to consider:

Challenge and α are *not* Identity-dependent ID_{U} is removed from the first message, encrypted using the key $\kappa_{\mathsf{U,H}}^{priv}$, and then added to the third message. The encrypted ID_{U} must be external to the contents of $E_{\kappa_{\mathsf{U,H}}^{enc}}(\cdot)$ as the key $\kappa_{\mathsf{U,H}}^{enc}$ is dependent on pwd_{U}^{ver}, which cannot be retrieved by H before ID_{U} is known.

Challenge or α is Identity-dependent ID_{U} is again encrypted with $\kappa_{\mathsf{U,H}}^{priv}$, and as either C or α is identity-dependent and *may* leak identifying information, both of these values are also encrypted with $\kappa_{\mathsf{U,H}}^{priv}$ before they are sent to U. Since H cannot know the correct values for C and α before it learns ID_{U}, an extra round of messages is required since H's first response cannot include these values and U cannot send $E_{\kappa_{\mathsf{U,H}}^{enc}}(\cdot)$ until these values are received.

4 Security Analysis

Attacking pwdArmor As with conventional password protocols, if Assumption 1 holds the $\mathcal{PWD}$, $\mathcal{LEAK}$, and $\mathcal{MITM}$ breaks cannot occur in pwdArmor. If Assumption 1 does not hold, then the following attacks are possible (note that both Type-0 and Type-1 protocols have the same assurances when used with pwdArmor):

	$\mathcal{S}_{clear}$	$\mathcal{S}_{tunnel}$
$\mathcal{E}$	None	None
$\mathcal{M}$	$\mathcal{LEAK}$, $\mathcal{MITM}$*	$\mathcal{LEAK}$

* Only if $\mathcal{M}$ uses a routing-MITM attack

The remainder of this section provides arguments which justify the claims made above and explores the impact of compromised session and long-term secrets.

Thwarting $\mathcal{PWD}$ breaks Neither $\mathcal{E}$ nor $\mathcal{M}$ can mount successful $\mathcal{PWD}$ breaks as the conventional password protocol authentication response R_{U} is encrypted using $\kappa_{\mathsf{U,H}}^{enc}$, which is based, in part, on pwd_{U}^{ver}. In effect, an attacker must know the password verifier before she can attack this

protocol to obtain pwd_{U}. Note that while obtaining pwd_{U}^{ver} constitutes a $\mathcal{LEAK}$ break, a $\mathcal{LEAK}$ break does not necessarily mean that the attacker has obtained pwd_{U}^{ver}.

Limiting $\mathcal{LEAK}$ breaks pwdArmor relies on a DH key exchange to generate $\kappa_{\mathsf{U,H}}$ and to prevent its passive observation. We assume the presence of pre-established, well-known, well-tested generator/prime pairs (g, p) for a specified key size (e.g., [20, 22]). This prevents $\mathcal{M}$ from injecting a pair for which she can compute discrete logs[5]. H ultimately decides on which pair to use for a specified run of the framework. As R_{U} is encrypted using $\kappa_{\mathsf{U,H}}^{enc}$, a key that is based, in part, on $\kappa_{\mathsf{U,H}}$, $\mathcal{E}$, who is limited to eavesdropping, cannot mount successful $\mathcal{LEAK}$ breaks as each password guess must also correctly guess the value of $\kappa_{\mathsf{U,H}}$.

The deployment scenario dictates the difficultly for $\mathcal{M}$ to mount $\mathcal{LEAK}$ breaks. In $\mathcal{S}_{clear}$, $\mathcal{M}$ can perform a DH-MITM attack by substituting the value of g^x sent by H with its own value $g^{x'}$. When $\mathcal{M}$ receives g^y from U she can compute $\kappa_{\mathsf{U,H}}$. Since she knows the DH component of $\kappa_{\mathsf{U,H}}^{enc}$ she can verify offline password guesses by first using the guess to compute $pwd_{\mathsf{U}}^{ver'}$, deriving $\kappa_{\mathsf{U,H}}^{enc}$, and then checking to see if the decryption of $E_{\kappa_{\mathsf{U,H}}^{enc}}(\cdot)$ contains the password guess. Note that a DH-MITM attack destroys the ability for U and H to establish the same value for $\kappa_{\mathsf{U,H}}$ and therefore relaying the encrypted R_{U} will be detected by H as it cannot successfully decrypt $E_{\kappa_{\mathsf{U,H}}^{enc}}(\cdot)$.

In $\mathcal{S}_{tunnel}$ a $\mathcal{LEAK}$ break is only possible if U creates the tunnel with $\mathcal{M}$ instead of H (i.e., succumbs to the certificate trick). In this case, $\mathcal{M}$ will be able to perform the $\mathcal{LEAK}$ break in the same manner described above for $\mathcal{S}_{clear}$.

Preventing $\mathcal{MITM}$ breaks Although a DH-MITM attack enables $\mathcal{LEAK}$ breaks, it destroys $\mathcal{M}$'s ability to mount $\mathcal{MITM}$ breaks as both U and H learn that something is amiss. H knows this because it is unable to decrypt R_{U} since its value for $\kappa_{\mathsf{U,H}}$ is different from U's value. U learns of this as $\mathcal{M}$ cannot produce a valid $\text{MAC}_{\kappa_{\mathsf{U,H}}^{mac}}(\cdot)$, due to her lack of knowledge of pwd_{U}^{ver}. Due to these factors, given a single run of the protocol to attack, $\mathcal{M}$ can choose to attempt $\mathcal{LEAK}$ breaks or $\mathcal{MITM}$ breaks, but not both.

Recall that server-authenticated tunnels (e.g., TLS, SSH) rely on users to detect MITM attacks. pwdArmor adds the ability for H to detect MITM attacks, which were missed by U, by including ID_{H} in $E_{\kappa_{\mathsf{U,H}}^{enc}}(\cdot)$. Therefore, in order to achieve $\mathcal{MITM}$ breaks, $\mathcal{M}$ must conceal its presence from both U and H. Again, the difficultly of this masquerade depends on the scenario.

In $\mathcal{S}_{clear}$ conventional phishing/pharming-MITM attacks, missed by U, should be detected by H since these attacks have difficultly masking all of the network identifiers that compose ID_{H}. A routing-MITM attack in this scenario does mask its presence effectively and therefore cannot currently be detected by pwdArmor. Note that $\kappa_{\mathsf{U,H}}^{other}$ may be used to establish a secure tunnel after the authentication completes, and this tunnel would prevent even a routing-MITM attack from achieving a $\mathcal{MITM}$ break.

In $\mathcal{S}_{tunnel}$ the entity with which U is connected is identified by the certificate used in the server-authentication of the tunnel. In order for H not to detect a MITM attack, $\mathcal{M}$ would have had to authenticate to U as if it was the legitimate H (e.g., in TLS/SSH this requires a proof of ownership of H's private key).

Compromise of Session Secrets If U's or H's DH exponent (x or y, respectively) is compromised, then $\mathcal{M}$ would be able to compute $\kappa_{\mathsf{U,H}}$ from a recorded session and therefore perform a successful $\mathcal{LEAK}$ break.

The nonces (n_{U}, n_{H}) in this protocol are used in the same manner as they are in IKEv2 [18] to create a seed key ($\kappa_{\mathsf{U,H}}$) from the result of the DH key exchange (g^{xy}). These nonces also enable U and H to reuse their DH parameters across multiple sessions (e.g., for performance reasons) while ensuring that each session is unique. Note that the forward secrecy of $\kappa_{\mathsf{U,H}}$ is maintained only between sessions where the DH values are not reused.

These nonces also introduce session-specific randomness from each participant to ensure the freshness of the keys used in this session. Also, deriving session-specific, purpose-specific symmetric keys using $\kappa_{\mathsf{U,H}}$ helps ensure that if one of these keys is ever compromised, the effects are limited to the scope of that key.

It is important to observe that since $\kappa_{\mathsf{U,H}}^{other}$ is not used as keying material in the resulting session, forward secrecy of that session is dependent on whether the underlying $\mathcal{S}_{tunnel}$ method provides forward secrecy.

Compromise of Long-term Secrets If pwd_{U} is compromised by $\mathcal{M}$, she can impersonate U until the password is changed. In $\mathcal{S}_{tunnel}$, if H's private key is stolen by $\mathcal{M}$, then she can perform $\mathcal{MITM}$ breaks, however, it does not improve her ability to directly learn pwd_{U} and obtain a $\mathcal{PWD}$ break. In pwdArmor, the ability to authenticate as the legitimate host within the context of the tunnel is only valuable in that it more effectively convinces U to initiate an authentication with $\mathcal{M}$.

If pwd_{U}^{ver} is compromised (e.g., H's verifier database is stolen), then two attacks are possible. First, $\mathcal{M}$'s possession of pwd_{U}^{ver} constitutes a $\mathcal{LEAK}$ break. Second, $\mathcal{M}$ can perform a DH-MITM attack and obtain an unencrypted R_{U}. This attack is possible since $\mathcal{M}$ now knows both private components of $\kappa_{\mathsf{U,H}}^{enc}$ and can decrypt $E_{\kappa_{\mathsf{U,H}}^{enc}}(\cdot)$. For Type-0

[5]For example, $p = 3$, $p - 1$ is composed of only small prime factors, and many cases of $p = 2^n$ [29].

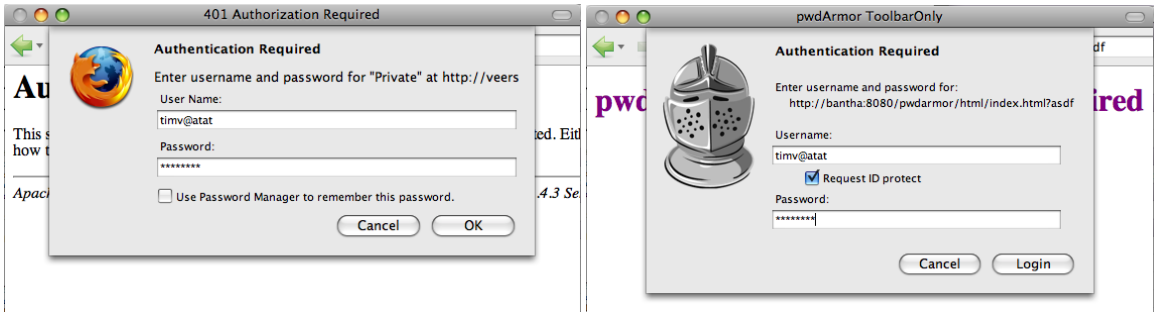


Figure 2. Modern web browsers can already prompt users for login information. The traditional HTTP Authentication dialog (left) is mimicked by our integration (right) of pwdArmor into Firefox.

protocols this constitutes a $\mathcal{PWD}$ break. For Type-1 protocols a $\mathcal{MITM}$ break is possible since $\mathcal{M}$ can re-encrypt R_{U} (and the correct value for ID_{H}) using the appropriate key as well as construct a valid value for $\text{MAC}_{\kappa^{mac}_{\mathsf{U,H}}}(\cdot)$.

5 Deployment Considerations

Client-side Support Due to the computational requirements of pwdArmor, client-side software support is essential. Existing systems (e.g., browsers, SSH clients, wireless supplicants) must be updated in order to support pwdArmor. Web browsers present an attractive avenue for incremental deployment, since, without intervention by browser vendors, client-side support for pwdArmor can be provided via browser extensions or zero-footprint clients (see Section 6). To prevent $\mathcal{M}$ from circumventing the login process, pwd_{U} must be delivered directly to the client-side software and not the server. This is not a problem for most client software (e.g., SSH clients, wireless supplicants), however, it is a significant problem for web browsers, which typically rely on a server-supplied HTML-based login page.

Entering login credentials through the browser's chrome is an attractive alternative to logins embedded in web pages because it provides a consistent authentication experience across all domains and avoids problems with malicious login pages. Modern web browsers are already capable of prompting users for login credentials (see Figure 2) when they recognize a server's request for HTTP Basic or Digest authentication. This practice, however, is not widely adopted by web sites.

Wireless Authentication Extensible Authentication Protocol (EAP) [1] provides a valuable framework for implementing and deploying pwdArmor for wireless (and also wired) network authentication in both $\mathcal{S}_{clear}$ and $\mathcal{S}_{tunnel}$ scenarios. Although EAP provides mechanisms to directly bind authentication protocols to a secure tunnel (see Section 7), this binding does not provide any guarantees if the host is not properly authenticated (i.e., the user has been tricked by a phisher). Therefore, pwdArmor is still useful as it protects against users negating the benefits of the tunnel. Support for a new pwdArmor EAP method must be added to the user's wireless supplicant software and to the network authentication server. For home users, it is preferable for pwdArmor to be supported by the access points themselves as opposed to requiring them to host a specialized authentication server attached to the access points (as it is typically done in enterprise environments).

6 Implementation

We have developed `libpwdarmor`, a general purpose library written in Java, that provides the functionality needed to build client/server pwdArmor modules. This library creates and processes pwdArmor messages in binary and human-readable formats (see Section 6.1), however, the parent application is responsible for transporting these messages to their intended destination.

Currently, `libpwdarmor` supports HTTP Basic and OTP authentication and the following verifiers:

- MD5-based BSD password algorithm (essentially, but a bit more complicated than, the password hashed 1000 times; used by most Linux distributions to create the password verifiers stored in /etc/shadow)
- Apache variant[6] of the MD5-based BSD password algorithm (used by the Apache web server to create verifier databases for use with HTTP Basic authentication)
- HTTP Digest $H(A1)$ verifier (the MD5 hash of "username:realm:password")

[6]Which only differs from the original algorithm by changing the "magic" string from 1 to apr1.

	HTTP Basic (MD5-based BSD Verifiers)	OTP	HTTP Basic (HTTP Digest Verifiers)
ID_{U}	"timv@atat"		
α	alg=apr1, salt=CGyXh...	alg=otp-sha1, seed=pongo, cnt=100	alg=http-digest, realm=Hoth
C	n/a	otp-sha1 99 pongo	n/a
R_{U}	password	AND FULL FAN GAFF BURT HOLM	password
R_{H}	n/a		

Table 1. The pwdArmor message contents for specific conventional password protocols (nonces, DH key exchange values, and encrypted values are omitted).

- MD5-based OTP verifiers (the n^{th} truncated hash of the seed and the password; these non-static verifiers are updated after each successful authentication to the $(n-1)^{th}$ hash)
- SHA1-based OTP verifiers (same as above except the result of the SHA-1 hash function is converted to little endian before it is truncated).

HTTP Basic can be adapted to use any of these verifiers, however, it only makes sense to use OTP authentication with the OTP verifiers.

`libpwdarmor` adds a user options element (O_{U}) to inform H of U's supported DH groups and password protocols as well as to request privacy protection for ID_{U}. It also adds a host options element (O_{H}) to enable H to notify U of the selected DH group and password protocol. This element also includes a session identifier that allows interaction with H over a stateless transport mechanism.

As `libpwdarmor` supports a variety of underlying protocols, $\mathcal{M}$ could manipulate O_{H} so that U will use the weakest password protocol or verifier format she supports. This has the potential to increase the efficiency of an offline guessing attack (e.g., one hash with HTTP Digest verifier vs. 1000 hashes with MD5-based BSD password verifier). A relatively time-intensive PRF could be used to limit the effectiveness of this attack.

Client-side Support We used `libpwdarmor` to create a browser extension for Firefox. This extension recognizes that a web site supports pwdArmor and transports the pwdArmor messages via HTTP headers. The extension uses a similar modal dialog box as browsers currently use to prompt users for login information (see Figure 2).

We also used `libpwdarmor` to create a signed Java Applet. As zero-footprint clients, like Java Applets, involve browsers running server-supplied code they cannot provide the same assurances as a pure client-based approach, but are attractive due to their portability. This applet is intended to be loaded from a trusted source and then used to login to any pwdArmor-enabled web site. After the applet is loaded the user enters the web site she wants to authenticate to, or selects from a previously established list, and enters her login information. After the authentication is complete the applet opens a new browser tab with the cookies it received from the host and, thus, transfers the authenticated session from the applet to the browser.

Server-side Support Server-side support is realized as a Servlet Filter in Tomcat. This filter prevents access-restricted pages from being retrieved by unauthorized users and relies on an HTTP header to indicate that pwdArmor logins are available. It also uses HTTP headers to exchange the pwdArmor messages created by `libpwdarmor`. Once authentication is successful it uses session cookies to maintain an authenticated state with the client.

6.1 Integrating with Existing Protocols

Table 1 gives examples of the specific contents of the pwdArmor messages for the following password protocols:

HTTP Basic with the Apache Variant of MD5-based BSD Verifiers In this protocol α specifies the method of verifier creation and contains the salt that is required to create the same verifier stored by H. HTTP Basic has no host-supplied challenge and expects pwd_{U}, which is sent in R_{U}.

If $\mathcal{M}$ steals pwd_{U}^{ver} from H and then tricks U into attempting an authentication to her then she will be able to use $\kappa_{\mathsf{U,H}}$ and pwd_{U}^{ver} to decrypt R_{U}. Since HTTP Basic is a Type-0 protocol, the value for R_{U}, in this case pwd_{U}, allows $\mathcal{M}$ to repeatedly impersonate U until pwd_{U} is reset.

Outfitting HTTP Basic with pwdArmor provides it with the same protections it would have normally received if it were contained within an encrypted tunnel, and adds the benefit of protecting R_{U} if that tunnel is circumvented.

OTP The value of α is equivalent, in content, to the challenge from the previous successful authentication. The challenge contains the information to use pwd_{U} to derive current one-time password, which is sent to H in R_{U}.

If $\mathcal{M}$ obtains a copy of pwd_{U}^{ver} and tricks U into attempting an authentication to her, she will be able to decrypt R_{U}. As OTP is a Type-1 protocol, the value for R_{U}, in this case the current one-time password, allows $\mathcal{M}$ to impersonate U a single time. It is interesting to note that when OTP is used normally it is vulnerable to a "small n" attack in which $\mathcal{M}$ reduces the hash index of the real host's challenge. If $\mathcal{M}$ manipulates the hash index contained in α, she will, at worst, obtain $\mathcal{LEAK}$ break since pwd_{U}^{ver} is needed to decrypt R_{U} and obtain the "small n" value generated by U. If $\mathcal{M}$ obtains a copy of pwd_{U}^{ver}, the small n attack is still not feasible as the value for pwd_{U}^{ver} that U will generate will not correspond to the stolen pwd_{U}^{ver} as their respective hash indices are not equal.

HTTP Basic with HTTP Digest Verifiers The algorithm and realm specified in α enables U to generate the $H(A1)$ HTTP Digest password verifier. Again, HTTP Basic has no host-supplied challenge and expects pwd_{U}.

pwdArmor provides HTTP Basic with the same protection from $\mathcal{PWD}$ breaks as HTTP Digest, without requiring password-equivalent verifiers. In this approach pwd_{U} is hashed and then compared to $H(A1)$, allowing what were password-equivalent verifiers to become password-dependent. If HTTP Digest is still employed elsewhere using these same verifiers, then they remain password-equivalent for those HTTP Digest authentications.

7 Related Work

As is evidenced by the success of password phishing, the assurances provided by encasing conventional password protocols within encrypted tunnels (e.g., TLS) can be easily circumvented by tricking the user. The key problems with encrypted tunnels are the difficultly for users to: 1) Determine if the host authentication ever occurred; and 2) Correctly verify the identity of the server. Several solutions have been proposed to improve the authentication of H by U. Visualization techniques [27] help improve verification of public keys. Additional human perceptible server authenticators (e.g., pictures, voice) [12] may also help users. Other systems, like BeamAuth [2], use bookmarks to ensure that logins only occur with legitimate sites. These solutions do not address the lack of creation of the tunnel. pwdArmor provides the same passive/active protections as encrypted tunnels, along with the additional benefit that if the host authentication is circumvented then pwd_{U} is not leaked.

Delayed password disclosure (DPD) [17] authenticates H without a secure tunnel by requiring U to verify the correctness of a host-supplied image after each character of pwd_{U} is entered. A *distinct* oblivious transfer for *each* password character ensures that actual password character is not disclosed. U then authenticates via a traditional PAKE protocol. Unlike pwdArmor, DPD is designed to thwart static phishing sites and does not protect against MITM attacks. Also, pwdArmor maintains the current practice of allowing users to quickly submit their usernames and passwords.

The need to couple an inner authentication protocol with its outer tunnel has been previously examined as the *compound authentication binding problem* [5, 28]. The solution proposed (and adopted by EAP [1]) requires that the resulting session key be derived from: 1) The tunnel key; and 2) A key created by the inner authentication protocol or from pwd_{U}. Although this binding prevents MITM attacks, the tunnel must be modified and the password protocol is not protected if the tunnel is incorrectly established.

IKEv2 [18] does not directly address the tunneling of Type-0/1 protocols, which it calls "legacy authentication" mechanisms, but does supports the optional binding of itself to EAP, provided the underlying EAP method produces a key. As with EAP, IKEv2 does not provide any protections against phishers if the tunnel is incorrectly established.

Oppliger et al. [26] couples a TLS session to a specific authentication through a hardware/software token. Client-side certificates (from the token) are used in the TLS handshake to prevent MITM attacks, not for client authentication. pwdArmor accomplishes the same goals by using nonces to ensure unique sessions and by using ID_{H} to ensure a tight coupling with the encrypted tunnel.

Halevi and Krawczyk [13] specifies a generic, password-based, encrypted challenge-response protocol and an instantiation that provides mutual authentication and key exchange. Even though this protocol encrypts pwd_{U} with H's public key, if the phisher successfully performs the certificate trick then the benefits of this protocol are completely negated. Also, the requirement that H have a public key pair may be unreasonable for the low-value services that typically operate in $\mathcal{S}_{clear}$ scenarios.

PAKE protocols are the strongest answer, to date, for the troubles with password-based authentication through encrypted tunnels. There are a variety of protocols: SRP [30], SPEKE [16], EKE [7], PDM [19], SNAPI [24], PAK [23], AuthA [6], AMP [21]. If executed correctly, these protocols are not vulnerable to $\mathcal{PWD}$, $\mathcal{LEAK}$, or $\mathcal{MITM}$ breaks. As pwdArmor only provides all of these assurances if U does not fall victim to a certificate trick, PAKE protocols provide superior benefits to pwdArmor, with two exceptions. First, and most significantly, is their inability to reuse existing password verifier databases as detailed in Section 2.3. Second, is their lack of privacy protection for ID_{U}. With the exception of SNAPI, this protection cannot be easily added to these protocols as their DH-based key exchanges are tightly coupled to ID_{U} (and as such cannot be used to privately transmit ID_{U}) and they do not require H to have a public key pair.

8 Conclusions and Future Work

pwdArmor is a more secure alternative to the current practice of encasing conventional password-based authentication protocols within a server-authenticated, encrypted tunnel. The key benefit of pwdArmor is evident when users think they are authenticating to one of their legitimate service providers, but are, in fact, being phished. In this scenario, all the benefits of using an encrypted tunnel can be negated by users, whereas in pwdArmor, users' passwords are never disclosed. This advantage comes from pwdArmor's treatment of server authentication as an added bonus, rather than the linchpin of its assurances.

Unlike other PAKE protocols, pwdArmor can reuse existing verifier databases while maintaining the password-dependent nature of those verifiers. Reuse of existing verifiers is valuable as it allows simplified deployment, avoids the overhead of creating new verifiers, and preserves compatibility with legacy systems. pwdArmor also, unlike other PAKE protocols, offers optional privacy protection to the user's identity during the authentication.

Operating system utilities (e.g., CardSpace [9]) for managing and using login credentials have the potential to unify user authentication across a variety of mediums (e.g., web site, wireless network, local application logins). As such, they represents an attractive avenue for deploying the client-side functionality required by pwdArmor.

Assuming the presence of HTTPS only for login pages, SessionLock [3] secures resulting web sessions from passive eavesdropping without TLS. A combination of pwdArmor and SessionLock, could eliminate the need for HTTPS for a large number of low-security web sites and therefore remove the additional costs associated with the performance overhead and caching behavior of TLS.

References

[1] B. Aboba, L. Blunk, J. Vollbrecht, and J. Carlson. RFC 3748: Extensible Authentication Protocol, June 2004.

[2] B. Adida. Beamauth: Two-factor web authentication with a bookmark. In *ACM Conference on Computer and Communications Security*, October 2007.

[3] B. Adida. Sessionlock: securing web sessions against eavesdropping. In *International Conference on World Wide Web*, April 2008.

[4] APWG. Phishing Trends Reports, January 2008. Available from http://anti-phishing.org.

[5] N. Asokan, V. Niemi, and K. Nyberg. Man-in-the-Middle in Tunnelled Authentication Protocols. In *Security Protocols Workshop*, 2003.

[6] M. Bellare and P. Rogaway. The AuthA Protocol for Password-Based Authenticated Key Exchange. Submission to IEEE P1363, February 2000.

[7] S. Bellovin and M. Merritt. Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks. In *IEEE Symposium on Security and Privacy*, 1992.

[8] Cain and Able. Available from http://www.oxid.it/cain.html.

[9] CardSpace. http://msdn.microsoft.com/CardSpace.

[10] W. Diffie and M. E. Hellman. New directions in cryptography. In *IEEE Trans. on Information Theory*, Nov 1976.

[11] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. RFC 2617: HTTP Authentication: Basic and Digest Access Authentication, June 1999.

[12] S. Gajek, M. Manulis, A. Sadeghi, and J. Schwenk. Provably secure browser-based user-aware mutual authentication over TLS. In *ACM Symposium on Information, computer and communications security*, 2008.

[13] S. Halevi and H. Krawczyk. Public-key cryptography and password protocols. *ACM Transactions on Information and System Security*, 1999.

[14] N. Haller. The S/KEY one-time password system. In *Symposium on Network and Distributed System Security*, 1994.

[15] N. Haller, C. Metz, P. Nesser, and M. Straw. RFC 2289: A One-Time Password System, Feb. 1998.

[16] D. P. Jablon. Strong password-only authenticated key exchange. *SIGCOMM Comput. Commun. Rev.*, 1996.

[17] M. Jakobsson and S. Myers. Delayed Password Disclosure. *SIGACT News*, 38(3):56–75, 2007.

[18] C. Kaufman. RFC 4306: Internet Key Exchange (IKEv2) Protocol, December 2005.

[19] C. Kaufman and R. Perlman. PDM: A New Strong Password-based Protocol. In *USENIX Security Symposium*, 2001.

[20] T. Kivinen and M. Kojo. RFC 3526: More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE), May 2003.

[21] T. Kwon. Authentication via Memorable Password. Submission to IEEE P1363, May 2000.

[22] M. Lepinski and S. Kent. RFC 5114: Additional Diffie-Hellman Groups for Use with IETF Standards, Jan 2008.

[23] P. MacKenzie. The PAK suite. Submission to IEEE P1363, May 2002.

[24] P. MacKenzie and R. Swaminathen. Secure Network Authentication with Password Identification. Submission to IEEE P1363, August 1999.

[25] NIST. The Keyed-Hash Message Authentication Code (HMAC). FIPS Pub 198a.

[26] R. Oppliger, R. Hauser, and D. Basin. SSL/TLS session-aware user authentication – Or how to effectively thwart the man-in-the-middle. *Computer Communications*, 29(12):2238–2246, August 2006.

[27] A. Perrig and D. Song. Hash Visualization: A New Technique to Improve Real World Security. In *Intl. Workshop on Cryptographic Techniques and E-commerce*, 1999.

[28] J. Puthenkulam, V. Lortz, A. Palekar, and D. Simon. Internet Draft: The Compound Authentication Binding Problem. draft-puthenkulam-eap-binding-04.txt, October 2003.

[29] B. Schneier. *Applied Cryptography, 2nd Edition*. John Wiley & Sons, Inc., New York, NY, 1996.

[30] T. Wu. The Secure Remote Password Protocol. In *Network and Distributed System Security Symposium*, 1998.

[31] G. Zorn. RFC 2759: Microsoft PPP CHAP Extensions, Version 2, January 2000.