RC23193 (W0404-133) April 23, 2004
Computer Science

IBM Research Report

Secure Client-Managed Authentication: A Passport-free Solution

Reiner Sailer, James Giles, Anca Dracinschi Sailer
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Secure Client-Managed Authentication: A Passport-free Solution

Reiner Sailer, James Giles, and Anca Dracinschi Sailer
IBM T.J. Watson Research Center
Hawthorne, NY 10532, USA
Email: {sailer, gilesjam, ancas}@watson.ibm.com

***Abstract*—This paper presents a novel authentication service that enforces security by assisting the management of the overwhelming and constantly increasing collections of user identifiers and passwords. As the number of these authentication credentials (i.e., userid and password) increases, maintaining and recalling them on demand becomes a challenge. Studies show that users typically choose the same easy-to-guess password for multiple services and store it unprotected. This behavior implies that credential leaks within poorly protected services can compromise or disrupt better protected critical services.**

The new secure client-managed authentication service proposed in this paper is suitable for a large spectrum of applications, including Internet Services and network management services. Our main contributions are (1) the delegation of credential management to a local secure agent while keeping the users in control of their credentials, (2) a three-level user control of credential release, and (3) generality, i.e., allowing secure credential release to authorized server applications without requiring client application or operating system modifications. Offering a key differentiation to centralized solutions such as Microsoft Passport, our authentication service empowers users to control the release of their identity and related credentials on demand. We compare the performances of our prototype (fully functioning implementation) to those of a conventional user authentication service and we show that our prototype is faster and easier to use.

***Index Terms*— Access Control, Authentication, Security Management**

I. Introduction

The problem of securely maintaining authentication credentials as considered in this paper is motivated by the increasing attention focused on security issues in applications and communication networks. The proliferation of user identifiers and passwords required for authentication purposes by new electronic services confirms that providers of these services are more and more concerned about the security of their applications and networks. Therefore, the secure maintenance of the authentication credentials becomes of major importance.

In the case of the electronic services, studies [1], [2], [3], [4], [5] have shown that users typically avoid the inconvenience of having to remember different credentials by choosing simple, easy-to-guess passwords for multiple services, by storing passwords insecurely, and by choosing default passwords (see for example the recent password-guessing virus [6]). While this behavior results in weakened security, related problems are compounded if some services offer less secure methods for credential exchange than others. Hence, a leak of a user's credential from one service allows an attacker to replay it when trying to access another service. More cautious users would like to avoid giving the same easy-to-guess passwords for multiple services. However, the key issue is how to conveniently and securely manage a large number of credentials authorizing the access to numerous services.

In this paper we develop a new authentication service that addresses this need for a secure solution to maintain authentication credentials. The proposed service is a client-based solution that allows users (e.g., Internet users, network administrators, network management tools) to maintain control over their credentials for privacy and accountability. It is also autonomous in the sense that it does not require changes to the operating system or client applications making use of its service. Note that other approaches to secure authentication have been proposed in the literature, by using a third party centralized server like Microsoft Passport [7] and Liberty Alliance [8], by using a client-based agent as Gator [9], Freedom [10] or Factotum-Plan 9 [11], or by using client certificates on behalf of the user. However, all these approaches have limitations in terms of either user control over personal information in the case of the server approaches, application/system compatibility in case of client-based agents, or management overhead and complexity in the case of client certificates.

Our approach takes into account all of the above issues by providing seamless backward compatibility with previous authentication mechanisms, by providing simple client and server migration paths, and by supporting multiple types of network services. We choose a client-centric approach rather than the server-centric approach of passport and Liberty Alliance so that users have more control over releasing their identity and credentials. Additionally, denial-of-service attacks against decentralized authentication services are less profitable and more difficult than against centralized credential servers.

As a last remark, security enhancements to ensure that unauthorized users cannot disrupt critical communication between system components or network devices is a vast area of research that, besides authentication, includes encryption, filtering, registration access control lists, etc. The overall security of a system depends on its architecture, implementation, and operation; security issues can exist in any of these. In this paper we focus on secure authentication and discuss its architecture, implementation, and operational issues.

The rest of the paper is organized as follows: We first outline in Section II the key ideas and requirements for the secure authentication design. Then, in Section III, we describe in detail the components of our architecture, their implementation, options and features, and how to use the service in order to achieve the desired authentication security level. Section IV is dedicated to performance studies where we show that our solution performs very well, and is faster and easier to use compared to a conventional user authentication service. Section V presents a comparison to existent authentication models. We conclude in Section VI.

II. Key Ideas and Design Requirements

In this section, we give a description of the key ideas used in the proposed authentication solution, the requirements to ensure security, and the different steps of the operational functionality. The main objective is to assist the management of authentication credentials and authentication tasks and, in the same time, improve the consistency of the security protection provided during credential exchanges without requiring modifications in the operating systems or client applications. A user's credentials are managed by a *client authentication service* on his or her electronic device. Before any credentials will be released, the user must first log into the client authentication service on the user's device. Whenever a server needs to authenticate a user, it requests credentials from the client authentication service at a well-known port. The client authentication service securely correlates this request for credentials with the user's active sessions to determine which credentials, if any, should be returned to the *system requiring authentication*. The architecture of the proposed authentication service is illustrated in Figure 1.

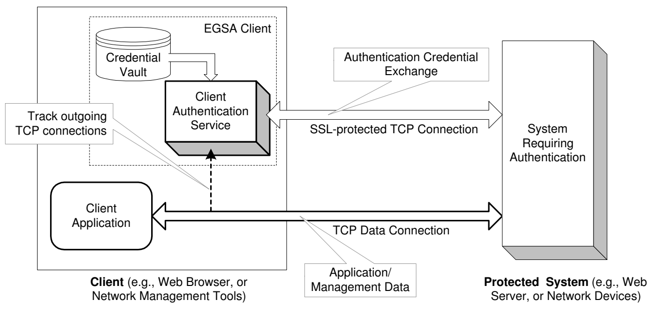
EGSA Client
Credential Vault
Client Authentication Service
Track outgoing TCP connections
Authentication Credential Exchange
SSL-protected TCP Connection
System Requiring Authentication
Client Application
TCP Data Connection
Application/ Management Data
Client (e.g., Web Browser, or Network Management Tools)
Protected System (e.g., Web Server, or Network Devices)

Fig. 1. Architecture of the Proposed Secure Authentication Service

There are two important requirements to ensure system security with our solution. Foremost, user credentials should only be released securely to authorized services at appropriate times. Second, data streams must be tied to appropriate authentication credentials. Therefore, a cornerstone of our solution is the correlation of credential requests with a user's data connection and authorization of credential releases. The proposed authentication passively tracks a user's outgoing data connections, noting which servers and services are being actively used. When a service requests a credential, it does so over an encrypted connection (e.g., SSL) so that (a) the credential is protected regardless of the protections for the data stream, and (b) the server can be securely identified. The overall operational process is illustrated in Figure 2.

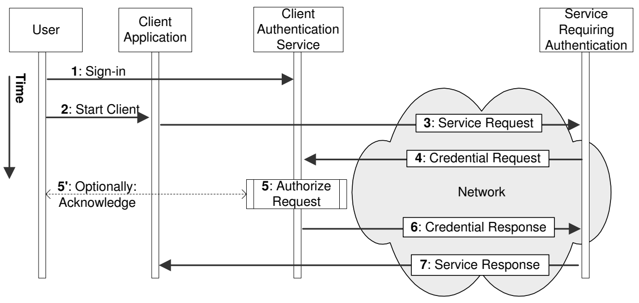

Fig. 2. Operational Functionality of the Proposed Secure Authentication Service

Once the user has activated the client authentication service by logging into it (1) and starts requesting remote services (2 and 3), the service uses the server certificate (4) and its record of active, client-initiated data sessions to determine (5) the credentials the server may be authorized to receive (6). Optionally (only for the maximum protection among the three user interaction preferences), the service can check with the user interactively (5') before sending any credential to a service (6). Finally, when the user has been successfully authenticated, the service sends the requested response (7).

Threat Model. Our focus is to defend against attacks involving the theft of user credentials. Our service does not defend against trusted services, but only against malicious services that can easily divulge and misuse any user's credentials for that service. Thus, it ensures that credentials are securely stored on the user's device and that authorized services can obtain credentials from the client authentication service. The proposed authentication also protects against disclosure of credentials to an eavesdropping attacker. However, attackers that obtain server certificates may be able to masquerade as a server gaining access to user credentials. Our service also does not defend against vulnerabilities and exploits in the client specific operating system: an attacker gaining sufficient privileges on a client system could make changes to the network subsystem or user interface that could circumvent some of our solution's protections. Our service copes with Denial of Service (DoS) attacks by detecting invalid authentication requests as soon as possible. As we will briefly describe in the next section, only authentication requests from servers to which the client has an open outgoing connection are considered valid; other requests are rejected before computing-intensive operations are started (e.g., SSL certificate validation).

III. ServiceFeatures and Implementation

In this section, we describe in more detail the four main components of our service: the credential vault, the client authentication service, the server authorization mechanism, and the user control. Before we describe these components in detail, we shall sketch their cooperation that implements the service. The client authentication service starts as a thread as soon as the user signs in successfully. To sign in, a user selects a credential vault file and enters the correct user identity and password to decrypt the vault file. The decrypted content of the vault file represents the credentials that are used by the authentication service to respond on behalf of the signed-in user to remote credential requests. This file is especially sensitive as it contains all the credentials of the user. In Section IV we describe in more detail how we particularly protect those credentials. The client authentication service thread is stopped when the user signs out of the service. The graphical user interface (GUI, c.f. Appendix I, Figure 6) allows the user to edit and save the credential file.

A. Credential Vault

The client authentication service needs access to a user's credentials (e.g., user identity and password) to respond on behalf of this user to authorized credential requests from remote services. Therefore, the first task of the user is to sign into the authentication client application to enable it to retrieve the user credentials from the vault file and decrypt them.

A user signs into the client by proving his or her identity, enabling at the same time the client application to decrypt the selected credential vault. For this purpose, the user identity and the password are hashed into a decryption key for the vault. If user identity and password are correct, the vault decrypts into the credential file and exhibits the proper format; if user identity or password are wrong, then the file decrypts into some file that does not satisfy the syntax rules for the credentials, and which makes the sign-in fail.

The vault password is not stored in the system and is only available throughout the sign-in phase to decrypt the vault content. We call the decrypted vault content a *profile*. A profile consists of user credential entries, each of which is structured as illustrated in Figure 3.

```
PROFILE[user@work]
  CREDENTIALENTRY {
       SERVER=www.intranet.ibm.com:
       SERVICE=HTTP:
       REALM=ANY:
       PROXY=proxy.intranet.ibm.com:
       CREDS=me@us.ibm.com:passw0rd:
       PROTECTION=SSL:  }
  CREDENTIALENTRY {
       SERVER=www.esorics.org:
       SERVICE=TELNET:
       REALM=ANY:
       PROXY=NONE:
       CREDS=admin:s3cre7:
       PROTECTION=:  }
  ...
ENDPROFILE[user@work]
```

Fig. 3. Example Profile

Each credential entry contains the *server* name and *realm* field. Those fields together should specify unambiguously an entry. The realm field offers an additional granularity if a user needs to store multiple credential entries for a single server; it can also be empty. The *service* entry distinguishes credentials for different services offered by the same server.

The *proxy* field can either be empty (NONE) or specify a proxy server that is expected to request credentials on behalf of the server providing the service. Running a proxy server represents an easy way to enable the services without the need to change the actual server, but it must be as trusted as the original server. The client authentication service responds with the contents of the *creds* field to authorized credential requests.

The credential exchange itself is always independently protected; however, the *protection* field enforces protection of the data exchange between the user client (web browser) and the server (web server). This field—if used—adds requirements to the authorization of a request.

The user profile is read completely into a hash-table in non-persistent memory. The key to the hash table is the proxy entry if available, or the server entry otherwise. The client authenticator searches this hash-table to find the requested credentials.

B. Client Authentication Service

The major purpose of the client authentication service is to respond to an authorized server request with the proper user credentials allowing the signed-in user to use this service. For this purpose, the client listens on a well-known port (our prototype uses port number 10000) on the client system for credential requests from servers. A server must set up an SSL connection to the well known client port and authenticate against the client using a valid certificate. As clients do usually not run a web server, the authentication service can optionally listen on port 80 or 443 as well, taking advantage of the firewalls configured to allow traffic only through port 80 and 443.

The pseudo-code in Figure 4 shows how the client authentication service handles credential requests from server applications. After signing in the user and retrieving his or her credentials from the vault file (the *if* condition holds), the client authentication service continues to handle requests until an exit condition holds. At this time, the user is signed out and the client authentication service stops. The main loop (the *while* loop) accepts an incoming SSL connection and parses the incoming request. If the request has a valid syntax, and the server certificate—used for authentication throughout the SSL connection establishment phase—is valid, then the request is accepted.

C. Server Authorization Mechanism

With the credential request, the server provides the client port number belonging to the TCP data connection for which the credential request was issued (and in case of HTTP optionally the original URL that triggered the credential request), the server certificate (implicitly through SSL authentication), and optionally the realm for the authentication. A *credential request is considered valid* if it is received from an authorized server (or proxy). A server is considered authorized if (i) there is a pending request to this server from this client, (ii) there is a credential entry

```
ClientAuthenticationService() {
    /* Authentication Client Logic */
    if (!sign_in_user())
        return;
    while(!EXIT) {
        request = SelectNextRequest();
        if (!ParseSyntax(request)) {
            reply(error); continue;
        }
        if (!Authorized(request)) {
            reply(error); continue;
        }
        creds = FindCreds(request);
        if (!UserApproved(request)) {
            reply(error); continue;
        }
        reply(creds);
    };
    sign_out_user();
}
```

Fig. 4. Client Logic of the Proposed Authentication Service

in the active profile whose server or proxy name matches the server's name in the SSL certificate, and (iii) the *protection* requirement field of this credential entry matches the protection type of the TCP data connection between the client and server. The order of the steps ensures that invalid authentication requests are discarded as soon as possible, while the lightweight mechanism that correlates incoming requests with outgoing connection data thwarts computing power based DoS attacks.

i. *Correlating client request:* To validate (i), the client maintains one hash-table that stores all active TCP connections that were initiated by the client system. Our service monitors the setup and release of TCP connections using the packet capture library for Windows [12]. To maintain the table, the service starts a separate thread for each network interface that has an IP address (*client IP*) and compiles and sets the PCAP packet filter rule detailed in Figure 5.

```
(((tcp[tcpflags]&(tcp-fin|tcp-rst)!=0) or
  ((tcp[tcpflags] & tcp-syn!=0) and
   (tcp[tcpflags] & tcp-ack=0))
 )
 and (src 'client IP'))
```

Fig. 5. PCAP Filter Rule

The authentication service considers that the client has initiated a TCP connection to the server if the TCP connection hash table contains an entry with the client source port as stated in the credential request and destination IP address equal to the server IP address. This client port number is not used for authentication purposes, but to distinguish between multiple client requests to the same server. On multi-user operating systems, the client port number of the request could be used to determine the user identifier under which the client application is running; this could be used to determine whether this user identifier is the one under which the client is running. Requirement (i) ensures that there is always a correlated client request to a server credential request.

ii. *Matching credential entry:* We validate (ii) by searching the profile data for a credential entry whose server or proxy field matches the server name of the SSL certificate that was used by the server to establish the SSL connection to the client authentication service. Using the client port number as submitted within the credential request, the service can find the server port number of the client connection in the TCP hash-table (see validation step i) and the client application. Thus, our solution can distinguish credential requests for multiple client applications accessing the same server. Additionally, we can assume well-known port numbers to determine the service entry of the requested credential by looking up the server port in the TCP connection data. If this does not unambiguously determine a credential, the service asks the user which entry to use, i.e., which user identity to authenticate. As an enhancement, the server certificate can be checked against a local or remote certificate revocation list. Our authentication service lends itself as well to using client certificates for authentication. In this case, the SSL server authentication required for an authentication request is extended to include an SSL client authentication based on the related client private key, and to pass the client certificate of the public key instead of password credentials to the server.

iii. *Matching protection policy:* We validate (iii) by retrieving additional connection characteristics from our TCP connection hash table. Our prototype supports the protection policies ANY and SSL. If the protection field reads SSL, our authentication service validates whether the outgoing TCP connection that proves the correlation in requirement (i) is using SSL. The service currently supports the validation of SSL for HTTP services simply by checking for the well-known HTTPS server port (443). This is a heuristic that shall protect from accidentally using HTTP instead of HTTPS, and not from more sophisticated attacks.

D. User Feedback Control

Before any credential reply is sent to the server, the authentication client checks whether the user needs to acknowledge the authentication. This allows fine-grained user control regarding the release of credentials. The client offers three interaction levels to its user: Low, Medium, and High. An interaction level set to Low indicates that the user asks the service to autonomously authenticate the user if possible; i.e., if credential entries for a requesting service are unambiguous. The High interaction level indicates that the user wants control over all authentications that are handled by the service. Each time a credential request is received, the user is prompted regarding whether or not the credentials shall be sent to the server; the user is thereby additionally informed about unauthorized requests. The Medium interaction level enables user feedback and confirmation for the initial authentication against a service. Subsequent authentications within the same session for the same service are autonomously satisfied by the client authentication service. A session ends when the TCP connection (with which the initial request was correlated) ends. For Web browsers using multiple TCP connections to access a Web server, experience shows that it is more appropriate to relate the end of the session with the exit of the client application, the user's sign-out, or a global timeout (whichever occurs first) rather than with the end of individual TCP connection.

Each credential entry in the profile can overwrite these GUI settings by including an *interaction* field in the entry. Thus, the decision whether to pop up a user confirmation window before replying to a credential request is done based on the matched credential (dominant) and on the GUI setting. Independently of the user interaction level, the activity indicator at the upper left corner of the authentication GUI visualizes the processing of credential requests by our client authentication service (see Appendix I, Figure 6).

The authentication client is programmed in C and comprises about 1600 lines of code, including the GUI, the client authenticator, and the handlers for the credential vault. We implemented the client as an application for the Windows 32-bit architecture (Windows 2000, Windows XP). We use standard Microsoft libraries [13] and the Packet Capture (PCAP) library for Windows operating systems [12]. Our prototype processes user identities and passwords as user credentials, and it can be easily extended to include certificates and one-time passwords.

IV. Evaluation and Performance Studies

In this section, we evaluate the proposed authentication solution. The performance objectives that we consider are security, overhead, compatibility, and ease of use.

A. Security

Our security goal is that credentials (c) are released (Rel) to authorized servers only. A server (s) is authorized (A) regarding a request (r) if and only if it has a client request pending that requires authentication and if its identity is proven. The released credentials are those that are found (F) in the users profile for this server and request. They are released only if the user approves (UA) to release respective credentials. Formally, our design and implementation ideally enforces:

$$Rel(c, s) \ \rightarrow\ A(r, s) \ \&\&\ (c == F(r, s)) \ \&\&\ UA(c)$$

We do not try to satisfy the $\leftrightarrow$, i.e., the reciprocal relationship, because it is not possible to prevent situations where all assertions are fulfilled and still a reply cannot be sent. For instance, this could happen if the client memory or CPU is exhausted or the network connection fails. Thus, it is not possible to guarantee availability of service. However, proving the above assertion holds in our implementation merely requires an analysis of the flow control of our application (see Section III-B). Therefore, it is more interesting to examine whether the predicates reflect the real-world assumptions, e.g., that $A(r, s)$ holds if and only if a server is actually authorized. For this reason, we focus on analyzing the implementation of the predicates when we evaluate possible limitations of our solution with regard to the above assertion in the context of attacks that exploit vulnerabilities of (1) the architecture, (2) the implementation, (3) the network, and (4) the client system.

1. *The architecture:* Correlating a client's service request with a server's credential request (this is determining the $A(r, s)$ predicate) constitutes a major problem when trying to separate service and authentication. Our solution solves this problem by ensuring that the client has initiated an open TCP connection to the server (for the initial request). Additionally, the server identity is verified using the SSL certificate that the server uses to authenticate. Ideally, we would like to also verify that the client request actually requires authentication. However, our service cannot always read (and thus verify) the client request because it might be protected by SSL or other end-to-end encryption. In this case, our authentication service validates the remote SSL endpoint; actual authentication requests are only accepted if they originate from there. Users are made aware that the authentication client cannot validate whether authentication is necessary for this request and that they must decide whether to go ahead with the authentication or not. Even in this case, a valid server must supply the client port number of the open connection in order to "get through" to the user. Nevertheless, with our solution this is the only restriction, while existing models (see Section V) cannot even securely determine if there is an initial client request for the server that demands authentication. To make sure that the user is aware of the authentication (and of the following authenticated actions), our service provides both a permanent indicator and a user approval option. As the credential exchange is protected by client-server authenticated and encrypted SSL, the released credentials can be disclosed by the server only. Some of the above correlation considerations could be overcome by having the client authentication service in the data path between the client and the server. However, this would destroy the client-server end-to-end relationship, especially by using SSL or TLS, and, additionally, would make the client authentication service platform-specific, which our solution avoids by providing generality and broad applicability.

2. *The implementation:* A major known implementation problem is that of programming errors, which can lead to security breaches. The service implementation comprises about 1600 lines of code, not including the shared libraries. Extensive studies showed [14] that even in well-tested code, there remain about 3-8 programming errors per 1000 lines of code. Hence, we must assume that any implementation of the size of our prototype may include multiple errors. However, it is more effective to evaluate and securely implement *one* single authentication service as the proposed solution, rather than *all* different specific authentication clients for applications and network management services needing authentication credentials today (HTTP, Telnet, Secure Shell).

3. *Network:* We protect credentials when they are transmitted by SSL, independent of the protection of the service data connection (e.g., HTTP, telnet). When authenticating the server using the server's trusted SSL certificate (and the ASCII name given therein), our service will not be deceived by spoofing or DNS attacks—this ensures that the $F(r, s)$ predicate holds. Our service can additionally check certificate revocation lists before authorizing a server. However, our solution is susceptible to denial of service attacks by any server because setting up the SSL connection requires CPU processing power for public key authentication. We mitigate this risk by validating that the client has an outgoing connection to a server before going through the SSL authentication for this server's credential request.

4. *Client system:* The credential vault is represented by

a flat encrypted file. We use the Triple-DES algorithm and a 128-bit key that is derived by hashing the sign-in user identity and password (optionally including a salt value). This file can be stored on a USB token and accompany the user. The hardware token can be write-protected by a hardware switch, which protects against integrity attacks. Additionally, we can store the authentication client application on the USB token as well because the code size of our implementation is rather small. Therefore, the token content cannot be corrupted. Noteworthy is also that a hardware token need only be attached to the client system when the authentication service is used and this restricts the time frame for disclosure attacks.

The authentication service stores multiple credentials within the authentication client application. This may seem attractive to attackers. However, compared to network-centered approaches that store credentials for hundreds of users, our service is of small interest to attackers. Nevertheless, users can mitigate the exposure of the service by using multiple profiles (such as @work and @home, or @MPLS_core and @MPLS_edge) allowing them to make only certain credentials available depending on the context.

Finally, the credentials are stored in clear-text in the main memory between sign-in and sign-out. If they are swapped out onto persistent memory, they are easily exposed. We are currently exploring possibilities to mark the memory that holds the credentials as non-swappable or to protect memory that is swapped out [15]. We also catch signals sent to the authentication client and the respective signal handlers overwrite the memory area holding the clear-text credentials before exiting or at sign-out time—this does not help in case of power failures or purposely manipulated client systems.

B. Overhead

The main benefit of our solution resides in its support to facilitate and secure the credential management for the user. In this section our goal is to additionally determine the cost of using the proposed service in terms of user perception as compared to the current legacy. We first qualitatively analyze the service overhead in terms of extra computation and communication required by the system. These two performance metrics reflect the time-impact of using the service from the user's perspective. Based on this study we have proceeded to test the prototype system. We provide here the most relevant performance results that we have obtained.

Qualitative Analysis. The first component of overhead is the additional processing required by the client authentication service. The contribution of this overhead is relatively small, since the client authentication service is only active when a service makes a credential request, or when the client sets up outgoing TCP connections. When handling a credential request, the client authentication service accepts the SSL connection, checks the server certificate, parses the request, looks up appropriate credentials, verifies that there is an active TCP data connection, and then optionally asks the user if the credentials should be sent. Accepting the authenticated SSL connection contributes most to the overhead introduced by our solution. For legitimate requests, checking for credentials and checking for active connections is rather fast even for a large number of credentials and active connections (e.g., in the case of network management) as binary search algorithms perform search with negligible (fractions of millisecond) response time and caches can be added as needed. Monitoring the outgoing TCP connections requires a relatively simple PCAP filter, and the rate of new outgoing connections is generally low when they are opened as a result of a user action (e.g., a new telnet connection or browsing to a new web page). Even in case of software originated automatic connection establishment (e.g., in network management professional software services) the PCAP filter has no impact in the service performances as it helps the outgoing connection monitoring.

The second overhead required by our solution is the additional communication between the server and the client. The largest fraction comes from the establishment of the SSL connection from the server to the client and from the time required to make the credential request over this connection. To reduce the impact of the connection set-up, the SSL connection can be cached and later resumed or left in place for subsequent credential requests. For initial authentication, most protocols (e.g. telnet, HTTP) require a round-trip message for credential requests like the proposed authentication does. Note that unlike our solution, for subsequent access requests, protocols like HTTP may include credentials in the HTTP header such that additional round-trip messages for credential requests are not needed again. A similar result can be achieved for authentication in case of HTTP traffic if servers set a cookie on the client side, via the data stream, with an encrypted, time-limited credential for the user after the initial authentication (c.f. [16]).

Performance Testing Environment. We present below the performance results of our service using HTTP traffic. The results obtained by using telnet traffic, which are even easier to generate as there is no HTTP interface involved, show similar performances. The server is a 2 GHz Pentium 4 (Windows 2000) workstation running the IBM HTTP Server. The client system is a 1.2 GHz Pentium III

(Windows XP) IBM Thinkpad running a Netscape 4.79 HTTP client. We enabled the authentication for the server by using a server-side HTTP proxy. All machines where lightly loaded on the same 100 Mbps Ethernet segment. The measurements were taken using IBM Page Detailer and reading the time between the HTTP request and the reception of the related HTTP response on the client.

Initial Authentication counts the time needed for the first authentication within a session after a user has requested access protected data. Initial authentication for the legacy HTTP case includes the time for the client data request, the related server authentication request (HTTP 401 - Authorization Required), the User Response Time (URT) for entering the user identity and password for this service, the sending of the user identity and password to the server, and the final content response of the server. Initial authentication for the case of our solution includes the time for the client data request, the related server credential request (over SSL), the reply of the credentials to the server (over SSL), and the final content response of the server. Table I compares the performance of initial authentication for the legacy case and for our service. It shows that the legacy authentication requires about 24 ms at which the variable user response time (of the order of seconds) is to be added. In the case of our proposed solution, the initial authentication requires only 92 ms in all, i.e., including the additional protection of the credential exchange via client-server authenticated SSL using 1024 bit RSA key certificates. We used the Low interaction setting to avoid user confirmation prompts and related delays. Given that the user response time to authentication requests in the legacy case usually requires multiple seconds, our result shows that our solution significantly improves the initial authentication performance, while additionally enhancing the credential security.

Authentication Method	Time (ms)
Legacy	24 + several seconds of URT
Our Service	92

TABLE I
PERFORMANCE INITIAL AUTHENTICATION

Repeated Authentication counts the time required for subsequent requests of access controlled data. Repeated authentication for the legacy HTTP case includes the time for the client data request (the browser includes the user credentials automatically in the request) and the final content response of the server. Repeated authentication using our solution includes the time for the client data request, the related server credential request (over resumed SSL), the reply of the credentials to the server (over resumed SSL), and the content response of the server. Table II compares the performance of repeated authentication for the legacy case and for our service. It shows that the legacy authentication requires about 12 ms. In the case of our service, the repeated authentication requires about 52 ms and includes additional protection of the credential exchange via client-server-authenticated SSL (resuming the former SSL session). However, the absolute response time of 52 ms is insignificant from a human perspective.

Authentication Method	Time (ms)
Legacy	12
Our Service	52

TABLE II
PERFORMANCE REPEATED AUTHENTICATION

In conclusion, using our solution brings its credential management benefits at a very low time cost for the user in case of repeated authentication (which can be further reduced for HTTP by using authentication-cookies as sketched in the next section) and improves the response time substantially for the initial authentication.

C. *Interoperability and Compatibility*

In this section we focus on compatibility issues when our service interacts with existent systems. Thus, we first describe backward compatibility and migration from existing authentication techniques. Next, we consider compatibility with NAT routers and firewalls. Finally, we describe the zero interaction repeated authentication solution for HTTP(S) connections.

Backward Compatibility and Migration Path. It is important to note that our service does not use the client's data connections for its signaling. That is, the communication for the client applications does not require any modification when using the proposed authentication. This fact allows seamless backward compatibility and a straightforward migration path as described by the two following scenarios.

Let us first consider a legacy authentication client connecting to our server. When the client application requests restricted content, our server will issue an authentication credential request specific to our service. Since the client does not support our solution, there will not be a client authentication service listening on a well-known port on the client, so the server will receive a connection-refused response. In this case, the server reverts to the legacy credential request over the data connection and manual authentication follows as supported by the existent system.

Let us now consider the case when a client using our authentication application communicates with a legacy server that doesn't support our solution. When the client application communicates with the server application, the server will issue its legacy credential requests over the data connection. Since the client application is not modified and our service does not modify the data connection, the legacy credential request will be received by the client application and satisfied as it is currently supported, e.g., requiring the user to manually provide userid and password. In this case the user cannot profit from the features of our service.

If the server cannot be controlled to send the credential request to the port specific to our service—e.g., our service is operated by an ISP that does not have control over third party servers—we enable the clients to make use of our service by introducing a proxy on the data stream between the client and the server. This proxy passes messages back and forth between the client and the server on the data connection, passively watching the data stream for a legacy credential request from the server. Rather than passing the legacy request to the client application over the data connection, the proxy issues a credential request specific to our solution and then forwards the credentials to the server using the legacy authentication format. If the server does not accept the provided credentials, the proxy passes the legacy credential request on to the client application through the data connection resulting in legacy user authentication; otherwise the proxy passes results from the server to the client and discards the legacy credential request from the server. This approach requires an extension for our service to identify and handle the legacy authentication in the proxy. Additionally, any SSL connection on the data stream terminates at the proxy. We have fully implemented and successfully tested these compatibility and migration options for our authentication prototype using a simple extended HTTP proxy java application and the Apache Web server [17] on the local IBM network. Our authentication service only requires that the authenticating party can reach the client and its authentication port. Thus, it supports Mobile IP as long as this client port is not blocked by a visited network.

NAT and Firewall Support. The current implementation of our authentication solution requires the server to initiate the request for credentials and set up a connection to the client system. This has technical drawbacks if used in an environment where network address translation (NAT) [18] is used or where firewalls block incoming traffic. Such a situation occurs if a user works remotely behind a router that offers NAT to enable multiple clients to share a single IP address—usually this is necessary when operating multiple clients connected by a single broadband cable or DSL connection. In these settings, it is possible to use port forwarding to expose our client authentication service for one machine behind the NAT. This port forwarding can be automated using NAT traversal with UPnP [19] in the case of NAT devices. Our authentication also works via VPN connections to remote clients without enabling port forwarding on the router since the server credential request is tunneled through the NAT router with IPSEC.

Zero Interaction Repeated Authentication. To optimize repeated authentication, a server aware of our service may embed credentials after the initial request into a protected cookie [20] in the HTTP response and store it this way on the client. The credentials are protected by using a random symmetric encryption key and adding this key—encrypted with the public key of the server certificate—to the cookie. Future client requests will include this cookie and the server can retrieve the protected information on demand from the cookie instead of requesting it from the client authenticator again. This extension only works for HTTP and Web Browsers supporting cookies.

D. Ease of Use

Experience has shown that one of the key-points to render a service successful, beside powerful technical features, is the ease-of-use. As we have seen, our solution is straightforward, does not introduce additional cost from the user perception point of view, and is easy to integrate with existing systems. Additionally, the proposed service allows users to off-load credential management and authentication handling to the client authentication service. Users store their credential profile in the vault and need only to update credentials if they change. If no credential information is found, the authentication service pops up a window and asks the user explicitly for authentication credentials; a checkbox allows the user to decide if these credentials are permanently added to the credential vault. An *edit* option in the graphical user interface of the authentication service, c.f. Figure 6, allows users to edit the protected vault, to delete, or change entries manually. A syntax check ensures that the structure of the file is not accidentally invalidated. Our authentication solution also permits the user to choose the level of involvement in authentication among three levels of interaction (user preferences). Note that specific interaction settings within a credential entry can be used to override the default settings for very sensitive or very insensitive services. Finally, the vault and the client authentication service program can be stored on the very convenient and easily accessible USB token (or other pervasive device) so that they can accompany a mobile user and be used to provide secure authentication.

V. Related Work

In this section we discuss some of the features of the most known authentication models proposed in the literature and relevant to the issues considered in this paper. We first give a brief description of the model, present its particular limitations, and then compare to how our service addresses them.

The Factotum user agent in Plan 9 [11] addresses secure storage of credentials and considers flexible, protected authentication for services. Factotum requires the authentication services to be integrated into the operating system, the client applications, as well as into the server applications. For this reason, Factotum's security depends on changing the operating system and specific libraries that support the client applications, which is impractical for most deployed operating systems. As mentioned before, our authentication service does not require changes to the operating system or client applications making use of its service.

Microsoft Passport [7] is a third-party cookie-based technology that allows automatic, repeated authentication for applications using the HTTP protocol. Passport controls highly sensitive information about a large number of users including their buying habits, preferences, account information, and credibility. It relies on a centralized architecture and, therefore, it is inherently vulnerable to concerted attacks that can use the system to compromise or manipulate confidential information of its users without consent or knowledge of those users. Users must fully trust Microsoft Passport to handle their information (privacy, credentials) in accordance with the user's wishes. The users must additionally trust Microsoft to protect the Passport infrastructure against strongly motivated attacks (see [21] for a description of vulnerabilities in Passport and the corresponding Wallet service) aimed at the valuable information of users and merchants. Liberty Alliance [8] is similar to Microsoft Passport. However, it allows groups of merchants to share federated user information reducing thereby the risks associated with a single repository. Our service originality consists in allowing automatic secure authentication while keeping the information totally decentralized and fully under the control of the user.

Zero Interaction Authentication [22] offers a token-based single sign-on solution for protecting user data on devices such as laptops. In this model, the files are stored encrypted on a laptop, and the user's authentication token provides decryption authority for laptops bound to the token once the user has authenticated with the token. The design of ZIA is focused on protecting a user's sensitive content from accidental disclosure with proximity being a key part of the access control. ZIA enhances user-to-client authentication, whereas our solution enhances client-to-server authentication. Our service can use ZIA to prevent client-hijacking due to forgotten authentication sign-outs.

Shibboleth [23] is an open, standards-based solution to the needs for organizations to exchange information about their users in a secure and privacy-preserving manner. This information exchange typically determines if a person using a web browser has the permissions to access a target resource based on information such as being a member of an institution or a particular class. Like Passport, this model places an intermediate party between services and users; users must trust this intermediate party regarding their privacy, and services must trust this intermediate party to authenticate users. Our solution enables users to keep control over their private data without need for intermediaries.

The Security Assertion Markup Language (SAML [24]) is an XML-based [25] security standard for exchanging authentication and authorization information that is likely to be used to encode information that is exchanged between Passport-like servers and merchants or clients. In our case, SAML offers one possibility for encoding information exchanged between the servers and clients.

The Simple Authentication and Security Layer (SASL [26], [27]) specification describes how to integrate authentication support to connection-based protocols. The authentication provided by our solution can be negotiated as a so-called EXTERNAL SASL authentication mechanism matching the requirements specified for SASL similarly to the methods exemplified in [28]. Having our solution available as an EXTERNAL SASL authentication mechanism, the server would directly use our service to determine the client identity and whether the client is authorized [26].

Kerberos [29] allows client applications modified to work with Kerberos the ability to have automated, repeated authentication to a particular service through the use of the Kerberos ticket. Our solution is orthogonal to Kerberos services because it supports the authentication of users to the Kerberos user agent, whereas Kerberos supports the authentication of client applications to server applications. The service we propose mitigates the risk of offline password guessing attacks against Kerberos tickets [30] by enhancing the likelihood of strong passwords.

The Identification Protocol [31] provides a means to determine the identity of a user of a particular TCP connection. Given a TCP port number pair, it returns a character string which identifies the owner of that connection on the client side to the server system. This mechanism has been used, for example, by FTP servers to get additional information about users that log into the system. The identification protocol is not used as a secure way to authenticate clients, rather as a supplement to the existing client-server password authentication. The proposed service uses a similar call-back mechanism. However, our solution provides a complete call-back mechanism for the client-server authentication, and, in addition, is secure.

VI. Concluding Remarks

We considered the problem of securely managing the authentication credentials in the context of the security issues of applications and communication networks. Authentication requirements of newly emerging services and vulnerabilities regarding denial-of-service attacks against centralized credential servers have motivated our development of a secure client-managed authentication service.

We introduced a client-based solution that keeps the users in control of their credentials and, thereby, avoids the user-unaware credential manipulation risks of centralized models. The proposed service maintains a database of user identifiers, credentials (e.g., passwords), and server information through which the client assists in authenticating users and applications. We described our authentication service, which securely correlates the server credential requests with the established data connections and validates the server's identity before authorizing any credential release. The service we propose allows a three level user control of the credential release. It does not require changes to the operating system or client applications making use of its service, and provides a common, consistently high protection level (SSL) for credential exchanges.

We implemented our service to support HTTP-based, HTTPS-based, and Telnet-based services, which account for most interactive Internet traffic today. The service can be extended to support local applications, such as SSH or Kerberos user agents, as well; these applications can be adapted to transform their local interactive user authentication requests into our authentication. In Linux, for instance, our solution naturally fits into the pluggable authentication module (PAM [32]) abstraction and through it can be made accessible to any PAM-enabled application.

We presented sample experimental results characterizing the performance of the proposed service and showed that our fully implemented prototype performs well in terms of overhead from the user perception point of view, integrates seamlessly with existing systems, is highly secure and convenient to use. Our service supports users in securely storing and maintaining their credentials for different types of applications. An immediate extension to this work will be the integration of non-password based credentials and the integration of our service into the context of pervasive devices. Finally, integrating our authentication into the Simple Authentication and Security Layer (SASL) framework as sketched in Section V enables seamless invocation and broad use of our solution from any SASL-enabled client-server application.

Acknowledgment

The authors would like to thank Dinesh Verma, Suresh Chari, and Alan Bivens for helpful discussions of our architecture, useful comments and suggestions.

Appendix: Authentication Service Gui

The Authentication Service GUI is implemented in C. The design is straightforward, containing six elements as described below:

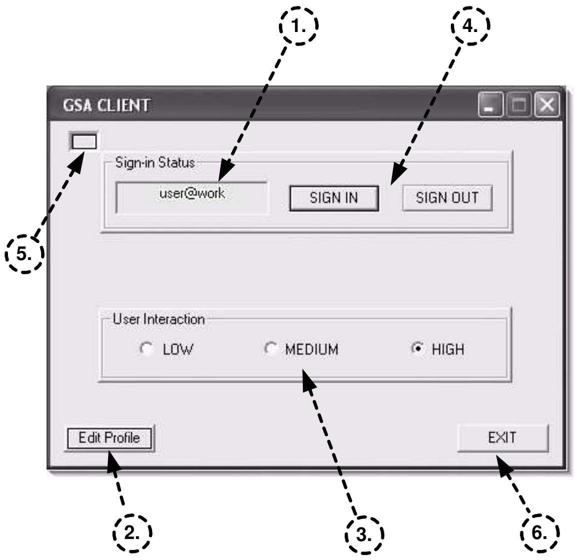

Fig. 6. The Authentication Service Graphical User Interface

1. Profile indicating a specific collection of credential entries that determines the context in which the user wants to operate (e.g., related to work, or personal accounts; to MPLS core routers, or MPLS edge routers, or ATM switches, etc.).

2. Edit Profile to fill in the details (e.g., userid, password, server, protection, etc. cf. Section III-A) of the credentials for a given profile.

3. User Interaction to choose the appropriate level of interaction as explained in Section III-D.

4. Sign In - Sign Out to start - stop the authentication client and, thereby, the service.

5. Activity Indicator to visualize the processing of credential requests by the client authentication service.

6. Exit the application GUI.

REFERENCES

[1] Robert Morris and Ken Thompson, "Password security: A case history," *CACM*, vol. 22, no. 11, pp. 594–597, 1979.

[2] David C. Feldmeier and Philip R. Karn, "UNIX password security - ten years later," in *CRYPTO*, 1989, pp. 44–63.

[3] Daniel V. Klein, "'Foiling the cracker' – A survey of, and improvements to, password security," in *Proceedings of the second USENIX Workshop on Security*, Summer 1990, pp. 5–14.

[4] Philip Leong, "UNIX password encryption considered unsecure," in *USENIX Conference Proceedings*, January 1991, pp. 269–280.

[5] Li Gong, T. Mark A. Lomas, Roger M. Needham, and Jerome H. Saltzer, "Protecting poorly chosen secrets from guessing attacks," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 5, pp. 648–656, 1993.

[6] "Virus description for BAT.Mumu.A.Worm," June 2003, http://securityresponse.symantec.com/ avcenter/ venc/ data/ bat.mumu.a.worm.html.

[7] Microsoft Corporation, ".NET Passport 2.0 Technical overview," http:// www.microsoft.com/ myservices/ passport/ technical.doc, October 2001.

[8] "Liberty alliance project," 2004, http:// www.projectliberty.org.

[9] GAIN Publishing, "Gator eWallet," 2004, http:// www.gator.com/home2.html.

[10] Zero-Knowledge Systems, "Freedom security and privacy suite," 2002, http:// www.freedom.net/ products/ suite.

[11] Russ Cox, Eric Grosse, Rob Pike, Dave Presotto, and Sean Quinlan, "Security in Plan 9," in *11th USENIX Security Symposium*, August 2002.

[12] L. Degioanni et. al., "Windows packet capture library," http://winpcap.polito.it/.

[13] "Microsoft developer network (msdn) library," http://www.msdn.microsoft.com/.

[14] T. Drake, "Measuring software quality: A case study," *IEEE Computer*, vol. 29, no. 11, pp. 78–87, November 1996.

[15] N. Provos, "Encrypting Virtual Memory," in *9th Usenix Security Symposium*. USENIX, 2000.

[16] J. Giles, R. Sailer, D. Verma, and S. Chari, "Authentication for Distributed Web Caches," in *7th European Symposium on Research in Computer Security (ESORICS)*, October 2002, pp. 126–145.

[17] "Apache project," http://www.apache.org.

[18] P. Srisuresh and K. Egevang, "Traditional ip network address translator (traditional nat)," January 2001, IETF RFC 3022.

[19] Microsoft Corporation, "Overview of network address translation in Windows XP," http://www.microsoft.com/ WindowsXP/ pro/ techinfo/ planning/ networking/ nattraversal.asp, July 2001.

[20] K. Fu, E. Sit, K. Smith, and N. Feamster, "Dos and don'ts of client authentication on the web," in *The 10th USENIX Security Symposium*. USENIX, August 2001.

[21] David Kormann and Aviel Rubin, "Risks of the Passport single signon protocol," *Computer Networks*, vol. 33, pp. 51–58, 2000, http://avirubin.com/ passport.html.

[22] Mark Corner and Brian Noble, "Zero-interaction authentication," in *ACM MOBICOM 2002*. MOBICOM, September 2002.

[23] Internet2, "Shibboleth project," 2004, http:// middleware.internet2.edu/shibboleth.

[24] OASIS, "Security assertion markup language (SAML)," 2004, http:// www.oasis-open.org/committees/security.

[25] World Wide Web Consortium (W3C), "Extensible markup language XML," 2002, http:// www.w3.org/XML.

[26] J. Myers, "Simple authentication and security layer (SASL)," October 1997, IETF RFC 2222.

[27] C. Newman and J. Myers, "ACAP – application configuration access protocol," November 1997, IETF RFC 2244.

[28] IETF Charters, "Simple authentication and security layer (SASL) working group," http://www.ietf.org/ ietf/ sasl/ sasl-charter.txt.

[29] J. Kohl and C. Neuman, "Kerberos network authentication service (V5)," September 1993, IETF RFC 1510.

[30] Thomas Wu, "A real-world analysis of Kerberos password security," in *Internet Society Network and Distributed System Security Symposium*, 1999.

[31] M. St. Johns, "Identification protocol," February 1993, IETF RFC 1413.

[32] "Pluggable authentication modules for linux," http://www.kernel.org/ pub/ linux/ libs/ pam/.