INTERNET-DRAFT
draft-petke-remote-pass-auth-00.txt
Expires: 19 November 1997

G Brown
CompuServe

19 May 1997

Remote Passphrase Authentication

Status of this Memo

Abstract

Remote Passphrase Authentication provides a way to authenticate a user to a service by using a pass phrase over an insecure network, without revealing the pass phrase to eavesdroppers. In addition, the service need not know and does not learn the user's pass phrase, making this scheme useful in distributed environments where it would be difficult or inappropriate to trust a service with a pass phrase database or to allow the server to learn enough to masquerade as the user in a future authentication attempt.

This scheme was inspired by Dave Raggett's Mediated Digest Authentication, draft-ietf-http-mda-00.txt.

This specification is divided into five parts. Part Zero contains an extended introduction to the problem and potential solutions. Part One explains the mechanism. Part Two explains how to incorporate the mechanism into HTTP. Part Three explains the protocol between the service and deity. Part Four explains the GSS-API token formats. Feel free to start with Part One; Part Zero provides background information and is not a prerequisite for Part One.

Table of Contents

Remote Passphrase Authentication
Part 0
Extended Introduction

1 Introduction

In this introduction we'll explain the problem--fundamentally, how to authenticate a user to a service without revealing a pass phrase, and without requiring the service to know the user's pass phrase--and consider several alternatives and their flaws, leading to the reasons for developing this authentication mechanism. If you're already familiar with the concept of authentication and the surrounding issues, you might prefer to skip to Part One of the specification, returning to this part only if you want more information about the motivation for the mechanism.

We'll speak of an environment in which a user communicates with a service that wishes to learn and authenticate the user's identity and vice versa. You may, of course, think in terms of client and server, but those terms generally refer to an implementation. We're speaking at a higher level where there's no direct correspondence between server and service nor user and client.

We'll use CompuServe and America Online as concrete examples of services, but the same concepts apply even to a single Web server or BBS that wants to authenticate users. There are three aspects of this environment of interest:

Identification--the way in which we refer to a user.

Authentication--the way in which a user may prove his or her identity.

Authorization--the way in which we determine what a given user may do.

The same aspects apply to services as well as users.

1.1 Identification

A user's identity consists of a user name and a realm name. A realm is a universe of identities; CompuServe Information Service user IDs and America Online screen names are two examples of realms. The combination of username and realm--typically shown as name@realm--identifies a user. Any given service will recognize some particular set of identities. A realm doesn't have to be large, though, either in number of users or size of service. For example, a single Web server might have its own realm of users.

Often, a service recognizes only one realm: CIS recognizes only

identities within the CIS realm, and AOL recognizes only identities within the AOL realm. But one can imagine a service that has agreements with both CIS and AOL. The service gives the user a choice of realms--"Please supply a CIS or AOL identity, and prove it"--and the user chooses a realm in which he has an identity.

1.2 Authentication

Identification provides the ability to identify, or refer to, a user. Authentication provides the ability to prove identity. When you ask to do something for which your identity matters, we ask for your identity--your username and realm--and we make you prove that you are who you say you are.

To accomplish this, we'll use a secret that we call a pass phrase, although it's not necessarily derived from text. Such a secret is sometimes called a secret key, but we won't be using it for encryption.

The fundamental problem to be solved is: How can you prove to me that you know your pass phrase without revealing the pass phrase in the process? We'll explore this problem in more detail momentarily.

1.3 Authorization

Authorization refers to the process of determining whether a given user is allowed to do something. For example, may he post a message? May he use a surcharged service? We won't say much about this topic, but it's important to realize that authentication and authorization are distinct processes, one related to proving an identity, and the other related to the properties of an identity.

Our mechanism has nothing to do with authorization, but it is designed to co-exist with authorization mechanisms.

2 The problem and how not to solve it

Imagine that I'm a service who wishes to authenticate you, a user. You must identify yourself and prove to me that you know your pass phrase. That's easy: I'll prompt you for your pass phrase.

But that doesn't work. We learned long ago that plaintext pass phrases cannot be transmitted through a network. X.25 networks have been compromised, and LANs, modem pools, and "The Internet" likewise are not suitable for plaintext pass phrases. Prompting for the pass phrase is not the answer.

2.1 Encrypt the pass phrase?

How about encrypting the pass phrase? Sounds good. You encrypt your pass phrase, send me the result, and I'll decrypt it. Techniques like Diffie-Hellman can create a one-time key that prevents an eavesdropper from decrypting your pass phrase.

But that doesn't work, either. What if somebody else--a spoofer--pretends to be the service? He'll decrypt the result, learning your pass phrase and gaining the ability to masquerade as you. Perhaps that sounds unlikely, but it's not; even in dial-up modem days, people have spoofed services--"Here's a new telephone number they left out of their directory. It's much faster than the listed numbers!"

We need a mechanism that won't reveal your pass phrase to anyone, even if you're not talking to whom you think you're talking.

2.2 A challenge-response mechanism?

How about a challenge-response mechanism? Now we're on the right track. I send you a challenge, which is a random number, and you use a one-way function to calculate a result that depends on the challenge and your pass phrase. You send me the result, and I perform the same calculation and see if my result matches yours. Done correctly, this reveals no information to eavesdroppers, nor does it allow a spoofer to acquire your pass phrase--if someone pretends to be me, they learn only your result for a particular challenge, which is of no value.

Although such a mechanism works, it doesn't quite solve our problem. If I'm the service, I must know your pass phrase in order to reproduce your calculation and verify your response to my challenge. But what if I don't know your pass phrase?

2.3 What if I don't know your pass phrase?

Why might I, the service, not know your pass phrase? Consider a set of services that share a set of users' identities. For example, imagine a collection of Web servers, scattered throughout the world, all of which are a part of Gary's Information Service; you may use your GIS name and pass phrase to identify yourself to any GIS service.

The obvious implementation--each physical server has a copy of all pass phrases or access to a master database--is awkward at best, especially if some are third-party servers, not directly under the control of our imaginary GIS.

Or consider a service that accepts identities in multiple realms. Imagine a service that has agreements with both CIS and AOL. The

service gives the user a choice of realms--"Please supply a CIS or AOL identity, and prove it"--and the user chooses a realm in which he has an identity. It's unlikely that CIS and AOL will entrust a copy of their pass phrase databases to a third-party service--or to each other.

So, if I don't know your pass phrase, how can you prove to me that you do know it? And that's the fundamental question addressed by this mechanism. We'll begin by pointing out a couple of solutions that don't work.

2.4 Two more ways not to solve the problem

Wrong answer #1--I'll prompt you for your pass phase. Let's make this example more concrete: I'll display an HTML form with a box that asks for your name and a box for your pass phrase. We'll use SSL or SHTTP so an eavesdropper can't see it. When I get your reply, I can use a challenge-response mechanism to verify your pass phrase with a server that knows the pass phrases.

But that won't work. It's important to teach users not to type their pass phrases just because somebody asks for it--that's a standard technique for cracking others' accounts. Teaching users to provide their pass phrases in an HTML form is a bad idea.

And I'll see your pass phrase, which is precisely what we want to avoid, especially if I'm a spoofer.

Wrong answer #2--We'll create a pass-phrase database server. I'll ask it for a copy of your pass phrase. Now that I know it, we can use an ordinary challenge-response mechanism.

That won't work. We'd need a way to get the pass phrase from that database to me, safely. And if I can look up your pass phrase, what's to stop somebody else from doing the same? (Don't say "a firewall." Services that need to verify your identity exist outside firewalls, too.)

If anything, this is even worse--I could dump the entire pass-phrase database--and, again, I should never see your pass phrase.

But there is a solution, which we'll cover in Part One of this specification.

Remote Passphrase Authentication
Part 1
The Mechanism

Table of Contents

3 Introduction

In this mechanism, we'll authenticate a user to a service and vice versa. We'll use pass phrases--actually, they're 128-bit shared secrets, but we'll define a way to use textual phrases--so the goal is to prove to the service that you know your pass phrase, and vice versa.

Of course, it's important not to reveal the pass phrase to an eavesdropper. It is equally important not to reveal the pass phrase to a spoofer.

Furthermore, the mechanism should work even if the service does not know the user's pass phrase. In a distributed environment, with many services that wish to authenticate the same set of users, it may be difficult to make users' pass phrases available to all services. And we might prefer not to do that, if we don't completely trust the services. So, not only should the service not have to know the user's pass phrase, but the service should not learn the user's pass phrase during the authentication process.

On the other hand, the mechanism should be simple enough to apply even in the traditional case where the service knows the user's pass phrase; there's no need to use a different mechanism in that case.

Part Zero of this specification contains an extended introduction that explains the problem and various potential solutions and their problems, leading to this mechanism. If you find yourself asking,

"Why not just...," it might be worth reading Part Zero to see if that explains it. However, it contains only background material, so you

needn't read Part Zero before reading the rest of this specification.

4 Terminology

Throughout this specification we'll speak of a "user" communicating with a "service" that wishes to learn and authenticate the user's identity. Often, the user is a "client" and the service is a "server," but those terms refer to an implementation.

The "deity" knows the users' and services' pass phrases, and the service talks to the deity during the authentication process. Although the term "authentication server" is more conventional, we call it a deity because it's got fewer syllables and the term "server" is overloaded. If the service knows the pass phrases, then it acts as its own deity, simplifying the implementation but otherwise having no effect on the mechanism.

Identities exist in some "realm," and we use that term in its usual sense. We often think of a realm as being a relatively large collection of users, like compuserve.com or aol.com, but it might well consist of a small set of users, e.g., user names and pass phrases associated with an individual Web server. We allow the service to specify a set of realms, to recognize an identity in any of the realms in which it participates.

5 Design criteria

This authentication mechanism is intended to meet the following criteria.

* The service learns and authenticates the user's identity.

* The user learns and authenticates the service's identity.

* The mechanism does not use public-key technology.

* The mechanism does not use encryption. (By encryption, we're referring to reversible encryption, the ability to encrypt something and later decrypt it. By avoiding encryption, we avoid restrictions on exportability.)

* The mechanism is based on shared secrets: "pass phrases," although they can be arbitrary bit patterns rather than text.

* Neither the user nor the service needs to know the other's pass phrase.

* Neither the user nor the service nor eavesdroppers will learn the other's pass phrase. However, if the pass phrase is based on text,

it's important to choose a "good" pass phrase to avoid a dictionary attack.

* The mechanism is reasonably easy to implement in clients and does not require the client to communicate with a third party nor to a possess a reliable clock.

* The mechanism derives a shared secret that may be used as a session key for subsequent authentication.

* The mechanism may be incorporated into almost any protocol. In other words, the mechanism is not designed around a protocol; the protocol is designed around the mechanism. But the mechanism must be suitable for incorporation into protocols like HTTP.

* The mechanism provides the ability to accept an identity in any of a set of realms in which the user and service are members.

6 The mechanism

This authentication mechanism consists of three related processes: authentication, reauthentication, and reauthentication cheating.

Authentication is the fundamental process by which a user and a service mutually authenticate one another within one of a set of realms, without revealing their pass phrases to one another.

Reauthentication is a process by which a user and service, having recently authenticated one another, may again authenticate one another. They could, of course, simply repeat the authentication process, but that requires interaction with an authentication deity. The reauthentication process is faster, requiring no communication with a third party. Reauthentication is useful when multiple connections between the user and service are established, whether sequential as in HTTP or simultaneous. Each connection must be authenticated, but the reauthentication process provides a shortcut.

Reauthentication cheating is a further optimization for HTTP, a protocol that is quite unfriendly to challenge-response mechanisms. Reauthentication cheating can be performed in parallel with an HTTP transaction. True reauthentication is just as simple, but requires two sequential requests because of the characteristics of HTTP. By using reauthentication cheating, we create a "one-way" handshake.

6.1 Authentication

There are three parties involved in the authentication process:

* the user;

* the service; and

* the authentication deity.

Each user has a name and a pass phrase in some realm of interest. Similarly, each service has a name and a pass phrase in that realm. The pass phrase isn't really text; it's a 128-bit (16-octet) string of bits.

However, it's often useful to use pass phrases in the conventional, textual sense, so we define a procedure for converting a textual phrase to the 128-bit value used by the authentication mechanism. If such a pass phrase is poorly chosen, it will be subject to dictionary attack, and that's why we never use the word password in this specification (well, except in this sentence)--use a phrase, not a word.

The service may specify a list of realms, and the user chooses one in which he has an identity. Thus, a service is not restricted to authenticating identities in a single realm. The service must possess a name and pass phrase in all realms it lists.

Each realm has an authentication deity, which knows the names and pass phrases of its members. It's the service's responsibility to know how to locate an authentication deity for each realm; the user never communicates directly with an authentication deity. If the service knows the user's pass phrase, it performs the role of the authentication deity itself, but this does not affect the mechanism.

6.1.1 Values and their representation

Following is a glossary of the values involved in the authentication process; we'll use these symbols in the following explanation.

As--Authentication deity's response to service; proves user's identity

Au--Authentication deity's response to user; proves service's identity

Cs--Challenge from service

Cu--Challenge from user

Kus--Session key for user and service

Kuss--Session key obscured so visible only to service

Kusu--Session key obscured so visible only to user

Nr--Realm name

Ns--Service name

Nu--User name

Ps--Service's pass phrase, a 128-bit value

Pu--User's pass phrase, a 128-bit value

Rs--Service's response to challenge (during authentication process, goes to authentication deity; during reauthentication, goes to user)

Ru--User's response to challenge (during authentication process, goes via service to authentication deity; during reauthentication, goes to service)

Ts--Service's time stamp

Z--Padding consisting of 48 octets (384 bits) with all bits set to zero

+--Concatenation of octet strings

xor--Bitwise exclusive or

Bit patterns for each value must be specified. Imagine, for example, that one implementation uses ASCII, another EBCDIC, and another Unicode for the user name. Or one implementation converts the name to lowercase, another to all caps. Each would generate a different result for the same calculation, and authentication would fail.

Should we leave such details to the underlying protocol? We could, but that would make the service-to-deity protocol dependent on the user-to-service protocol, so we couldn't have a single deity for each realm. If we specify the bit patterns, we can allow any mixture of user-to-service and service-to-deity protocols to operate on the same data. Therefore, we adopt the following conventions.

Text strings are represented in the Unicode character set, in big-endian byte order, without a trailing null character. Note that ASCII can be converted to ISO 8859-1 by prefixing a single 0 bit, and ISO 8859-1 can be converted to Unicode by prefixing eight 0 bits. Each 16-bit Unicode character is stored in two octets, with its high-order 8 bits in the first octet. Representation of characters with multiple encodings is for further study. For example, e-acute has more than one representation. The form that uses combining characters, in character-code order, is probably the most logical.

Note, by the way, that this specification refers only to values used in the authentication calculations, not the underlying protocol. For

example, it's quite reasonable for a protocol to use ASCII for user names, if that character set is adequate. Those ASCII characters must be converted to Unicode before using them in authentication calculations, but the protocol need not transmit Unicode characters.

* Names--Nr, Ns, Nu--are converted to lowercase Unicode. Note that there is no trailing null character.

* Challenges--Cs, Cu--are arbitrary strings of octets, not text. They may contain any bit patterns, including nulls, and must be at least eight octets in length.

* The time stamp--Ts--is the ISO 646 (ASCII) textual representation of the current universal time--UTC--in exactly 14 octets, using 24-hour time, with leading zeroes: 19950805011344.

* Pass phrases--Ps, Pu--are 16-octet quantities that contain arbitrary bit patterns, including nulls. If the pass phrase is based on a textual phrase, the textual phrase is converted to a 16-octet quantity by the following process.

  * Convert the text string to a sequence of characters in either the Unicode or ISO 8859-1 character sets, as appropriate for the realm.

  * Convert each character to its lowercase equivalent, or its uppercase equivalent, or leave it alone, as appropriate for the realm.

  * Store the sequence of characters in an octet stream, with each Unicode character in two consecutive octets in big-endian order, or each ISO 8859-1 character in one octet. Do not append a trailing null character.

  * Take the MD5 digest of the resulting string of octets. The result is the 128-bit value to use in the authentication calculations.

A realm will specify which of the preceding options--character set, case conversion, and hash function--it uses for the text-to-128-bit value transformation; the defaults are Unicode, convert to lowercase, and MD5. More options might be added in the future. The user-service protocol should be designed to convey the appropriate options for each realm from the service to the user, if other than the defaults are to be supported, to avoid requiring the (human) user to manually configure software.

6.1.2 The authentication process

Here we describe the individual steps. Taken literally, one might

envision many messages between the service and deity, but an actual implementation within a protocol combines steps. For example, "The

user sends a random challenge" is shown as a separate step for clarity, but it needn't be a separate message to the service, nor must it be sent at the point shown--if it makes sense in the underlying protocol, the user's challenge might be included with the user's response to the service.

* The service supplies a sequence of realms, with the service's name in each realm, to the user. For example,

    foo@compuserve.com bar@aol.com

  means "Please identify yourself with a CIS user ID. If you don't have one, your AOL ID will do." The service indicates its realm preferences in most-preferred to least-preferred order; by specifying only one realm, the service requires identification in that realm.

* The user chooses a realm, Nr, and gives it and his name in that realm, Nu, to the service. That, in turn, determines Ns, the service's name in that realm. Note that a protocol might allow the service to include a null realm name, meaning "I'll accept you as an anonymous user if you wish." The user might make this choice by supplying a null name; the the process stops here, and no authentication is performed.

* The service transmits a random challenge, Cs, and a time stamp, Ts. The challenges are random values that make each authentication unique. The time stamp is, in effect, a third challenge, which the deity will ensure is recent. The user may examine it, but most users lack an accurate source of universal time, so most users will treat it as an opaque value.

* The user sends a random challenge, Cu.

* The user calculates a response, Ru:

    Ru = MD5(Pu + Z + Nu + Ns + Nr + Cu + Cs + Ts + Pu)

  and sends it to the service.

  Only the real user can generate the correct response, because it depends on the user's pass phrase, Pu. No one can determine the user's pass phrase from a captured response, because it's generated by a one-way function, although there is the risk of a dictionary attack if Pu is based on a poorly chosen pass phrase.

* The service calculates a response, Rs:

    Rs = MD5(Ps + Z + Nu + Ns + Nr + Cu + Cs + Ts + Ru + Ps)

This response is not sent to the user; it would do no harm if the

user saw it, but the user won't need it.

* The service sends a request to the authentication deity for the realm in question. The request contains

  * The realm name, Nr (included so the same deity can serve more than one realm)

  * The user's name, Nu

  * The service's name, Ns

  * The user's challenge, Cu

  * The service's challenge, Cs

  * The time stamp, Ts

  * The user's response, Ru

  * The service's response, Rs

* The deity verifies the time stamp per previously agreed upon criteria. In some applications, one might require it within a few minutes; in others, one might want to allow 25 hours to eliminate problems of misconfigured time zones. Beware of overzealousness, though; this time stamp went from the service to the user, then back to the service, then to the deity, perhaps with human interaction--typing a pass phrase--introducing further delay. The deity might implement a replay cache.

* The deity uses Nr, Ns, and Nu to look up the user's and service's pass phrases.

* The deity uses the values in the request, plus the service's pass phrase, Ps, to verify Rs. If it is incorrect, the deity returns a negative response; this request apparently did not come from a legitimate service.

* Having verified the requesting service's identity, the deity uses the values in the request, plus the user's pass phrase, Pu, to verify Ru. If it is incorrect, the deity returns a failure response to the service; the user does not know the correct pass phrase.

* Having verified both the user's and service's identity, the deity creates a random, 128-bit session key, Kus, for use by the user and service. They might use it for session encryption; in addition, it will be used in the reauthentication process described later.

* The deity generates two obscured copies of the session key:

  * Kuss = Kus xor MD5(Ps + Z + Ns + Nu + Nr + Cs + Cu + Ts + Ps)

  * Kusu = Kus xor MD5(Pu + Z + Ns + Nu + Nr + Cs + Cu + Ts + Pu)

    The obscuring masks resemble Ru and Rs, but differ, of course, so an eavesdropper cannot recover Kus.

* The deity generates a pair of authentication "proofs":

  * Au = MD5(Pu + Z + Ns + Nu + Nr + Kusu + Cs + Cu + Ts + Kus + Pu)

  * As = MD5(Ps + Z + Ns + Nu + Nr + Kuss + Cs + Cu + Ts + Kus + M + Ps)

    Here "M" is the message transmitted from the deity to the service; it is included in the calculation to authenticate the response to the service. Refer to Part Three of this specification for more details.

* The deity sends the four values Kuss, Kusu, As, and Au to the service.

* The service extracts its copy of the session key from Kuss by calculating the obscuring mask value and XORing. (The service can determine the user's key-obscuring value by calculating Kus xor Kusu; and if the user sees Kuss, it can do likewise. But the obscuring masks reveal nothing.)

* The service verifies As by performing the same calculation and comparing the result. If it matches, the service knows that someone who knows its pass phrase--the deity--replied, having verified that the user is who he claims to be.

* The service forwards Kusu and Au to the user.

* The user extracts its copy of the session key from Kusu by calculating the mask value and XORing.

* The user verifies Au by computing it and comparing. If it matches, the user knows that someone who knows his pass phrase--the deity--replied, having verified that the service is who it claims to be. Of course, if the service itself knows the user's pass phrase, it can assert any service identity; but this is the case where the service is trusted and acts as its own deity.

Now the user and service are confident of each others' identities, and the two parties share a session key that they may use for encryption, if they so choose.

[Perhaps we should add another value to the authentication calculations, opaque to the mechanism, provided by the

protocol in which this mechanism is embedded. This value would, of course, have to be added to the service-to-deity protocol, and its generation and interpretation would be up to the lower-level protocol. For example, HTTP might choose to include the Web server's IP address and, perhaps, the URL in the authentication calculations, making it harder to do a man-in-the-middle attack. (Of course, that problem cannot be completely solved without using the session key to authenticate data, which is a protocol issue outside the scope of this mechanism.)]

6.2 Reauthentication

Reauthentication is a process by which a user and service, having recently authenticated each other, may again mutually authenticate without talking to a deity. This is useful with protocols like HTTP, which involve a sequence of connections that must be independently authenticated. It's also useful with parallel connections--imagine a scheme in which a user and service are connected, and wish to establish a second connection.

To reauthenticate one another, the user and service prove to each other that they both possess a secret 128-bit key--the session key, Kus, derived during the authentication process. The reauthentication process is essentially an ordinary challenge-response mechanism in which the session key is used as a pass phrase.

* The service sends a challenge, Cs, to the user.

* The user sends a challenge, Cu, to the service.

* The user calculates

  Ru = MD5(Kus + Z + Ns + Nu + Nr + Cs + Cu + Kus)

  and sends it to the service.

* The service verifies the result. If correct, it calculates

  Rs = MD5(Kus + Z + Nu + Ns + Nr + Cu + Cs + Kus)

  and sends it to the user. Both responses involve the same set of values, but they're used in a different order, so the responses are different.

* The user verifies the result.

6.3 Reauthentication cheating

In HTTP, one can shortcut the reauthentication process by cheating, for an increase in efficiency.

A naive approach allows the user to repeat its authentication data, presumably in the form of an Authorization header. If the service recognizes the same Authorization header, it presumes that it's talking to the previously authenticated user; essentially, we pretend that we reauthenticated with the same challenges. But this approach is vulnerable to replay attacks during the period of time the service considers the data valid. The service can check the user's IP address to reduce the risk, but IP addresses mean surprisingly little. Even neglecting address spoofing, multiple users share an IP address when they're on the same host or routed through a proxy or SOCKS server.

There's a better solution. We begin by noting why it's desirable--from an efficiency, not security, point of view--to allow the Authorization header to be replayed. To embed a challenge-response mechanism in HTTP, we require at least two HTTP transactions for authentication, because we cannot send a challenge and receive a response in one HTTP transaction. If we could challenge the user without sending a challenge to the user, we could authenticate in one HTTP transaction. And we can do exactly that by treating the URI as a challenge.

* The first time, the user and service perform the authentication process.

* The user and service remember the session key (Kus), challenges (Cu and Cs), and timestamp (Ts).

* When the user generates an HTTP request, he includes an Authorization header containing a response calculated as

    MD5(Kus + Z + Ns + Nu + Nr + Cs + Cu + Ts + method + URI + Kus)

The method and URI are canonicalized by taking the big-endian Unicode representation and converting all characters to lowercase; the URI should not include the scheme://host:port. It always begins with a slash; for "http://www.foo.com" the one-character string "/" would be used.

Now the authentication response is unique for each URI, and calculable only by the authenticated user, even without a unique challenge. This doesn't completely eliminate the risk of replay, of course, but an attacker can replay only a previously referenced URI during the window in which the service considers the session key to be valid. Is that acceptable?

Sometimes. If we're reading Web pages, and the only impact of replay is that the attacker could re-read the page, it might be acceptable--after all, the attacker saw the page, anyway, when he

captured it along with the original request. On the other hand, if we're charging the user per page, or if the request "did" something,

replay might not be so harmless.

One strategy is to maintain some history. In its simplest form, the service sets a flag for this session when it does something for which replay would be harmful. If the user tries reauthentication cheating, and the flag is set, the service forces reauthentication. Because the cheating response is based on Cu and Cs, and those values change during reauthentication, the correct response for a given URI changes after reauthentication. Thus, reauthentication creates a boundary after which previous requests cannot be replayed.

Or the service can maintain a history of URIs for which replay would be harmful, and force reauthentication only if the user tries reauthentication cheating on one of those URIs.

Remote Passphrase Authentication
Part 2
HTTP Authentication Scheme

Table of Contents

7 Introduction

See Part One of this series for an explanation of the mechanism, its motivation, and its specification. This part describes only the HTTP encapsulation of the mechanism.

8 Using this authentication mechanism in HTTP

The HTTP client may indicate that it supports this authentication mechanism by whatever technique is appropriate.

> [For example, a header like "Extension: Security/Remote-Passphrase" might be appropriate, if that extension mechanism is adopted. The extension mechanism is, of course, independent of authentication, but we mention it here to point out the issue. Theoretically, the server does not need to know ahead of time whether the client supports a particular authentication scheme.]

We begin by defining a security context, which represents a logical connection between a user and Web server. Because the context spans HTTP connections, the server assigns a security context identifier, an opaque string, when it creates a context, and it informs the client of its value in the Security-Context attribute of the WWW-Authenticate header. The client includes the identifier in the Authorization header of subsequent requests that refer to the same context.

From the client's point of view, the pair (server IP address, security context identifier) uniquely identifies a context; the same is essentially true for the server, although a server can make its security context identifiers unique, rather than (client IP address, identifier) pairs.

Note that a client might refer to the same security context from

different IP addresses, if he switches proxies (is that possible?). Note also that the client IP address alone is not adequate to identify the security context. A multiple-user host, an HTTP proxy, and a SOCKS server are examples of situations in which the same IP address may be involved in many security contexts. And even an individual PC running two browsers falls into this category--if I connect to you from both browsers, I'll establish two security contexts, which might or might not refer to the same user identity.

The server should assign security context identifiers that are unique over time. If the client refers to an old context identifier--the user returns to his PC tomorrow morning and clicks a link that was displayed yesterday--it will do no harm if that identifier had been reused, but the server won't be able to recognize it as such.

The security context "contains" information appropriate to the context, such as the realm name, user and service names, session key, challenges, state, etc. We'll gloss over the details in this explanation. Note that a session using this mechanism is secure; unlike other "cookie"-type mechanisms, we do not depend on the secrecy of the context identifier. However, the content of requests and responses is not authenticated, in this version of the protocol.

We define the authentication scheme name "Remote-Passphrase", used as described below. The client begins by making a request for which the server requires identification and authentication; because there is no Authorization header in the request, the server will demand authentication.

All WWW-Authenticate and Authorization headers used with this scheme may include a Version attribute. When omitted, as in the examples below, Version="1" is implied, for this version of the protocol.

8.1 Authentication

The server creates a new security context, assigns it an identifier, and responds 401 Unauthorized and includes the header

```
WWW-Authenticate:
        Remote-Passphrase
        Realm="compuserve.com",
        State="Initial",
        Realms="foo@compuserve.com
                bar@aol.com:iso-8859-1,lc,md5",
        Challenge="base64 encoding of service challenge",
        Timestamp="19950808132430",
        Security-Context="opaque"
```

The first token specifies the authentication scheme,

Remote-Passphrase. That's followed by a comma-separated list of attribute-value pairs. HTTP requires the first attribute to be called

"Realm" and specify the realm in which the user must indicate his identity, but we support multiple realms, so this is merely one realm acceptable to the server, perhaps its preferred realm.

The State attribute distinguishes this as the initial request for authentication.

The Realms attribute provides a list of realms in the order preferred by the server, with the server's name in each realm. Each may be followed by a colon and a list of parameters separated by commas, to drive the transformation from pass phrase to 128-bit shared secret for that particular realm. Refer to Part One of this specification for more information about the transformation.

The default transformation, if the colon and parameters are omitted, is specified in Part One of this specification--the Unicode character set in big-endian ("network") byte order, with all characters converted to lowercase, and the MD5 hash algorithm.

Otherwise, a single parameter, "none", implies that the client must already possess a 128-bit value, and no transformation from a textual pass phrase is defined.

Otherwise, three parameters control the transformation from a textual pass phrase to the 128-bit shared secret used by the authentication mechanism, if such a transformation takes place (it might not, if the client believes it already knows a 128-bit value for this user). The three parameters specify the character set: Unicode 1.1 ("unicode-1-1") or ISO 8859-1 ("iso-8859-1"); case conversion: convert to all caps ("uc"), all lowercase ("lc"), or as-is with no case conversion ("nc"); and hash function: MD5 ("md5"). Omitting the colon and parameters is equivalent to specifying "unicode-1-1,lc,md5".

> [There's no need for US-ASCII as a character set, because ISO 8859-1 will give the same results. Note that these parameters are part of the base authentication mechanism specification; only the means of conveying them, and the textual names shown above, are specific to this HTTP authentication scheme. Other variations can be added, but they must be added to the authentication mechanism defined by Part One of this specification as well as here in Part Two.]

We convey this information to the client because there's no reason the client would otherwise know whether a particular realm's pass phrases are case sensitive, etc. The server, on the other hand, simply must "know" how its particular realm uses pass phrases; these

characteristics are a part of server's configuration along with its name in the realm, deity addresses, etc.

The Challenge attribute specifies the service's challenge. It is an arbitrarily long sequence of octets containing arbitrary bit patterns, represented in base64. The client must decode it before using it in the authentication calculations; it might contain nulls or any other bit patterns. The client may decline to trust the server and abort at this point, if it deems the challenge to be too short.

The Timestamp attribute specifies the server's timestamp. This is a UTC date and time in the format specified by the authentication standard. It may be treated as an opaque string by the client, unless the client chooses to interpret it to make a judgement about its reality; but beware that you probably don't have a reliable source of universal time.

The Security-Context attribute contains the server-assigned security context identifier, an opaque string.

The client creates its security context and repeats the request with an Authorization header:

```
Authorization:
        Remote-Passphrase
        State="Initial",
        Security-Context="opaque",
        Realm="compuserve.com",
        Username="70003.1215",
        Challenge="base64 encoding of user challenge",
        Response="base64 encoding of response"
```

The first token specifies the authorization scheme. That's followed by the state, "Initial" for the initial authentication; the security context identifier; the realm chosen by the user; the user's identity in that realm; the user's challenge; and the user's response.

The service looks up the security context. If the security context identifier refers to no context or refers to a context that is already established, the server creates a new security context with a new identifier, then responds 401 Unauthorized and includes a fresh WWW-Authenticate header as shown above, with which the client can repeat the request with correct authentication information.

> [Or does this risk a loop? We could just respond with an error.]

Any existing security context is unaffected; if I send you a request that specifies someone else's security context, you should not delete his context.

Otherwise--the context identifier is recognized and that context is

in the awaiting authentication state--the server performs the authentication process.

The server may verify that the client's IP address matches that in the previous request that created the "pending" context. The only risk is that someone might change proxies at whim, which seems unlikely.

If the authentication process fails, the server refuses to process the request, but does not delete the "pending" security context. It generates a 401 Unauthorized response with a WWW-Authenticate header that indicates failure:

```
WWW-Authenticate:
        Remote-Passphrase
        Realm="nonsense",
        State="Failed"
```

It is up to the client to try the request again (without an Authorization header), restarting the entire process, if it believes that it was using the wrong pass phrase but it now has the right pass phrase.

[Sending another "Initial" WWW-Authenticate header would provoke a loop: the browser would calculate a new response and retry the request, which is pointless if the browser's idea of the pass phrase is wrong, so we indicate the failure.]

[One could argue that the browser should forget whichever cached pass phrase it used, in order to prompt for it again if the user tries to next time. But the pass phrase might have been correct, depending on what exactly went wrong at the server.]

Otherwise, having successfully authenticated the user, the server processes the client's request and returns an appropriate response, including in its reply:

```
WWW-Authenticate:
        Remote-Passphrase
        Realm="realm in use",
        State="Authenticated",
        Session-Key="base64 encoding of session key",
        Response="base64 encoding of response"
```

The "Authenticated" state indicates that the user was successfully authenticated, and includes the session key, masked so only the user can extract it (Kusu), and the authentication deity's proof of the service's identity (Au, not Rs). The realm is ignorable, but should indicate the realm in which the identity was authenticated.

8.2 Reauthentication cheating

In subsequent requests, the client tries to cheat by including an Authorization header in its request:

```
Authorization:
        Remote-Passphrase
        State="Cheating",
        Security-Context="opaque",
        Response="base64 encoding of response"
```

where the response is calculated based on the previously agreed-upon values plus the canonicalized method and URI of this request as explained in Part One of this specification.

> [The HTTP specification suggests that clients be allowed to replay the previous Authorization header, but it includes an escape clause--"for a period of time determined by the authentication scheme"--so we simply declare that period of time to be zero.]

If the server is willing to accept the use of reauthentication cheating, and the response is correct, the server processes the request without comment. If it recognizes the security context but is not willing to cheat--e.g., it recognizes a replay--the server demands reauthentication. If it does not recognize the security context or if it recognizes the context but the client's response is incorrect, the server demands authentication but does not delete the existing security context.

> [Perhaps the user is referring to a security context that has expired because it's been a long time since the user last referred to it. And this can happen legitimately, if the user refers to an expired security context and the server reuses context identifiers. We do not delete an existing context because that would provide a way for an attacker to delete security contexts.]

In either of these cases, the server responds 401 Unauthorized and includes the appropriate WWW-Authenticate header. To require authentication, refer to the preceding section; to require reauthentication, refer to the next section.

8.3 Reauthentication

If the server chooses to require reauthentication, it replies 401 Unauthorized and includes the header

```
WWW-Authenticate:
```

```
                Remote-Passphrase
                Realm="realm in use",
```

```
            State="Reauthenticate",
            Challenge="base64 encoding of service challenge"
```

The client retries the request with an Authorization field:

```
    Authorization:
            Remote-Passphrase
            State="Reauthenticate",
            Security-Context="opaque",
            Challenge="base64 encoding of user challenge",
            Response="base64 encoding of response"
```

If the response is correct--the user has proven his knowledge of the previously generated Kus for this context--the server processes the request and includes in its reply:

```
    WWW-Authenticate:
            Remote-Passphrase
            Realm="realm in use",
            State="Reauthenticated",
            Response="base64 encoding of response"
```

The past-tense state indicates successful reauthentication, and includes the server's response; this response is of debatable relevance to HTTP, of course, given that the client's use of reauthentication cheating implies its willingness to trust that the server's identity has not changed.

If the client's response is incorrect, the server does not process the request. However, there's a possibility that the client attempted to do reauthentication with an old security context identifier that has been reused by the server. Although the server should avoid reusing security context identifiers, it can attempt to avert the problem by forcing authentication by responding 401 Unauthorized and including the header described above under Authentication.

Remote Passphrase Authentication
Part 3
Service-to-Deity Protocol

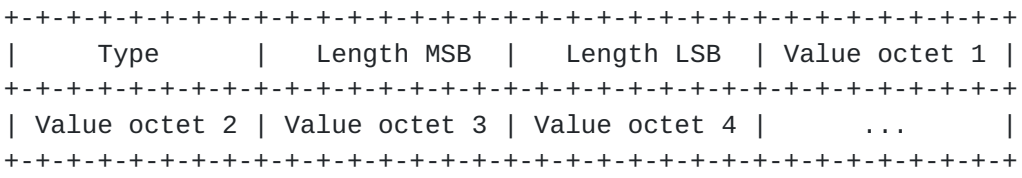
9 Introduction

The service sends a request to the authentication diety and receives a reply. The requests and replies may be packaged in UDP datagrams, or as byte streams over a TCP connection. The tradeoff is primarily that opening a TCP connection requires multiple round trip delays, where UDP doesn't; but TCP avoids the "I wonder whether it's actually running" issue.

How to find the deity is a service configuration issue. The service must know the IP addresses, TCP or UDP port numbers, etc., for the deities for a particular realm; it must also know its name and pass phrase in that realm.

10 Object formats

Every message is an object composed of other objects. Every object consists of a type-length-value encoded structure:

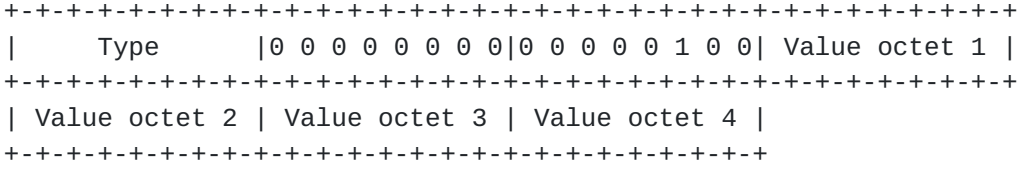


In this picture, each box represents one octet. Octets are transmitted in order from left to right, top to bottom.

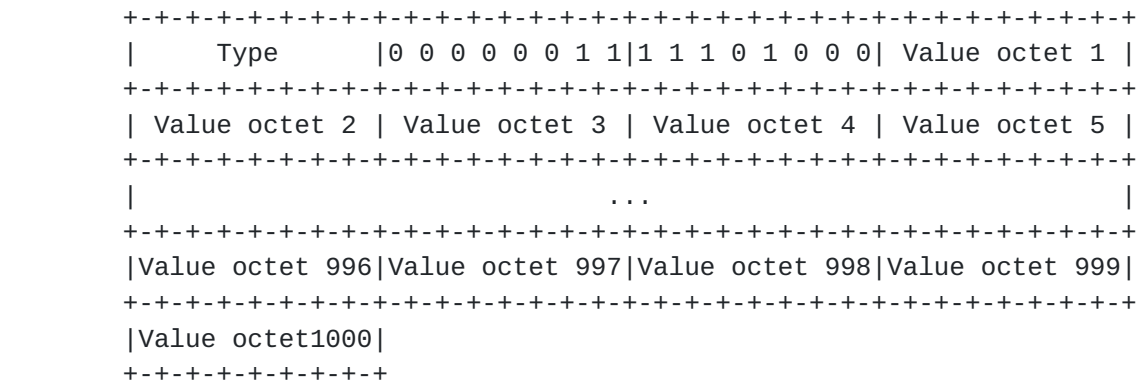
"Type" is a single octet that identifies the type of the object.

"Length" indicates the number of octets following the length field, as a 16-bit, big-endian value. The appropriate number of value octets--possibly none--follow the length field. Their meaning is determined by the type of the object; in some cases, the value octets contain a sequence of other objects.

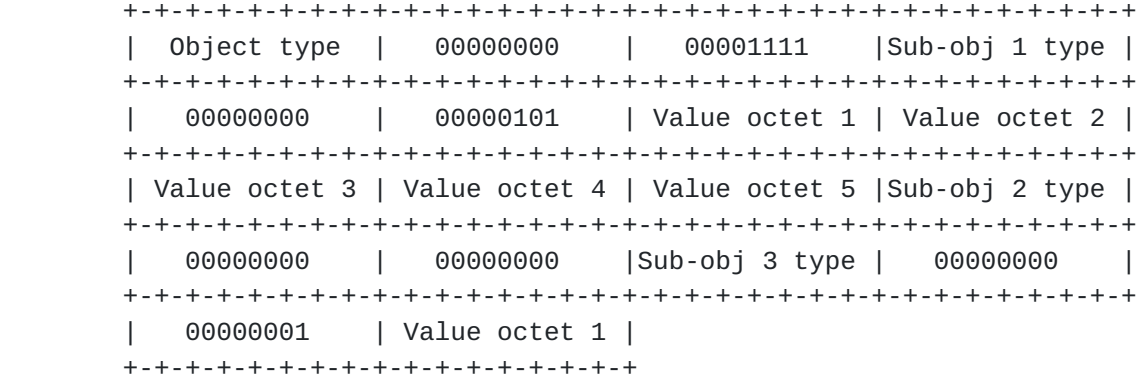
Here is an example of an object that contains 4 value octets:



And here is an example of an object that contains 1,000 value octets:



No padding or alignment is used; if an object contains sub-objects, they follow one another with no padding. For example, an object whose value consists consists of three sub-objects might look like this:



In this example, we have a single object whose value contains 15 octets. In this example, the value is a sequence of three objects, the first of which contains five octets, the second of which is zero length, and third of which contains one octet. The meaning of each object depends on its type; we'll describe all object types in detail after describing the message objects.

We'll sometimes use the term "sub-object" to refer to an object when it is a part of another object, but this is merely a matter of terminology. There is no difference in encoding nor in the meaning of the type field, regardless of whether the object is contained in some other object or not.

All messages may contain a "blob" that conveys information defined by a particular deity. The blob is envisioned for use in three contexts.

* In a request, a service may use the blob to tell the deity the nature of the action for which authentication is being performed, if there's some reason to do so. In addition, the service might ask

the diety for particular information about the username being authenticated, although, in the general case, the deity will already know what additional information to return to a particular

service.

* In an affirmative response, the deity may return additional information about the username.

* In other responses, the blob might indicate something about the nature of the problem.

In general, different deities and services will have different information that's appropriate for inclusion in the blob, so it is difficult to conceive of a truly "standard" set of information. However, we define one format that we'll describe below.

11 Message object types

There are six message object types, one for a request and six kinds of replies.

* Authentication request
* Authentication response, affirmative
* Authentication response, no service
* Authentication response, negative
* Authentication response, invalid service
* Authentication response, problem

The various response flavors indicate various conditions of the account as described below.

Remember, a message is simply an object that contains other objects. The message itself is encoded as a type, length, and value, as described above, where the value consists of the concatenation of the component objects of that message; each component object consists of its own type, length, and value. Unless stated to the contrary, all messages must contain exactly the objects indicated in the order shown. Optional components, such as the blob, may be omitted.

> [When appropriate, it is possible to add extensions, or make a sub-object optional, yet parse the containing object successfully. But in a security protocol, it is best to stick to well-defined formats, rather than adopting a "construct them any way you wish" attitude.]

Contents of the component objects are explained in more detail following the descriptions of the message objects.

11.1 Authentication request

An authentication request contains the following sub-objects.

Request identifier
Nr (Realm name)
Ns (Service name)
Nu (User name)
Cu (Challenge from user)
Cs (Challenge from service)
Ts (Timestamp from service)
Ru (Response from user)
Blob (optional)
Rs (Response from service)

The value contained in most of the sub-objects matches the value described in Part One of this specification.

The request identifier contains arbitrary data that is not interpreted by the deity; it is simply echoed in a response to provide a way for the requesting service to match requests and responses.

The blob contains additional information about the request, and is described below. Usually, it will be omitted or null, i.e., zero-length.

Rs is calculated as MD5(Ps + Z + M + Ps), where M is the request shown above, octet by octet, from the type octet for the message object itself through the last length octet of the length field of the Rs object. Thus, it serves to protect the entire request, including its structure, length, etc., and is a different calculation from that shown in the authentication document.

11.2 Authentication response, affirmative

An affirmative response indicates that the username is recognized, and is indeed the user you're talking to.

Request identifier
Canonical Nu (User name, case corrected)
Kuss (Key obscured for service)
Kusu (Key obscured for user)
Au (Authentication value for user)
Blob (optional)
As (Authentication value for service)

The response contains the canonical username in the desired case; this is not the same object type as Nu in the request. In an environment that is not case sensitive, this is the preferred form of the name, which might differ from the name in the request.

The blob may contain additional information about the username; see

below.

As is calculated as

MD5(Ps + Z + Ns + Nu + Nr + Kuss + Cs + Cu + Ts + Kus + M + Ps)

where M is the request shown above, octet by octet, from the type octet for the containing object through the last octet of the length field of the As object, inclusive. This serves to protect the entire request, and differs from the calculation in the authentication document by the addition of the message contents as shown. Note that the Nu mentioned as the third component in the formula is the originally specified username, not the altered-case version in the response message.

Beware that an affirmative response does not necessarily mean that it is reasonable to provide service to the user. Often, there are criteria beyond a "yes" answer, which could mean anything from "it's a valid user" to "it's a valid user but not billable" to "it's an account that was signed up five minutes ago and we haven't had a chance to look at it yet."

Typically, the authentication deity applies criteria appropriate to the requesting service. For example, if the service doesn't want to allow "free" users, the authentication deity would be configured to return a no-service response for such a user. Alternatively, the deity could be configured to provide an affirmative response but include information in the blob that would permit the service to distinguish "free" from "paying" users and treat them differently.

11.3 Authentication response, no service

The no-service response is an indication that the user is whom he claims to be, but you should not provide service to him for one reason or another. For example, he might be a "free" user but your service is provided only to paying accounts; his billing choices might not include your service; or Customer Service might be waiting for him to provide a new credit card number.

The authentication deity's configuration for this particular service determines the criteria applied by the deity when making the decision to reply affirmative or no service.

Request identifier
Canonical Nu (User name, case corrected)
Kuss (Key obscured for service)
Kusu (Key obscured for user)
Au (Authentication value for user)
Blob (optional)
As (Authentication value for service)

The contents of this object are identical to those for an affirmative response, but the service would not normally use the keys or Au

values. The blob might include information useful in distinguishing the reason for the no service response, if appropriate for this service.

11.4 Authentication response, negative

A negative response means the user is not who he says he is. Whether there is such a username, but that's not who you're talking to; there is such a username, but it is not an enabled account; or there is no such username, is not specified.

Request identifier
Blob (optional)
As (Authentication value for service)

As is calculated as MD5(Ps + Z + M + Ps). The message may contain a blob if there is additional information about the problem, e.g., for logging, but it may be omitted.

11.5 Authentication response, invalid service

An invalid request response means the request could not be processed because you (the service) are not whom you claim to be, based on your apparently not knowing the service's pass phrase or based on any other kind of authentication checking done by the deity.

Request identifier
Blob (optional)

The blob, if present, contains information that allows the deity administrators to trace the problem. There is no As field, because there is no shared secret to authenticate the response. This presents some obvious denial of service issues.

11.6 Authentication response, problem

A "problem" response indicates that the request could not be processed for some reason. This could indicate a failure in the system, an unparsable request, or a request for a realm that isn't handled by this deity.

Request identifier
Blob (optional)
As (optional)

The blob may contain information that allows the deity administrators to trace the problem. As might or might not be present, depending on the nature of the problem, i.e., whether there is a known shared secret with the server; if present, it is calculated as MD5(Ps + Z +

M + Ps).

12 Object types

The following types of objects are defined in this protocol. These object types apply to the messages themselves and objects contained in messages. These types do not apply to the contents of the blob.

> [Numbers for the object type field are indicated for each type, but are not necessarily accurate in this draft of the document.]

Authentication request--type 1--The request to the authentication deity. Its contents consist of a sequence of other objects as described elsewhere in this document.

Authentication response, affirmative--type 2

Authentication response, no service--type 3

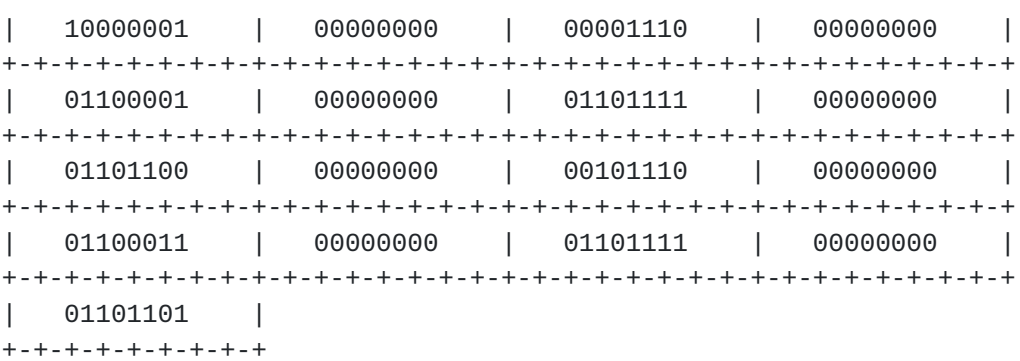
Authentication response, negative--type 4

Authentication response, invalid service--type 5

Authentication response, problem--type 6

Request identifier--type 128--A request must contain an identifier to assist in matching replies to requests. This identifier is opaque to the deity, and is simply echoed in the reply, so its value is defined only by the requesting entity. The value should, of course, be unique for each request, but it is otherwise meaningless. It may be of any length.

Realm name--type 129--The name of the realm in which the user's and service's identities exist. This is included in the request to allow a deity to serve more than one realm. The value consists of the name in Unicode, in big-endian order. There is no terminating null character, and the realm is generally treated as being case insensitive. For example, the realm aol.com might look like this:



That's type 129, fourteen octets follow, and the big-endian Unicode

representation of the seven characters aol.com.

Service name--type 130--The name of the service in big-endian Unicode.

User name--type 131--The name of the user in big-endian Unicode, e.g., gsb.

User challenge--type 132--The user's challenge, a sequence of random octets. The length is not bounded by the protocol, but the deity will impose length restrictions, e.g., a minimum and maximum length. All bit patterns are legal in the challenge.

Service challenge--type 133--The service's challenge, a sequence of random octets. The length is not bounded by the protocol, but the deity will impose length restrictions, e.g., a minimum and maximum length. All bit patterns are legal in the challenge.

Time stamp--type 134--The time stamp, containing 14 octets with the value specified in Part One of this specification.

User's response--type 135--The user's response, containing 16 octets with the value specified in Part One of this specification. This is a binary value, so any bit pattern is possible in this value.

Service's response--type 136--The service's response, calculated as described elsewhere in this document. This is a binary value, so any bit pattern is possible in this value.

Key obscured for user--type 137--The key for the user, containing 16 octets as described in Part One of this specification. This is a binary value, so any bit pattern is possible in this value.

Key obscured for service--type 138--The key for the service, containing 16 octets as described in Part One of this specification. This is a binary value, so any bit pattern is possible in this value.

Authentication proof for user--type 139--The authentication proof, Au, for the user, containing 16 octets as described in Part One of this specification. This is a binary value, so any bit pattern is possible in this value.

Authentication proof for service--type 140--The authentication proof, As, for the service, containing 16 octets calculated as described elsewhere in this document (not as described in Part One of this specification). This is a binary value, so any bit pattern is possible in this value.

Canonical user name--type 141--The username adjusted to canonical

case, in big-endian Unicode.

Blob--type 142--Deity-specific request and response information.

13 The blob

The first two octets in the blob contain a major/minor version number to indicate the format of the blob. This format is version 1.0, the two octets 0x01 0x00.

This is followed by multiple null-terminated attribute name-value pairs with the final attribute followed by an additional null. Names and values are represented in the ISO 8859-1 character set. The format of an attribute/value pair is

```
name[=[value]]
```

That is, an attribute name may exist alone, implying that just its existence is significant. Additionally, the value may be a null string, in which case the '=' character is followed immediately by a null character.

Attribute names may consist of letters, digits, hyphens, and underscores; letter case is not significant. The first character must be a letter or underscore. Attribute values may contain any ISO 8859-1 graphic character; they may not contain control characters, but they may contain spaces (i.e., octet values 00-1F and 7F-9F are illegal). If, for some reason, a particular attribute value should contain arbitrary octet values, it must be encoded somehow, e.g., by using base64 or MIME quoted-printable encoding. (We presume that you know, when you get around to using an attribute value, how it's formatted, so as to know whether some form of decoding is necessary.)

Remove Passphrase Authentication
Part 4
GSS-API Handshake

Table of Contents

14 Introduction

This section explains the GSS-API handshake for RPA, used by connection-oriented protocols such as NNTP, POP3, etc. We define "tokens" that are conveyed from end to end; they contain arbitrary binary data, so a text-based protocol would generally use base64 encoding to transport the token.

15 The handshake

RPA uses a four-way or five-way handshake. The protocol requires only a four-way handshake, and the additional hop is used to support servers that require the last authentication step to go from the client to the server. Clients should support both the four-way and the five-way handshake. The server indicates which type it requires via the "Selected Version" field described below.

15.1 Token 1 (negotiation, client-to-server)

Token Length
Mechanism Type
Earliest Version
Latest Version
Flags

15.2 Token 2 (challenge, server-to-client)

Token Length

Mechanism Type
Selected Version

Service Challenge
Timestamp
Realm List

15.3 Token 3 (authentication, client-to-server)

Token Length
Mechanism Type
Identity
User Challenge
User Response

15.4 Token 4 (authentication, server-to-client)

Token Length
Mechanism Type
User Authentication
Obscured Session-Key
Status

Note: If the Status field is non-zero, indicating an authentication error, the User Authentication and Obscured Session Key values should be ignored; although, the values must still be legal values (that is, the field lengths should be valid).

15.5 Token 5 (hack, client-to-server)

Token Length
Dummy Byte

15.6 Field descriptions

Dummy Byte: A single octet with the value 0.

Earliest Version: Two binary octets indicating the earliest version of the RPA protocol that the client-side implementation of the mechanism supports. First octet is major version; second octet is minor version. The client should indicate that it supports version 1.0.

Flags: Two binary octets, in network (big-endian) byte order, consisting of bit flags. In versions 1.0 and 2.0, bit 0 indicates that mutual authentication is requested. Mutual authentication should always be selected. In version 3.0 of the protocol, there are no flags defined because Token 4 is always returned from the server to the client (that is, mutual authentication is always selected).

Note: Bit 0 is the low-order bit of the second octet.

Identity: The selected user identity, in the form "name@realm,"
preceded by two binary octets, in network (big-endian) byte order,

representing the length of the identity string (in characters, not octets). The identity is encoded using the ISO-8859-1 character set.

Latest Version: Two binary octets indicating the latest version of the RPA protocol that the client supports. First octet is major version; second octet is minor version. The client should indicate that it supports version 3.0.

Mechanism Type: An OID (object identifier) specifying the RPA SSPI scheme. The value is the following sequence of bytes 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x86, 0xF8, 0x73, 0x01, 0x01.

Obscured Session-Key: The session-key, generated and obscured by the deity, encoded as length plus binary value as described above.

Realm List: The list of realms in which the service can accept identities, along with the service name for each realm. Each realm is separated by a single space character; the entire string is preceded by two binary octets, in network (big-endian) byte order, representing the length of the realm list string (in characters, not octets):

"foo@compuserve.com bar@aol.com"

The realm name begins with the first character to the right of the rightmost '@' character; realm names cannot contain '@' characters. The realm list is encoded using the ISO-8859-1 character set.

Selected Version: Two binary octets indicating the version of RPA the server-side implementation of the mechanism has elected to use. It must be no earlier than "earliest version" and no later than "latest version" as specified in Token 1. If the server indicates version 1.0 or version 3.0, then the client MUST perform the five-way handshake. If version 2.0 is specified, the client should perform the 4-way handshake. Version 3.0 indicates that Token 4 contains the Status field that is used to report certain errors from the server to the client. Previous versions do not contain the Status field.

Service Challenge: The service's random challenge, consisting of a one octet length field followed by the specified number of octets containing the challenge. This is raw, binary data, where any bit patterns are allowed.

Status: A single octet indicating the status of the authentication attempt. A non-zero value indicates there was an error authenticating the user. The following status codes are defined for version 3.0 of the protocol:

0 Success

1 Restricted user (something wrong with user's account)
2 Invalid user ID or passphrase

3 Deity error

Timestamp: The service's timestamp encoded using the ISO 646 (ASCII) textual representation of the current universal time (UTC) in exactly 14 octets, using 24-hour time with leading zeroes, e.g. 19950805011344.

Token Length: To conform to the GSS-API standard for token formats, the token is encapsulated in an ASN.1 SEQUENCE that includes its length. This consists of one octet containing the value 0x60, an ASN.1 SEQUENCE, followed by the length of the token, indicating the number of octets following the length field. This length is ASN.1 DER encoded: if the length is less than 128 octets, it consists of a single octet containing the length. Otherwise, the length is represented using the minimum number of octets required, in network (big-endian) byte order, preceded by an octet whose MSB is set and whose remaining bits indicate the number of octets in the length.

For example, if there are 126 octets, the length is one octet with a value of 0x7E (126 decimal). If there are 150 octets, the length is two octets: 0x81 0x96. If there are 258 octets, the length is three octets: 0x82 0x01 0x02.

User Authentication: The deity's response to the user, encoded as length plus binary value as described above.

User Challenge: The user's random challenge, consisting of a one octet length field followed by the specified number of octets containing the challenge. This is raw, binary data, where any bit patterns are allowed.

User Response: The user's response, encoded as length plus binary value as described above.

16 Sample conversation

RPA SSPI authentication can be used with the POP3 AUTH command to perform authentication between a client and server. The following is an example conversation between a client and server:

Server: listens at TCP port 110

Client: connects to port 110

Server: +OK <server message>

Client: AUTH RPA

```
   Server: +
```

The client and server communicate Base64-encoded tokens using one or more of the following sequence, ending with the client:

a) Client: <Base64-encoded token>
b) Server: + <Base64-encoded token>

Server: +OK <server message>

Security Considerations

This entire document is about security.

Author's Address

Gary S. Brown
CompuServe Incorporated
5000 Arlington Centre Blvd
Columbus OH 43220
USA

<gsb@csi.compuserve.com>