GSS-API のプログラミング

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part Number 806-4504–10
2000 年 7 月

原典: *GSS-API Programming Guide*

Part No: 806-3814-10

Revision A

目次

はじめに

『*GSS-API* のプログラミング』では、Generic Security Services Application Programming Interface (GSS-API) について説明します。GSS-API は、開発者が、特定の機構用に明示的にプログラムする必要なく、Kerberos v5 などのセキュリティ機構を利用するアプリケーションを作成できるようにするフレームワークです。したがって、GSS-API を使用するプログラムは移植性が高く、あるプラットフォームから別のプラットフォームに移植できるだけでなく、あるセキュリティ設定から別のセキュリティ設定に移植したり、ある転送プロトコルから別の転送プロトコルに移植したりできます。GSS-API は、システムに実装されている実際のセキュリティ機構と整合性のある、複数のレベルのデータ保護を提供します。

対象読者

『*GSS-API* のプログラミング』は、クライアントサーバープログラムなどのように、あるアプリケーションから別のアプリケーションにデータを安全に転送するプログラムを作成する C 言語開発者を対象としています。GSS-API を理解または使用するには、転送プロトコルやネットワークプログラミングの特別の知識は必要ありません。もちろん、GSS-API 自身は転送を行わないため、ネットワークアプリケーションを作成するときには、ネットワークプログラミングの知識は必要です。

このマニュアルをお読みになる前に

C プログラミングの知識があることが必要です。セキュリティ機構の基本的な知識があると便利ですが、必須ではありません。このマニュアルを使用するにあたっては、ネットワークプログラミングについての専門知識は必要ありません。

内容の紹介

第 1 章では、GSS-API の概要を示します。この章では、GSS-API を使用するための一般的な手順を説明し、基本的な概念を示し、さらに、最も重要な機能を一部詳細に説明します。

第 2 章では、付録 A にリストされているサンプルプログラムを使って、GSS-API の使用法を概略的に説明します。

付録 A では、2 つのサンプルプログラムである GSS-API クライアントと GSS-API サーバーのプログラムリストを示します。

付録 B では、GSS-API 機能、状態コード、およびデータ型の参照情報を示します。

付録 C では、GSS-API においてセキュリティ機能を指定する方法について簡単に説明します。

付録 D では、GSS-API の Sun の実装に固有な特徴について説明します。

付録 E では、Kerberos v5 セキュリティ機構が戻す状態コードの表を示します。

用語集では、このマニュアルで使用されている用語の定義を示します。

関連マニュアル

次のマニュアルも参考になります。

- 『*ONC+* 開発ガイド』

GSS-API については、次の 2 つのマニュアルでも説明されています。この 2 つの文書はアプリケーション開発者向けというよりも GSS-API 実装者向けです。『*Generic Security Service Application Program Interface*』(`ftp://ftp.isi.edu/in-notes/rfc2743.txt`) は、GSS-API の概念的な概要を示し、『*Generic Security Service API Version 2: C-Bindings*』(`ftp://ftp.isi.edu/in-notes/rfc2744.txt`) は C 言語ベースの GSS-API の特徴について説明します。

Sun のマニュアルの注文方法

専門書を扱うインターネットの書店 Fatbrain.com から、米国 Sun Microsystems™, Inc. (以降、Sun™ とします) のマニュアルをご注文いただけます。

マニュアルのリストと注文方法については、`http://www1.fatbrain.com/documentation/sun` の Sun Documentation Center をご覧ください。

Sun のオンラインマニュアル

`http://docs.sun.com` では、Sun が提供しているオンラインマニュアルを参照することができます。マニュアルのタイトルや特定の主題などをキーワードとして、検索を行うこともできます。

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 **P–1** 表記上の規則

字体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 system%
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	system% **su** password:
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm *filename* と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「 」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	sun% **grep '^#define ** **XV_VERSION_STRING'**

ただし AnswerBook2™ では、ユーザーが入力する文字と画面上のコンピュータ出力は区別して表示されません。

コード例は次のように表示されます。

- C シェルプロンプト

```
system% command y|n [filename]
```

- Bourne シェルおよび Korn シェルのプロンプト

```
system$ command y|n [filename]
```

- スーパーユーザーのプロンプト

```
system# command y|n [filename]
```

[ ] は省略可能な項目を示します。上記の例は、*filename* は省略してもよいことを示しています。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: Shift キーを押します)。ただし、キーボードによっては Enter キーが Return キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、Ctrl-D は Control キーを押したまま D キーを押すことを意味します。

一般規則

- このマニュアルでは、「IA」という用語は、Intel 32 ビットのプロセッサアーキテクチャを意味します。これには、Pentium、Pentium Pro、Pentium II、Pentium II Xeon、Celeron、Pentium III、Pentium III Xeon の各プロセッサ、および AMD、Cyrix が提供する互換マイクロプロセッサチップが含まれます。

第 1 章

GSS-API の概要

GSS-API の紹介

Generic Security Standard Application Programming Interface (GSS-API) は、ピアとなるアプリケーションに送信されるデータを保護する方法をアプリケーションに提供します。通常、ピアとなるアプリケーションとは、同じマシン上のクライアントから別のマシン上のサーバーまで、様々な可能性があります。名前が示すとおり、GSS-API を使用すると、プログラマはセキュリティの点で汎用的なアプリケーションを作成できます。つまり、特定のプラットフォーム、セキュリティ機構、保護の種類、または転送プロトコル向けに特定のセキュリティ実装を施す必要はありません。GSS-API を実装したアプリケーションはセキュリティを制御できます。しかし、GSS-API を使用するプログラマは、ネットワークデータを保護する方法についての詳細を知らなくてもプログラムの作成を行うことができます。したがって、GSS-API を使用するプログラムはネットワークセキュリティに関して移植性がより高くなります。この移植性が Generic Security Standard API の最も優れた特徴です。

GSS-API 自身はセキュリティサービスを実際に提供しません。GSS-API はセキュリティサービスを汎用的な方法で呼び出し側に提供するためのフレームワークであり、実際の機構やテクノロジなど (Kerberos v5 や公開鍵テクノロジなど) を幅広くサポートできます。図 1–1 を参照してください 。

図 1–1　GSS-API 層

GSS-API の主な機能は簡単に言うと次の 2 つです。

1. セキュリティコンテキストを作成し、アプリケーション間でデータを受け渡します。コンテキストとは、2 つのアプリケーション間における一種の「信用状態」であると考えられます。コンテキストを共有するアプリケーションは誰が相手であるかを知っており、したがって、そのコンテキストが継続する限り、アプリケーション間でデータを転送できます。
2. 1 つまたは複数の種類の保護 (セキュリティサービス) を転送されるデータに適用します。セキュリティサービスについては、15ページの「セキュリティサービス」を参照してください。

もちろん、GSS-API は上記よりももっと複雑です。GSS-API の他の機能には、データ変換、エラー検査、ユーザー特権の委託、情報の表示、および ID の比較などがあります。GSS-API にはさまざまなサポート機能や便利な機能があります。

アプリケーションの移植性

上記のとおり、GSS-API は複数の種類の移植性をアプリケーションに提供します。

- 機構非依存性。GSS-API は実装されている機構に対して、汎用的なインタフェースを提供します。デフォルトのセキュリティ機構を指定することによって、アプリケーションはどの機構 (たとえば Kerberos v5 など) を使用しているかや、どの種類の機構を利用しているかを知る必要がありません。たとえば、アプリケーションがユーザーの資格 (credential) をサーバーに転送するとき、その資格が Kerberos 形式であるか、他の機構の形式であるかを知る必要はありません。あるいは、その資格が機構によってどのように格納されるか、その資格がアプリケーションによってどのようにアクセスされるかを知る必要もありません。必要であれば、アプリケーションは使用する特定の機構を指定できます。

- プロトコル非依存性。GSS-API は特定の通信プロトコルまたはプロトコル群に依存しません。GSS-API は、ソケット、RCP、TCP/IP などを使用するアプリケーションで使用できます。

  RPCSEC_GSS は、GSS-API と RPC をスムースに統合するために追加される層です。詳細は、16ページの「RPCSEC_GSS 層」を参照してください。

- プラットフォーム非依存性。GSS-API は、アプリケーションが動作しているオペレーティングシステムにまったく依存しません。

- 保護品質に対する非依存性。保護品質 (Quality of Protection: QOP) とは、データを暗号化したり、暗号タグを生成したりするときに使用されるアルゴリズムの種類を示します。GSS-API を使用し、GSS-API が提供するデフォルトを使用すると、プログラマは QOP を無視することができます。必要であれば、アプリケーションは QOP を指定することもできます。

セキュリティサービス

GSS-API が提供する基本的なセキュリティは認証です。認証とは ID の検証のことです。あるユーザーが認証されるということは、そのユーザーが自分が宣言したユーザーとして認識されたことを意味します。

実際の機構がサポートする場合、GSS-API は認証以外にも次の 2 種類のセキュリティサービスを提供します。

- 整合性。データを送信しているアプリケーションが、本当に自ら主張しているとおりのアプリケーションであるかどうかを知るだけでは、必ずしも常に十分であるとは言えません。データ自身が破壊または破損している可能性もあります。GSS-API はメッセージ整合性コード (MIC) と呼ばれる暗号化タグをデータに添付することによって、ユーザーのところに到着したデータとアプリケーション

が送信したデータが同じであることを証明します。このようにデータの有効性を検証することを整合性と呼びます。

- 機密性。認証と整合性はどちらもデータ自身をそのままにしておくため、誰かに横取りされたときは読まれてしまいます。したがって、実際の機構がサポートする場合、GSS-API はデータを暗号化できます。このようにデータを暗号化することを機密性と呼びます。

GSS-API で使用できる機構

GSS-API の現在の実装では、Kerberos v5 セキュリティ機構だけに適用されます。この中には、Sun が変更して作成した Solaris Enterprise Authentication Mechanism (SEAM) も含まれます。詳細は、『*Solaris* のシステム管理 *(*第 *2* 巻*)*』の「SEAM の概要」を参照してください。したがって、GSS-API を使用するプログラムが動作しているシステムには、Kerberos v5 または SEAM がインストールされている必要があります。

RPCSEC_GSS 層

RPC (Remote Procedure Call) プロトコルをネットワークアプリケーションに使用するプログラマは、RPCSEC_GSS を使用してセキュリティを提供できます。RPCSEC_GSS は GSS-API 上にある別の層であり、GSS-API のすべての機能を RPC 向けの方法で提供します。事実、RPCSEC_GSS は GSS-API の多くの側面をプログラマが意識する必要がないようにするため、特に、RPC セキュリティのアクセス性と移植性が向上します。RPCSEC_GSS についての詳細は、『*ONC+* 開発ガイド』を参照してください。

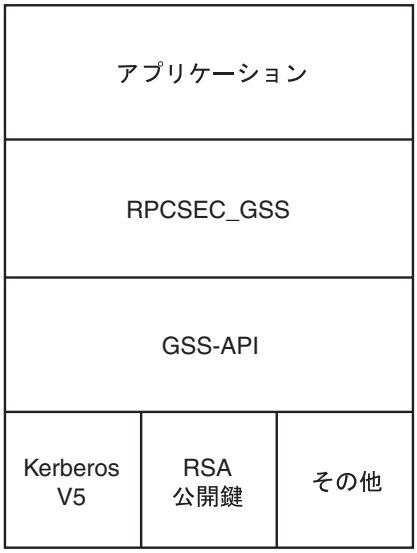

図 1–2　RPCSEC_GSS と GSS-API

GSS-API が行わないこと

GSS-API はデータの保護を簡単にしますが、汎用性という性質を最大限にするために、次のことは行いません。

- セキュリティ資格をユーザーまたはアプリケーションに提供すること。セキュリティ資格は実際のセキュリティ機構が提供する必要があります。GSS-API では、アプリケーションが資格を自動的または明示的に獲得することを許可しています。
- アプリケーション間でデータを転送すること。セキュリティ関連のデータまたは通常のデータのどちらの場合でも、アプリケーション間でデータの転送を処理するのはアプリケーションの責任です。
- 転送されたデータの種類を区別すること。たとえば、データパケットが通常のデータであり、GSS-API 関連のデータではないことを判断するなどです。
- リモート (非同期) エラーによる状態を示すこと。
- マルチプロセスプログラムのプロセス間で送信される情報を自動的に保護すること。
- GSS-API 関数に渡される文字列バッファを割り当てること。19ページの「文字列および類似のデータ」を参照してください。

- GSS-API データ領域を解放すること。GSS-API データ領域の解放は、`gss_release_buffer()` や `gss_delete_name()` などの関数で明示的に行う必要があります。

言語のバインディング

このマニュアルでは現在、GSS-API の C 言語バインディング (関数とデータ型) だけをカバーしています。将来的には、GSS-API の Java バインディングも使用できるようになる予定です。

参照箇所

GSS-API については、次の 2 つのマニュアルでも説明されています。この 2 つの文書はアプリケーション開発者向けというよりも GSS-API 実装者向けです。『*Generic Security Service Application Program Interface*』 (`ftp://ftp.isi.edu/in-notes/rfc2743.txt`) は、GSS-API の概念的な概要を示し、『*Generic Security Service API Version 2: C-Bindings*』 (`ftp://ftp.isi.edu/in-notes/rfc2744.txt`) は C 言語ベースの GSS-API の特徴について説明します。

基本概念

実際に GSS-API を使用するプロセスを考える前に、4 つの基本的な概念について検証します。プリンシパル、GSS-API データ型、状態コード、およびトークンです。

プリンシパル

ネットワークセキュリティの用語では、「プリンシパル」とは、ユーザー、プログラム、またはマシンを指します。プリンシパルはクライアントまたはサーバーのどちらにでもなり得ます。たとえば、他のマシンにログインしているユーザー (*joe@machine*)、ネットワークサービス (*nfs@machine*)、アプリケーションを実行しているマシン (*swim2birds@eng.company.com*) などがプリンシパルです。

GSS-API では、プリンシパルは特別なデータ型で示されます。20ページの「名前」を参照してください。

GSS-API データ型

次の節では、より重要な表示できる GSS-API データ型について説明します。詳細は、144ページの「GSS-API データ型と値」を参照してください。

注意 - 割り当てられたすべてのデータ領域を解放するのは呼び出し元アプリケーションの責任です。

整数

`int` のサイズはプラットフォームによって異なるため、GSS-API は次の整数型を提供します。

```
OM_uint32
```

これは、32 ビットの符号なし整数です。

文字列および類似のデータ

GSS-API はすべてのデータを内部形式で処理するため、文字列は GSS-API 関数に渡す前に GSS-API 形式に変換しておく必要があります。GSS-API は文字列を `gss_buffer_desc` 構造体で処理します。`gss_buffer_t` は `gss_buffer desc` 構造体へのポインタです。

```
typedef struct gss_buffer_desc_struct {
     size_t     length;
     void       *value;
}  gss_buffer_desc *gss_buffer_t;
```

したがって、文字列は GSS-API 関数に渡す前に `gss_buffer_desc` 構造体に変換しておく必要があります。ここでは、メッセージを受け取ってなんらかの方法で処理する (たとえば、転送する前にメッセージに保護を適用するなど)、次のような汎用的な GSS-API 関数を考えます。

例 1–1　文字列の使用例

```
char *message_string;
gss_buffer_desc input_msg_buffer;

input_msg_buffer.value = message_string;
input_msg_buffer.length = strlen(input_msg_buffer.value) + 1;

gss_generic_function(arg1, &input_msg_buffer, arg2...);

gss_release_buffer(input_msg_buffer);
```

終了時には gss_release_buffer() で input_msg_buffer を解放する必要があることに注意してください。

gss_buffer_desc オブジェクトは文字列だけに使用されるわけではありません。たとえば、トークンも gss_buffer_desc オブジェクトとして処理されます。31 ページの「GSS-API トークン」を参照してください。

名前

「名前」はプリンシパルを指します。つまり、*joe@company* や *nfs@machinename* などのように、人、マシン、またはアプリケーションを指します。GSS-API では、名前は gss_namae_t オブジェクトとして格納され、アプリケーションでは意識する必要はありません。名前は gss_import_name() 関数によって gss_buffer_t オブジェクトから gss_name_t 形式に変換されます。インポートされたすべての名前には関連する名前型が割り当てられます。名前型とは、その名前の形式の種類を示すものです。名前型についての詳細は、27ページの「OID」を参照してください。また、有効な名前型のリストについては、146ページの「名前型」を参照してください。

次に、gss_import_name() の例を示します。

```
OM_uint32 gss_import_name (
       OM_uint32          *minor_status,
       const gss_buffer_t input_name_buffer,
       const gss_OID      input_name_type,
       gss_name_t         *output_name)
```

minor_status　実際の機構から戻される状態コード。29ページの「状態コード」を参照してください。

input_name_buffer	インポートされた名前が格納される gss_buffer_desc 構造体。アプリケーションはこの構造体を明示的に割り当てる必要があります。例 1–2、および 19ページの「文字列および類似のデータ」を参照してください。使用し終わったとき、この引数は gss_release_buffer() で解放する必要があります。
input_name_type	*input_name_buffer* の形式を示す gss_OID。29ページの「名前型」を参照してください。また、有効な名前型のリストについては、146ページの「名前型」を参照してください。
output_name	名前を受け取る gss_name_t 構造体。

次に、例 1-1 で使用した汎用例を少しだけ変更します。ここでは、どのように gss_import_name() を使用するかを示します。まず、通常の文字列を gss_buffer_desc 構造体に変換して、次に、gss_import_name() で gss_name_t 構造体に変換します。

例 **1–2** gss_import_name() の使用例

```
char *name_string;
gss_buffer_desc input_name_buffer;
gss_name_t      output_name_buffer;

input_name_buffer.value = name_string;
input_name_buffer.length = strlen(input_name_buffer.value) + 1;

gss_import_name(&minor_status, input_name_buffer,
                    GSS_C_NT_HOSTBASED_SERVICE, &output_name);

gss_release_buffer(input_name_buffer);
```

インポートされた名前を gss_buffer_t オブジェクトに戻して、gss_display_name() で人が読める形式で表示することも可能です。しかし、実際の機構が名前を保存する方法の違いにより、gss_display_name() は結果の文字列がオリジナルと同じであることを保証できません。GSS-API には他にも名前を処理する関数があります。135ページの「GSS-API 関数」を参照してください。

gss_name_t 構造体は、単一の名前について複数のバージョンを持つことができます。つまり、GSS-API がサポートする機構ごとに 1 つのバージョンが生成されま

す。たとえば、「joe@company」の gss_name_t 構造体は Kerberos v5 用の名前と他の機構用の名前を持つなどです。GSS-API は gss_canonicalize_name() という関数を提供します。この関数は、内部名 (つまり、gss_name_t 構造体) と機構を入力として受け取り、その機構に特定な名前のバージョンを 1 つだけ持つ別の内部名 (これも gss_name_t 構造体) を出力します。

このような機構に固有な名前のことを「機構名 (Mechanism Name: MN)」と呼びます。機構名は、機構自身の名前を指すのではなく、その機構が生成したプリンシパルの名前を指すため、すこし紛らわしいかもしれません。図 1–3 に、このプロセスを図示します。

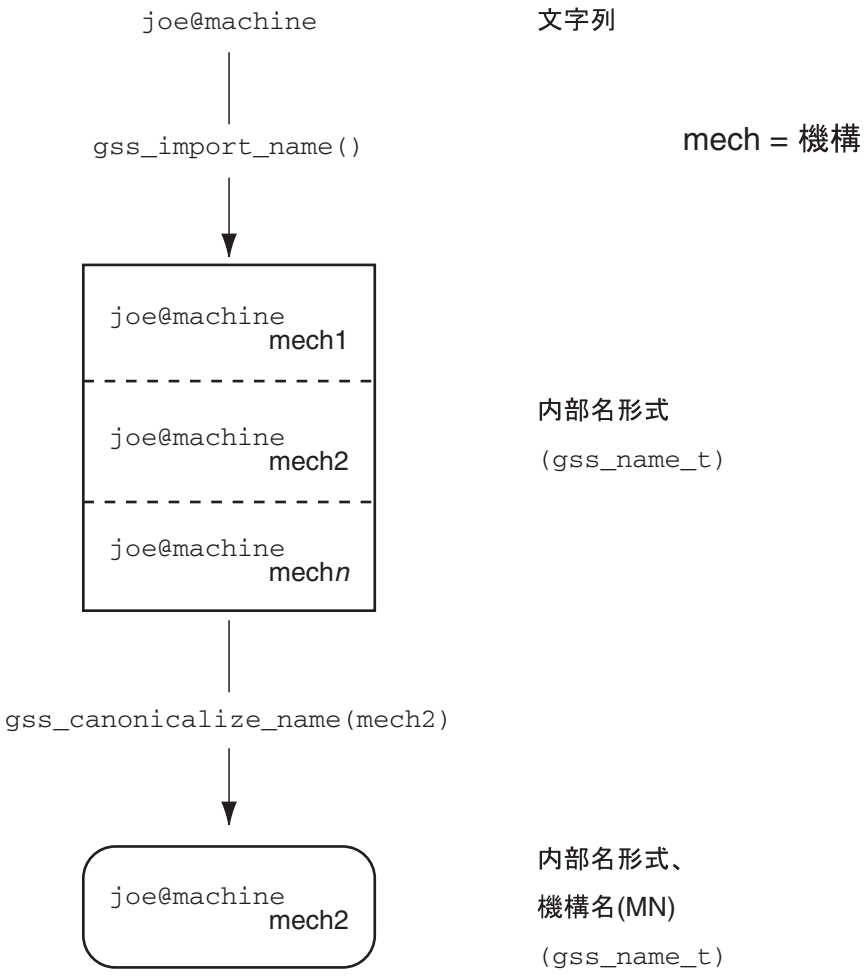

図 1–3　内部名と機構名 (MN)

名前の比較

なぜこのような関数が便利なのでしょうか。ここでは、サーバーがクライアントから名前を受け取り、その名前をアクセス制御リストから検索する例を考えます。アクセス制御リスト (Access Control List: ACL) とは、特定のアクセス権を備えたプリンシパルのリストのことです。このためには、次のような方法が考えられます。

1. gss_import_name() で、クライアント名を GSS-API 内部形式でインポートします (まだインポートされていない場合)。

   サーバーの中には、内部形式で名前を受け取るものもあります。この場合、この手順は必要ありません。特に、サーバーがクライアント独自の名前を検索する場合です。コンテキストの起動中、クライアント独自の名前は内部形式で渡されます。

2. gss_import_name() で、各 ACL 名を インポートします。
3. gss_compare_name() で、インポートした ACL 名とインポートしたクライアント名をそれぞれ比較します。

図 1–4 に、このプロセスを示します。ここでは手順 1 が必要であると仮定します。

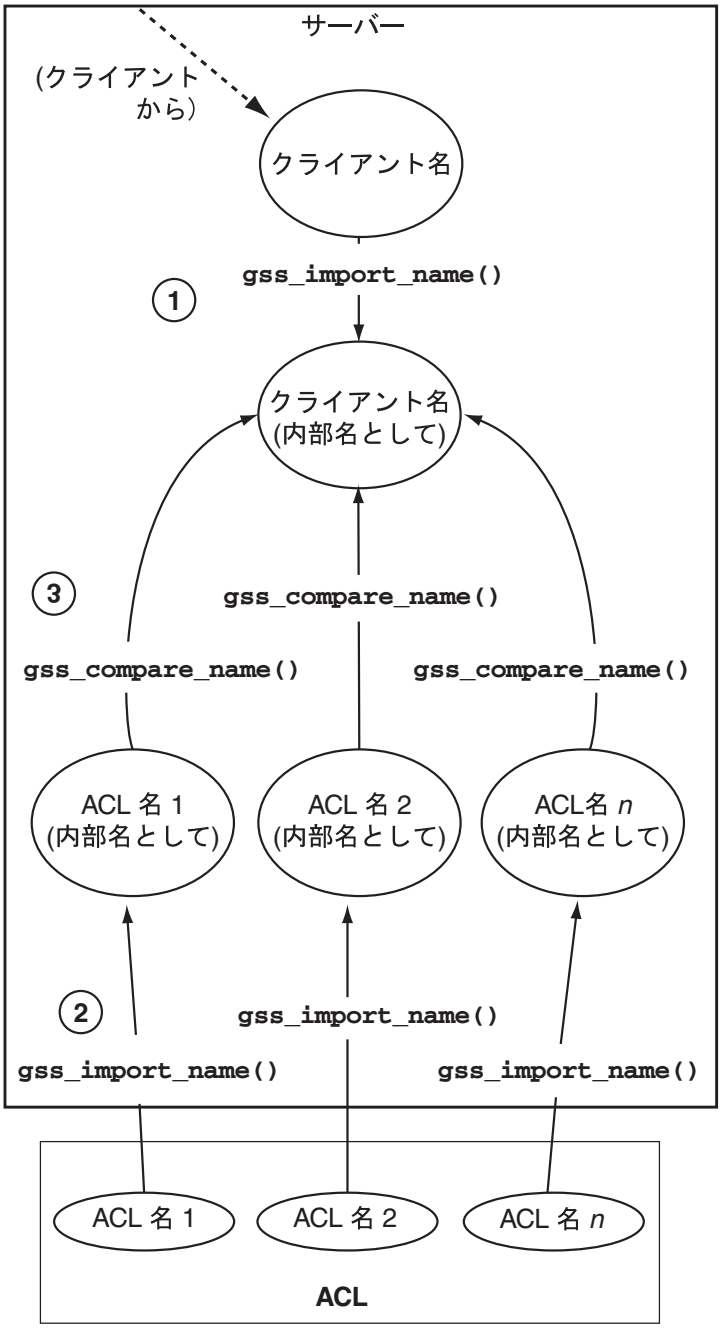

図 1–4　名前の比較 (遅い)

この手順は、クライアント名と比較する名前の数が少ない場合には有効です。しかし、非常に処理が遅いため、大きなリストを比較するときには不都合です。ACL 名

ごとに gss_import_name() と gss_compare_name() を実行すると、膨大な CPU サイクルが必要です。したがって、次のような方法を使用します。

1. gss_import_name() で、クライアント名をインポートします (まだインポートされていない場合)。

   前述の方法と同様に、サーバーが内部形式で名前を受け取る場合、この手順は必要ありません。

2. gss_canonicalize_name() で、クライアント名の MN を生成します。
3. gss_export_name() で、「エクスポート名」を生成します。エクスポート名とは、クライアント名の連続する文字列バージョンのことです。
4. memcmp() で、エクスポートされたクライアント名を ACL 内のすべての名前と比較します。memcmp() は高速で、オーバーヘッドの少ない関数です。

図 1–5 に、このプロセスを示します。ここでも、サーバーがクライアントから名前をインポートする必要があると仮定します。

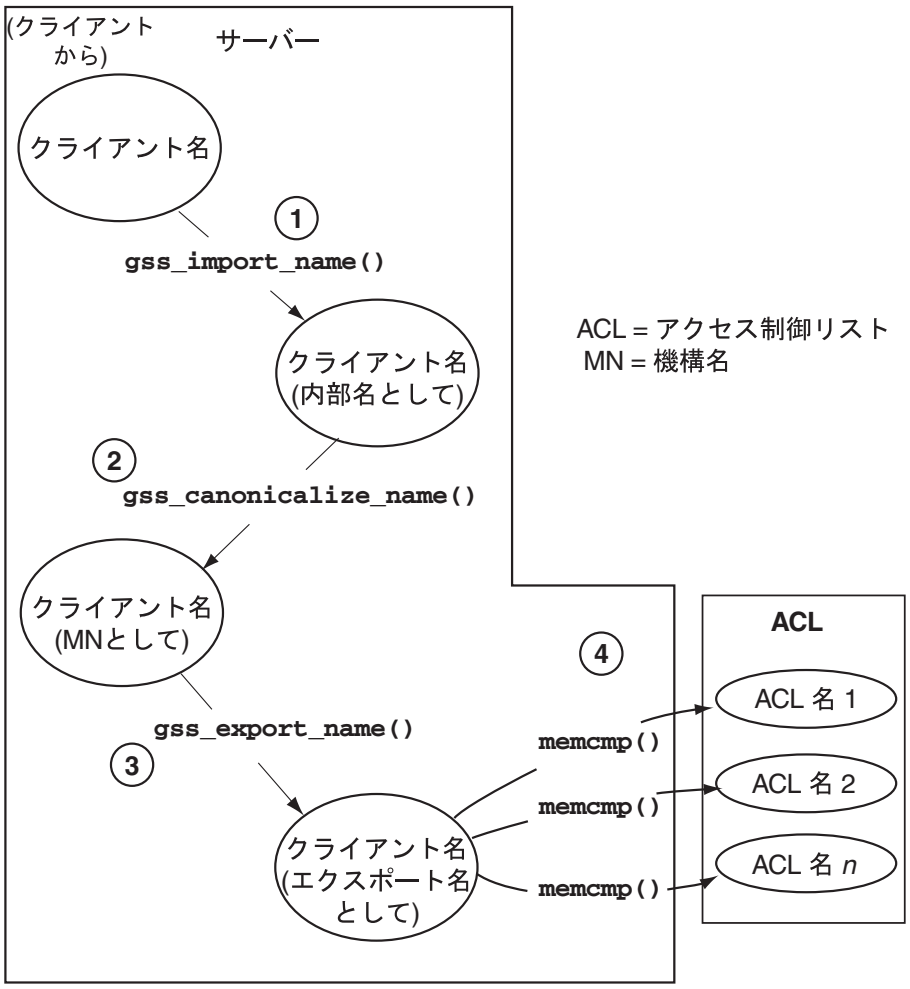

図 1–5　名前の比較 (速い)

gss_export_name() は機構名 (MN) を期待するため、あらかじめ、クライアント名に対して gss_canonicalize_name() を実行する必要があります。

詳細は、gss_canonicalize_name(3GSS)、gss_export_name(3GSS)、および gss_import_name(3GSS) のマニュアルページを参照してください。

OID

オブジェクト識別子 (Object Identifier: OID) は、セキュリティ機構、QOP の値 (保護品質の値)、および名前型などのデータを格納するときに使用します。OID は

GSS-API の ggss_OID_desc 構造体に格納されます。GSS-API は次のような gss_OID_desc 構造体へのポインタ (gss_OID) を提供します。

例 **1–3** OID

```
typedef struct gss_OID_desc_struct {
        OM_uint32   length;
        void        *elements;
     } gss_OID_desc, *gss_OID;
```

さらに、gss_OID_set_desc 構造体は 1 つまたは複数の OID を含むことができます。

例 **1–4** OID セット

```
typedef struct gss_OID_set_desc_struct {
        size_t    count;
        gss_OID   elements;
     } gss_OID_set_desc, *gss_OID_set;
```

注意 - アプリケーションは free() で OID を解放してはいけません。

機構と *QOP*

GSS-API を使用すると、アプリケーションはどの実際のセキュリティ機構を使用するかを選択できます。しかし、アプリケーションは可能な限り、GSS-API が選択したデフォルトの機構を使用するべきです。同様に、GSS-API を使用すると、アプリケーションはどの QOP (保護品質) でデータを保護するかを指定できます。QOP とは、データの暗号化や暗号識別タグの生成に使用されるアルゴリズムのことです。しかし、可能な限り、デフォルトの QOP を使用するべきです。デフォルトの機構を表すには、機構または QOP を期待する関数に値 GSS_C_NULL_OID を引数として渡します。

注意 - セキュリティ機構または QOP を明示的に指定するとアプリケーションの移植性を制限してしまうため、GSS-API を使用する目的が多かれ少なかれ損なわれます。他の実装の GSS-API がその機構または QOP をサポートしなかったり、サポートする方法や範囲が異なる場合があるためです。ただし、付録 C では、使用できる機構と QOP を示して、その選択方法を示しています。

名前型

QOP とセキュリティ機構の他に、OID でも名前型を指定できます。名前型とは、関連する名前の形式を示すものです。たとえば、`gss_import_name()` 関数はプリンシパルの名前を文字列から `gss_name_t` 型に変換しますが、この関数は変換すべき文字列の形式を引数の 1 つとして受け取ります。たとえば、名前型が `GSS_C_NT_HOSTBASED_SERVICE` である場合、`gss_import_name()` 関数は入力された名前が "service@host" 形式 (たとえば、"nfs@swim2birds") であることが分かります。また、たとえば名前型が `GSS_C_NT_EXPORT_NAME` である場合、`gss_import_name()` 関数は入力された名前が GSS-API エクスポート名であることが分かります。アプリケーションは `gss_inquire_names_for_mech()` 関数を使用すると、指定した機構で使用できる名前型を知ることができます。GSS-API が使用する名前型のリストについては、146ページの「名前型」を参照してください。

状態コード

すべての GSS-API 関数は 2 種類のコードを戻して、関数が成功したか失敗したかについての情報を提供します。どちらの種類の状態コードも `OM_uint32` 値として戻されます。次に、この 2 種類の戻りコードについて説明します。

- メジャー状態コード。メジャー状態コードは、

  a) 汎用 GSS-API ルーチンエラー (ルーチンに無効な機構を指定したなど)

  b) 特定の GSS-API 言語バインディングに固有な呼び出しエラー (つまり、関数の引数が読めない、書き込めない、または形式が間違っているなど)

  c) a) と b) の両方

  のいずれかを示します。さらに、メジャー状態コードはルーチンの状態について補助的な情報も提供できます。たとえば、操作が終了していない、トークンが送信された順番が間違っているなどです。なにもエラーが発生しなかった場合、ルーチンは値が `GSS_S_COMPLETE` のメジャー状態コードを戻します。

メジャー状態コードは次のように戻されます。

```
OM_uint32 major_status ;    /* GSS-API が返す状態 */

major_status = gss_generic_function(arg1, arg2 ...);
```

メジャー状態戻りコードは他の OM_uint32 と同じように処理できます。次に例を示します。

```
OM_uint32 maj_stat;

maj_sta = gss_generic_function(arg1, arg2 ...);

if (maj_stat == GSS_CREDENTIALS_EXPIRED)
     <何らかの処理を行う…>
```

メジャー状態コードは、マクロ GSS_ROUTINE_ERROR()、GSS_CALLING_ERROR()、および GSS_SUPPLEMENTARY_INFO() で処理できます。メジャー状態コードの読み方と GSS-API 状態コードのリストについては、139ページの「GSS-API 状態コード」 を参照してください。

- マイナー状態コード。マイナー状態コードは実際の機構が戻すものです。したがって、このマニュアルでは特に説明しません。

すべての GSS-API 関数は最初の引数として OM_uint32 のマイナーコード状態を受け取ります。関数が呼び出された関数に戻るとき、マイナー状態コードは次のように格納されます。

```
OM_uint32 *minor_status ;    /* mech が返す状態 */

major_status = gss_generic_function(&minor_status, arg1, arg2 ...);
```

GSS-API が致命的なメジャーコードエラーを戻す場合でも、*minor_status* パラメータは常に GSS-API ルーチンによって設定されますが、他のほとんどの出力パラメータは設定されません。しかし、GSS-API ルーチンによって割り当てられた記憶領域へのポインタが戻されると期待される出力パラメータは NULL に設定されます。つまり、記憶領域は実際には割り当てられていないと示されます。このようなポインタに関連する長さフィールド (gss_buffer_desc 構造体を参照) は 0 に設定されます。このような場合は、アプリケーションはこれらのバッファを解放する必要はありません。

GSS-API トークン

GSS-API における流通の基本単位は「トークン」です。GSS-API を使用するアプリケーションはトークンを使用してお互いに通信し、データを交換したり、セキュリティを取り決めたりします。トークンは `gss_buffer_t` データ型として宣言され、アプリケーションでは意識する必要はありません。

トークンには 2 種類あります。コンテキストレベルトークンとメッセージ毎トークンです。コンテキストレベルトークンは、主に、コンテキストが確立される (起動され、受け入れられる) ときに使用されます。しかし、その後でコンテキストを管理するためにも使用されます。

メッセージ毎トークンは、コンテキストが確立された後で使用されます。そして、データへの保護サービスを提供します。たとえば、アプリケーションがメッセージを別のアプリケーションに送信したい場合、そのアプリケーションは GSS-API を使用して暗号識別子を生成し、そのメッセージと一緒に送信します。すると、その識別子はトークンに格納されます。

メッセージ毎トークンは「メッセージ」という観点からは次のように考えることができます。メッセージとは、アプリケーションがピアに送信するデータの一部です。たとえば、`ls` コマンドは `ftp` サーバーにメッセージを送信します。メッセージ毎トークンとは、このようなメッセージに対して GSS-API が生成するもの (暗号化タグやメッセージの暗号化形式など) です。意味論上、最後の例は若干正確ではありません。トークンとは GSS-API が生成した情報にしか過ぎないため、暗号化メッセージはやはりメッセージであり、トークンとは言えません。しかし、正式にではありませんが、「メッセージ」と「メッセージ毎トークン」は同じ意味で使用されることがあります。

次の作業は、GSS-API ではなく、アプリケーションの責任です。

1. トークンを送受信すること。開発者はこのようなアクションを実行するために、通常、汎用的な読み取り関数と書き込み関数を作成する必要があります。このような関数の例については、131ページの「`send_token()`」と 132ページの「`recv_token()`」を参照してください。
2. トークンの種類を区別し、種類に従って操作すること。

   トークンはアプリケーションでは意識する必要がないため、アプリケーションにとっては各トークンに違いはありません。アプリケーションはトークンを適切な GSS-API 関数に渡す前に、トークンの内容が分からなくても、トークンを区別できる方法があります。次に、アプリケーションがトークンを区別するための方法を示します。

- 状態によって (つまり、プログラムの制御フローを通じて)。たとえば、アプリケーションがコンテキストを受け入れるために待機している場合、アプリケーションは、ピアがコンテキストが完全に確立されるまで待機することが判明している、つまり、メッセージ (データ) トークンは送信されないと仮定できるため、受け取るトークンがコンテキスト確立に関連するコンテキストレベルトークンであると仮定できます。コンテキストが確立された後、アプリケーションは受け取るトークンがメッセージトークンであると仮定できます。これは、トークンを処理する最も一般的な方法です。後述のサンプルプログラムでもこの方法を使用しています。
- トークンを送受信するとき、アプリケーションはトークンの種類を区別できます。たとえば、アプリケーションが独自の関数でピアにトークンを送信する場合、送信するトークンの種類を示すフラグを含めることができます。

```
gss_buffer_t token;     /* トークンを宣言 */
   OM_uint32 token_flag        /* トークンの型を記述するフラグ */

   <get token from a GSS-API function>

   token_flag = MIC_TOKEN;     /* トークンの種類を指定 */
   send_a_token(&token, token_flag);
```

受信側のアプリケーションは受信関数 (この例では `get_a_token()`) で *token_flag* 引数をチェックします。

- 3 番目の方法は明示的なタグ付けを行うことです。たとえば、アプリケーションは独自の「メタトークン」を使用できます。メタトークンとはユーザー定義の構造体であり、GSS-API 関数から受け取ったトークンとともに、GSS-API が提供するトークンをどのように使用するかを示すユーザー定義フィールドを格納できます。

プロセス間トークン

GSS-API では、マルチプロセスアプリケーションにおいて、あるプロセスから別のプロセスにセキュリティコンテキストを送信できます。一般的に、マルチプロセスアプリケーションはクライアントのコンテキストを受け入れ、プロセス間で共有します。マルチプロセスアプリケーションについては、61ページの「コンテキストのエクスポートとインポート」を参照してください。

`gss_export_context()` 関数が作成するプロセス間トークンには、2 番目のプロセスがコンテキストを再構築できるような情報が含まれています。このプロセス間トー

クンをあるプロセスから別のプロセスに渡すのはアプリケーションの責任であり、また、トークンを別のアプリケーションに渡すのもアプリケーションの責任です。

このプロセス間トークンには鍵となる、つまり他の重要な情報が格納されることもあります。ところが、必ずしもすべての GSS-API 実装がプロセス間トークンを暗号化で保護するとは保証できません。したがって、アプリケーションはプロセス間トークンを送受信する前に保護する必要があります。たとえば、暗号化を使用できる場合は、`gss_wrap()` でプロセス間トークンを暗号化するなどです。

注 - 異なる GSS-API 実装間では、プロセス間トークンを転送できるとは限りません。

GSS-API を使用するプログラミング

この節では、GSS-API を使用する安全なデータ交換を実装する方法を、一般的な手順で説明します。すべての GSS-API 関数を説明するわけではありません。その代わりに、GSS-API を使用する上で最も中心となる関数に注目します。詳細は、すべての GSS-API 関数のリスト (および GSS-API 状態コードとデータ型) が載っている付録 B を参照してください。また、個々の GSS-API 関数については、マニュアルページも参照してください。

簡単に理解できるように、このマニュアルでは単純なモデルを使用します。クライアントアプリケーションがリモートサーバーにデータを送信します。クライアントは直接、つまり、RPC などの転送プロトコル層を介さずに、データを送信します。付録 A には、サンプルプログラム (クライアントとサーバー) を示します。第 2 章では、これらのサンプルプログラムについて段階的に説明しています。

概要

次に、GSS-API を使用するための基本的な手順を示します。

1. 各アプリケーション (送信側と受信側) は資格を明示的に獲得します (資格を自動的に獲得していない場合)
2. 送信側はセキュリティコンテキストを起動し、受信側はそれを受け入れます。

3. 送信側は転送するメッセージ (データ) にセキュリティ保護を適用します。すなわち、メッセージを暗号化するか、識別タグを付けます。送信側は保護したメッセージを転送します。

   送信側はセキュリティ保護を適用しなくてもかまいません。この場合、関連するデフォルトの GSS-API セキュリティサービスだけがメッセージに適用されます。つまり、認証です。認証を使用すると、受信側は、送信側が本当に受信側の要求したとおりのアプリケーションであるかどうかを知ることができます。

4. 受信側はメッセージを復号化し (必要であれば)、検証します (該当する場合)。
5. (省略可能) 確認のため、受信側は識別タグを送信側に返送します。
6. 送信側と受信側のアプリケーションは両方とも共有セキュリティコンテキストを無効にします。必要であれば、残っている GSS-API データも解放できます。

GSS-API を使用するアプリケーションは `gssapi.h` ファイルをインクルード (`include`)します。

図 1–6 に、このプロセスの全体的な概要を示します。この図では、GSS-API が使用できる 1 つの方法しか示していません。しかし、他の場合も考えられます。

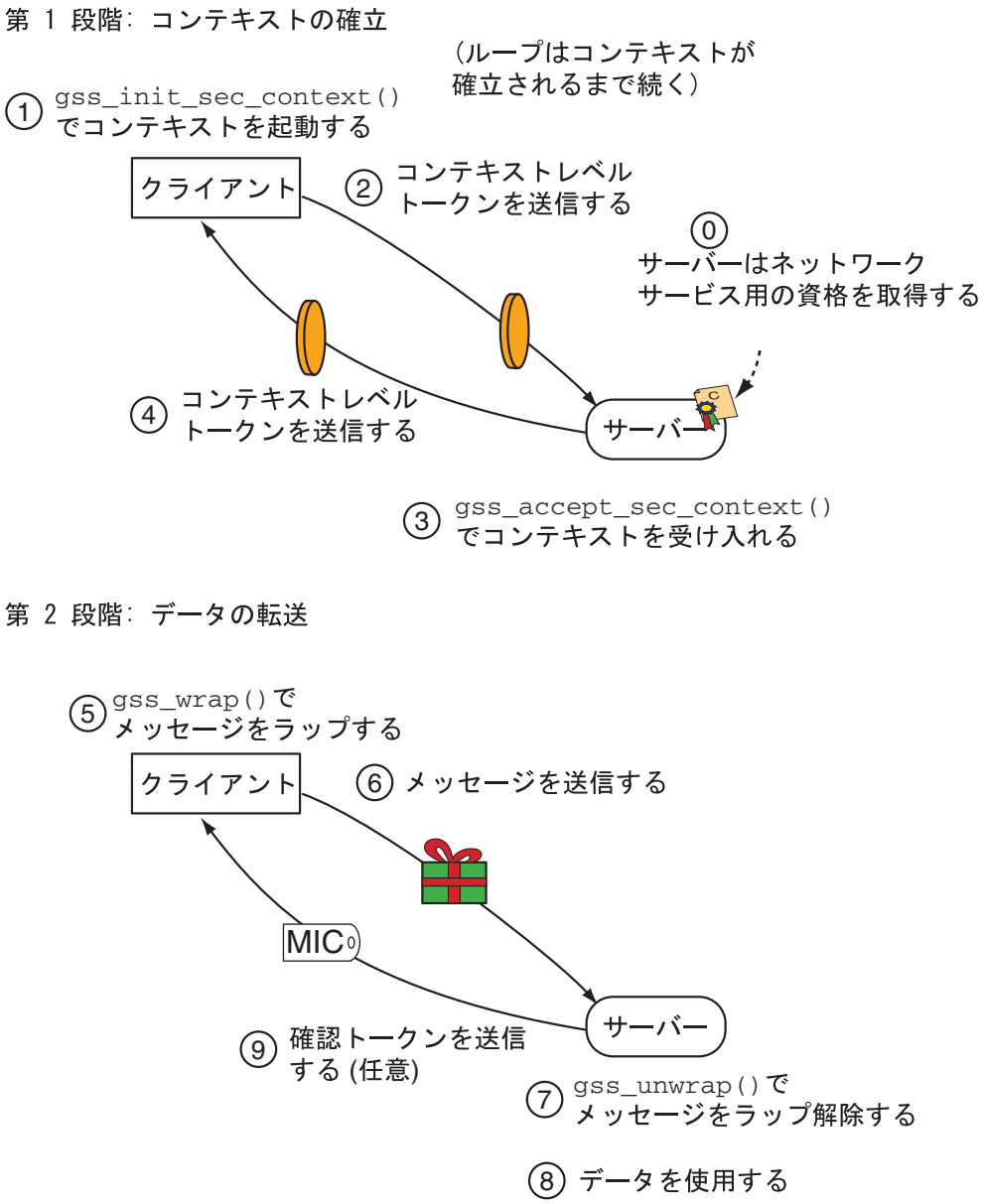

図 1–6　GSS-API の使用: 概要

資格

資格とは、プリンシパル名に対するアプリケーションの要求の証明を提供するデータ構造です。アプリケーションは資格を使用して、ID を確立します。資格はプリンシパルの「識別バッジ」であると考えることもできます。つまり、人、マシン、またはプログラムが本当に要求したとおりのプリンシパルであるかどうか、また多くの場合は、どのような特権を持っているかを証明する情報の集合です。

GSS-API 自身は資格を提供しません。資格は、GSS-API 関数が呼び出される前に、GSS-API の下にあるセキュリティ機構によって作成されます。たとえば、多くの場合、ユーザーはシステムにログインするときに資格を受け取ります。

1 つの GSS-API 資格は単一のプリンシパルだけに有効です。単一の資格は、その単一のプリンシパルに対して複数の (つまり、機構ごとに作成される) 要素を持つことができます。図 1–7 を参照してください。つまり、複数のセキュリティ機構を持つ 1 台のマシン上で獲得された資格は、それらの機構のサブセットだけを持つマシンに転送されるときに有効です。GSS-API は `gss_cred_id_t` 構造体を通じて資格にアクセスします。この構造体のことを「資格ハンドル」と呼びます。資格はアプリケーションでは意識する必要はありません。ユーザーは、与えられた資格の詳細を知っている必要はありません。

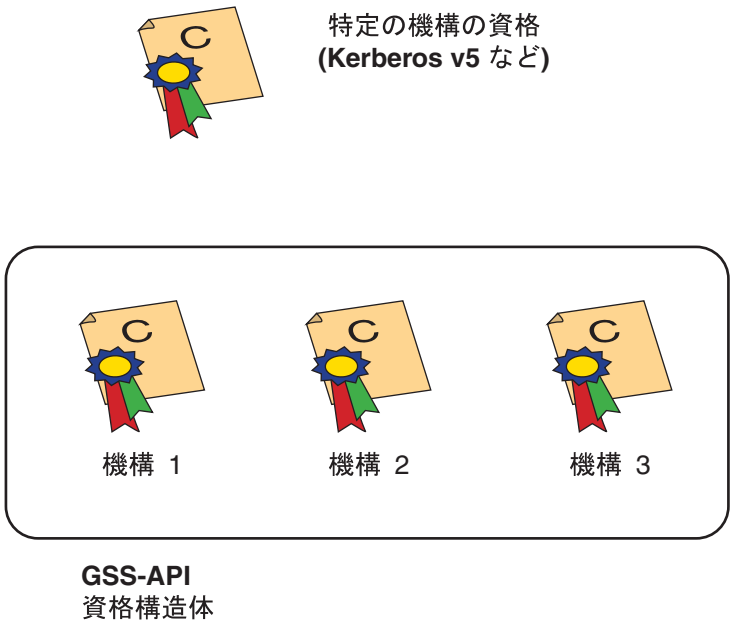

図 1–7　汎用的な GSS-API 資格

資格には 3 つの形式があります。

- GSS_C_INITIATE: この種類の資格は、セキュリティコンテキストを起動するだけのアプリケーションを識別します。
- GSS_C_ACCEPT: この種類の資格は、セキュリティコンテキストを受け入れるだけのアプリケーションを識別します。
- GSS_C_BOTH: この種類の資格は、セキュリティコンテキストを起動または受け入れることができるアプリケーションを識別します。

資格の獲得

セキュリティコンテキストが確立できるようになるまでに、サーバーとクライアントはそれぞれの資格を獲得する必要があります。獲得後、資格は有効期間が満了するまで何度も使用できます。資格が満了したときは、もう一度資格を獲得し直す必要があります。クライアントが使用する資格とサーバーが使用する資格とでは、その有効期間が異なる場合があります。

GSS-API ベースのアプリケーションは、次の 2 つの方法のどちらかで資格を獲得します。

- gss_acquire_cred() 関数 (または、gss_add_cred() 関数) を使用する方法
- コンテキストを確立するときに、デフォルトの資格 (値 GSS_C_NO_CREDENTIAL) を指定する方法

ほとんどの場合、コンテキスト起動側 (クライアント) はログイン時に資格を受け取るため、gss_acquire_cred() を呼び出すのはコンテキスト受け入れ側 (サーバー) だけです。したがって、コンテキスト起動側は通常、デフォルトの資格だけを指定します。コンテキスト受け入れ側は gss_acquire_cred() を使用せずに、デフォルトの資格を使用することもできます。

起動側の資格は、他のプロセスに対してその ID を証明します。一方、受け入れ側は、セキュリティコンテキストを受け入れるための資格を獲得します。ここでは、クライアントがサーバーに ftp 要求を行う場合を考えます。クライアントはログイン時からすでに資格を持っています。そして、クライアントがコンテキストを起動しようとすると、GSS-API は自動的にその資格を取り出します。しかし、サーバープログラムは要求されたサービス (ftp) の資格を明示的に獲得します。

次に、gss_acquire_cred() の構文を示します。

例 **1–5**

```
OM_uint32 gss_acquire_cred (
OM_uint32          *minor_status,
const gss_name_t   desired_name,
OM_uint32          time_req,
const gss_OID_set  desired_mechs,
gss_cred_usage_t   cred_usage,
gss_cred_id_t      *output_cred_handle,
gss_OID_set        *actual_mechs,
OM_uint32          *time_rec)
```

minor_status	実際の機構が返す状態コード。
desired_name	資格を獲得すべきプリンシパルの名前。上記例では「ftp」です。この引数は gss_import_name() で作成されます (20ページの「名前」を参照)。

	desired_name を GSS_C_NO_NAME に設定した場合、汎用の資格が戻されます。つまり、GSS-API のコンテキスト起動ルーチンとコンテキスト受け入れルーチンは資格に関してデフォルトの動作を使用します。言い換えると、gss_acquire_cred() に GSS_C_NO_NAME を渡すと、gss_init_sec_context() または gss_accept_sec_context() にデフォルトの資格要求 (GSS_C_NO_CREDENTIAL) を渡したときと同じ資格が戻ります。詳細は 40ページの「コンテキストの起動 (クライアント)」 と 48 ページの「コンテキストの受け入れ (サーバー)」を参照してください。
time_req	資格が有効であるべき時間 (秒)。GSS_C_INDEFINITE を指定すると、最大の可能な有効期間を要求することができます。
desired_mechs	アプリケーションがこの資格で使用したい実際の機構の集合。これは gss_OID_set データ構造体であり、それぞれが適切な機構を表す 1 つまたは複数の gss_OID 構造体を持ちます。可能な限り、GSS_C_NO_OID_SET を指定して、GSS-API からデフォルトのセットを取得してください。
cred_usage	この資格をどのように使用するかを示すフラグ。コンテキストを起動する場合は GSS_C_INITIATE、コンテキストを受け入れる場合は GSS_C_ACCEPT、または両方の場合は GSS_C_BOTH に設定します。
output_cred_handle	この関数から戻される資格ハンドル。
actual_mechs	この資格で使用できる機構のセット。どの機構であるかを知る必要がない場合は、NULL に設定します。
time_rec	資格が実際に有効である時間 (秒)。時間が重要でない場合は、NULL に設定します。

正常に終了した場合、gss_acquire_cred() は GSS_S_COMPLETE を戻します。有効な資格を戻すことができない場合、gss_acquire_cred() は GSS_S_NO_CRED を戻します。他のエラーコードについては、gss_acquire_cred(3GSS) のマニュアルページを参照してください。資格を獲得する例については、91ページの「資格の獲得」(プログラムリストは 118ページの「server_acquire_creds()」) を参照してください。

gss_add_cred() は gss_acquire_cred() と似ています。しかし gss_add_cred() は、既存の資格に基づいて新しい資格ハンドルを作成するか、既存の資格に新しい資格要素を追加することを、アプリケーションに可能にします。GSS_C_NO_CREDENTIAL を既存の資格として指定した場合、gss_add_cred() はデフォルトの動作に基づいて新しい資格を作成します。詳細は、gss_add_cred(3GSS) のマニュアルページを参照してください。

コンテキストの確立

前述のとおり、GSS-API がセキュリティの提供において行う最も重要な 2 つの仕事は、セキュリティコンテキストを作成することと、データを保護することです。必要な資格を取得した後、アプリケーションはセキュリティコンテキストを確立します。このためには、一方のアプリケーション (通常はクライアント) がコンテキストを起動して、もう一方のアプリケーション (通常はサーバー) がそのコンテキストを受け入れます。ピア間で複数のコンテキストが存在してもかまいません。

通信中のアプリケーションは、認証トークンを交換することによって、結合セキュリティコンテキストを確立します。セキュリティコンテキストは、2 つのアプリケーション間で共有される情報が入っている一対の GSS-API データ構造体です。この情報は、各アプリケーションの (セキュリティにおける) 状態を記述します。セキュリティコンテキストはデータの保護のために必要です。

コンテキストの起動 (クライアント)

アプリケーションとリモートピア間でのセキュリティコンテキストの起動は、gss_init_sec_context() 関数で行います。成功した場合、gss_init_sec_context() 関数は確立中のコンテキストのコンテキストハンドルと、受け入れ側に送信するコンテキストレベルトークンを戻します。しかし、gss_init_sec_context() を呼び出す前に、クライアントは次のことを行う必要があります。

1. 必要であれば、gss_acquire_cred() で資格を獲得します。しかし、通常、クライアントはログイン時に資格を受け取っているため、その場合、この手順は飛ばすことができます。
2. gss_import_name() で、サーバー名を GSS-API 内部形式にインポートします。名前と gss_import_name() についての詳細は、20ページの「名前」を参照してください。

gss_init_sec_context() を呼び出すとき、通常、クライアントは次の引数値を渡します。

- GSS_C_NO_CREDENTIAL を *cred_handle* 引数に渡して、デフォルトの資格を示します。
- GSS_C_NULL_OID を *mech_type* 引数に渡して、デフォルトの機構を示します。
- GSS_C_NO_CONTEXT を *context_handle* 引数に渡して、初期コンテキストが NULL であることを示します。gss_init_sec_context() は通常ループ内で呼び出されるため、後続の呼び出しは以前の呼び出しで戻されたコンテキストハンドルを渡す必要があります。
- GSS_C_NO_BUFFER を *input_token* 引数に渡して、トークンが最初は空であることを示します。あるいは、アプリケーションは length が 0 に設定されている gss_buffer_desc オブジェクトへのポインタを渡すこともできます。
- gss_import_name() で GSS-API 内部形式にインポートされたサーバー名。

アプリケーションは必ずしもこのようなデフォルト値を使用する必要はありません。さらに、クライアントは *req_flags* 引数を使用して、他のセキュリティパラメータに対する要件を指定することもあります。gss_init_sec_context() 引数についての詳細は、以降の節で説明します。

コンテキストを確立するために、コンテキスト受け入れ側はいくつかの「ハンドシェーク」を要求できます。つまり、コンテキストが完全に確立されたと考えられるまでに、複数のコンテキスト情報を送信するように起動側に要求できます。したがって、移植性のため、コンテキストの起動は常に、コンテキストが完全に確立されたかどうかを検査するループの一部として行われる必要があります。

コンテキストが完全に確立されていない場合、gss_init_sec_context() はメジャー状態コードとして GSS_C_CONTINUE_NEEDED を戻します。したがって、ループは gss_init_sec_context() の戻り値を使用して、起動ループを継続するかどうかをテストする必要があります。

クライアントはコンテキスト情報をサーバーに、gss_init_sec_context() から戻された出力トークンの形式で渡します。その後、クライアントはこの情報をサーバーから入力トークンとして受け取ります。すると、後続の gss_init_sec_context() の呼び出しに引数として渡すことができます。受け取った入力トークンの長さが 0 の場合、サーバーはこれ以上出力トークンを要求していないことが分かります。

したがって、gss_init_sec_context() の戻り状態を検査することに加えて、ループは入力トークンの長さを検査して、さらにトークンをサーバーに送信するのかどうかを判断する必要があります。ループが始まる前には、入力トークンを GSS_C_NO_BUFFER に設定するか、構造体の長さフィールドを 0 に設定することによって、入力トークンの長さを 0 に初期化する必要があります。

次に、このようなループの例を示します。かなり一般化されています。

```
context = GSS_C_NO_CONTEXT を入れる
input token = GSS_C_NO_BUFFER を入れる

do {

      gss_init_sec_context(credential, context, name, input_token,
            output_token, other args...)

      if (受け入れ側に送信する output_token がある時)

            受け入れ側に output_token を送信
            output_token を解放する

      if (context が完全でない時)

            受け入れ側から input_token を受け取る

      if (GSS-API エラーがある時)
            context を削除する

} while (context が完全になるまで)
```

当然、実際のループはより複雑になります。たとえば、より多くのエラー検査が必要になるなどです。このようなコンテキスト起動ループの実際の例については、83 ページの「コンテキストの確立」(プログラムリストは 110ページの「client_establish_context()」) を参照してください。さらに、gss_init_sec_context(3GSS) のマニュアルページにも、上記例ほど一般化されていない例があります。

繰り返しますが、GSS-API 自身はトークンを送受信しません。トークンの送受信はアプリケーションが処理する必要があります。トークン転送関数の例について

は、131ページの「send_token()」と 132ページの「recv_token()」を参照してください。

次に、gss_init_sec_context() の形式を示します。詳細は、gss_init_sec_context(3GSS) のマニュアルページを参照してください。

例 **1–6** gss_init_sec_context()

```
OM_uint32 gss_init_sec_context (
  OM_uint32                     *minor_status,
  const gss_cred_id_t           initiator_cred_handle,
  gss_ctx_id_t                  *context_handle,
  const gss_name_t              target_name,
  const gss_OID                 mech_type,
  OM_uint32                     req_flags,
  OM_uint32                     time_req,
  const gss_channel_bindings_t  input_chan_bindings,
  const gss_buffer_t            input_token
  gss_OID                       *actual_mech_type,
  gss_buffer_t                  output_token,
  OM_uint32                     *ret_flags,
  OM_uint32                     *time_rec )
```

minor_status 実際の機構から戻される状態コード。

initiator_cred_handle アプリケーションの資格ハンドル。デフォルトの資格を使用することを示すには、GSS_C_NO_CREDENTIAL に初期化する必要があります。

context_handle 戻すべきコンテキストハンドル。ループが始まる前には、GSS_C_NO_CONTEXT に設定する必要があります。

target_name 接続先のプリンシパルの名前。たとえば、「nfs@machinename」などです。

mech_type 使用されるセキュリティ機構。GSS-API が提供するデフォルトを取得するには、GSS_C_NO_OID に設定します。

req_flags このコンテキストで要求される追加のサービスまたはパラメータを示すフラグ。*req_flags* フラグは次のようにビット論理和をとって、希望のビットマスク値を作成する必要があります。

```
GSS_C_MUTUAL_FLAG | GSS_C_DELEG_FLAG
```

GSS_C_DELEG_FLAG

起動側の資格を委託できるように要求します。55ページの「委託」を参照してください。

GSS_C_MUTUAL_FLAG

相互認証を要求します。56ページの「相互認証」を参照してください。

GSS_C_REPLAY_FLAG

繰り返しメッセージを検出するように要求します。57ページの「誤順序の検出とリプレイの検出」を参照してください。

GSS_C_SEQUENCE_FLAG

誤順序メッセージを検出するように要求します。57ページの「誤順序の検出とリプレイの検出」を参照してください。

GSS_C_CONF_FLAG

転送されるメッセージに機密性サービスを適用するように、つまり、メッセージを暗号化するように要求します。機密性が許可されない場合、データ起点認証と (GSS_C_INTEG_FLAG が偽でなければ) 整合性サービスだけを適用できます。

GSS_C_INTEG_FLAG

メッセージに整合性サービスを適用するように、つまり、メッセージに MIC を付けて有効性を保証するように要求します。

GSS_C_ANON_FLAG

起動側が匿名のままであるように要求します。59ページの「匿名認証」を参照してください。

time_req

コンテキストが有効であるべき時間 (秒)。デフォルトを要求するには、0 に設定します。

input_chan_bindings

セキュリティコンテキストに接続された特定のピアツーピアのチャネル識別情報。チャネルバインディングについての詳細は、59ページの「チャネルバインディング」を参照してください。チャネルバインディングを使用しない場合は、GSS_C_NO_CHANNEL_BINDINGS に設定します。

input_token

コンテキスト受け入れ側から受信したトークン (もしあれば)。関数が呼び出される前に、GSS_C_NO_BUFFER に初期化する (あるいは、長さフィールドを 0 に設定する) 必要があります。

actual_mech_type

コンテキストで実際に使用される機構。どの機構が使用されるかを知る必要がない場合は、NULL に設定します。

output_token

受け入れ側に送信すべきトークン。

ret_flags

このコンテキストで要求された、追加のサービスまたはパラメータを示すフラグ。*ret_flags* フラグは次のようにビット論理積をとって、戻されたビットマスク値をテストする必要があります。

```
if (ret_flags & GSS_C_CONF_FLAG)
     confidentiality = TRUE;
```

GSS_C_DELEG_FLAG

真の場合、起動側の資格が委託できることを示します。55ページの「委託」を参照してください。

GSS_C_MUTUAL_FLAG

真の場合、相互認証が使用できることを示します。56ページの「相互認証」を参照してください。

GSS_C_REPLAY_FLAG

真の場合、繰り返しメッセージの検出が有効であることを示します。57ページの「誤順序の検出とリプレイの検出」を参照してください。

GSS_C_SEQUENCE_FLAG

真の場合、誤順序メッセージの検出が有効であることを示します。57ページの「誤順序の検出とリプレイの検出」を参照してください。

GSS_C_CONF_FLAG

真の場合、転送されるメッセージに機密性サービスを適用できます。つまり、メッセージを暗号化できます。機密性が許可されない場合、データ起点認証と (`GSS_C_INTEG_FLAG` が偽でなければ) 整合性サービスだけを適用できます。

GSS_C_INTEG_FLAG

真の場合、メッセージに整合性サービスを適用できます。つまり、メッセージに MIC (Message Integrity Code) を付けて有効性を保証できます。

GSS_C_ANON_FLAG

真の場合、コンテキスト起動側が匿名のままであることを示します。59ページの「匿名認証」を参照してください。

GSS_C_PROT_READY_FLAG

コンテキストの確立に時間がかかり、完了するまで、クライアントが待機する場合もありま

す。コンテキストが完全に確立されていない場合でも、gss_init_sec_context() は、コンテキストが完全に確立された後にどの保護サービスが使用できるかを (もしあれば) 示すことができます。したがって、アプリケーションはデータをバッファに格納しておいて、コンテキストが完全に確立された後で、バッファに格納しておいたデータを送信できます。

ret_flags が GSS_C_PROT_READY_FLAG を示す場合、コンテキストが完全に確立されていない場合でも (つまり、gss_init_sec_context() が GSS_S_CONTINUE_NEEDED を戻す場合でも)、GSS_C_CONF_FLAG と GSS_C_INTEG_FLAG が示す保護サービスを使用できます。すると、アプリケーションは希望の保護サービスに適切なラップ関数 (gss_wrap() または gss_get_mic()) を呼び出し、さらに、コンテキストが完全に確立されたときに転送する出力をバッファに格納できます。

GSS_C_PROT_READY_FLAG が偽の場合、アプリケーションはデータ保護を仮定できず、コンテキストの確立が完了するまで (つまり、gss_init_sec_context() が GSS_S_COMPLETE を戻すまで) 待機する必要があります。

注 - GSS-API の以前のバージョンは GSS_C_PROT_READY_FLAG 引数をサポートしていません。したがって、移植性を最大限にしたい開発者は、コンテキストの確立が完了した後で GSS_C_CONF_FLAG と GSS_C_INTEG_FLAG フラグを調べて、どのメッセージ毎サービスを使用できるかを判断する必要があります。

GSS_C_TRANS_FLAG

このコンテキストがエクスポートできるかどうかを示します。コンテキストのインポートとエクスポートについての詳細は、61ページの「コンテキストのエクスポートとインポート」を参照してください。

time_rec 資格が有効である時間 (秒)。時間が重要でない場合は、NULL に設定します。

一般に、コンテキストが完全に確立されていない時に戻されるパラメータ値は、コンテキストが完了する時に入力されるはずの値です。詳細は、gss_init_sec_context() のマニュアルページを参照してください。

正常に終了した場合、gss_init_sec_context() は GSS_S_COMPLETE を戻します。コンテキスト確立トークンがピアとなるアプリケーションから要求された場合、gss_init_sec_context() は GSS_S_CONTINUE_NEEDED を戻します。エラーが発生した場合、gss_init_sec_context() はエラーコードを戻します。エラーコードについては、gss_init_sec_context(3GSS) のマニュアルページを参照してください。

コンテキストの起動が失敗した場合、クライアントはサーバーから切断する必要があります。

コンテキストの受け入れ (サーバー)

コンテキストの確立におけるもう一つの仕事は、コンテキストの受け入れです。コンテキストの受け入れは gss_accept_sec_context() 関数で行います。通常、クライアントが (gss_init_sec_context() で) コンテキストを起動し、サーバーがそのコンテキストを受け入れます。

gss_accept_sec_context() は、起動側から送信された入力トークンを主な入力として使用します。gss_accept_sec_context() は、コンテキストハンドルと起動側に戻すべき出力トークンを戻します。しかし、gss_accept_sec_context() を呼び出す前に、サーバーはクライアントから要求されたサービスの資格を獲得しておく必要があります。サーバーはこのような資格を gss_acquire_cred() 関数で獲得します。あるいは、サーバーは資格を明示的に獲得するのではなく、gss_accept_sec_context() を呼び出すときにデフォルトの資格を指定する (GSS_C_NO_CREDENTIAL で示す) ことも可能です。

gss_accept_sec_context() を呼び出すとき、サーバーは次の引数値を渡します。

- gss_acquire_cred() から戻された資格ハンドル。あるいは、GSS_C_NO_CREDENTIAL を *cred_handle* 引数に渡して、デフォルトの資格を示します。
- GSS_C_NO_CONTEXT を *context_handle* 引数に渡して、初期コンテキストが NULL であることを示します。gss_init_sec_context() は通常ループ内で呼び出されるため、後続の呼び出しは以前の呼び出しで戻されたコンテキストハンドルを渡す必要があります。
- クライアントから受け取ったコンテキストトークンを *input_token* 引数に渡します。

gss_accept_sec_context() 引数についての詳細は、以降の節で説明します。

セキュリティコンテキストを確立するためには、いくつかの「ハンドシェーク」が必要です。つまり、コンテキストが完全に確立されるまでに、起動側と受け入れ側は複数のコンテキスト情報を送信する必要があることがあります。したがって、移植性のため、コンテキストの受け入れは常に、コンテキストが完全に確立されたかどうかを検査するループの一部として行われる必要があります。コンテキストが完全に確立されていない場合、gss_accept_sec_context() はメジャー状態コードとして GSS_C_CONTINUE_NEEDED を戻します。したがって、ループは gss_accept_sec_context() の戻り値を使用して、受け入れループを継続するかどうかをテストする必要があります。

コンテキスト受け入れ側はコンテキスト情報をコンテキスト起動側に、gss_accept_sec_context() から戻された出力トークンの形式で渡します。その後、受け入れ側は以降の情報を起動側から入力トークンとして受け取ります。すると、後続の gss_accept_sec_context() の呼び出しに引数として渡すことができます。起動側に送信するトークンがなくなったとき、gss_accept_sec_context() は、length の値が 0 の出力トークンを戻します。したがって、gss_accept_sec_context() の戻り状態を検査することに加えて、ループは出力トークンの length を検査して、さらにトークンを送信するのかどうかを判断する必要があります。ループが始まる前には、出力トークンを GSS_C_NO_BUFFER に設定するか、構造体の length フィールドを 0 に設定することによって、出力トークンの length を 0 に初期化する必要があります。

次に、このようなループの例を示します。かなり一般化されています。

```
context = GSS_C_NO_CONTEXT を入れる
input_token = GSS_C_NO_BUFFER を入れる
```

(続く)

```
do {

      起動側から input_token を受け取る

      gss_accept_sec_context(context, cred_handle, input_token,
            output_token, other args...)

      if (起動側に送信する output_token がある時)

            起動側に output_token を送信
            output_token を解放する

      if (GSS-API エラーがある時)

            context を削除する

} while (context が完全になるまで)
```

当然、実際のループはより複雑になります。たとえば、より多くのエラー検査が必要になるなどです。このようなコンテキスト受け入れループの実際の例については、94ページの「コンテキストの受け入れ」(プログラムリストは 121ページの「server_establish_context()」) を参照してください。さらに、gss_accept_sec_context(3GSS) のマニュアルページにも、上記例ほど一般化されていない例があります。

繰り返しますが、GSS-API 自身はトークンを送受信しません。トークンの送受信はアプリケーションが処理する必要があります。トークン転送関数の例については、131ページの「send_token() と recv_token()」を参照してください。

次に、gss_accept_sec_context() の形式を示します。詳細は、gss_accept_sec_context(3GSS) のマニュアルページを参照してください。

例 1–7 gss_accept_sec_context()

```
OM_uint32 gss_accept_sec_context (
  OM_uint32                    *minor_status,
  gss_ctx_id_t                 *context_handle,
  const gss_cred_id_t          acceptor_cred_handle,
  const gss_buffer_t           input_token_buffer,
  const gss_channel_bindings_t input_chan_bindings,
  const gss_name_t             *src_name,
  gss_OID                      *mech_type,
  gss_buffer_t                 output_token,
```

(続く)

```
OM_uint32                   *ret_flags,
OM_uint32                   *time_req,
gss_cred_id_t               *delegated_cred_handle)
```

minor_status 実際の機構から戻される状態コード。

context_handle 起動側に戻すべきコンテキストハンドル。ループが始まる前には、GSS_C_NO_CONTEXT に設定する必要があります。

acceptor_cred_handle 受け入れ側が (通常は、gss_acquire_cred() で) 獲得した資格のハンドル。GSS_C_NO_CREDENTIAL に初期化すると、デフォルトの資格を使用することを示すことができます。デフォルトの資格が定義されていない場合、GSS_C_NO_CRED を戻します。

(注: プリンシパル名として GSS_C_NO_NAME が渡された場合、gss_acquire_cred() は gss_accept_sec_context() がデフォルトの資格として扱うような資格を生成します。)

input_token_buffer コンテキスト起動側から受け取ったトークン。

input_chan_bindings セキュリティコンテキストに接続された特定のピアツーピアチャネル識別情報。チャネルバインディングについての詳細は、59ページの「チャネルバインディング」を参照してください。チャネルバインディングを使用しない場合は、GSS_C_NO_CHANNEL_BINDINGS に設定します。

src_name 起動しているプリンシパルの名前。たとえば、nfs@machinename などです。プリンシパルの名前が重要でない場合は、NULL に設定します。

mech_type

使用されるセキュリティ機構。どの機構が使用されるかが重要でない場合は、NULL に設定します。

output_token

起動側に送信すべきトークン。関数が呼び出される前に、GSS_C_NO_BUFFER に初期化する (あるいは、長さフィールドを 0 に設定する) 必要があります。長さが 0 の場合、送信する必要のあるトークンはありません。

ret_flags

このコンテキストで要求された追加のサービスまたはパラメータを示すフラグ。*ret_flags* フラグは次のようにビット論理積をとって、戻されたビットマスク値をテストする必要があります。

```
if (ret_flags & GSS_C_CONF_FLAG)
     confidentiality = TRUE;
```

GSS_C_DELEG_FLAG

起動側の資格が *delegated_cred_handle* 引数経由で委託できることを示します。55ページの「委託」を参照してください。

GSS_C_MUTUAL_FLAG

相互認証が使用できることを示します。56ページの「相互認証」を参照してください。

GSS_C_REPLAY_FLAG

繰り返しメッセージの検出が有効であることを示します。57ページの「誤順序の検出とリプレイの検出」を参照してください。

GSS_C_SEQUENCE_FLAG

誤順序メッセージの検出が有効であることを示します。57ページの「誤順序の検出とリプレイの検出」を参照してください。

GSS_C_CONF_FLAG

真の場合、転送されるメッセージに機密性サービスを適用できます。つまり、メッセージを暗号化できます。機密性が許可されない場合、データ起点認証と (GSS_C_INTEG_FLAG が偽でなければ) 整合性サービスだけを適用できます。

GSS_C_INTEG_FLAG

真の場合、メッセージに整合性サービスを適用できます。つまり、メッセージに MIC (Message Integrity Code) を付けて有効性を保証できます。

GSS_C_ANON_FLAG

コンテキスト起動側が匿名のままであることを示します。59ページの「匿名認証」を参照してください。

GSS_C_PROT_READY_FLAG

コンテキストの確立に時間がかかる場合、コンテキストが完全に確立されていない場合でも、受け入れ側がコンテキスト関連のデータを処理できるように、クライアントは起動パスに十分な情報を送信できます。このような状況では、受け入れ側は情報がどのように保護されているかを (もしあれば) 知る必要があります。

GSS_C_PROT_READY_FLAG が真の場合、GSS_C_CONF_FLAG と GSS_C_INTEG_FLAG が示す保護サービスを指定します。すると、受け入れ側は希望の保護サービスに適切なデータ受け入れ関数 (gss_unwrap() または gss_verify_mic()) を呼び出すことができます。

(さらに、コンテキスト起動側と同様に、受け入れ側はこれらのフラグを使用して、起動側に送

信したいデータをバッファに格納しておいて、コンテキストが完全に確立された後で、バッファに格納しておいたデータを送信できます。)

GSS_C_PROT_READY_FLAG が偽の場合、受け入れ側はデータ保護を仮定できず、コンテキストの確立が完了するまで (つまり、gss_accept_sec_context() が GSS_S_COMPLETE を戻すまで) 待機する必要があります。

注 - GSS-API の以前のバージョンは GSS_C_PROT_READY_FLAG 引数をサポートしていません。したがって、移植性を最大限にしたい開発者は、コンテキストの確立が完了した後で GSS_C_CONF_FLAG と GSS_C_INTEG_FLAG フラグを調べて、どのメッセージ毎サービスを使用できるかを判断する必要があります。

GSS_C_TRANS_FLAG

真の場合、このコンテキストがエクスポートできることを示します。コンテキストのインポートとエクスポートについての詳細は、61ページの「コンテキストのエクスポートとインポート」を参照してください。

time_rec

コンテキストが有効である時間 (秒)。時間が重要でない場合は、NULL に設定します。

delegated_cred_handle

コンテキスト起動側から受け取った資格 (つまり、クライアントの資格) の資格ハンドル。受け入れ側がプロクシとして動作するように起動側が要求した場合だけ、つまり、*ret_flags* 引数に GSS_C_DELEG_FLAG が含まれている場合だけ有効です。委託についての詳細は、55ページの「委託」を参照してください。

正常に終了した場合、gss_accept_sec_context() は GSS_S_COMPLETE を戻します。コンテキストが完全に確立されていない場合、gss_accept_sec_context() は GSS_S_CONTINUE_NEEDED を戻します。エラーが発生した場合、gss_accept_sec_context() はエラーコードを戻します。エラーコードについては、gss_accept_sec_context(3GSS) のマニュアルページを参照してください。

追加のコンテキストサービス

gss_init_sec_context() 関数 (40ページの「コンテキストの起動 (クライアント)」を参照) を使用すると、アプリケーションは基本のコンテキスト確立以外にも追加のデータ保護サービスも要求できます。このようなサービス (以降の節で説明) を要求するには、gss_init_sec_context() の *req_flags* 引数を使用します。

すべての機構が追加の保護サービスを提供するわけではありません。したがって、特定のコンテキストで使用できる追加の保護サービスを調べるには、gss_init_sec_context() の *ret_flags* 引数を使用します。同様に、コンテキスト受け入れ側も gss_accept_sec_context() 関数から戻された *ret_flags* の値を見て、どのサービスを使用できるかを判断します。以降の節では、追加のサービスについて説明します。

委託

可能であれば、コンテキスト起動側はコンテキスト受け入れ側がプロクシとして動作するように要求できます。この場合、受け入れ側は起動側の代わりに別のコンテキストを起動できます。このような委託の例としては、マシン A 上のユーザーがマシン B に rlogin し、次に、マシン B からマシン C に rlogin するなどです (図 1–8 を参照)。委託された資格がマシン B をマシン A として識別するか、あるいは、マシン A の代わりに動作しているマシン B と識別するかは、機構によって異なります。

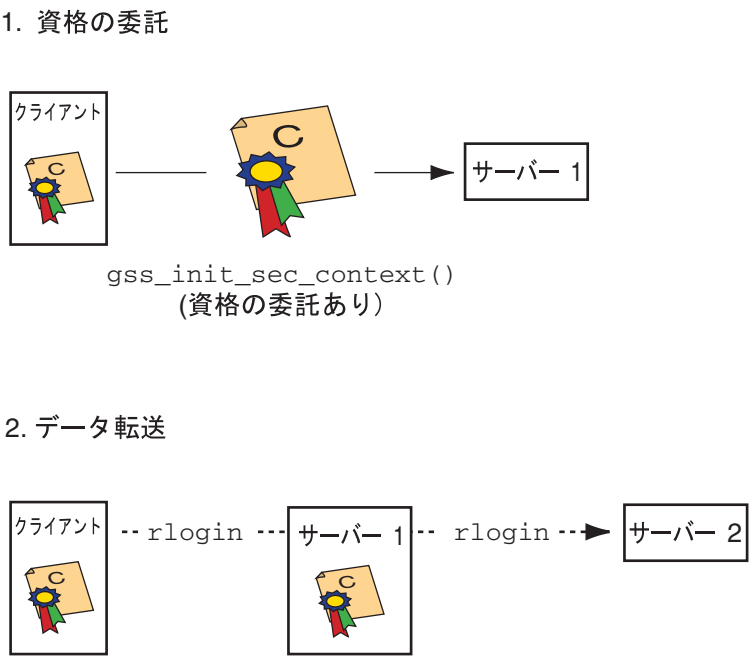

図 1–8　資格の委託

委託が許可されると、*ret_flags* に値 GSS_C_DELEG_FLAG が設定されます。受け入れ側は委託された資格を gss_accept_sec_context() の *delegated_cred_handle* 引数として受け取ります。資格の委託はコンテキストのエクスポートとは異なります(61ページの「コンテキストのエクスポートとインポート」 を参照)。その違いのひとつは、アプリケーションはその資格を一度に複数回委託できますが、コンテキストは一度に 1 つのプロセスでしか保持されません。

相互認証

ftp でファイルを公的な ftp サイトにダウンロードする場合、サイトがユーザーの ID を証明するように要求することはあっても、ユーザーがサイトの ID を証明するように要求することはありません。ところが、パスワードやクレジットカードの番号をアプリケーションに提供する場合、ユーザーは受け取り側が信頼できるサイトであることを確認したいはずです。このような場合、「相互認証」が必要で

す。つまり、コンテキストの起動側と受け入れ側が両方とも自身の ID を証明する必要があるということです。

コンテキスト起動側が相互認証を要求するには、gss_init_sec_context() の *req_flags* 引数に値 GSS_C_MUTUAL_FLAG を設定します。相互認証が承認されると、*ret_flags* 引数にも値 GSS_C_MUTUAL_FLAG が設定されます。相互認証が要求されたが使用できない場合、結果に応じて応答するのは起動側アプリケーションの責任です。つまり、このような理由では、GSS-API はコンテキストを終了しません。機構の中には、要求されたかどうかに関わらずに相互認証を実行するものもあります。

誤順序の検出とリプレイの検出

通常、コンテキスト起動側が複数の連続するデータパケットを受け入れ側に転送する場合、到達したパケットが正しい順序であり、不必要な重複がないことをコンテキスト受け入れ側がチェックできるような機構もあります (図 1–9 を参照)。受け入れ側がこのような 2 つの状態をチェックするのは、パケットの有効性を検証するとき、あるいは、パケットをラップ解除するときです。詳細は、71ページの「ラップ解除と検証」 を参照してください。

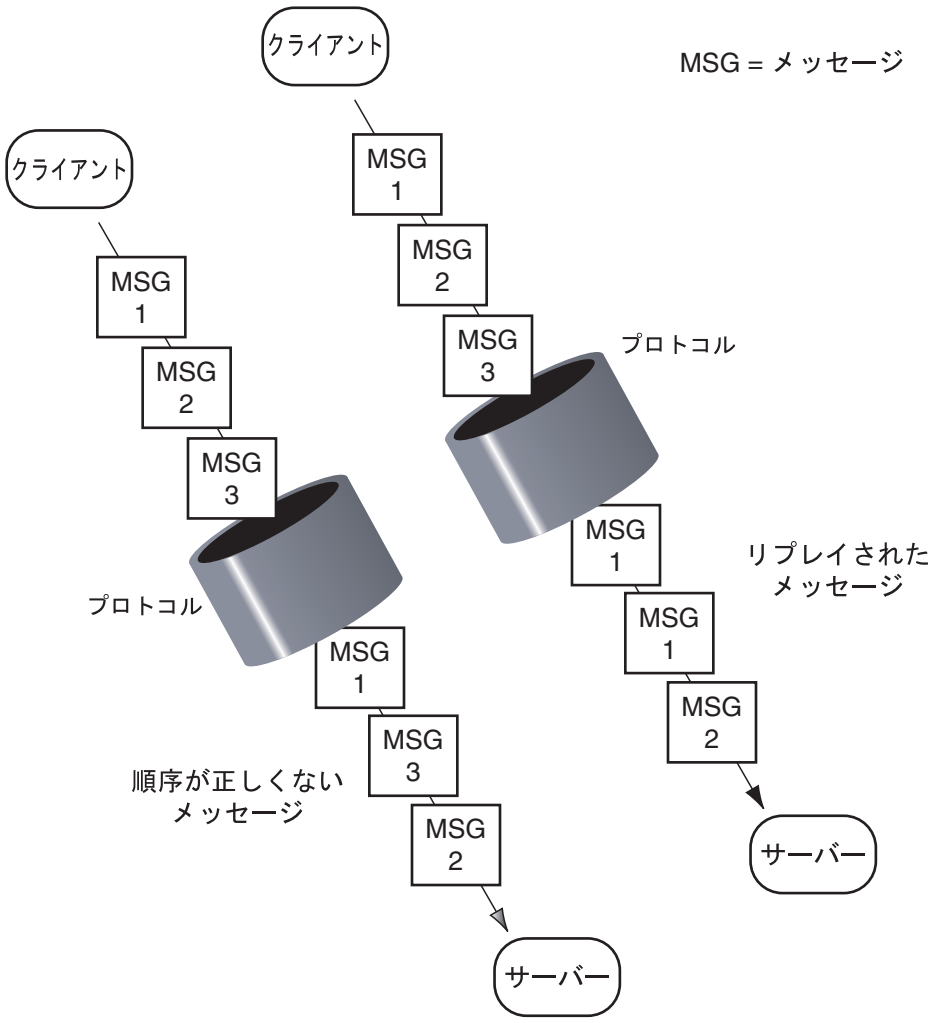

図 1–9　リプレイされたメッセージと順序が正しくないメッセージ

このような 2 つの状態を検査するように要求するには、起動側は、`gss_init_sec_context()` でコンテキストを起動するときに、*req_flags* 引数に値 `GSS_C_REPLAY_FLAG` または `GSS_C_SEQUENCE_FLAG` をビット論理和で設定する必要があります。

匿名認証

GSS-API の通常の使用においては、起動側の識別情報 (ID) はコンテキスト確立プロセスの結果として、受け入れ側に使用できるようになります。しかし、コンテキスト起動側は自身の ID をコンテキスト受け入れ側に知らせないように要求することもできます。

たとえば、医療情報が含まれているデータベースへのアクセスを提供するアプリケーションで、サービスへの制限なしのアクセスを提供している場合を想定してください。このようなサービスのクライアントは、サービスを認証したい (つまり、そこから取得できる情報の信頼性を確立したい) と希望しますが、サービスによりクライアントの識別情報が取得されることは希望しません (特定の照合に関するプライバシの問題や、メールリストなどに利用されるのを防ぐなどのため)。

匿名性を要求するには、`gss_init_sec_context()` の *req_flags* 引数に `GSS_C_ANON_FLAG` を設定します。匿名性を使用できるかどうかをチェックするには、`gss_init_sec_context()` または `gss_accept_sec_context()` の *ret_flags* 引数に `GSS_C_ANON_FLAG` が設定されているかどうかを調べます。

匿名性が有効であり、`gss_accept_sec_context()` または `gss_inquire_context()` が戻したクライアント名上で `gss_display_name()` が呼び出された場合、`gss_display_name()` は汎用的な匿名を生成します。

注 - 匿名性が要求されたが使用できない場合、適切な対処を行うのはアプリケーションの責任です。つまり、このような理由では、GSS-API はコンテキストを終了しません。

チャネルバインディング

多くのアプリケーションでは、コンテキスト起動側を適切に認証するには、基本的なコンテキスト確立だけで十分です。追加のセキュリティが必要な場合、GSS-API ではチャネルバインディングを使用します。チャネルバインディングとは、使用される特定のデータチャネル (つまり、コンテキストの起点と終点 (起動側と受け入れ側)) を識別するタグのことです。このようなタグは起動側と受け入れ側のアプリケーションに固有であるため、より有効な ID の証明となります。

チャネルバインディングは、`gss_channel_bindings_struct` 構造体へのポインタである `gss_channel_bindings_t` データ型で指定します。例 1–8 を参照してください。

例 **1–8** gss_channel_bindings_t

```
typedef struct gss_channel_bindings_struct {
OM_uint32       initiator_addrtype;
gss_buffer_desc initiator_address;
OM_uint32       acceptor_addrtype;
gss_buffer_desc acceptor_address;
gss_buffer_desc application_data;
} *gss_channel_bindings_t;
```

最初の 2 つのフィールドは起動側のアドレスとアドレス型 (起動側のアドレスが送信される形式) を示します。たとえば、*initiator_addrtype* を GSS_C_AF_INET に設定した場合、*initiator_address* がインターネットアドレス形式 (つまり、IP アドレス) であることを示します。同様に、3 番目と 4 番目のフィールドは受け入れ側のアドレスとアドレス型を示します。最後のフィールド (*application_data*) はアプリケーションが自由に使用することができます。使用する予定がない場合は GSS_C_NO_BUFFER に設定するように習慣付けましょう。アプリケーションがアドレスを指定したくない場合、アドレス型フィールドを GSS_C_AF_NULLADDR に設定します。有効なアドレス型の値については、147ページの「チャネルバインディングのアドレス型」を参照してください。

このようなアドレス型は、特定のアドレス形式を示すのではなく、アドレスファミリを示します。アドレスファミリが複数の代替アドレス形式を持つ場合、どのアドレス形式を使用するかを判断できるだけの十分な情報を *initiator_address* と *acceptor_address* のフィールドに指定する必要があります。特に指定しない限り、アドレスはネットワークのバイト順 (つまり、アドレスファミリにネイティブなバイト順) で指定します。

チャネルバインディングを使用してコンテキストを確立するには、割り当てられたチャネルバインディング構造体を gss_init_sec_context() の *input_chan_bindings* 引数で指します。gss_init_sec_context() 関数は、構造体のフィールドをオクテット文字列に連結し、この文字列から MIC (Message Integrity Code) を計算し、この MIC と gss_init_sec_context() が生成した出力トークンを結合します。次に、アプリケーションはこのトークンをコンテキスト受け入れ側に送信します。コンテキスト受け入れ側はこのトークンを受け取り、gss_accept_sec_context() を呼び出します (48ページの「コンテキストの受け入れ (サーバー)」を参照)。gss_accept_sec_context() は受け取ったチャネルバインディングから MIC を計算し、MIC が一致しない場合は、GSS_C_BAD_BINDINGS を戻します。

gss_accept_sec_context() は転送されたチャネルバインディングを戻すため、受け入れ側は受け取ったチャネルバインディング値に基づいて独自のセキュリティ検査を行うことができます。たとえば *application_data* の値をセキュアデータベースに保存しておいたコードワードと比較するなどです。しかし、多くの場合、これは冗長です。

注 - チャネルバインディング情報の機密性を提供するかどうかは、実際の機構によって異なります。したがって、アプリケーションは、機密性が確保されたと断定できるまで、チャネルバインディングに重要な情報を含めてはなりません。機密性を使用できるかどうかを判断するには、アプリケーションが gss_init_sec_context() または gss_accept_sec_context() の *ret_flags* 引数 (特に値 GSS_C_CONF_FLAG と GSS_C_PROT_READY_FLAG) をチェックする方法もあります。*ret_flags* については、40ページの「コンテキストの起動 (クライアント)」 と 48ページの「コンテキストの受け入れ (サーバー)」を参照してください。

機構は個々に、チャネルバインディングにおけるアドレスとアドレス型に追加の制限を課すことができます。たとえば、機構は、gss_init_sec_context() に指定されたチャネルバインディングの *initiator_address* フィールドが持つホストシステムのネットワークアドレスが正しいかどうかを検証することができるなどです。したがって、移植性のあるアプリケーションは、アドレスフィールドに正しい情報を提供するか、あるいは、アドレス情報を省略して、アドレス型として GSS_C_AF_NULLADDR を指定する必要があります。

コンテキストのエクスポートとインポート

GSS-API はコンテキストをエクスポートおよびインポートする方法を提供します。この機能の主な目的は、マルチプロセスアプリケーション (通常は、コンテキスト受け入れ側) があるプロセスから別のプロセスにコンテキストを転送できるようにすることです。たとえば、受け入れ側がコンテキスト起動側の応答を待つプロセスと、コンテキストに送信されたデータを処理するプロセスを持っている場合などです。これらの関数でコンテキストを保存および復元する方法については、125ページの「test_import_export_context()」を参照してください。

gss_export_sec_context() 関数は、エクスポートされるコンテキストについての情報が入ったプロセス間トークンを作成します。32ページの「プロセス間トークン」を参照してください。gss_export_sec_context() を呼び出す前に、このトークンを受信するバッファは GSS_C_NO_BUFFER に設定する必要があります。

次に、アプリケーションはこのトークンを他のプロセスに渡します。他のプロセスはトークンを受け取り、gss_import_sec_context() に渡します。多くの場合、アプリケーション間でトークンを渡すときに使用される関数は、プロセス間でトークンを渡すときにも使用されます。

セキュリティプロセスのインスタンスは一度に 1 つしか存在できません。gss_export_sec_context() はエクスポートされたコンテキストを無効にし、そのコンテキストハンドルを GSS_C_NO_CONTEXT に設定します。さらに、そのコンテキストに関連するプロセス規模のリソースの割り当てをすべて解除します。コンテキストのエクスポートが完了しなかった場合、gss_export_sec_context() はプロセス間トークンを戻さず、既存のセキュリティコンテキストを元のままにします。

すべての機構でコンテキストをエクスポートできるわけではありません。したがって、コンテキストをエクスポートできるかどうかを調べるには、gss_accept_sec_context() または gss_init_sec_context() の *ret_flags* 引数をチェックします。このフラグに GSS_C_TRANS_FLAG が設定されている場合、コンテキストはエクスポートできます。48ページの「コンテキストの受け入れ (サーバー)」と 40ページの「コンテキストの起動 (クライアント)」を参照してください。

図 1–10 に、マルチプロセスの受け入れ側がコンテキストをエクスポートしてマルチタスクを実現している様子を示します。この例では、プロセス 1 はトークンを受け取って処理し、コンテキストレベルトークンをそのトークンから分離し、そのトークンをプロセス 2 に渡します。プロセス 2 はこのデータトークンをアプリケーション固有な方法で処理します。この図では、クライアントはすでに gss_init_sec_context() からエクスポートトークンを取得しています。クライアントはエクスポートトークンをユーザー定義関数 send_a_token() に渡します。send_a_token() は転送中のトークンがコンテキストレベルトークンまたはメッセージトークンのどちらであるかを示しています。send_a_token() はトークンをサーバーに転送します。この図には示されていませんが、おそらく、send_a_token() はスレッド間でトークンを渡すときにも使用されます。

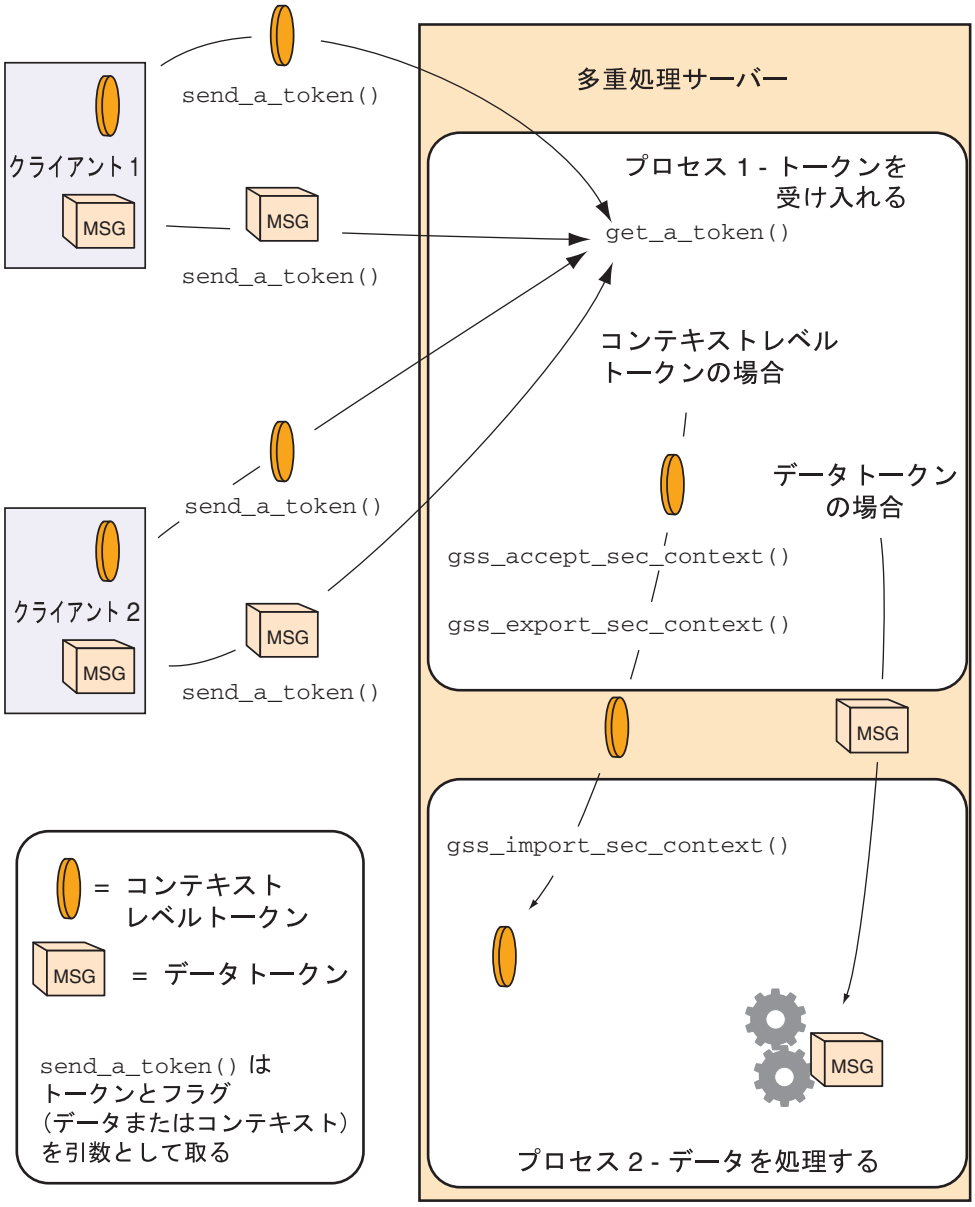

図 1–10　コンテキストのエクスポート: マルチスレッド化された受け入れ側の例

コンテキスト情報

GSS-API は、指定されたセキュリティコンテキスト (不完全なものであっても) についての情報を獲得する `gss_inquire_context()` 関数を提供します。コンテ

キストハンドルを指定すると、gss_inquire_context() はそのコンテキストについて次の情報を提供します。

- コンテキスト起動側の名前
- コンテキスト受け入れ側の名前
- コンテキストが有効である時間 (秒)
- コンテキストで使用されるセキュリティ機構
- いくつかのコンテキストパラメータフラグ。これらのフラグは gss_accept_sec_context() 関数の *ret_flags* 引数と同じで（48ページの「コンテキストの受け入れ (サーバー)」を参照)、委託や相互認証などをカバーします。
- 照会側アプリケーションがコンテキスト起動側であるかどうかを示すフラグ
- コンテキストが完全に確立されているかどうかを示すフラグ

詳細は、gss_inquire_context(3GSS) のマニュアルページを参照してください。

データ保護

ピア間 (つまり、クライアントとサーバー間) でコンテキストが確立された後、メッセージは送信する前に保護できます。

コンテキストを確立しメッセージを送信するだけの場合、最も基本的な GSS-API 保護である「認証」が使用されます。認証を使用すると、受信側は、送信側であるべきと要求されているプリンシパルからのメッセージであるかどうかを知ることができます。使用される実際のセキュリティ機構によって異なりますが、GSS-API は次の 2 つの保護も提供します。

- 整合性。メッセージ整合性コード (MIC) をメッセージに添付します。MIC を使用すると、受信側は受信したメッセージが送信されたメッセージと同じであることをチェックできます。MIC を生成するのは、GSS-API の gss_get_mic() 関数です。
- 機密性。MIC の添付に加えて、メッセージを暗号化します。暗号化を実行するのは、GSS-API の gss_wrap() 関数です。

図 1–11 に、2 つの関数の違いを示します。

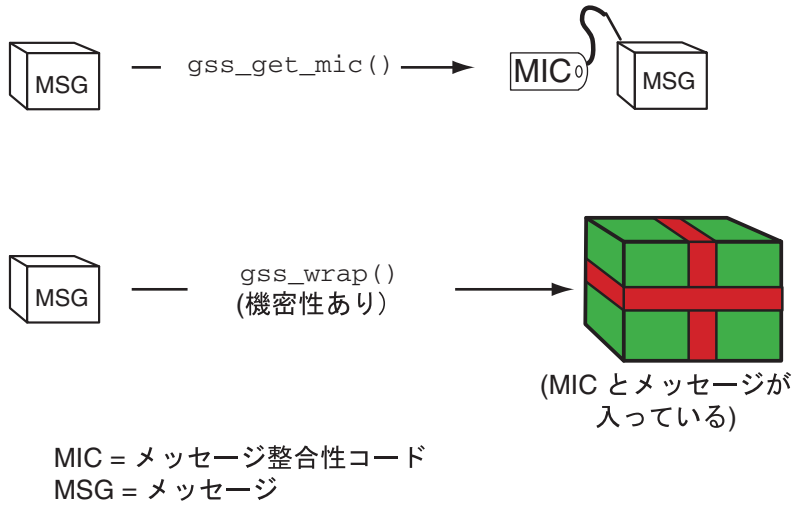

図 1–11　gss_get_mic() と gss_wrap()

どちらの関数を使用するかは、ユーザーの必要性によって異なります。gss_wrap() は整合性サービスも含むため、多くのプログラムは gss_wrap() を使用します。機密性サービスを使用できるかどうかをテストするには、機密性サービスを付けて (または、付けずに) gss_wrap() を呼び出します。使用例は、87 ページの「データの送信」(プログラムリストは 103ページの「call_server()」) を参照してください。しかし、gss_get_mic() で保護されたメッセージは受信側がラップ解除する必要がないため、gss_wrap() を使用するときよりも CPU サイクルを節約できます。したがって、機密性が必要ないプログラムは gss_get_mic() でメッセージを保護する傾向にあります。

gss_get_mic() によるメッセージのタグ付け

gss_get_mic() を使用すると、プログラムは暗号化 MIC をメッセージに追加できます。受信側は gss_verify_mic() を呼び出し、この MIC をチェックすることによって、受信したメッセージが送信されたメッセージと同じであるかどうかを調べることができます。次に、gss_get_mic() の形式を示します。

```
OM_uint32 gss_get_mic (
OM_uint32          *minor_status,
const gss_ctx_id_t context_handle,
```

(続く)

```
gss_qop_t           qop_req,
const gss_buffer_t message_buffer,
gss_buffer_t        msg_token)
```

minor_status	実際の機構から戻される状態コード。
context_handle	メッセージが送信されるコンテキスト。
qop_req	要求する QOP (保護品質)。MIC を生成するときに使用される暗号化アルゴリズムです。移植性のためには、アプリケーションは可能な限りデフォルトの QOP を指定するべきです。つまり、この引数に GSS_C_QOP_DEFAULT を設定します。デフォルト以外の QOP の指定については、付録 C を参照してください。
message_buffer	MIC をタグ付けするメッセージ。この引数は gss_buffer_desc オブジェクトの形式である必要があります (19ページの「文字列および類似のデータ」を参照)。使用し終わったときには、gss_release_buffer() で解放する必要があります。
msg_token	メッセージと MIC が入っているトークン。使用し終わったときには、gss_release_buffer() で解放する必要があります。

gss_get_mic() はメッセージと MIC を別々に出力します。これは、gss_wrap() とは異なります。gss_wrap() は両方を一緒にして出力します。このように別々に出力するということは、送信側アプリケーションがメッセージと MIC の両方を送信するためにアレンジを行う必要があるということを意味します。さらに重要なことは、受信側アプリケーションがメッセージと MIC を受信および区別できる必要があるということです。メッセージと MIC を適切に処理するには、次のような方法があります。

- プログラム制御 (つまり、状態) を通じて。受信側アプリケーションは受信関数を 2 回呼び出す (つまり、1 回目はメッセージを取得するため、2 回目はメッセージの MIC を取得するため) ことをあらかじめ知ることができます。

- フラグを通じて。送信と受信の関数はどの種類のトークンを含めるかをフラグで示すことができます。
- メッセージと MIC の両方を入れることができるユーザー定義トークン構造体を通じて。

正常に終了した場合、gss_get_mic() は GSS_S_COMPLETE を戻します。指定した QOP が有効でなかった場合、gss_get_mic() は GSS_S_BAD_QOP を戻します。詳細は、gss_get_mic(3GSS) のマニュアルページを参照してください。

gss_wrap() によるメッセージのラップ

メッセージは gss_wrap() 関数で「ラップ」することも可能です。gss_get_mic() と同様に、gss_wrap() は MIC を提供します。機密性が要求される場合 (かつ、実際の機構で使用できる場合)、gss_wrap() はさらにメッセージを暗号化します。メッセージの受信側は gss_unwrap() でメッセージを「ラップ解除」します。次に、gss_wrap() の形式を示します。

```
OM_uint32 gss_wrap (
OM_uint32          *minor_status,
const gss_ctx_id_t context_handle,
int                conf_req_flag,
gss_qop_t          qop_req
const gss_buffer_t input_message_buffer,
int                *conf_state,
gss_buffer_t       output_message_buffer )
```

minor_status	実際の機構から戻される状態コード。
context_handle	このメッセージが送信されるコンテキスト。
conf_req_flag	機密性サービス (暗号化) を要求するためのフラグ。0 以外の場合、機密性と整合性の両方を要求します。0 の場合、整合性サービスだけを要求します。
qop_req	要求する QOP (保護品質)。MIC を生成するときと暗号化を行うときに使用される暗号化アルゴリズムです。移植性のためには、アプリケーションは可能な限りデフォルトの QOP を指定するべきです。つまり、この引数に

GSS_C_QOP_DEFAULT を設定します。デフォルト以外の QOP の指定については、付録 C を参照してください。

input_message_buffer — ラップするメッセージ。この引数は gss_buffer_desc オブジェクトの形式である必要があります (19ページの「文字列および類似のデータ」を参照)。使用し終わったときには、gss_release_buffer() で解放する必要があります。

conf_state — 関数が戻ったときに、機密性が適用されたかどうかを示すフラグ。0 以外の場合、機密性、メッセージ起点認証、および整合性サービスが適用されたことを示します。0 の場合、メッセージ起点認証と整合性だけが適用されたことを示します。必要ない場合は、NULL を指定します。

output_message_buffer — ラップしたメッセージ用のバッファ。アプリケーションがメッセージを処理した後には、gss_release_buffer() で解放する必要があります。

gss_get_mic() とは違い、gss_wrap() はメッセージと MIC を一緒にラップし、1 つの出力メッセージにします。したがって、メッセージを転送する関数は 1 回呼び出すだけでかまいません。メッセージの受信側では、gss_unwrap() でメッセージを抽出します。なお、MIC はアプリケーションからは見えません。

メッセージが正常にラップされた場合、gss_wrap() は GSS_S_COMPLETE を戻します。要求した QOP が有効でなかった場合、gss_wrap() は GSS_S_BAD_QOP を戻します。gss_wrap() の使用例は、87ページの「データの送信」(プログラムリストは 103ページの「call_server()」を参照してください。詳細は、gss_wrap(3GSS) のマニュアルページを参照してください。

ラップのサイズ

gss_wrap() でメッセージをラップすると、メッセージのサイズが増加します。保護されたメッセージパケットは、転送プロトコルを通過できるぐらいのサイズである必要があります。したがって、GSS-API の gss_wrap_size_limit 関数を使用して、ラップしても転送プロトコルを通過できるメッセージの最大サイズを計算し

ます。この最大サイズを超える場合、アプリケーションは gss_wrap() を呼び出す前にメッセージを分割できます。メッセージを実際にラップする前にはラップサイズの制限値をチェックするように習慣付けましょう。

サイズの増加量は次の 2 つの影響を受けます。

- メッセージをラップするためにどの QOP (保護品質) アルゴリズムを使用するか。GSS-API の実装によってデフォルトの QOP が異なるため、デフォルトの QOP を指定した場合でも、ラップされたメッセージのサイズは異なる可能性があります。図 1–12 に、この様子を示します。

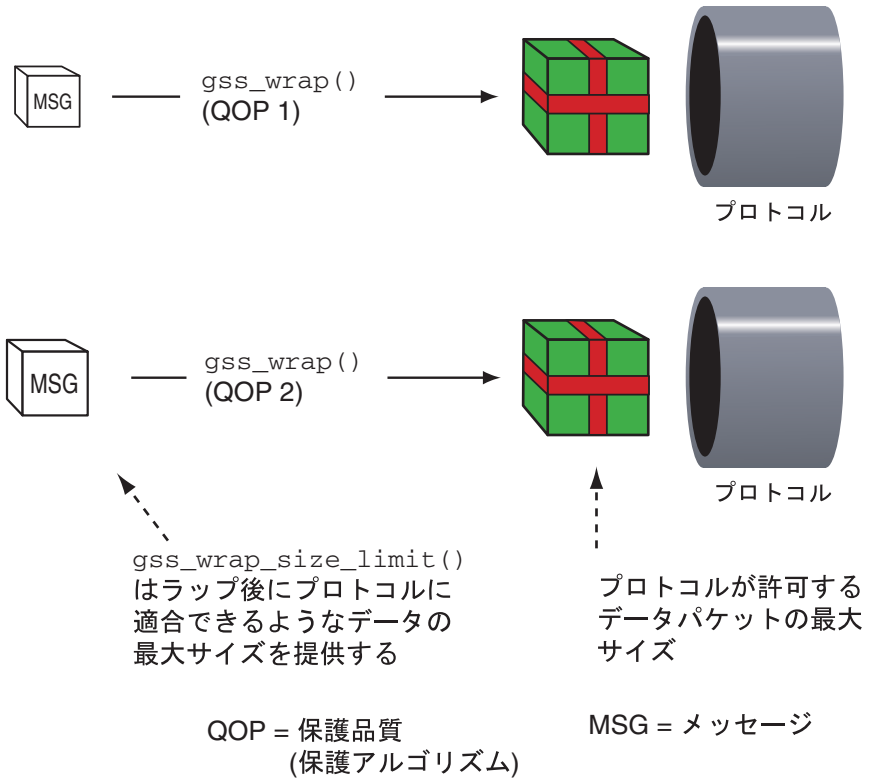

図 1–12　ラップのサイズ (QOP が異なる場合)

- 機密性を呼び出すかどうか。機密性を適用するかどうかに関わらず、gss_wrap() は転送されるメッセージに MIC を添付するため、メッセージのサイズは増加します。しかし、メッセージを暗号化すると (機密性を適用すると)、メッセージのサイズはさらに増加します。図 1–13 に、この様子を示します。

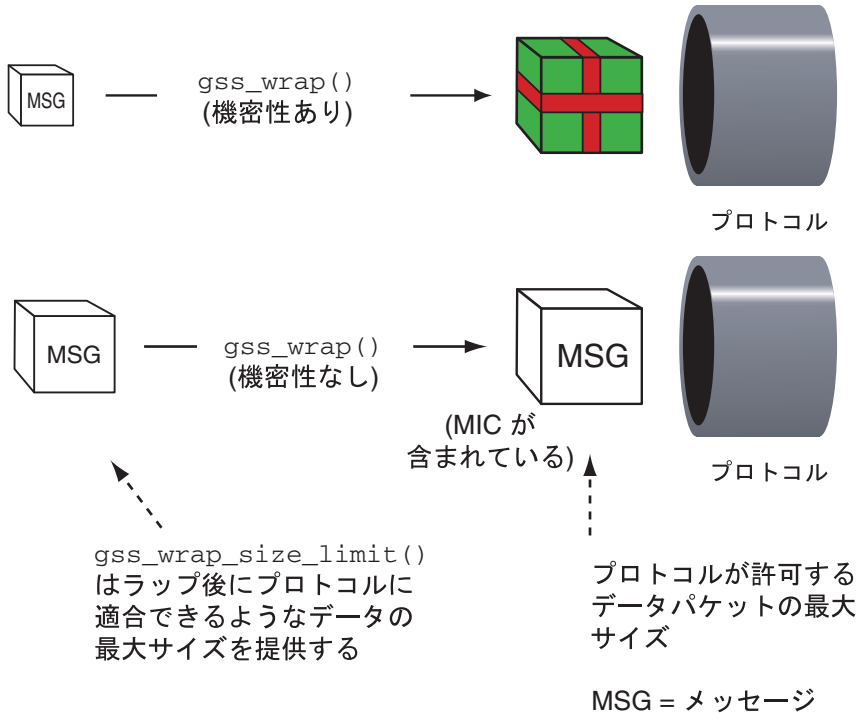

図 1–13　ラップのサイズ (機密性がある場合とない場合)

次に、gss_wrap_size_limit() の形式を示します。

```
OM_uint32 gss_wrap_size_limit (
OM_uint32          *minor_status,
const gss_ctx_id_t context_handle,
int                conf_req_flag,
gss_qop_t          qop_req,
OM_uint32          req_output_size,
OM_uint32          *max_input_size)
```

minor_status　実際の機構から戻される状態コード。

context_handle　データが転送されるコンテキスト。

conf_req_flag　機密性サービス (暗号化) を要求するためのフラグ。0 以外の場合、機密性と整合性の両方を要求します。0 の場合、整合性サービスだけを要求します。

qop_req	要求する QOP (保護品質)。MIC を生成するときと暗号化を行うときに使用される暗号化アルゴリズムです。移植性のためには、アプリケーションは可能な限りデフォルトの QOP を指定するべきです。つまり、この引数に GSS_C_QOP_DEFAULT を設定します。デフォルト以外の QOP の指定については、付録 C を参照してください。
req_output_size	指定した転送プロトコルが処理できるデータ片の最大サイズ (型は int)。この情報は、ユーザーが自ら提供する必要があります。つまり、GSS-API はプロトコルに依存しないため、どのプロトコルが使用されるのかを知る方法がありません。
max_input_size	関数から戻される値。つまり、ラップしたときに *req_output_size* を超えないような、ラップしていないメッセージの最大サイズ。

正常に終了した場合、gss_wrap_size_limit() は GSS_S_COMPLETE を戻します。指定した QOP が有効でなかった場合、gss_wrap_size_limit() は GSS_S_BAD_QOP を戻します。gss_wrap_size_limit() でオリジナルのメッセージの最大サイズを求める例 (機密性を使用する場合と使用しない場合の両方) については、103ページの「call_server()」を参照してください。

この機能は gss_wrap() を呼び出した時点でのシステムリソースの可用性に依存するため、この呼び出しが正常に終了したとしても、必ずしも、gss_wrap() が *max_input_size* 以下のバイトの長さを持つメッセージを保護できるとは保証できません。詳細は、gss_wrap_size_limit(3GSS) のマニュアルページを参照してください。

ラップ解除と検証

ラップされたメッセージを受信した後は、gss_unwrap() でメッセージをラップ解除する必要があります。gss_unwrap() は、ラップされたメッセージに埋め込まれている MIC に対してメッセージを自動的に検証します。送信側がメッセージをラップしなかったが、gss_get_mic() で MIC を生成している場合は、gss_verify_mic() で、その MIC に対して受信したメッセージを検証できま

す。後者の場合、受け入れ側はメッセージと MIC の両方を受信するようアレンジする必要があります。

gss_unwrap()

次に、gss_unwrap() の形式を示します。

```
OM_uint32 gss_unwrap (
OM_uint32          *minor_status,
const gss_ctx_id_t context_handle,
const gss_buffer_t input_message_buffer,
gss_buffer_t       output_message_buffer,
int                *conf_state
gss_qop_t          *qop_state)
```

minor_status 実際の機構から戻される状態コード。

context_handle このメッセージが送信されるコンテキスト。

input_message_buffer ラップされたメッセージ。この引数は gss_buffer_desc オブジェクトの形式である必要があります (19ページの「文字列および類似のデータ」を参照)。使用し終わったときには、gss_release_buffer() で解放する必要があります。

output_message_buffer ラップ解除したメッセージ用のバッファ。アプリケーションがラップ解除したメッセージを処理した後には、gss_release_buffer() でこのバッファを解放する必要があります。この引数も gss_buffer_desc オブジェクトです。

conf_state 機密性が適用されたかどうかを示すフラグ。0 以外の場合、機密性、メッセージ起点認証、および整合性サービスが適用されたことを示します。0 の場合、メッセージ起点認証と整合性だけが適用されたことを示します。必要ない場合は、NULL を指定します。

qop_state 使用する QOP (保護品質)。MIC を生成するときと暗号化を行うときに使用される暗号化アルゴ

リズムです。必要ない場合は、NULL を指定します。

メッセージが正常にラップ解除された場合、gss_unwrap() は GSS_S_COMPLETE を戻します。MIC に対してメッセージを検証できなかった場合、gss_unwrap() は GSS_S_BAD_SIG を戻します。

gss_verify_mic()

メッセージがラップ解除された場合、あるいは、初めからメッセージがラップされていない場合は、gss_verify_mic() でメッセージを検証できます。次に、gss_verify_mic() の形式を示します。

```
OM_uint32 gss_verify_mic (
OM_uint32          *minor_status,
const gss_ctx_id_t context_handle,
const gss_buffer_t message_buffer,
const gss_buffer_t token_buffer,
gss_qop_t          qop_state)
```

minor_status 実際の機構から戻される状態コード。

context_handle メッセージが送信されるコンテキスト。

message_buffer 受信したメッセージ。この引数は gss_buffer_desc オブジェクトの形式である必要があります (19ページの「文字列および類似のデータ」を参照)。アプリケ-ションが使用し終わったときには、gss_release_buffer() で解放する必要があります。

token_buffer 受信した MIC が入っているトークン。この引数は gss_buffer_desc オブジェクトの形式である必要があります (19ページの「文字列および類似のデータ」を参照)。アプリケーションが使用し終わったときには、gss_release_buffer() で解放する必要があります。

qop_state MIC を生成するときに適用される QOP (保護品質)。必要ない場合は、NULL を指定します。

メッセージが正常に検証された場合、`gss_verify_mic()` は `GSS_S_COMPLETE` を戻します。MIC に対してメッセージを検証できなかった場合、`gss_verify_mic()` は `GSS_S_BAD_SIG` を戻します。

転送の確認 (任意)

転送されたメッセージをラップ解除または検証した後、受信側は送信側に確認を送信することもできます。つまり、そのメッセージの MIC を返送します。送信側がラップはしなかったが `gss_get_mic()` で MIC をタグ付けしているメッセージの場合を考えます。図 1–14 に、このプロセスの様子を示します。

1. 起動側は `gss_get_mic()` でメッセージにタグ付けします。
2. 起動側はメッセージと MIC を受け入れ側に送信します。
3. 受け入れ側は `gss_verify_mic()` でメッセージを検証します。
4. 受け入れ側は MIC を起動側に返送します。
5. 起動側は `gss_verify_mic()` で、オリジナルのメッセージに対して受信した MIC を検証します。

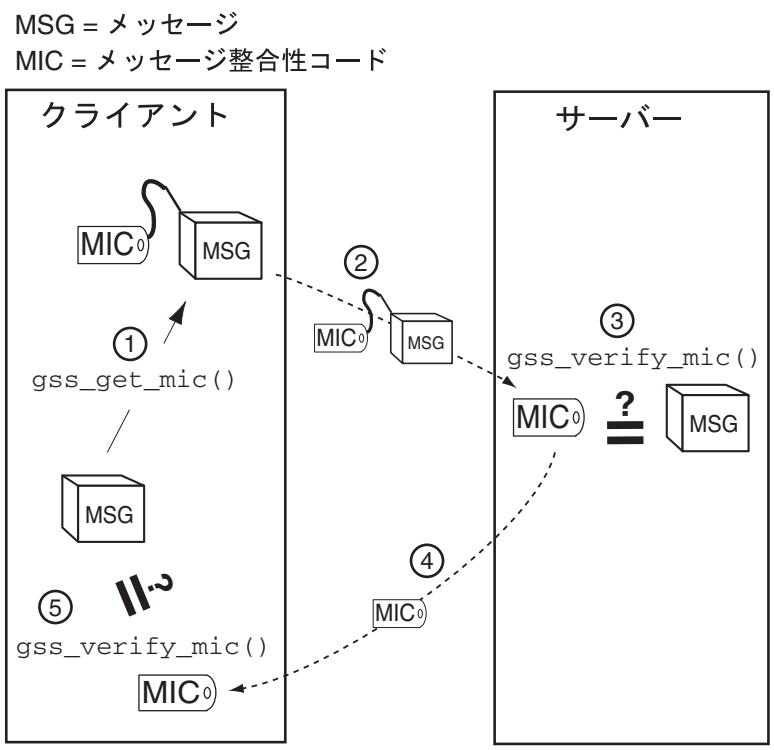

図 1–14　MIC 付きデータの確認

ラップされたデータの場合、gss_unwrap() 関数はメッセージと MIC を別々に生成しません。したがって、受信側は、受信した (およびラップ解除した) メッセージから MIC を生成する必要があります。図 1–15 に、このプロセスの様子を示します。

1. 起動側は gss_wrap() でメッセージをラップします。
2. 起動側はラップしたメッセージを送信します。
3. 受け入れ側は gss_unwrap() でメッセージをラップ解除します。
4. 受け入れ側は gss_get_mic() でラップ解除されたメッセージの MIC を生成します。
5. 受け入れ側は抽出した MIC を起動側に返信します。
6. 起動側は gss_verify_mic() で、オリジナルのメッセージに対して受信した MIC を検証します。

MSG = メッセージ
MIC = メッセージ整合性コード

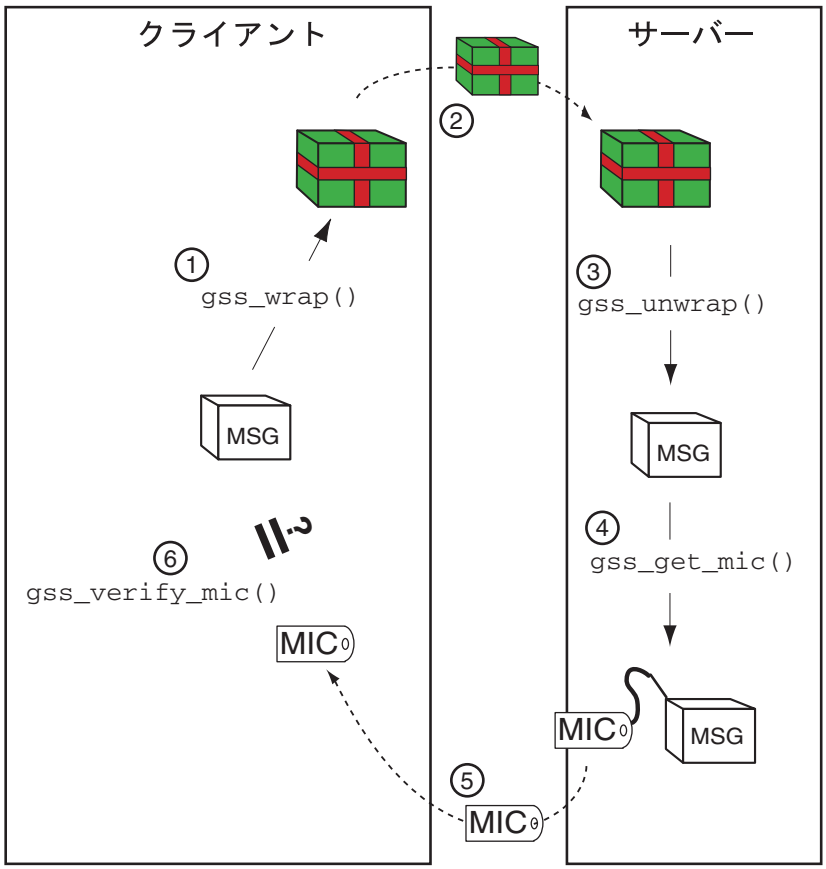

図 1–15　ラップされたデータの確認

コンテキストの削除とデータの解放

すべてのメッセージを送受信し、終了した後、起動側と受け入れ側のアプリケーションは両方とも gss_delete_sec_context() で共有コンテキストを破壊する必要があります。gss_delete_sec_context() はコンテキストに関連するローカルのデータ構造体を削除します。次に、gss_delete_sec_context() の形式を示します。

```
OM_uint32 gss_delete_sec_context (
OM_uint32     *minor_status,
gss_ctx_id_t *context_handle,
```

```
gss_buffer_t output_token)
```

minor_status	実際の機構から戻される状態コード。
context_handle	削除されるコンテキスト。
output_token	GSS_C_NO_BUFFER に設定します。

詳細は、gss_delete_sec_context(3GSS) のマニュアルページを参照してください。

用心のため、アプリケーションは GSS-API データ用に割り当てたデータ領域をすべて解放するべきです。このような関数には、gss_release_buffer()、gss_release_cred()、gss_release_name()、および gss_release_oid_set() があります。詳細は、それぞれのマニュアルページを参照してください。

第 2 章

GSS-API サンプルプログラムについての概略説明

サンプルプログラムの概要

付録 A には、GSS-API を使用する 2 つの C 言語アプリケーション (クライアント用とサーバー用) のソースコードが掲載されています。この章では、これらのアプリケーションを使用して GSS-API を段階的に説明します。この章は付録 A を参照しながら読んでいくことを想定しています。アプリケーションのすべての面を詳細に説明するわけではなく、GSS-API の使用に関連する面に焦点を置いて説明します。

注意 - GSS-API は自動的に自分自身をクリーンアップしないため、GSS-API を使用するアプリケーションや関数はこのクリーンアップを行う必要があります。つまり、たとえば、GSS-API バッファや GSS-API 名前空間を使用する関数は、終了時に、`gss_release_buffer()` や `gss_release_name()` などの GSS-API 関数を呼び出す必要があります。

繰り返しを避けるため、以降のコードでは一般的にこのようなクリーンアップを省いてあります。実行する必要があることは必ず覚えておいてください。いつどこでクリーンアップ関数を使用するかが不明な場合は、付録 A のサンプルプログラムを参照してください。

クライアント側の GSS-API: gss-client

サンプルのクライアント側プログラム gss-client は、サーバーとのセキュリティコンテキストを作成し、セキュリティパラメータを確立し、文字列 (メッセージ) をサーバーに送信します。接続にはシンプルな TCP ベースのソケット接続を使用します。

次に、gss-client のコマンド行形式を示します。

```
gss-client [–port port] [–d] [–mech mech] host service [–f] msg
```

gss-client は主に次の作業を行います。

1. コマンド行を解析します。
2. 機構の OID (オブジェクト ID) を作成します (指定されている場合)。
3. サーバーとの接続を設定します。
4. コンテキストを確立します。
5. メッセージをラップします。
6. メッセージを送信します。
7. サーバーが正しくメッセージに署名していることを検証します。

以降では、gss-client がどのように動作するかを段階的に説明します。機能性を示すために設計されたサンプルプログラムであるため、上記の手順にあまり関係のない部分は省略しています。コンテキストのインポートやエクスポート、ラップサイズの取得などの機能については、このマニュアルの他の場所を参照してください。

概要: main() (クライアント)

すべての C プログラムと同様に、プログラムの外部シェルはエントリポイント関数 main() に含まれます。main() は次の 4 つの機能を実行します。

1. コマンド行引数を解析し、各引数を変数に割り当てます。
   - *port* が指定されている場合、*port* は *host* で指定されたリモートマシンへの接続を確立するポート番号です。
   - –d フラグが設定されている場合、セキュリティ資格はサーバーに委託されます。特に、*deleg_flag* 変数は GSS-API 値 GSS_C_DELEG_FLAG に設定されます。–d フラグが設定されていない場合、*deleg_flag* は 0 に設定されます。

- (省略可能) *mech* は使用されるセキュリティ機構名 (Kerberos v5 や X.509 など) です。機構が指定されていない場合、GSS-API はデフォルトの機構を使用します。
- クライアントから要求されたネットワークサービス名 (`telnet`、`ftp`、`login` などのサービス) は *service_name* に割り当てられます。
- 最後に、*msg* は保護されたデータとしてサーバーに送信される文字列です。`–f` オプションが指定されている場合、*msg* は文字列を読み取るべきファイル名です。

次に、コマンド行の例を示します。

```
% gss-client -port 8080 -d -mech kerberos_v5 erebos.eng nfs "ls"
```

次のコマンド行は機構もポートも指定せず、委託も使用しません。

```
% gss-client erebos.eng nfs "ls"
```

2. `parse_oid()` を呼び出して、(コマンド行に指定されている場合) セキュリティ機構名から、GSS-API OID (オブジェクト識別子) を作成します。

```
if (mechanism)
        parse_oid(mechanism, &g_mechOid);
```

mechanism は変換される文字列で、*g_mechOid* は機構の `gss_OID` オブジェクトへのポインタです。デフォルト以外の機構の指定については、付録 C を参照してください。

3. `call_server()` を呼び出します。この関数は、コンテキストの作成とデータの送信を実際に行います。

```
if (call_server(hostname, port, g_mechOid, service_name,
                    deleg_flag, msg, use_file) < 0)
          exit(1);
```

4. OID の記憶領域を解放します (まだ解放されていない場合)。

```
if (g_mechOID != GSS_C_NULL_OID)
     (void) gss_release_oid(&min_stat, &g_mechoid);
```

gss_release_oid() は、GSS-API の Sun の実装ではサポートされますが、すべての GSS-API 実装でサポートされるわけではなく、標準ではないと考えられます。アプリケーションは、gss_str_to_oid() で機構を割り当てるのではなく、可能な限り GSS-API が提供するデフォルトの機構を使用するべきです。したがって、gss_release_oid() コマンドは通常使用するべきではありません。

デフォルト以外の機構の指定

一般的に、GSS-API を使用するアプリケーションは特定の機構を指定せず、GSS-API 実装が提供するデフォルトの機構を使用すべきです。デフォルトの機構を指定するには、機構の表す gss_OID を値 GSS_C_NULL_OID に設定します。

デフォルト以外の機構を設定することは推奨されないため、このプログラムでは取り上げません。クライアントアプリケーションがユーザー指定の (デフォルト以外の) 機構名を解析する方法については、102ページの「parse_oid()」のコードを参照してください。また、デフォルト以外の OID を使用する方法については、付録 C を参照してください。

サーバーの呼び出し

機構を gss_OID 形式で格納した後、実際の作業を行うことができます。つまり、main() は、コマンド行引数とほとんど同じ引数で、call_server() 関数を呼び出します。

```
call_server(hostname, port, g_mechOid, service_name,
     deleg_flag, msg, use_file);
```

use_file は、送信されるメッセージがファイルに格納されているかどうかを示すフラグです。

サーバーへの接続

変数を宣言した後、call_server() はまずサーバーとの接続を設定します。

```
if ((s = connect_to_server(host, port)) < 0)
     return -1;
```

s はファイル記述子で (型は int)、最初は socket() の呼び出しから戻されます。

connect_to_server() は、ソケットを使用して接続を作成するシンプルな関数です。この関数は GSS-API を使用しないため、ここでは省略します。connect_to_server() についての詳細は、112ページの「connect_to_server()」を参照してください。

コンテキストの確立

接続が確立された後、call_server() は client_establish_context() 関数を使用して、セキュリティコンテキストを確立します。

```
int client_establish_context(s, service_name, deleg_flag, oid,
     &context, &ret_flags)
```

- *s* は、connect_to_server() で確立される接続を表すファイル記述子です。
- *service_name* は要求するネットワークサービスです (nfs など)。
- *deleg_flag* は、サーバーがクライアントのプロキシとして動作するかどうかを指定します。
- *oid* は機構です。
- *context* は作成されるコンテキストです。
- *ret_flags* は GSS-API 関数 gss_init_sec_context() から戻されるフラグを指定します (型は int)。

コンテキストを起動するために、アプリケーションは gss_init_sec_context() 関数を使用します。ほとんどの GSS-API 関数と同様に、この関数でも名前は GSS-API 内部形式である必要があります。したがって、アプリケーションはまず、サービス名を文字列から内部形式に変換する必要があります。このためには、gss_import_name() を使用します。

```
maj_stat = gss_import_name(&min_stat, &send_tok,
     (gss_OID) GSS_C_NT_HOSTBASED_SERVICE, &target_name);
```

この関数は引数としてサービス名を受け取り (参照できない (不透明な) GSS-API バッファ send_tok に格納される)、GSS-API 内部名 target_name に変換します。send_tok は新しい gss_buffer_desc を宣言せず、領域を節約するために使用されます。3 番目の引数は gss_OID 型で、send_tok に格納されている名前の形式を示します。この場合は GSS_C_NT_HOSTBASED_SERVICE で、サービスの形式が *service@host* であることを意味します。この引数に有効なその他の値については、146ページの「名前型」を参照してください。

サービスを GSS-API 内部形式に変換した後、コンテキストを確立する作業に進むことができます。移植性を最大限にするには、コンテキストの確立を常にループとして実行する必要があります。40ページの「コンテキストの起動 (クライアント)」を参照してください。

まず、アプリケーションはコンテキストを NULL に初期化します。

```
*gss_context = GSS_C_NO_CONTEXT;
```

次に、サーバーから受け取るトークンも NULL に初期化します。

```
token_ptr = GSS_C_NO_BUFFER;
```

次に、アプリケーションはループに入ります。ループは 2 つのことを検査しながら進みます。gss_init_sec_context() から戻される状態と、サーバーに送信されるトークンのサイズ (この値も gss_init_sec_context() で生成される) です。トークンのサイズが 0 の場合、サーバーがクライアントからこれ以上のトークンを期待していないことを意味します。次に、このループの疑似コードを示します。

```
do{
      gss_init_sec_context()
      if (コンテキストが全く確立されなかった場合)
                  エラーを出力して終了する
      if (状態が「完了」または「処理中」のどちらでもない場合
                  サービスの名前空間を解放し、エラーを出力して終了する
      if (サーバーに送信するトークンがある場合。つまり、サイズが 0 以外の場合)
                  トークンを送信する
                  if (トークンの送信が失敗した場合)
                  トークンとサービスの名前空間を解放し、エラーを出力して終了する
                  先ほど送信したトークンの名前空間を解放する
      if (コンテキストの確立が完了していない場合
                  サーバーからトークンを受け取る
```

(続く)

```
} while (コンテキストが完全になるまで)
```

次に、gss_init_sec_context() への呼び出しを示します。

```
do {
     maj_stat = gss_init_sec_context(&min_stat,
                              GSS_C_NO_CREDENTIAL,
                              gss_context,
                              target_name
                              oid
                              GSS_C_MUTUAL_FLAG |
                                 GSS_C_REPLAY_FLAG |
                                 deleg_flag,
                              0,
                              NULL,
                              &send_tok,
                              ret_flags,
                              NULL);
```

次に、各引数について説明します。

- 実際の機構が設定する状態コード。
- 資格ハンドル。デフォルトのプリンシパルとして動作させるために、GSS_C_NO_CREDENTIAL を使用します。
- (*gss_context*) 作成するコンテキストハンドル。
- (*target_name*) GSS_API 内部名形式のサービス。
- (*oid*) 機構。
- 要求フラグ。この場合、クライアントは、

  a) サーバーが自分自身を認証すること

  b) メッセージの複製をオンにすること

  c) サーバーがプロキシとして動作すること (要求された場合)

  を要求します。
- コンテキストの時間制限はありません。
- チャネルバインディングの要求はありません。
- (*token_ptr*) サーバーから受け取るトークンへのポインタ (ある場合)。

- サーバーが実際に使用する機構。アプリケーションがこの値に関与していないため、ここでは NULL に設定されています。
- (*&send_tok*) サーバーに送信するために gss_init_sec_context() が作成するトークン。
- 戻りフラグ。ここでは無視するため、NULL に設定されています。

コンテキストを起動する前には、クライアントは資格を獲得する必要がないことに注意してください。クライアント側では、資格の管理は GSS-API によって透過的に処理されます。つまり、(通常はログイン時に) プリンシパルのために機構が作成した資格をどのように取得するかを、GSS-API は知っているということです。このため、アプリケーションは gss_init_sec_context() にデフォルトの資格を渡しています。しかし、サーバー側では、サーバーアプリケーションはコンテキストを受け入れる前に、サービスの資格を明示的に獲得する必要があります。91ページの「資格の獲得」を参照してください。

コンテキスト (完成されている必要はない) があること
と、gss_init_sec_context() が有効な状態を戻したことを検査した後、アプリケーションは gss_init_sec_context() がサーバーに送信すべきトークンをコンテキストに渡しているかどうかを調べます。渡していない場合、これは、トークンが (これ以上) 必要でないことをサーバーが示しているためです。渡している場合、トークンをサーバーに送信します。トークンの送信が失敗した場合、トークンの名前空間とサービスを解放し、(エラーで) 終了します。トークンの存在をチェックするには、トークンの長さ (サイズ) を調べることに注意してください。

```
if (send_tok_length != 0) {
                if (send_token(s, &send_tok) < 0) {
                     (void) gss_release_buffer(&min_stat, &send_tok);
                     (void) gss_release_name(&min_stat, &target_name);
                     return -1;
                }
          }
```

send_token() は GSS-API 関数ではなく、ユーザーが作成した基本的なファイル書き込み関数です。詳細は、131ページの「send_token()」を参照してください。GSS-API 自身はトークンを送受信しないことに注意してください。GSS-API が作成したトークンを送受信するのは、呼び出し側アプリケーションの責任です。

サーバーが (これ以上) 送信するトークンを持っていない場合、gss_init_sec_context() は GSS_S_COMPLETE を戻します。つまり、gss_init_sec_context() がこの値を戻さなかった場合、アプリケーションはまだ受け取るトークンがサーバーに残っていると判断できます。受け取りが失敗した場合、アプリケーションはサービスの名前空間を解放し、(エラーで) 終了します。

```
if (maj_stat == GSS_S_CONTINUE_NEEDED) {
               if (recv_token(s, &recv_tok) < 0) {
                    (void) gss_release_name(&min_stat, &target_name);
                    return -1;
```

最後に、プログラムはトークンのポインタをリセットします。そして、コンテキストが完全に確立されるまで、ループを繰り返します。したがって、do ループの終了は次のようになります。

```
} while (maj_stat == GSS_S_CONTINUE_NEEDED);
```

データの送信

セキュリティコンテキストが確立された後、gss-client はデータをラップし、データを送信し、そして、サーバーが戻した「署名」を検証する必要があります。他にも行うことはありますが (コンテキストについての情報の表示など)、gss-client はプログラム例であるため、ここでは省略して、データの送信と検証を行います。まず、送信すべきメッセージ (ls など) をバッファに格納します。

```
     if (use_file) {
         read_file(msg, &in_buf);
     } else {
         /* メッセージをラップする */
         in_buf.value = msg;
         in_buf.length = strlen(msg) + 1;
     }
```

ラップする前に、データを暗号化できるかどうかを検査します。

```
     if (ret_flag & GSS_C_CONF_FLAG) {
          state = 1;
     else
          state = 0;
     }
```

次に、データをラップします。

```
maj_stat = gss_wrap(&min_stat, context, conf_req_flag, GSS_C_QOP_DEFAULT,
                    &in_buf, &state, &out_buf);
if (maj_stat != GSS_S_COMPLETE) {
     display_status("wrapping message", maj_stat, min_stat);
     (void) close(s);
     (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
     return -1;
} else if (! state) {
     fprintf(stderr, "Warning!  Message not encrypted.\n");
}
```

このように、*in_buf* に格納されたメッセージは、*context* で参照されるサーバーに送信されます。このとき、機密性サービスとデフォルトの保護品質 (QOP) も要求されます。保護品質とは、データを変換するときに適用されるアルゴリズムのことです。移植性のためには、可能な限りデフォルトの保護品質を使用するようにします。`gss_wrap()` はメッセージをラップし、その結果を *out_buf* に格納し、機密性が実際にラップで適用されたかどうかをフラグ (*state*) に設定します。

クライアントは独自の `send_token()` 関数で、ラップされたメッセージをサーバーに送信します。`send_token()` 関数については、83ページの「コンテキストの確立」を参照してください。

```
send_token(s, &outbuf)
```

メッセージの検証

次に、送信したメッセージの有効性を検証します。送信したメッセージの MIC がサーバーから戻されることが判明しているため、プログラムは独自の `recv_token()` 関数で MIC (Message Integrity Code) を受け取ります。そして、`gss_verify_mic()` でサーバーの「署名」(MIC) を検証します。

```
    maj_stat = gss_verify_mic(&min_stat, context, &in_buf,
                              &out_buf, &qop_state);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("verifying signature", maj_stat, min_stat);
        (void) close(s);
        (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
        return -1;
    }
```

gss_verify_mic() はサーバーのトークン (*out_buf* に格納されている) とともに受け取った MIC を、オリジナルのラップ解除されたメッセージ (*in_buf* に格納されている) から作成した MIC と比較します。2 つの MIC が一致した場合、メッセージの有効性は検証されたことになります。次に、クライアントは受け取ったトークンのバッファ (*out_buf*) を解放します。

終了時、call_server() はコンテキストを削除し、main() に戻ります。

サーバー側の GSS-API: gss-server

通常クライアントは、セキュリティハンドシェークを実行するためにサーバーを必要とします。クライアントはセキュリティコンテキストを起動し、データを送信します。一方、サーバーはコンテキストを受け入れ、クライアントの ID を検証する必要があります。この中で、サーバーは自分自身をクライアントに対して認証したり (要求された場合)、データの「署名」をクライアントに提供したりする場合もあります。もちろん、これに加えて、サーバーはデータを処理する必要があります。

次に、gss-server のコマンド行形式を示します。

```
gss-server [-port port] [-verbose] [-inetd] [-once] [-logfile file] \
           [-mech mechanism] service_name
```

gss-server は次の作業を行います。

1. コマンド行を解析します。
2. コマンド行に指定された機構名を、内部形式に変換します (ある場合)。
3. 呼び出し側の資格を獲得します。
4. inetd デーモンを使用して接続するようにユーザーが指定しているかどうかをチェックします。

5. 接続を確立します。
6. データを取得します。
7. データに署名し、署名を戻します。
8. 名前空間を解放し、終了します。

以降では、gss-server がどのように動作するかを段階的に説明します。機能性を示すために設計されたサンプルプログラムであるため、上記の手順にあまり関係のない部分は省略しています。

概要: main() (サーバー)

gss-client は main() 関数から始めます。main() は次の作業を実行します。

1. コマンド行引数を解析し、各引数を変数に割り当てます。
   - *port* が指定されている場合、*port* は待機するポート番号です。*port* が指定されていない場合、プログラムはデフォルトでポート 4444 を使用します。
   - –verbose が指定されている場合、プログラムはデバッグに類似したモードで動作します。
   - –inetd オプションは、プログラムが inetd デーモンを使用してポートで待機することを指示します。inetd は stdin と stdout を使用してクライアントとの接続を処理します。
   - –once が指定されている場合、プログラムは 1 つのインスタンス接続だけを作成します。
   - (省略可能) *mechanism* は使用されるセキュリティ機構名 (Kerberos v5 など) です。機構が指定されていない場合、GSS-API はデフォルトの機構を使用します。
   - クライアントから要求されたネットワークサービス名 (telnet、ftp、login などのサービス) は *service_name* で指定されます。

   次に、コマンド行の例を示します。

```
% gss-server -port 8080 -once -mech kerberos_v5 erebos.eng nfs "hello"
```

2. 機構を GSS-API オブジェクト識別子 (OID) に変換します (指定されている場合)。これは、GSS-API 関数が名前を内部形式で処理するためです。
3. 使用される機構 (Kerberos v5 など) のために、サービス (ftp など) の資格を獲得します。

4. sign_server() 関数を呼び出します。この関数は、接続の確立、メッセージの受け取り、その署名など、ほとんどの作業を実際に行います。

   inetd を使用するようにユーザーが指定している場合、プログラムは標準出力と標準エラーを閉じ、標準入力で sign_server() を呼び出します。inetd はこの sign_server() を使用して接続を渡します。そうでない場合、プログラムはソケットを作成し、そのソケットの接続を TCP 関数 accept() で受け入れ、accept() から戻されたファイル記述子で sign_server() を呼び出します。

   inetd を使用しない場合、プログラムは終了されるまで接続とコンテキストを作成します。しかし、ユーザーが –once オプションを指定している場合、ループは最初の接続の後で終了します。

5. 獲得した資格を解放します。
6. 機構 OID の名前空間を解放します。
7. 接続を閉じます (まだ開いている場合)。

機構の OID の作成

gss-client プログラムの例と同様に、サンプルのサーバープログラムでもユーザーは機構を指定できます。しかし、すべてのアプリケーションで、GSS-API 実装が提供するデフォルトの機構を使用することを強く推奨します。デフォルトの機構を指定するには、機構の表す gss_OID を値 GSS_C_NULL_OID に設定します。コードについては、117ページの「createMechOid()」を参照してください。また、デフォルト以外の機構を使用する方法については、付録 C を参照してください。

資格の獲得

クライアントアプリケーションと同様に、サーバーアプリケーションと GSS-API はどちらも資格を作成しません。資格は実際の機構が作成します。クライアントプログラムとは異なり、サーバーは必要な資格を明示的に獲得する必要があります。クライアントアプリケーションの中にも、明示的に資格を獲得するものはあります。この場合も、ここで説明する方法を使用します。しかし、一般的には、クライアントはログイン時にすでに資格を獲得しており、GSS-API はこのような資格を自動的に獲得します。

gss-server プログラムは独自の関数 server_acquire_creds() を持っています。この関数は提供されるサービスの資格を取得します。この関数は、入力としてサービス名と使用されるセキュリティ機構を受け取り、そのサービスの資格を戻します。

server_acquire_creds() は GSS-API 関数 gss_acquire_cred() を使用して、サーバーが提供するサービスの資格を取得します。しかし、これを行う前には、次の 2 つのことを処理する必要があります。

1 つの資格を複数の機構で共有できる場合、gss_acquire_cred() はこのような機構すべての資格を戻します。したがって、gss_acquire_cred() は入力として 1 つの機構ではなく、機構の集合を受け取ります (36ページの「資格」を参照)。しかし、ほとんどの場合 (このプログラムも含む)、1 つの資格は複数の機構で機能しません。さらに、サーバーアプリケーションでは、1 つの機構をコマンド行に指定するか、デフォルトの機構を使用します。したがって、最初に行うことは、gss_acquire_cred() に渡される機構の集合に 1 つの機構 (デフォルトまたはそれ以外) だけが入っていることを確認することです。

```
if (mechOid != GSS_C_NULL_OID) {
     desiredMechs = &mechOidSet;
     mechOidSet.count = 1;
     mechOidSet.elements = mechOid;
} else
     desiredMechs = GSS_C_NULL_OID_SET;
```

GSS_C_NULL_OID_SET は、デフォルトの機構を使用することを示します。

gss_acquire_cred() はサービス名を gss_name_t 構造体の形式で受け取ります。したがって、2 番目に行うことは、サービス名を gss_name_t 構造体の形式にインポートすることです。このためには、gss_import_name() を使用します。ほとんどの GSS-API 関数と同様に、この関数でも引数は GSS-API 型である必要があるため、まず、サービス名を GSS-API バッファにコピーする必要があります。

```
     name_buf.value = service_name;
     name_buf.length = strlen(name_buf.value) + 1;
     maj_stat = gss_import_name(&min_stat, &name_buf,
                (gss_OID) GSS_C_NT_HOSTBASED_SERVICE, &server_name);
     if (maj_stat != GSS_S_COMPLETE) {
          display_status("importing name", maj_stat, min_stat);
          if (mechOid != GSS_C_NO_OID)
                gss_release_oid(&min_stat, &mechOid);
          return -1;
```

(続く)

```
    }
```

今度も標準でない関数 gss_release_oid() を使用していることに注意してください。80ページの「概要: main() (クライアント)」を参照してください。

入力のサービス名は *name_buf* に文字列として格納されます。出力は *server_name* という gss_name_t 構造体へのポインタです。3 番目の引数 GSS_C_NT_HOSTBASED_SERVICE は *name_buf* に格納されている文字列の名前型です。この場合、文字列が *service@host* というサービスの形式で解釈されることを示します。

これで、サーバープログラムは gss_acquire_cred() を呼び出すことができます。

```
maj_stat = gss_acquire_cred(&min_stat, server_name, 0,
                                desiredMechs, GSS_C_ACCEPT,
                                server_creds, NULL, NULL);
```

次に、各引数について説明します。

- *min_stat* は関数から戻されるエラーコードです。
- *server_name* は (上記のとおり) サーバー名です。
- 0 は、プログラムが資格の有効期間の最大値には関心がないことを示します。
- *desiredMechs* は (上記のとおり) この資格が適用する機構の集合です。
- GSS_C_ACCEPT は、資格がセキュリティコンテキストを受け入れるためだけに使用できることを示します。
- *server_creds* は関数から戻される資格ハンドルです。
- NULL は、適用される機構や、資格が有効である期間を、プログラムが知る必要がないことを示します。

コンテキストの受け入れと、データの取得と署名

サービスの資格を獲得した後、サーバープログラムは、ユーザーが inetd を使用するように指定しているかどうかを調べて (90ページの「概要: main() (サーバー)」を参照)、次に、sign_server() を呼び出します。sign_server() はプログラムの主な作業を行います。sign_server() が最初に行うことは、server_establish_context() を呼び出してコンテキストを確立することです。

注 - ここでは、inetd については説明しません。基本的に、inetd が指定された場合、プログラムは標準入力で sign_server() を呼び出します。そうでない場合、プログラムはソケットを作成し、接続を受け入れ、次に、その接続で sign_server() を呼び出します。

---

sign_server() は次の作業を行います。

1. コンテキストを受け入れます。
2. データをラップ解除します。
3. データに署名します。
4. データを戻します。

コンテキストの受け入れ

コンテキストの確立では、クライアントとサーバー間で一連のトークンが交換されます。したがって、プログラムの移植性を保持するために、コンテキストの受け入れとコンテキストの起動はループで行われる必要があります。実際、コンテキストを受け入れるためのループとコンテキストを確立するためのループは (まったく逆ですが) 非常によく似ています。83ページの「コンテキストの確立」と比較してください。

1. サーバーが最初に行うことは、クライアントがコンテキスト起動プロセスの一部として送信したトークンを探すことです。GSS-API 自身はトークンを送受信しないことを思い出してください。したがって、この作業を行うには、プログラムは独自のルーチンを持つ必要があります。サーバーがトークンの受信用に使用するルーチンを、この例では recv_token() としています。詳細は、132ページの「recv_token()」を参照してください。

```
    do {
         if (recv_token(s, &recv_tok) < 0)
```

(続く)

```
            return -1;
```

2. 次に、プログラムは GSS-API 関数 gss_accept_sec_context() を呼び出します。

```
maj_stat = gss_accept_sec_context(&min_stat,
                                  context,
                                  server_creds,
                                  &recv_tok,
                                  GSS_C_NO_CHANNEL_BINDINGS,
                                  &client,
                                  &doid,
                                  &send_tok,
                                  ret_flags,
                                  NULL,    /* time_rec を無視する */
                                  NULL);   /* del_cred_handle を無視する */
```

次に、各引数について説明します。

- *min_stat* は実際の機構から戻されるエラー状態です。
- *context* は確立されているコンテキストです。
- *server_creds* は提供されているサーバーの資格です (91ページの「資格の獲得」を参照)。
- *recv_tok* は recv_token() でクライアントから受信したトークンです。
- GSS_C_NO_CHANNEL_BINDINGS はチャネルバインディングを使用しないことを示すフラグです (59ページの「チャネルバインディング」を参照)。
- *client* はクライアント名 (ASCII 文字) です。
- *oid* は機構です (OID 形式)。
- *send_tok* はクライアントに送信するトークンです。
- *ret_flags* は、コンテキストがサポートするオプション (メッセージ順序検出など) を示すさまざまなフラグです。
- NULL は、コンテキストが有効である期間の長さにも、サーバーがクライアントのプロキシとして動作できるかどうかにも、プログラムが関与していないことを示します。

gss_accept_sec_context() が *maj_stat* に GSS_S_CONTINUE_NEEDED を設定している限り、受け入れループは継続します (エラーの場合を除く)。*maj_stat*

の値が GSS_S_CONTINUE_NEEDED でも GSS_S_COMPLETE でもない場合、問題が発生し、ループが終了したことを示します。

3. クライアントに返送するトークンが存在する場合、gss_accept_sec_context() は *send_tok* の長さ (正の数) を戻します。次に行うことは、送信するトークンがあるかどうかを調べて、もしあれば、そのトークンを送信することです。

```
     if (send_tok.length != 0) {
          . . .
          if (send_token(s, &send_tok) < 0) {
               fprintf(log, "failure sending token\n");
               return -1;
          }

          (void) gss_release_buffer(&min_stat, &send_tok);
          }
```

メッセージのラップ解除

コンテキストを受け入れた後、サーバーはクライアントから送信されたメッセージを受信します。GSS-API はこの作業を行う関数を提供しないため、プログラムは独自の関数 recv_token() を使用します。

```
if (recv_token(s, &xmit_buf) < 0)
     return(-1);
```

メッセージは暗号化されている可能性があるため、プログラムは GSS-API 関数 gss_unwrap() でメッセージをラップ解除します。

```
maj_stat = gss_unwrap(&min_stat, context, &xmit_buf, &msg_buf,
                      &conf_state, (gss_qop_t *) NULL);
     if (maj_stat != GSS_S_COMPLETE) {
        display_status("unwrapping message", maj_stat, min_stat);
        return(-1);
     } else if (! conf_state) {
        fprintf(stderr, "Warning!  Message not encrypted.\n");
     }

     (void) gss_release_buffer(&min_stat, &xmit_buf);
```

gss_unwrap() は、recv_token() が *xmit_buf* に格納したメッセージを入力として受け取り、そのメッセージを変換し、その結果を *msg_buf* に格納しま

す。gss_unwrap() への 2 つの引数に注目してください。*conf_state* は、このメッセージに機密性が適用されたかどうか (つまり、メッセージが暗号化されているかどうか) を示すフラグです。次に、最後の NULL は、メッセージを保護するときに使用された QOP にプログラムが関与していないことを示します。

メッセージへの署名とメッセージの返送

最後にサーバーがすることは、メッセージに署名することです。つまり、メッセージの MIC (メッセージ整合性コード。メッセージに関連付けられた一意なタグ) をクライアントに戻すことによって、メッセージの送信とラップ解除が正常に完了したことをクライアントに証明します。このためには、プログラムは gss_get_mic() 関数を使用します。

```
maj_stat = gss_get_mic(&min_stat, context, GSS_C_QOP_DEFAULT,
                              &msg_buf, &xmit_buf);
```

この関数は *msg_buf* 内のメッセージを調べて、そのメッセージから MIC を生成し、その結果を *xmit_buf* に格納します。次に、サーバーは send_token() で MIC をクライアントに返送します。すると、クライアントは gss_verify_mic() で MIC を検証します。88ページの「メッセージの検証」を参照してください。

最後に、sign_server() はいくつかのクリーンアップを実行します。つまり、gss_release_buffer() で GSS-API バッファの *msg_buf* と *xmit_buf* を解放し、次に、gss_delete_sec_context() でコンテキストを無効にします。

コンテキストのインポートとエクスポート

61ページの「コンテキストのエクスポートとインポート」で説明したとおり、GSS-API を使用すると、コンテキストをエクスポートおよびインポートできます。通常、これを行う理由は、マルチプロセスプログラムにおける異なるプロセス間でコンテキストを共有することです。

sign_server() には検証用の関数 test_import_export_context() があります。この関数は、コンテキストのエクスポートとインポートがどのように機能するかを示します。この関数は実際にはコンテキストをプロセス間で渡しません。コンテキストをエクスポートするのにかかった時間を表示し、次に、インポートするのにかかった時間を表示するだけです。実際には機能しませんが、この関数は GSS-API のインポート関数とエクスポート関数をどのように使用するかを示すとと

もに、コンテキストの操作に関連するタイムスタンプをどのように使用するかの参考にもなります。test_import_export_context() の詳細は、125ページの「test_import_export_context()」を参照してください。

クリーンアップ

main() 関数に戻ると、アプリケーションは gss_delete_cred() でサービスの資格を削除します。機構の OID が指定されていた場合は、gss_delete_oid() で削除します。そして、終了します。

付属の関数

クライアントとサーバーのプログラムはいくつかのサポート関数を使用しています。たとえば、戻されたフラグの値を表示するなどです。このような関数は GSS-API に固有ではないため、あるいは、それほど重要ではないため、ここでは説明しません。このような関数については、126ページの「補助的な関数」で説明されているものもあります。しかし、このような関数の中でも send_token() と recv_token() の 2 つは特別で、重要です。したがって、131ページの「send_token() と recv_token()」で説明されています。

付録 **A**

C ベース の GSS-API サンプルプログラム

GSS-API を使用するプログラム

この付録では、GSS-API を使用して安全なネットワーク接続を行う 2 つのサンプルアプリケーションのソースコードを示します。一方はクライアントで、もう一方はサーバーです。2 つのプログラムは実行時にベンチマークを表示するため、ユーザーは GSS-API が使用されていることを見ることができます。さらに、クライアントアプリケーションとサーバーアプリケーションが使用する補助的な関数もいくつか示します。便宜上、クライアント側アプリケーションとサーバー側アプリケーションに分けて説明しています。

各プログラムについての詳細は、第 2 章を参照してください。

クライアント側アプリケーション

この節では、クライアント側プログラム `gss_client` について説明します。

プログラムヘッダー

プログラムヘッダーはクライアントプログラムの宣言部分です。また、コマンド行が間違っていた場合に構文を説明する関数もあります。

例 **A–1**　クライアントプログラムのヘッダー

```
/*
 * Copyright 1994 by OpenVision Technologies, Inc.
 *
 * Permission to use, copy, modify, distribute, and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appears in all copies and
 * that both that copyright notice and this permission notice appear in
 * supporting documentation, and that the name of OpenVision not be used
 * in advertising or publicity pertaining to distribution of the software
 * without specific, written prior permission. OpenVision makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 * OPENVISION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
 * EVENT SHALL OPENVISION BE LIABLE FOR ANY SPECIAL, INDIRECT OR
 * CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
 * USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
 * OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
 * PERFORMANCE OF THIS SOFTWARE.
 */


#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <error.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <gssapi/gssapi.h>
#include <gssapi/gssapi_ext.h>
#include "gss-misc.h"

/* ディスプレイ状態に必要な大域機構 oid、および資格の取得 */
gss_OID g_mechOid = GSS_C_NULL_OID;


void usage()
{
     fprintf(stderr, "Usage: gss-client [-port port] [-d]"
                        " [-mech mechOid] host service msg\n");
     exit(1);
}
```

main()

main() はプログラムのエントリポイントです。次に、コマンド行構文を示します。

```
gss-client [–port port] [–d] [–mech mech] host service msg
```

コマンド行を解析した後、main() は (指定されていれば) 該当するセキュリティ機構名を OID に変換し、安全な接続を確立し、そして、必要であれば、機構の OID を破棄します。

注 - main() は標準ではない関数 gss_release_oid() を使用します。この関数は GSS-API のすべての実装でサポートされているわけではないため、可能な限り使用するべきではありません。アプリケーションは独自の機構を割り当てるのではなく、デフォルトの機構を使用するため (GSS_C_NULL_OID で指定)、gss_release_oid() 関数はあまり使用するべきではありません。ここで使用している理由は下位互換性のためと、この GSS-API 実装の全体を示すためです。

例 A–2 main()

```
int main(argc, argv)
     int argc;
     char **argv;
{
    /* char *service_name, *hostname, *msg; */
     char *msg;
     char service_name[128];
     char hostname[128];
     char *mechanism = 0;
     u_short port = 4444;
     int use_file = 0;
     OM_uint32 deleg_flag = 0, min_stat;

     display_file = stdout;

     /* 引数を解析する */

        argc--; argv++;
     while (argc) {
          if (strcmp(*argv, "-port") == 0) {
               argc--; argv++;
               if (!argc) usage();
               port = atoi(*argv);
           } else if (strcmp(*argv, "-mech") == 0) {
               argc--; argv++;
               if (!argc) usage();
               mechanism = *argv;
           } else if (strcmp(*argv, "-d") == 0) {
               deleg_flag = GSS_C_DELEG_FLAG;
```

(続く)

```
          } else if (strcmp(*argv, "-f") == 0) {
               use_file = 1;
          } else
                break;
          argc--; argv++;
     }
     if (argc != 3)
          usage();

     if (argc > 1) {
                strcpy(hostname, argv[0]);
        } else if (gethostname(hostname, sizeof(hostname)) == -1) {
                       perror("gethostname");
                       exit(1);
        }


     if (argc > 2) {
        strcpy(service_name, argv[1]);
        strcat(service_name, "@");
        strcat(service_name, hostname);

     }

      msg = argv[2];

     if (mechanism)
         parse_oid(mechanism, &g_mechOid);

     if (call_server(hostname, port, g_mechOid, service_name,
                  deleg_flag, msg, use_file) < 0)
          exit(1);

     if (g_mechOid != GSS_C_NULL_OID)
         (void) gss_release_oid(&min_stat, &gmechOid);

     return 0;
}
```

parse_oid()

GSS-API が処理できるように、(指定されていれば) コマンド行で指定されたセキュリティ機構名を OID に変換します。

注意 - このようなサンプルを示していますが、可能な限り、独自の機構を指定するのではなく、GSS-API 実装が提供するデフォルトの機構を使用することを強く推奨します。デフォルトの機構を取得するには、機構 OID 値を `GSS_C_NULL_OID` に設定します。また、`gss_str_to_oid()` 関数はすべての GSS-API 実装でサポートされているわけではありません。

例 **A–3** `parse_oid()`

```
static void parse_oid(char *mechanism, gss_OID *oid)
{
    char        *mechstr = 0, *cp;
    gss_buffer_desc tok;
    OM_uint32 maj_stat, min_stat;

    if (isdigit(mechanism[0])) {
        mechstr = malloc(strlen(mechanism)+5);
        if (!mechstr) {
            printf("Couldn't allocate mechanism scratch!\n");
            return;
        }
        sprintf(mechstr, "{ %s }", mechanism);
        for (cp = mechstr; *cp; cp++)
            if (*cp == '.')
                *cp = ' ';
        tok.value = mechstr;
    } else
        tok.value = mechanism;
    tok.length = strlen(tok.value);
    maj_stat = gss_str_to_oid(&min_stat, &tok, oid);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("str_to_oid", maj_stat, min_stat);
        return;
    }
    if (mechstr)
        free(mechstr);
}
```

call_server()

`call_server()` はプログラムの中心部分です。

例 **A–4**　call_server()

```
/*
 * 関数: call_server
 *
 * 目的: 「署名」サービスを呼び出す
 *
 * 引数:
 *
 *      host            (r) サービスを提供するホスト
 *      port            (r) ホストに接続するためのポート
 *      service_name    (r) 認証する GSS-API サービス名
 *      msg             (r) 「署名」されたメッセージ
 *
 * 戻り値: 成功した場合は 0、失敗した場合は -1
 *
 * 効果:
 *
 * call_server は <host:port> への TCP 接続を開き、その接続上で
 * service_name との GSS-API コンテキストを確立する。次に、gss_wrap で
 * msg を GSS-API トークンにラップし、サーバーに送信し、サーバーから
 * 戻された msg の GSS-API の署名ブロックを読み取り、gss_verify で検証する。
 * どこかで失敗した場合は -1 を戻し、そうでない場合は 0 を戻す。
 */
int call_server(host, port, oid, service_name, deleg_flag, msg, use_file)
     char *host;
     u_short port;
     gss_OID oid;
     char *service_name;
     OM_uint32 deleg_flag;
     char *msg;
     int use_file;
{
     gss_ctx_id_t context;
     gss_buffer_desc in_buf, out_buf, context_token;
     int s, state;
     OM_uint32 ret_flags;
     OM_uint32 maj_stat, min_stat;
     gss_name_t          src_name, targ_name;
     gss_buffer_desc     sname, tname;
     OM_uint32           lifetime;
     gss_OID             mechanism, name_type;
     int                 is_local;
     OM_uint32           context_flags;
     int                 is_open;
     gss_qop_t           qop_state;
     gss_OID_set         mech_names;
     gss_buffer_desc     oid_name;
     int         i;
     int conf_req_flag = 0;
     int req_output_size = 1012;
     OM_uint32 max_input_size = 0;
        char *mechStr;

/* 接続を開く */
     if ((s = connect_to_server(host, port)) < 0)
```

(続く)

```
          return -1;

     /* 接続を確立する */
     if (client_establish_context(s, service_name, deleg_flag, oid, &context,
                                  &ret_flags) < 0) {
          (void) close(s);
          return -1;
     }

 /* 保存後、コンテキストを復元する */
     maj_stat = gss_export_sec_context(&min_stat,
                                       &context,
                                       &context_token);
     if (maj_stat != GSS_S_COMPLETE) {
             display_status("exporting context", maj_stat, min_stat);
             return -1;
     }
     maj_stat = gss_import_sec_context(&min_stat,
                                       &context_token,
                                       &context);
     if (maj_stat != GSS_S_COMPLETE) {
        display_status("importing context", maj_stat, min_stat);
        return -1;
     }
     (void) gss_release_buffer(&min_stat, &context_token);

     /* フラグを表示する */
     display_ctx_flags(ret_flags);

     /* コンテキスト情報を取得する */
     maj_stat = gss_inquire_context(&min_stat, context,
                                    &src_name, &targ_name, &lifetime,
                                    &mechanism, &context_flags,
                                    &is_local,
                                    &is_open);
     if (maj_stat != GSS_S_COMPLETE) {
         display_status("inquiring context", maj_stat, min_stat);
         return -1;
     }

     if (maj_stat == GSS_S_CONTEXT_EXPIRED) {
     printf(" context expired\n");
         display_status("Context is expired", maj_stat, min_stat);
         return -1;
     }

/* gss_wrap_size_limit をテストする */
     maj_stat = gss_wrap_size_limit(&min_stat, context,
                                 conf_req_flag,
                                 GSS_C_QOP_DEFAULT,
                                 req_output_size,
                                 &max_input_size
                                 );
     if (maj_stat != GSS_S_COMPLETE) {
```

(続く)

```
        display_status("wrap_size_limit call", maj_stat, min_stat);
    } else
        fprintf (stderr, "gss_wrap_size_limit returned "
            "max input size = %d \n"
            "for req_output_size = %d with Integrity only\n",
            max_input_size , req_output_size , conf_req_flag);

    conf_req_flag = 1;
    maj_stat = gss_wrap_size_limit(&min_stat, context,
                                   conf_req_flag,
                                   GSS_C_QOP_DEFAULT,
                                   req_output_size,
                                   &max_input_size
                                   );
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("wrap_size_limit call", maj_stat, min_stat);
    } else
        fprintf (stderr, "gss_wrap_size_limit returned "
            " max input size = %d \n"
            "for req_output_size = %d with "
            "Integrity & Privacy \n",
            max_input_size , req_output_size );

    maj_stat = gss_display_name(&min_stat, src_name, &sname,
                                &name_type);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("displaying source name", maj_stat, min_stat);
        return -1;
    }
    maj_stat = gss_display_name(&min_stat, targ_name, &tname,
                                (gss_OID *) NULL);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("displaying target name", maj_stat, min_stat);
        return -1;
    }
    fprintf(stderr, "\"%.*s\" to \"%.*s\", lifetime %u, flags %x, %s, %s\n",
            (int) sname.length, (char *) sname.value,
            (int) tname.length, (char *) tname.value, lifetime,
            context_flags,
            (is_local) ? "locally initiated" : "remotely initiated",
            (is_open) ? "open" : "closed");

    (void) gss_release_name(&min_stat, &src_name);
    (void) gss_release_name(&min_stat, &targ_name);
    (void) gss_release_buffer(&min_stat, &sname);
    (void) gss_release_buffer(&min_stat, &tname);


    maj_stat = gss_oid_to_str(&min_stat,
                              name_type,
                              &oid_name);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("converting oid->string", maj_stat, min_stat);
        return -1;
```

(続く)

```
    }
    fprintf(stderr, "Name type of source name is %.*s.\n",
            (int) oid_name.length, (char *) oid_name.value);
    (void) gss_release_buffer(&min_stat, &oid_name);

    /* この機構によってサポートされる名前を取得する */
    maj_stat = gss_inquire_names_for_mech(&min_stat,
                                          mechanism,
                                          &mech_names);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("inquiring mech names", maj_stat, min_stat);
        return -1;
    }

    maj_stat = gss_oid_to_str(&min_stat,
                              mechanism,
                              &oid_name);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("converting oid->string", maj_stat, min_stat);
        return -1;
    }
       mechStr = (char *)__gss_oid_to_mech(mechanism);
    fprintf(stderr, "Mechanism %.*s (%s) supports %d names\n",
               (int) oid_name.length, (char *) oid_name.value,
               (mechStr == NULL ? "NULL" : mechStr),
               mech_names->count);
    (void) gss_release_buffer(&min_stat, &oid_name);

    for (i=0; i < mech_names->count; i++) {
        maj_stat = gss_oid_to_str(&min_stat,
                                  &mech_names->elements[i],
                                  &oid_name);
        if (maj_stat != GSS_S_COMPLETE) {
            display_status("converting oid->string", maj_stat, min_stat);
            return -1;
        }
        fprintf(stderr, "  %d: %.*s\n", i,
                (int) oid_name.length, (char *) oid_name.value);

        (void) gss_release_buffer(&min_stat, &oid_name);
    }
    (void) gss_release_oid_set(&min_stat, &mech_names);

    if (use_file) {
        read_file(msg, &in_buf);
    } else {
        /* メッセージ内容を構造体へ設定完了 */
        in_buf.value = msg;
        in_buf.length = strlen(msg) + 1;
    }

    if (ret_flag & GSS_C_CONF_FLAG) {
         state = 1;
    else
```

(続く)

```
        state = 0;
    }

    maj_stat = gss_wrap(&min_stat, context, 1, GSS_C_QOP_DEFAULT,
                        &in_buf, &state, &out_buf);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("wrapping message", maj_stat, min_stat);
        (void) close(s);
        (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
        return -1;
    } else if (! state) {
        fprintf(stderr, "Warning!  Message not encrypted.\n");
    }

    /* サーバーに送信する */
    if (send_token(s, &out_buf) < 0) {
        (void) close(s);
        (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
        return -1;
    }
    (void) gss_release_buffer(&min_stat, &out_buf);

    /* 署名ブロックを out_buf に読み取る */
    if (recv_token(s, &out_buf) < 0) {
        (void) close(s);
        (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
        return -1;
    }

    /* 署名ブロックを検証する */
    maj_stat = gss_verify_mic(&min_stat, context, &in_buf,
                              &out_buf, &qop_state);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("verifying signature", maj_stat, min_stat);
        (void) close(s);
        (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
        return -1;
    }
    (void) gss_release_buffer(&min_stat, &out_buf);

    if (use_file)
        free(in_buf.value);

    printf("Signature verified.\n");

    /* コンテキストを削除する */
    maj_stat = gss_delete_sec_context(&min_stat, &context, &out_buf);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("deleting context", maj_stat, min_stat);
        (void) close(s);
        (void) gss_delete_sec_context(&min_stat, &context, GSS_C_NO_BUFFER);
        return -1;
    }
```

(続く)

```
     (void) gss_release_buffer(&min_stat, &out_buf);
     (void) close(s);
     return 0;
}
```

read_file()

転送されるメッセージがファイルに格納されている場合、read_file() 関数は (call_server() から呼び出されて) ファイルを開いて読み取ります。

例 **A–5** read_file()

```
void read_file(file_name, in_buf)
    char                *file_name;
    gss_buffer_t        in_buf;
{
    int fd, bytes_in, count;
    struct stat stat_buf;

    if ((fd = open(file_name, O_RDONLY, 0)) < 0) {
        perror("open");
        fprintf(stderr, "Couldn't open file %s\n", file_name);
        exit(1);
    }
    if (fstat(fd, &stat_buf) < 0) {
        perror("fstat");
        exit(1);
    }
    in_buf->length = stat_buf.st_size;
    in_buf->value = malloc(in_buf->length);
    if (in_buf->value == 0) {
        fprintf(stderr, "Couldn't allocate %ld byte buffer for reading file\n",
                in_buf->length);
        exit(1);
    }
    memset(in_buf->value, 0, in_buf->length);
    for (bytes_in = 0; bytes_in < in_buf->length; bytes_in += count) {
        count = read(fd, in_buf->value, (OM_uint32)in_buf->length);
        if (count < 0) {
            perror("read");
            exit(1);
        }
        if (count == 0)
            break;
    }
    if (bytes_in != count)
        fprintf(stderr, "Warning, only read in %d bytes, expected %d\n",
```

(続く)

```
                bytes_in, count);
}
```

client_establish_context()

client_establish_context() は gss_init_sec_context() を呼び出して、サーバーとのコンテキストを確立します。

例 **A–6** client_establish_context()

```
/*
 * 関数: client_establish_context
 *
 * 目的: 指定されたサービスとの GSS-API コンテキストを確立し、
 * コンテキストハンドルを戻す
 *
 * 引数:
 *
 *      s               (r) サーバーとの間で確立された TCP 接続
 *      service_name    (r) サービスの ASCII 名
 *      context         (w) 確立された GSS-API コンテキスト
 *      ret_flags       (w) init_sec_context から戻されたフラグ
 *
 * 戻り値: 成功した場合は 0、失敗した場合は -1
 *
 * 効果:
 *
 * service_name が GSS-API 名としてインポートされ、
 * 対応するサービスとの間で GSS-API コンテキストが確立される。
 * サービスは TCP 接続 s 上で応答待ちする。デフォルトの
 * GSS-API 機構が使用され、相互認証とリプレイの検出が
 * 要求される。
 *
 * 成功した場合、コンテキストハンドルが context に戻される。
 * 失敗した場合、GSS-API エラーメッセージが stderr に表示され、
 * -1 が戻される。
 */
     int client_establish_context(s, service_name, deleg_flag, oid,
                                  gss_context, ret_flags)
     int s;
     char *service_name;
     gss_OID oid;
     OM_uint32 deleg_flag;
     gss_ctx_id_t *gss_context;
     OM_uint32 *ret_flags;
{
     gss_buffer_desc send_tok, recv_tok, *token_ptr;
     gss_name_t target_name;
```

(続く)

```
OM_uint32 maj_stat, min_stat;

/*
 * 名前を target_name にインポートする。send_tok で
 * ローカル変数空間を保存する。
 */

send_tok.value = service_name;
send_tok.length = strlen(service_name) + 1;
maj_stat = gss_import_name(&min_stat, &send_tok,
                  (gss_OID) GSS_C_NT_HOSTBASED_SERVICE, &target_name);
if (maj_stat != GSS_S_COMPLETE) {
     display_status("parsing name", maj_stat, min_stat);
     return -1;
}


/*
 * コンテキスト確立ループを実行する。
 *
 * ループを通過するごとに、token_ptr はサーバーに送信される
 * トークンに設定される (最初に通過するときは GSS_C_NO_BUFFER)。
 * 生成された各トークンは send_tok に格納され、サーバーに送信される。
 * 受信された各トークンは recv_tok に格納され、token_ptr は次の
 * gss_init_sec_context の呼び出しで処理されるトークンに
 * 設定される。
 *
 * GSS-API は、以下の 2 つのことを保証する。
 * 1.サーバーがクライアントからこれ以上トークンを期待していない場合のみ、
 *   send_tok の length の値が 0 になる。
 * 2.サーバーにクライアントへ送るトークンが存在する場合のみ、
 *   gss_init_sec_context が GSS_S_CONTINUE_NEEDED を戻す。
 */

token_ptr = GSS_C_NO_BUFFER;
*gss_context = GSS_C_NO_CONTEXT;

do {
     maj_stat =
          gss_init_sec_context(&min_stat,
                               GSS_C_NO_CREDENTIAL,
                               gss_context,
                               target_name,
                               oid,
                               GSS_C_MUTUAL_FLAG | GSS_C_REPLAY_FLAG |
                                                   deleg_flag,
                               0,
                               NULL,       /* チャネルバインディングなし */
                               token_ptr,
                               NULL,       /* 機構の型を無視する */
                               &send_tok,
                               ret_flags,
                               NULL);      /* time_rec を無視する */
     if (gss_context == NULL){
```

(続く)

```
                printf("Cannot create context\n");
                return GSS_S_NO_CONTEXT;
           }
           if (token_ptr != GSS_C_NO_BUFFER)
                (void) gss_release_buffer(&min_stat, &recv_tok);
           if (maj_stat!=GSS_S_COMPLETE && maj_stat!=GSS_S_CONTINUE_NEEDED) {
                display_status("initializing context", maj_stat, min_stat);
                (void) gss_release_name(&min_stat, &target_name);
                return -1;
           }

           if (send_tok.length != 0) {
                fprintf(stdout, "Sending init_sec_context token (size=%ld)...",
                        send_tok.length);
                if (send_token(s, &send_tok) < 0) {
                     (void) gss_release_buffer(&min_stat, &send_tok);
                     (void) gss_release_name(&min_stat, &target_name);
                     return -1;
                }
           }
           (void) gss_release_buffer(&min_stat, &send_tok);

           if (maj_stat == GSS_S_CONTINUE_NEEDED) {
                fprintf(stdout, "continue needed...");
                if (recv_token(s, &recv_tok) < 0) {
                     (void) gss_release_name(&min_stat, &target_name);
                     return -1;
                }
                token_ptr = &recv_tok;
           }
           printf("\n");
      } while (maj_stat == GSS_S_CONTINUE_NEEDED);

      (void) gss_release_name(&min_stat, &target_name);
      return 0;
}
```

connect_to_server()

connect_to_server() は TCP 接続を作成するだけの基本的な関数です。

例 A–7 connect_to_server()

```
/*
 * 関数: connect_to_server
 *
 * 目的: 指定されたホストとポートへの TCP 接続を開く
 *
 * 引数:
```

(続く)

```
 *
 *      host            (r) ターゲットのホスト名
 *      port            (r) ターゲットのポート。ホストのバイト順。
 *
 * 戻り値: 成功した場合は、確立されたソケットのファイル記述子。
 * 失敗した場合は -1。
 *
 * 効果:
 *
 * ホスト名が gethostbyname() で解釈処理され、ソケットが開かれ、
 * 接続される。エラーが発生した場合、エラーメッセージが表示され、
 * -1 が戻される。
 */
int connect_to_server(host, port)
     char *host;
     u_short port;
{
     struct sockaddr_in saddr;
     struct hostent *hp;
     int s;

     if ((hp = gethostbyname(host)) == NULL) {
          fprintf(stderr, "Unknown host: %s\n", host);
          return -1;
     }

     saddr.sin_family = hp->h_addrtype;
     memcpy((char *)&saddr.sin_addr, hp->h_addr, sizeof(saddr.sin_addr));
     saddr.sin_port = htons(port);

     if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
          perror("creating socket");
          return -1;
     }
     if (connect(s, (struct sockaddr *)&saddr, sizeof(saddr)) < 0) {
          perror("connecting to server");
          (void) close(s);
          return -1;
     }

     return s;
}
```

サーバー側アプリケーション

この節では、前述のクライアント関数からメッセージを受信するサーバー側アプリケーションについて説明します。

プログラムヘッダー

プログラムヘッダーはサーバープログラムの宣言部分です。また、コマンド行が間違っていた場合に構文を説明する関数もあります。ここでは、GSS-API が提供するデフォルトのセキュリティ機構を使用するように設定されています。

例 A–8 プログラムヘッダー

```
/*
 * Copyright 1994 by OpenVision Technologies, Inc.
 *
 * Permission to use, copy, modify, distribute, and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appears in all copies and
 * that both that copyright notice and this permission notice appear in
 * supporting documentation, and that the name of OpenVision not be used
 * in advertising or publicity pertaining to distribution of the software
 * without specific, written prior permission. OpenVision makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 * OPENVISION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
 * EVENT SHALL OPENVISION BE LIABLE FOR ANY SPECIAL, INDIRECT OR
 * CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
 * USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
 * OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
 * PERFORMANCE OF THIS SOFTWARE.
 */

#if !defined(lint) && !defined(__CODECENTER__)
static char *rcsid = "$Header: /afs/athena.mit.edu/astaff/project/krbdev/.cvsroot
/src/appl/gss-sample/gss-server.c,v 1.17 1996/10/22 00:07:59 tytso Exp $";
#endif

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif
#include <stdlib.h>
#include <ctype.h>

#include <gssapi/gssapi.h>
#include <gssapi/gssapi_ext.h>
#include "gss-misc.h"

#ifdef USE_STRING_H
#include <string.h>
#else
#include <strings.h>
#endif
```

(続く)

```
/* 資格の獲得と状態の表示で使用される大域的な機構 OID */
gss_OID g_mechOid = GSS_C_NULL_OID;

void usage()
{
     fprintf(stderr, "Usage: gss-server [-port port] [-verbose]\n");
     fprintf(stderr, "       [-inetd] [-logfile file]");
     fprintf(stderr, " [-mech mechoid] [service_name]\n");
     exit(1);
}

FILE *log;

int verbose = 0;
```

main()

main() はプログラムのエントリポイントです。次に、コマンド行構文を示します。

```
gss-server [–port port] [–d] [–mech mech] host service msg
```

コマンド行を解析した後、main() は (指定されていれば) 希望のセキュリティ機構名を OID に変換し、資格を獲得し、コンテキストを確立し、データを受信し、そして、必要であれば機構 OID を破棄します。

注 - 通常、アプリケーションは機構を設定せずに、GSS-API が提供するデフォルトを使用するべきです。

例 **A–9** main()

```
int
main(argc, argv)
     int argc;
     char **argv;
{
     char *service_name, *mechType = NULL;
     gss_cred_id_t server_creds;
     OM_uint32 min_stat;
     u_short port = 4444;
     int s;
     int once = 0;
     int do_inetd = 0;
```

(続く)

```
log = stdout;
display_file = stdout;
argc--; argv++;
while (argc) {
     if (strcmp(*argv, "-port") == 0) {
          argc--; argv++;
          if (!argc) usage();
          port = atoi(*argv);
     } else if (strcmp(*argv, "-verbose") == 0) {
         verbose = 1;
     } else if (strcmp(*argv, "-once") == 0) {
         once = 1;
     } else if (strcmp(*argv, "-inetd") == 0) {
         do_inetd = 1;
     } else if (strcmp(*argv, "-mech") == 0) {
           argc--; argv++;
           if (!argc)      usage();
           mechType = *argv;
     } else if (strcmp(*argv, "-logfile") == 0) {
         argc--; argv++;
         if (!argc) usage();
         log = fopen(*argv, "a");
         display_file = log;
         if (!log) {
             perror(*argv);
             exit(1);
         }
     } else
          break;
     argc--; argv++;
}
if (argc != 1)
     usage();

if ((*argv)[0] == '-')
     usage();

service_name = *argv;

if (mechType != NULL) {
        if ((g_mechOid = createMechOid(mechType)) == NULL) {
                usage();
                exit(-1);
        }
}

if (server_acquire_creds(service_name, g_mechOid, &server_creds) < 0)
    return -1;

if (do_inetd) {
    close(1);
    close(2);
```

(続く)

```
        sign_server(0, server_creds);
        close(0);
    } else {
        int stmp;

        if ((stmp = create_socket(port))) {
            do {
                /* TCP 接続を受け入れる */
                if ((s = accept(stmp, NULL, 0)) < 0) {
                    perror("accepting connection");
                } else {
                    /* 失敗したとしても有効な対処はここではできないため、
                        この戻り値は検査されない */
                    sign_server(s, server_creds);
                }
            } while (!once);
        }

        close(stmp);
    }

    (void) gss_release_cred(&min_stat, &server_creds);
    if (g_mechOid != GSS_C_NULL_OID)
            gss_release_oid(&min_stat, &g_mechOid);

    /*NOTREACHED*/
    (void) close(s);
    return 0;
}
```

createMechOid()

createMechOid() は、プログラムの完全性のためだけに示しています。通常はデフォルトの機構を使用するべきです (GSS_C_NULL_OID で指定)。

例 **A–10** createMechOid()

```
gss_OID createMechOid(const char *mechStr)
{
        gss_buffer_desc mechDesc;
        gss_OID mechOid;
        OM_uint32 minor;

        if (mechStr == NULL)
                return (GSS_C_NULL_OID);

        mechDesc.length = strlen(mechStr);
```

(続く)

```
        mechDesc.value = (void *) mechStr;

        if (gss_str_to_oid(&minor, &mechDesc, &mechOid) !
= GSS_S_COMPLETE) {
                fprintf(stderr, "Invalid mechanism oid specified <%s>",
                                mechStr);
                return (GSS_C_NULL_OID);
        }

        return (mechOid);
}
```

server_acquire_creds()

server_acquire_creds() は要求されたネットワークサービスの資格を取得します。

例 **A–11** server_acquire_creds()

```
/*
 * 関数: server_acquire_creds
 *
 * 目的: サービス名をインポートし、その資格を獲得する
 *
 * 引数:
 *
 *      service_name    (r) サービスの ASCII 名
        mechType        (r) 使用される機構の型
 *      server_creds    (w) GSS-API サービスの資格
 *
 * 戻り値: 成功した場合は 0、失敗した場合は -1
 *
 * 効果:
 *
 * サービス名が gss_import_name でインポートされ、サービスの資格が
 * gss_acquire_cred で獲得される。どちらかの操作が失敗した場合、
 * エラーメッセージが表示され、-1 が戻される。そうでない場合、0 が戻される。
 */
int server_acquire_creds(service_name, mechOid, server_creds)
     char *service_name;
     gss_OID mechOid;
     gss_cred_id_t *server_creds;
{
     gss_buffer_desc name_buf;
     gss_name_t server_name;
     OM_uint32 maj_stat, min_stat;
     gss_OID_set_desc mechOidSet;
     gss_OID_set desiredMechs = GSS_C_NULL_OID_SET;
```

(続く)

```
     if (mechOid != GSS_C_NULL_OID) {
               desiredMechs = &mechOidSet;
               mechOidSet.count = 1;
               mechOidSet.elements = mechOid;
     } else
               desiredMechs = GSS_C_NULL_OID_SET;


     name_buf.value = service_name;
     name_buf.length = strlen(name_buf.value) + 1;
     maj_stat = gss_import_name(&min_stat, &name_buf,
               (gss_OID) GSS_C_NT_HOSTBASED_SERVICE, &server_name);
     if (maj_stat != GSS_S_COMPLETE) {
          display_status("importing name", maj_stat, min_stat);
          if (mechOid != GSS_C_NO_OID)
               gss_release_oid(&min_stat, &mechOid);
          return -1;
     }

     maj_stat = gss_acquire_cred(&min_stat, server_name, 0,
                                 desiredMechs, GSS_C_ACCEPT,
                                 server_creds, NULL, NULL);

     if (maj_stat != GSS_S_COMPLETE) {
          display_status("acquiring credentials", maj_stat, min_stat);
          return -1;
     }

     (void) gss_release_name(&min_stat, &server_name);

     return 0;
}
```

sign_server()

sign_server() はプログラムの中心部分です。server_establish_context() を呼び出してコンテキストを受け入れ、データを受信、ラップ解除、および検証し、そしてクライアントに返送する MIC を生成します。最後に、コンテキストを削除します。

例 A–12 sign_server()

```
/*
 * 関数: sign_server
 *
 * 目的: 「署名」サービスを実行する
 *
 * 引数:
```

```
 *
 *      s               (r) 接続が受け入れられた TCP ソケット
 *      service_name    (r) コンテキストを確立する GSS-API サービスの ASCII 名
 *
 * 戻り値: エラーが発生した場合は -1
 *
 * 効果:
 *
 * sign_server はコンテキストを確立し、単一の署名要求を実行する
 *
 * 署名要求は単一の GSS-AP ラップ済みトークンである。まず、
 * このトークンがラップ解除される。次に、署名ブロックが
 * gss_get_mic で生成され、送信側に戻される。コンテキストが
 * 破棄され、接続が閉じられる。
 *
 * エラーが発生した場合、-1 が戻される。
 */
int sign_server(s, server_creds)
     int s;
     gss_cred_id_t server_creds;
{
     gss_buffer_desc client_name, xmit_buf, msg_buf;
     gss_ctx_id_t context;
     OM_uint32 maj_stat, min_stat;
     int i, conf_state, ret_flags;
     char       *cp;

     /* クライアントとのコンテキストを確立する */
     if (server_establish_context(s, server_creds, &context,
                                  &client_name, &ret_flags) < 0)
        return(-1);

     printf("Accepted connection: \"%.*s\"\n",
            (int) client_name.length, (char *) client_name.value);
     (void) gss_release_buffer(&min_stat, &client_name);

     for (i=0; i < 3; i++)
             if (test_import_export_context(&context))
                     return -1;

     /* ラップ済みメッセージトークンを受信する */
     if (recv_token(s, &xmit_buf) < 0)
        return(-1);

     if (verbose && log) {
        fprintf(log, "Wrapped message token:\n");
        print_token(&xmit_buf);
     }

     maj_stat = gss_unwrap(&min_stat, context, &xmit_buf, &msg_buf,
                           &conf_state, (gss_qop_t *) NULL);
     if (maj_stat != GSS_S_COMPLETE) {
        display_status("unwrapping message", maj_stat, min_stat);
```

(続く)

```
        return(-1);
    } else if (! conf_state) {
        fprintf(stderr, "Warning!  Message not encrypted.\n");
    }

    (void) gss_release_buffer(&min_stat, &xmit_buf);

    fprintf(log, "Received message: ");
    cp = msg_buf.value;
    if (isprint(cp[0]) && isprint(cp[1]))
        fprintf(log, "\"%s\"\n", cp);
    else {
        printf("\n");
        print_token(&msg_buf);
    }

    /* メッセージの署名ブロックを生成する */
    maj_stat = gss_get_mic(&min_stat, context, GSS_C_QOP_DEFAULT,
                           &msg_buf, &xmit_buf);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("signing message", maj_stat, min_stat);
        return(-1);
    }

    (void) gss_release_buffer(&min_stat, &msg_buf);

    /* 署名ブロックをクライアントに送信する */
    if (send_token(s, &xmit_buf) < 0)
        return(-1);

    (void) gss_release_buffer(&min_stat, &xmit_buf);

    /* コンテキストを削除する */
    maj_stat = gss_delete_sec_context(&min_stat, &context, NULL);
    if (maj_stat != GSS_S_COMPLETE) {
        display_status("deleting context", maj_stat, min_stat);
        return(-1);
    }

    fflush(log);

    return(0);
}
```

server_establish_context()

server_establish_context() はコンテキスト確立ループの一部として gss_accept_sec_context() を呼び出します。

例 **A–13** server_establish_context()

```
/*
 * 関数: server_establish_context
 *
 * 目的: 着信クライアントで指定されたサービスとして
 * GSS-API コンテキストを確立し、コンテキストハンドルと
 * 関連するクライアント名を戻す。
 *
 * 引数:
 *
 *      s               (r) クライアントとの間で確立された TCP 接続
 *      service_creds   (r) gss_acquire_cred で取得したサーバーの資格
 *      context         (w) 確立された GSS-API コンテキスト
 *      client_name     (w) クライアントの ASCII 名
 *
 * 戻り値: 成功した場合は 0、失敗した場合は -1
 *
 * 効果:
 *
 * 有効なクライアント要求はすべて受け入れられる。コンテキストが
 * 確立された場合、そのハンドルが context に戻され、クライアント名が
 * client_name に戻され、0 が戻される。失敗した場合、エラーメッセージが
 * 表示され、-1 が戻される。
 */
int server_establish_context(s, server_creds, context, client_name, ret_flags)
     int s;
     gss_cred_id_t server_creds;
     gss_ctx_id_t *context;
     gss_buffer_t client_name;
     OM_uint32 *ret_flags;
{
     gss_buffer_desc send_tok, recv_tok;
     gss_name_t client;
     gss_OID doid;
     OM_uint32 maj_stat, min_stat;
     gss_buffer_desc    oid_name;
        char *mechStr;

     *context = GSS_C_NO_CONTEXT;

     do {
          if (recv_token(s, &recv_tok) < 0)
               return -1;

          if (verbose && log) {
              fprintf(log, "Received token (size=%d): \n", recv_tok.length);
              print_token(&recv_tok);
          }

          maj_stat =
               gss_accept_sec_context(&min_stat,
                                      context,
                                      server_creds,
                                      &recv_tok,
                                      GSS_C_NO_CHANNEL_BINDINGS,
                                      &client,
```

(続く)

```
                                    &doid,
                                    &send_tok,
                                    ret_flags,
                                    NULL,     /* time_rec を無視する */
                                    NULL);    /* del_cred_handle を無視する */

          if (maj_stat!=GSS_S_COMPLETE && maj_stat!=GSS_S_CONTINUE_NEEDED) {
               display_status("accepting context", maj_stat, min_stat);
               (void) gss_release_buffer(&min_stat, &recv_tok);
               return -1;
          }

          (void) gss_release_buffer(&min_stat, &recv_tok);

          if (send_tok.length != 0) {
              if (verbose && log) {
                  fprintf(log,
                          "Sending accept_sec_context token (size=%d):\n",
                          send_tok.length);
                  print_token(&send_tok);
              }
               if (send_token(s, &send_tok) < 0) {
                    fprintf(log, "failure sending token\n");
                    return -1;
               }

               (void) gss_release_buffer(&min_stat, &send_tok);
          }
          if (verbose && log) {
              if (maj_stat == GSS_S_CONTINUE_NEEDED)
                  fprintf(log, "continue needed...\n");
              else
                  fprintf(log, "\n");
              fflush(log);
          }
     } while (maj_stat == GSS_S_CONTINUE_NEEDED);

     /* フラグを表示する */
     display_ctx_flags(*ret_flags);

     if (verbose && log) {
         maj_stat = gss_oid_to_str(&min_stat, doid, &oid_name);
         if (maj_stat != GSS_S_COMPLETE) {
             display_status("converting oid->string", maj_stat, min_stat);
             return -1;
         }
                mechStr = (char *)__gss_oid_to_mech(doid);
         fprintf(log, "Accepted connection using mechanism OID %.*s (%s).\n",
                 (int) oid_name.length, (char *) oid_name.value,
                       (mechStr == NULL ? "NULL" : mechStr));
         (void) gss_release_buffer(&min_stat, &oid_name);
     }

     maj_stat = gss_display_name(&min_stat, client, client_name, &doid);
```

(続く)

```
     if (maj_stat != GSS_S_COMPLETE) {
          display_status("displaying name", maj_stat, min_stat);
          return -1;
     }
     return 0;
}
```

create_a_socket()

create_a_socket() はクライアントとの転送接続を作成するだけの関数です。

例 **A–14** create_a_socket()

```
/*
 * 関数: create_socket
 *
 * 目的: 応答待ちする TCP ソケットを開く
 *
 * 引数:
 *
 *      port            (r) 応答待ちするポート番号
 *
 * 戻り値: 成功した場合は、応答待ちするソケットのファイル記述子。
 * 失敗した場合は -1。
 *
 * 効果:
 *
 * 指定されたポート上で応答待ちするソケットが作成され、そのファイル記述子が
 * 戻される。エラーが発生した場合、エラーメッセージが表示され、-1 が戻される。
 */
int create_socket(port)
    u_short port;
{
    struct sockaddr_in saddr;
    int s;
    int on = 1;

    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(port);
    saddr.sin_addr.s_addr = INADDR_ANY;

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
          perror("creating socket");
          return -1;
    }
    /* ソケットを再使用できるようにする */
    (void) setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)&on, sizeof(on));
    if (bind(s, (struct sockaddr *) &saddr, sizeof(saddr)) < 0)
{
```

(続く)

```
            perror("binding socket");
            (void) close(s);
            return -1;
    }
    if (listen(s, 5) < 0) {
            perror("listening on socket");
            (void) close(s);
            return -1;
    }
    return s;
}
```

test_import_export_context()

最後に、test_import_export_context() は gss_export_sec_context() と gss_import_sec_context() がどのように機能するかを示す小さな関数です。この関数は実用性はなく、ここでは上記 2 つの GSS-API 関数がどのように使用されるかだけを示しています。

例 A–15 test_import_export_context()

```
int test_import_export_context(context)
      gss_ctx_id_t *context;
{
      OM_uint32       min_stat, maj_stat;
      gss_buffer_desc context_token, copied_token;
      struct timeval tm1, tm2;

      /*
       * 保存後、コンテキストを復元する
       */
      gettimeofday(&tm1, (struct timezone *)0);
      maj_stat = gss_export_sec_context(&min_stat, context, &context_token);
      if (maj_stat != GSS_S_COMPLETE) {
            display_status("exporting context", maj_stat, min_stat);
            return 1;
      }
      gettimeofday(&tm2, (struct timezone *)0);
      if (verbose && log)
            fprintf(log, "Exported context: %d bytes, %7.4f seconds\n",
                  context_token.length, timeval_subtract(&tm2, &tm1));
      copied_token.length = context_token.length;
      copied_token.value = malloc(context_token.length);
      if (copied_token.value == 0) {
          fprintf(log, "Couldn't allocate memory to copy context token.\n");
          return 1;
      }
```

(続く)

```
        memcpy(copied_token.value, context_token.value, copied_token.length);
        maj_stat = gss_import_sec_context(&min_stat, &copied_token, context);
        if (maj_stat != GSS_S_COMPLETE) {
                display_status("importing context", maj_stat, min_stat);
                return 1;
        }
        gettimeofday(&tm1, (struct timezone *)0);
        if (verbose && log)
                fprintf(log, "Importing context: %7.4f seconds\n",
                        timeval_subtract(&tm1, &tm2));
        (void) gss_release_buffer(&min_stat, &context_token);
        return 0;
}
```

timeval_subtract()

timeval_subtract() は test_import_export_context() が使用する簡便な関数です。

例 **A–16** timeval_subtract()

```
static float timeval_subtract(tv1, tv2)
        struct timeval *tv1, *tv2;
{
        return ((tv1->tv_sec - tv2->tv_sec) +
                ((float) (tv1->tv_usec - tv2->tv_usec)) / 1000000);
}
```

補助的な関数

クライアントプログラムとサーバープログラムが期待どおりに機能するには、他にもいくつかの関数が必要です。このような関数はほとんどが値を表示したりするもので、プログラムの基本機能には必ずしも必要ではありません。ここではプログラムの完全性のためだけに示しています。

ただし、2 つの関数 send_token() と recv_token() は重要です。この 2 つの関数はコンテキストのトークンとメッセージを実際に転送します。この 2 つの関数は純粋に基本的な関数であり、ファイル記述子を開いて読み書きします。普通でしかも GSS-API には直接関係はありませんが、別に説明するだけの価値はあります。

さまざまなサポート関数

次に、さまざまなサポート関数について説明します。

- `display_status()` — 最後に呼び出した GSS-API 関数から戻された状態を表示します。
- `write_all()` — バッファをファイルに書き込みます。
- `read_all()` — ファイルからバッファに読み取ります。
- `display_ctx_flags()` — 現在のコンテキストについての情報を人が読める形式で表示します。たとえば、機密性または相互認証が許可されているかどうかなどです。
- `print_token()` — トークンの値を出力します。

例 A–17

```
/*
 * Copyright 1994 by OpenVision Technologies, Inc.
 *
 * Permission to use, copy, modify, distribute, and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appears in all copies and
 * that both that copyright notice and this permission notice appear in
 * supporting documentation, and that the name of OpenVision not be used
 * in advertising or publicity pertaining to distribution of the software
 * without specific, written prior permission. OpenVision makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 * OPENVISION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
 * EVENT SHALL OPENVISION BE LIABLE FOR ANY SPECIAL, INDIRECT OR
 * CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
 * USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
 * OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
 * PERFORMANCE OF THIS SOFTWARE.
 */

#if !defined(lint) && !defined(__CODECENTER__)
static char *rcsid = "$Header: /afs/athena.mit.edu/astaff/project/krbdev/.cvsroot
/src/appl/gss-sample/gss-misc.c,v 1.15 1996/07/22 20:21:20 marc Exp $";
#endif

#include <stdio.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <errno.h>
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif
```

(続く)

```
#include <string.h>

#include <gssapi/gssapi.h>
#include "gss-misc.h"
#include <stdlib.h>

FILE *display_file;
extern gss_OID g_mechOid;


static void display_status_1(char *m, OM_uint32 code, int type);

static int write_all(int fildes, char *buf, unsigned int nbyte)
{
     int ret;
     char *ptr;

     for (ptr = buf; nbyte; ptr += ret, nbyte -= ret) {
          ret = write(fildes, ptr, nbyte);
          if (ret < 0) {
               if (errno == EINTR)
                    continue;
               return(ret);
          } else if (ret == 0) {
               return(ptr-buf);
          }
     }

     return(ptr-buf);
}

static int read_all(int fildes, char *buf, unsigned int nbyte)
{
     int ret;
     char *ptr;

     for (ptr = buf; nbyte; ptr += ret, nbyte -= ret) {
          ret = read(fildes, ptr, nbyte);
          if (ret < 0) {
               if (errno == EINTR)
                    continue;
               return(ret);
          } else if (ret == 0) {
               return(ptr-buf);
          }
     }

     return(ptr-buf);
}

static void display_status_1(m, code, type)
     char *m;
     OM_uint32 code;
     int type;
```

(続く)

```
{
     OM_uint32 maj_stat, min_stat;
     gss_buffer_desc msg = GSS_C_EMPTY_BUFFER;
     OM_uint32 msg_ctx;

     msg_ctx = 0;
     while (1) {
          maj_stat = gss_display_status(&min_stat, code,
                                        type, g_mechOid,
                                        &msg_ctx, &msg);
          if (maj_stat != GSS_S_COMPLETE) {
                  if (display_file) {
                        fprintf(display_file, "error in gss_display_status"
                                         " called from <%s>\n", m);
                  }
                  break;
          }
          else if (display_file)
                fprintf(display_file, "GSS-API error %s: %s\n", m,
                      (char *)msg.value);
          if (msg.length != 0)
                (void) gss_release_buffer(&min_stat, &msg);

          if (!msg_ctx)
               break;
     }
}

/*
 * 関数: display_status
 *
 * 目的: GSS-API メッセージを表示する
 *
 * 引数:
 *
 *      msg             メッセージと一緒に表示する文字列
 *      maj_stat        GSS-API メジャー状態コード
 *      min_stat        GSS-API マイナー状態コード
 *
 * 効果:
 *
 * maj_stat と min_stat に関連する GSS-API メッセージが stderr に表示される。
 * 各メッセージの前には「GSS-API error <msg>:」が、後には復帰改行が
 * 出力される。
 */
void display_status(msg, maj_stat, min_stat)
     char *msg;
     OM_uint32 maj_stat;
     OM_uint32 min_stat;
{
     display_status_1(msg, maj_stat, GSS_C_GSS_CODE);
     display_status_1(msg, min_stat, GSS_C_MECH_CODE);
}
```

(続く)

```
/*
 * 関数: display_ctx_flags
 *
 * 目的: コンテキスト起動から戻されたフラグを人が読める形式で表示する。
 *
 * 引数:
 *
 *      int             ret_flags
 *
 * 効果:
 *
 * コンテキストフラグに対応する文字列が stdout に出力される。各文字列の
 * 前には「context flag: 」が、後には復帰改行が出力される。
 */

void display_ctx_flags(flags)
     OM_uint32 flags;
{
     if (flags & GSS_C_DELEG_FLAG)
          fprintf(display_file, "context flag: GSS_C_DELEG_FLAG\n");
     if (flags & GSS_C_MUTUAL_FLAG)
          fprintf(display_file, "context flag: GSS_C_MUTUAL_FLAG\n");
     if (flags & GSS_C_REPLAY_FLAG)
          fprintf(display_file, "context flag: GSS_C_REPLAY_FLAG\n");
     if (flags & GSS_C_SEQUENCE_FLAG)
          fprintf(display_file, "context flag: GSS_C_SEQUENCE_FLAG\n");
     if (flags & GSS_C_CONF_FLAG )
          fprintf(display_file, "context flag: GSS_C_CONF_FLAG \n");
     if (flags & GSS_C_INTEG_FLAG )
          fprintf(display_file, "context flag: GSS_C_INTEG_FLAG \n");
}

void print_token(tok)
     gss_buffer_t tok;
{
    int i;
    unsigned char *p = tok->value;

    if (!display_file)
        return;
    for (i=0; i < tok->length; i++, p++) {
        fprintf(display_file, "%02x ", *p);
        if ((i % 16) == 15) {
            fprintf(display_file, "\n");
        }
    }
    fprintf(display_file, "\n");
    fflush(display_file);
}
```

send_token() と recv_token()

この 2 つの関数はクライアントとサーバー間でデータを送受信します。なお、マルチプロセスアプリケーションでは、プロセス間でデータを送受信できます。トークンと同様にメッセージも送受信するため、名前が若干間違っているように思えるかもしれません。この 2 つの関数は自分が処理する内容を意識しません。

send_token()

send_token() はトークンまたはメッセージを送信します。

例 A–18 send_token()

```
/*
 * 関数: send_token
 *
 * 目的: トークンをファイル記述子に書き込む
 *
 * 引数:
 *
 *      s               (r) 開いたファイル記述子
 *      tok             (r) 書き込むトークン
 *
 * 戻り値: 成功した場合は 0、失敗した場合は -1
 *
 * 効果:
 *
 * send_token はまずトークンの長さ (ネットワーク上の長さ) を、次にトークンの
 * データをファイル記述子 s に書き込みます。成功した場合は 0 を戻し、
 * エラーが発生したり、すべてのデータを書き込むことができなかった場合は
 * -1 を戻します。
 */
int send_token(s, tok)
     int s;
     gss_buffer_t tok;
{
     int len, ret;

     len = htonl((OM_uint32)tok->length);
     ret = write_all(s, (char *) &len, sizeof(int));
     if (ret < 0) {
          perror("sending token length");
          return -1;
     } else if (ret != 4) {
         if (display_file)
             fprintf(display_file,
                     "sending token length: %d of %d bytes written\n",
                     ret, 4);
          return -1;
     }
```

(続く)

```
     ret = write_all(s, tok->value, (OM_uint32)tok->length);
     if (ret < 0) {
          perror("sending token data");
          return -1;
     } else if (ret != tok->length) {
         if (display_file)
             fprintf(display_file,
                     "sending token data: %d of %d bytes written\n",
                     ret, tok->length);
         return -1;
     }

     return 0;
}
```

recv_token()

recv_token() はトークンまたはメッセージを受信します。

例 **A–19** recv_token()

```
/*
 * 関数: recv_token
 *
 * 目的: ファイル記述子からトークンを読み取ります
 *
 * 引数:
 *
 *      s               (r) 開いたファイル記述子
 *      tok             (w) 読み取るトークン
 *
 * 戻り値: 成功した場合は 0、失敗した場合は -1
 *
 * 効果:
 *
 * recv_token はまずトークンの長さ (ネットワーク上の長さ) を読み取り、その
 * データを保持するメモリーを割り当て、そして、ファイル記述子 s からトークン
 * のデータを読み取る。必要であれば、長さとデータの読みとりを中止する。
 * 成功した場合、gss_release_buffer でトークンを解放する必要がある。
 * 成功した場合は 0 を戻し、エラーが発生したり、すべてのデータを読み取る
 * ことができなかった場合は -1 を戻す。
 */
int recv_token(s, tok)
     int s;
     gss_buffer_t tok;
{
     int ret, len;

     ret = read_all(s, (char *) &len, sizeof(int));
```

(続く)

```
     if (ret < 0) {
          perror("reading token length");
          return -1;
     } else if (ret != 4) {
         if (display_file)
             fprintf(display_file,
                     "reading token length: %d of %d bytes read\n",
                     ret, 4);
         return -1;
     }

     tok->length = ntohl(len);
     tok->value = (char *) malloc(tok->length);
     if (tok->value == NULL) {
         if (display_file)
             fprintf(display_file,
                     "Out of memory allocating token data\n");
          return -1;
     }

     ret = read_all(s, (char *) tok->value, (OM_uint32)tok->length);
     if (ret < 0) {
          perror("reading token data");
          free(tok->value);
          return -1;
     } else if (ret != tok->length) {
          fprintf(stderr, "sending token data: %d of %d bytes written\n",
                  ret, tok->length);
          free(tok->value);
          return -1;
     }

     return 0;
}
```

付録 **B**

GSS-API リファレンス

この付録は、次のような節から構成されています。

- 135ページの「GSS-API 関数」では、GSS-API 関数の表を提供します。
- 139ページの「GSS-API 状態コード」では、GSS-API 関数が戻す状態コードについて説明し、状態コードのリストを提供します。
- 144ページの「GSS-API データ型と値」では、GSS-API で使用されるさまざまなデータ型について説明します。

これ以外の GSS-API 定義については、ファイル `gssapi.h` を参照してください。

GSS-API 関数

次の表に、GSS-API 関数のリストを示します。各関数の詳細は、それぞれのマニュアルページを参照してください。また、138ページの「旧バージョンの GSS-API 関数」も参照してください。

表 B–1 GSS-API 関数

ヘッダー	機能
`gss_acquire_cred()`	大域的な ID を取得する。つまり、すでに存在している資格の GSS-API 資格ハンドルを取得する。
`gss_add_cred()`	資格を増分的に作成する。

表 **B–1** GSS-API 関数 続く

ヘッダー	機能
`gss_inquire_cred()`	資格についての情報を取得する。
`gss_inquire_cred_by_mech()`	資格についての機構ごとの情報を取得する。
`gss_release_cred()`	資格ハンドルを破棄する。
`gss_init_sec_context()`	ピアとなるアプリケーションとのセキュリティコンテキストを起動する。
`gss_accept_sec_context()`	ピアとなるアプリケーションが起動したセキュリティコンテキストを受け入れる。
`gss_delete_sec_context()`	セキュリティコンテキストを破棄する。
`gss_process_context_token()`	ピアとなるアプリケーションからのセキュリティコンテキストでトークンを処理する。
`gss_context_time()`	コンテキストが有効である時間を決定する。
`gss_inquire_context()`	セキュリティコンテキストについての情報を取得する。
`gss_wrap_size_limit()`	`gss_wrap()` をコンテキストに実行するためにトークンのサイズの制限を決定する。
`gss_export_sec_context()`	セキュリティコンテキストを別のプロセスに転送する。
`gss_import_sec_context()`	転送されたコンテキストをインポートする。
`gss_get_mic()`	メッセージの暗号化メッセージ整合性コード (MIC) を計算する。つまり、整合性サービス。
`gss_verify_mic()`	MIC をメッセージに対して検査する。つまり、受信したメッセージの整合性を検証する。
`gss_wrap()`	MIC をメッセージに添付し、メッセージの内容を暗号化する (後者は省略可能)。

表 B–1 GSS-API 関数 続く

ヘッダー	機能
gss_unwrap()	添付された MIC でメッセージを検証し、必要であれば、メッセージの内容を復号化する。
gss_import_name()	連続する文字列名を内部形式に変換する。
gss_display_name()	内部形式名をテキストに変換する。
gss_compare_name()	2 つの内部形式名を比較する。
gss_release_name()	内部形式名を破棄する。
gss_inquire_names_for_mech()	指定した機構がサポートする名前型のリストを表示する。
gss_inquire_mechs_for_name()	指定した名前型をサポートする機構のリストを表示する。
gss_canonicalize_name()	内部名を MN に変換する。
gss_export_name()	MN をエクスポート形式に変換する。
gss_duplicate_name()	内部名のコピーを作成する。
gss_add_oid_set_member()	オブジェクト識別子を集合に追加する。
gss_display_status()	GSS-API 状態コードをテキストに変換する。
gss_indicate_mechs()	使用できる実際の認証機構を決定する。
gss_release_buffer()	バッファを破棄する。
gss_release_oid_set()	オブジェクト識別子の集合を破棄する。

表 B–1　GSS-API 関数　続く

ヘッダー	機能
gss_create_empty_oid_set()	オブジェクト識別子の空の集合を作成する。
gss_test_oid_set_member()	オブジェクト識別子が集合のメンバーであるかどうかを決定する。

旧バージョンの GSS-API 関数

この節では、旧バージョンの GSS-API 関数について説明します。

OID を処理する関数

次の関数は GSS-API の Sun の実装でサポートされています。これは、便宜上、旧バージョンの GSS-API で作成されたプログラムとの下位互換性のために提供されています。しかし、GSS-API の Sun 以外の実装ではサポートされていないため、これらの関数には依存すべきではありません。

- gss_delete_oid()
- gss_oid_to_str()
- gss_str_to_oid()

これらの関数は機構名を文字列から OID に変換します。しかし、これらの関数を指定するのではなく、可能な限り、GSS-API が提供するデフォルトの機構を使用するべきです。

名前が変更された関数

次の関数は新しい関数に差し替えられました。どの場合も、新しい関数は古い関数と機能的に同等です。古い関数もサポートされますが、可能な限り、新しい関数に置き換えていくべきです。

- gss_sign() は gss_get_mic() に差し替えられました。
- gss_verify() は gss_verify_mic() に差し替えられました。
- gss_seal() は gss_wrap() に差し替えられました。

- `gss_unseal()` は `gss_unwrap()` に差し替えられました。

GSS-API 状態コード

メジャー状態コードは `OM_uint32` に符号化されます (図 B–1 を参照)。

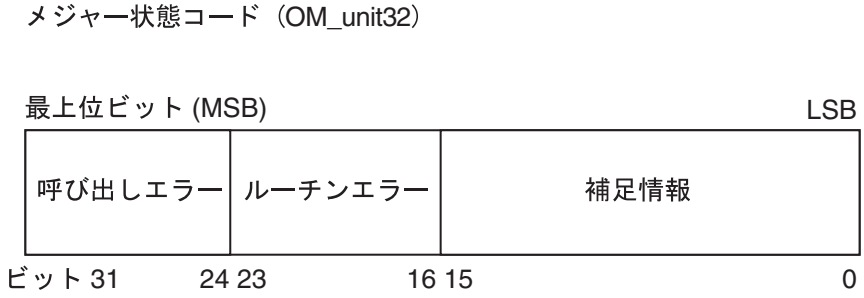

図 B–1　メジャー状態の符号化

GSS-API ルーチンが上位 16 ビットに 0 以外の値が入った GSS 状態コードを戻す場合、その呼び出しは失敗したことを示します。呼び出しエラーフィールドが 0 以外の場合、呼び出し側アプリケーションのルーチンの呼び出しにエラーがあったことを示します。表 B–2 に、呼び出しエラーのリストを示します。ルーチンエラーフィールドが 0 以外の場合、ルーチン固有のエラーのためにルーチンが失敗したことを示します。表 B–3 に、ルーチンエラーのリストを示します。上位 16 ビットが失敗または成功のどちらを示すかに関わらず、ルーチンは追加情報を状態コードの補足情報フィールドに設定できます。表 B–4 に、補足情報フィールドの個々のビットの意味を示します。

GSS-API メジャー状態コードの値

次の表に、GSS-API が戻す呼び出しエラーのリストを示します。これは、特定の言語バインディング (この場合は C) に固有なエラーのことです。

表 B–2　呼び出しエラー

エラー	フィールドの値	意味
GSS_S_CALL_INACCESSIBLE_READ	1	要求された入力パラメータを読み取れない。
GSS_S_CALL_INACCESSIBLE_WRITE	2	要求された出力パラメータに書き込めない。
GSS_S_CALL_BAD_STRUCTURE	3	パラメータの形式が間違っている。

次の表に、GSS-API が戻すルーチンエラーのリストを示します。つまり、GSS-API 関数が戻す一般的なエラーのことです。

表 B–3　ルーチンエラー

エラー	フィールドの値	意味
GSS_S_BAD_MECH	1	要求された機構がサポートされていない。
GSS_S_BAD_NAME	2	提供された名前が無効である。
GSS_S_BAD_NAMETYPE	3	提供された名前型がサポートされていない。
GSS_S_BAD_BINDINGS	4	提供されたチャネルバインディングが間違っている。
GSS_S_BAD_STATUS	5	提供された状態コードが無効である。
GSS_S_BAD_MIC, GSS_S_BAD_SIG	6	トークンが持っている MIC が無効である。
GSS_S_NO_CRED	7	資格が提供されていない。あるいは、資格を使用またはアクセスできない。
GSS_S_NO_CONTEXT	8	コンテキストが全く確立されていない。
GSS_S_DEFECTIVE_TOKEN	9	トークンが無効である。

表 B–3 ルーチンエラー 続く

エラー	フィールドの値	意味
GSS_S_DEFECTIVE_CREDENTIAL	10	資格が無効である。
GSS_S_CREDENTIALS_EXPIRED	11	参照された資格の有効期間が終了している。
GSS_S_CONTEXT_EXPIRED	12	コンテキストの有効期間が終了している。
GSS_S_FAILURE	13	その他のエラー **(**テキストを参照**)**。
GSS_S_BAD_QOP	14	要求された保護品質を提供できない。
GSS_S_UNAUTHORIZED	15	当該操作はローカルのセキュリティポリシーによって禁止されている。
GSS_S_UNAVAILABLE	16	当該操作またはオプションは使用できない。
GSS_S_DUPLICATE_ELEMENT	17	要求された資格要素はすでに存在している。
GSS_S_NAME_NOT_MN	18	提供された名前が機構名 (MN) ではない。

ルーチンの説明では、`GSS_S_COMPLETE` という名前も使用していました。この値は 0 で、API エラーと補足情報ビットがどちらも存在しないことを示します。

次の表に、GSS-API 関数が戻す補足情報の値のリストを示します。

表 B–4　補足情報コード

コード	ビット番号	意味
GSS_S_CONTINUE_NEEDED	0 (LSB)	gss_init_sec_context() または gss_accept_sec_context() だけが戻す。関数を完了させるには、もう一度ルーチンを呼び出す必要があることを示す。
GSS_S_DUPLICATE_TOKEN	1	トークンは以前のトークンの複製である。
GSS_S_OLD_TOKEN	2	トークンの有効期間が終了している。
GSS_S_UNSEQ_TOKEN	3	後方にあるトークンをすでに処理している。
GSS_S_GAP_TOKEN	4	期待していたメッセージ毎トークンを受信していない。

GSS メジャー状態コードの GSS_S_FAILURE は、実際の機構が特定の GSS–API 状態コードに定義されていないエラーを検出したことを示します。この場合、機構に固有な状態コード (マイナー状態コード) にエラーの詳細が提供されます。

状態コードについての詳細は、29ページの「状態コード」を参照してください。

状態コードの表示

gss_display_status() 関数は GSS-API 状態コードをテキスト形式に変換して、ユーザーに表示したり、テキストログに格納したりできます。gss_display_status() は一度に 1 つの状態コードしか表示できず、さらに、複数の状態を戻す関数も存在するため、gss_display_status() はループの一部として呼び出す必要があります。gss_display_status() が 0 以外の値コード (例 B–1 の *message_context* パラメータに戻される値) を示している間、関数は別の状態コードを取得できます。

例 B–1　gss_display_status() による状態コードの表示

```
OM_uint32 message_context;
OM_uint32 status_code;
OM_uint32 maj_status;
```

```
OM_uint32 min_status;
gss_buffer_desc status_string;

...

message_context = 0;

do {

     maj_status = gss_display_status(
               &min_status,
               status_code,
               GSS_C_GSS_CODE,
               GSS_C_NO_OID,
               &message_context,
               &status_string);

     fprintf(stderr, "%.*s\n", \
               (int)status_string.length, \
               (char *)status_string.value);

     gss_release_buffer(&min_status, &status_string,);

} while (message_context != 0);
```

状態コードのマクロ

マクロ GSS_CALLING_ERROR()、GSS_ROUTINE_ERROR()、および GSS_SUPPLEMENTARY_INFO() はそれぞれ、GSS 状態コードを引数として受け取り、関係のないフィールドをすべて削除します。たとえば、GSS_ROUTINE_ERROR() を状態コードに適用して得た値には、呼び出しエラーフィールドと補足情報フィールドが削除され、ルーチンエラーフィールドだけが残ります。このようなマクロが提供する値は、適切な型の GSS_S_*xxx* シンボルと直接比較できます。また、マクロ GSS_ERROR() もあります。このマクロを GSS–API 状態コードに適用すると、状態コードが呼び出しエラーまたはルーチンエラーを示す場合は 0 以外の値を戻し、それ以外の場合は 0 の値を戻します。GSS-API で定義されているすべてのマクロは引数を 1 つだけしか受け取りません。

GSS-API データ型と値

この節では、さまざまなタイプの GSS-API データ型と値を説明します。gss_cred_id_t や gss_name_t などのいくつかのデータ型はユーザーに不透明であり、その構造を知っても利点はないため、ここでは説明しません。この節では、次のことについて説明します。

- 144ページの「基本 GSS-API データ型」 — OM_uint32、gss_buffer_desc、gss_OID_desc、gss_OID_set_desc_struct、および gss_channel_bindings_struct データ型の定義を示します。
- 146ページの「名前型」 — 名前を指定するときに GSS-API が認識する、各種の名前の形式を示します。
- 147ページの「チャネルバインディングのアドレス型」 — gss_channel_bindings_t 構造体の *initiator_addrtype* と *acceptor_addrtype* のフィールドで使用できる値を示します。

基本 GSS-API データ型

次に、GSS-API で使用されるデータ型をいくつか示します。

OM_uint32

OM_uint32 はプラットフォームに依存しない 32 ビットの符号なし整数です。

gss_buffer_desc

次に、gss_buffer_desc と gss_buffer_t ポインタの定義を示します。

```
typedef struct gss_buffer_desc_struct {
        size_t length;
        void *value;
} gss_buffer_desc, *gss_buffer_t;
```

gss_OID_desc

次に、gss_OID_desc と gss_OID ポインタの定義を示します。

```
typedef struct gss_OID_desc_struct {
        OM_uint32 length;
        void*elements;
} gss_OID_desc, *gss_OID;
```

gss_OID_set_desc

次に、gss_OID_set_desc と gss_OID_set ポインタの定義を示します。

```
typedef struct gss_OID_set_desc_struct  {
        size_t  count;
        gss_OID elements;
} gss_OID_set_desc, *gss_OID_set;
```

gss_channel_bindings_struct

次に、gss_channel_bindings_struct 構造体と gss_channel_bindings_t ポインタの定義を示します。

```
typedef struct gss_channel_bindings_struct {
        OM_uint32 initiator_addrtype;
        gss_buffer_desc initiator_address;
        OM_uint32 acceptor_addrtype;
        gss_buffer_desc acceptor_address;
        gss_buffer_desc application_data;
} *gss_channel_bindings_t;
```

名前型

名前型とは、関連する名前の形式のことです。名前と名前型についての詳細は、20ページの「名前」と 27ページの「OID」を参照してください。次に、GSS-API がサポートする名前型を示します。これらはすべて `gss_OID` 型です。

表 B–5 名前型

名前型	意味
GSS_C_NO_NAME	`GSS_C_NO_NAME` (推奨されるシンボリック名) は、名前の転送で使用されるパラメータの特定の値に名前が渡されないことを意味する。
GSS_C_NO_OID	実際のオブジェクト識別子ではなく、NULL の入力値に相当する。指定した場合、関連する名前が機構に固有なデフォルトの印刷可能な構文に基づいて解釈されることを示す。
GSS_C_NT_ANONYMOUS	匿名を指定する方法として提供される。この名前型と比較することによって、その名前が匿名のプリンシパルを参照するかどうかを機構に依存しない方法で決定できる。
GSS_C_NT_EXPORT_NAME	`gss_export_name()` 関数でエクスポートされた名前。
GSS_C_NT_HOSTBASED_SERVICE	ホストコンピュータと関連するサービスを表す。この名前型は 2 つの要素「サービス (service)」と「ホスト名 (hostname)」からなり、*service@hostname* の形式を取る。
GSS_C_NT_MACHINE_UID_NAME	ローカルシステム上のユーザーの、数値によるユーザー識別子を示す。解釈の方法は OS に固有である。`gss_import_name()` 関数はこの UID を *username* に解釈処理し、ユーザー名形式として扱う。

表 B–5　名前型　続く

名前型	意味
GSS_C_NT_STRING_STRING_UID_NAME	ローカルシステム上のユーザーの数値によるユーザー識別子を表す数字文字列を示す。解釈の方法は OS に固有である。この名前型はマシン UID 形式に似ているが、バッファにはユーザー ID を表す文字が入っている。
GSS_C_NT_USER_NAME	ローカルシステム上の指定されたユーザー。解釈の方法は OS に固有である。*username* の形式を取る。

チャネルバインディングのアドレス型

表 B–6 に、`gss_channel_bindings_struct` 構造体の *initiator_addrtype* と *acceptor_addrtype* のフィールドで使用できる値を示します。この 2 つのフィールドは名前を受け取ることができる形式を示します (たとえば ARPAnet IMP アドレス形式や AppleTalk アドレス形式など)。チャネルバインディングについては、59ページの「チャネルバインディング」を参照してください。

表 B–6　チャネルバインディングのアドレス型

フィールド	値 (10 進数)	アドレス型
GSS_C_AF_UNSPEC	0	未定のアドレス型
GSS_C_AF_LOCAL	1	ホスト - ローカル
GSS_C_AF_INET	2	インターネットアドレス型 (IP など)
GSS_C_AF_IMPLINK	3	ARPAnet IMP
GSS_C_AF_PUP	4	pup プロトコル (BSP など)
GSS_C_AF_CHAOS	5	MIT CHAOS プロトコル
GSS_C_AF_NS	6	XEROX NS
GSS_C_AF_NBS	7	nbs

表 **B–6** チャネルバインディングのアドレス型 続く

フィールド	値 **(10** 進数**)**	アドレス型
GSS_C_AF_ECMA	8	ECMA
GSS_C_AF_DATAKIT	9	データキットプロトコル
GSS_C_AF_CCITT	10	CCITT
GSS_C_AF_SNA	11	IBM SNA
GSS_C_AF_DECnet	12	DECnet
GSS_C_AF_DLI	13	ダイレクトデータリンクインタフェース
GSS_C_AF_LAT	14	LAT
GSS_C_AF_HYLINK	15	NSC ハイパーチャネル
GSS_C_AF_APPLETALK	16	AppleTalk
GSS_C_AF_BSC	17	BISYNC
GSS_C_AF_DSS	18	分散システムサービス
GSS_C_AF_OSI	19	OSI TP4
GSS_C_AF_X25	21	X.25
GSS_C_AF_NULLADDR	255	アドレスは指定されていない。

付録 C

OID の指定

機構と QOP (Quality of Protection)

できるだけ GSS-API が提供するデフォルトの機構と QOP を使用することを強く推奨しますが (27ページの「OID」を参照)、なんらかの理由で機構または QOP の OID を指定する必要がある場合もあります。したがって、この章では機構または QOP の OID を指定する方法について簡単に説明します。

OID 値が含まれるファイル

GSS-API では、機構と QOP を人が読める形式で表示することができます。Solaris システムでは、/etc/gss/mech と /etc/gss/qop の 2 つのファイルに、使用できる機構と QOP についての情報が含まれています。この 2 つのファイルへのアクセス権がない場合 (ほとんどの場合、リモートマシンが許可していないため)、たとえば、その機構または QOP 用に公開されているインターネット標準などの他のソースから文字列リテラルを提供する必要があります。

/etc/gss/mech ファイル

/etc/gss/mech ファイルを調べると、どの機構を使用できるかが分かります。/etc/gss/mech には、機構名が ASCII 文字列と数値の両方の形式で格納されています。/etc/gss/mech の各行は、機構名 (ASCII 文字列)、機構の OID (数値)、その機構が提供するサービスを実装する共有ライブラリ、およびオプションで

サービスを実装するカーネルモジュールから構成されます。例 C–1 に /etc/gss/mech ファイルの例を示します。

例 **C–1** /etc/gss/mech ファイル

```
#
# Copyright (c) 2000, by Sun Microsystems, Inc.
# All rights reserved.
#
#ident  "@(#)mech 1.6     00/12/04 SMI"
#
# This file contains the GSS-API based security mechanism names,
# its object identifier (OID) and a shared library that implements
# the services for that mechanism under GSS-API.
#
# Mechanism Name       Object Identifier      Shared Library  Kernel Module
#
diffie_hellman_640_0   1.3.6.4.1.42.2.26.2.4  dh640-0.so.1
diffie_hellman_1024_0  1.3.6.4.1.42.2.26.2.5  dh1024-0.so.1
kerberos_v5            1.2.840.113554.1.2.2   gl/mech_krb5.so gl_kmech_krb5
```

/etc/gss/qop ファイル

/etc/gss/qop ファイルには、インストールされている各機構がサポートするすべての QOP が、ASCII 文字列と対応する 32 ビット整数の両方の形式で格納されています。例 C–2 に /etc/gss/qop ファイルの例を示します。

例 **C–2** /etc/gss/qop ファイル

```
#
# Copyright (c) 2000, by Sun Microsystems, Inc.
# All rights reserved.
#
#ident  "@(#)qop 1.3     00/11/09 SMI"
#
# This file contains information about the GSS-API based quality of
# protection (QOP), its string name and its value (32-bit integer).
#
# QOP string                     QOP Value       Mechanism Name
#
GSS_KRB5_INTEG_C_QOP_DES_MD5     0               kerberos_v5
GSS_KRB5_CONF_C_QOP_DES          0               kerberos_v5
```

gss_str_to_oid()

旧バージョンの GSS-API との下位互換性のため、この実装の GSS-API は gss_str_to_oid() 関数をサポートします。gss_str_to_oid() は機構または QOP を表す文字列を (ASCII 文字列または数値のどちらでも) OID に変換します。

注意 - デフォルトの機構と QOP を使用することが強く推奨されているため、`gss_str_to_oid()`、`gss_oid_to_str()`、および `gss_release_oid()` をサポートしていない実装の GSS-API もあります。

機構または QOP を表す文字列は、アプリケーション内でハードコード化することも、ユーザー入力から取得することも可能です。しかし、必ずしもすべての実装の GSS-API がこの関数をサポートしているわけではないため、アプリケーションはこの関数に依存すべきではありません。

機構を表す数値には、次の 2 種類の形式があります。1 つの形式は次のとおりです。

{ 1 2 3 4 }

この形式は GSS-API 仕様で正式に指定されています。もう 1 つの形式は次のとおりです。

1.2.3.4

この形式は広く使用されていますが、正式な標準形式ではありません。`gss_str_to_oid()` は機構の数値として最初の形式を期待します。したがって、2 番目の形式を使用している場合は、`gss_str_to_oid()` を呼び出す前に 1 番目の形式に変換する必要があります。この例については、102ページの「`parse_oid()`」を参照してください。機構が有効でない場合、`gss_str_to_oid()` は `GSS_S_BAD_MECH` を戻します。

`gss_str_to_oid()` は GSS-API データ領域を割り当てるため、終了時には、割り当てられた OID を `gss_release_oid()` 関数で削除する必要があります。`gss_str_to_oid()` と同様に、`gss_release_oid()` も一般的にサポートされている関数ではありません。したがって、移植性を最大限にしたいプログラムはこの関数に依存すべきではありません。

機構 OID の構築

`gss_str_to_oid()` は常に使用できるわけではなく、また推奨されるものではないため、使用できる機構を調べて選択するために、より好ましい方法が (若干複雑

ですが) いくつか存在します。1 つは、機構 OID を「手動で」構築し、使用できる機構の集合と比較する方法です。もう 1 つは、使用できる機構の集合を取得して、その中から 1 つを選択する方法です。

次に、gss_OID 型の形式を示します。

```
typedef struct gss_OID_desc struct {
     OM_uint32 length;
     void           *elements;
} gss_OID_desc, *gss_OID;
```

この構造体の *elements* フィールドは、gss_OID の通常の BER TLV エンコーディングの値の部分の ASN.1 BER エンコーディングが格納されているオクテット文字列の最初のバイトを指します。*length* フィールドには、この値のバイト数が格納されています。たとえば、DASS X.509 認証機構に対応する gss_OID 値の場合、*length* フィールドは 7 で、*elements* フィールドは「53,14,2,207,163,7,5」という 8 進数値を含む 7 つのオクテットを指します。

機構 OID を構築する 1 つの方法は、gss_OID を宣言して、次に、その要素を手作業で初期化して、機構の OID を表すようにします。前述のとおり、*elements* 値はハードコード化するか、表から検索するか、あるいは、ユーザー入力から取得できます。この方法は gss_str_to_oid() を使用するよりも手がかかりますが、同じ効果が得られます。

次に、手作業で構築した gss_OID を、使用できる機構の集合と比較します。使用できる機構の集合は、gss_indicate_mechs() または gss_inquire_mechs_for_name() の関数から戻されます。手作業で構築した機構 OID が、使用できる機構の集合の中に存在するかどうかを調べるには、gss_test_oid_set_member() 関数を使用します。gss_test_oid_set_member() がエラーを戻さなかった場合、手作業で構築した OID は GSS-API トランザクション用の機構として使用できます。

OID を手作業で構築する代わりに、gss_indicate_mechs() または gss_inquire_mechs_for_name() を使用すると、使用できる機構の gss_OID_set を取得できます。次に、gss_OID_set の形式を示します。

```
typedef struct gss_OID_set_desc_struct {
     OM_uint32 length;
     void           *elements;
} gss_OID_set_desc, *gss_OID_set;
```

elements は機構を表す gss_OID です。すると、アプリケーションは各機構を解析し、それぞれの elements の値 (つまり、機構の数値表記) を表示できます。この結果、ユーザーは表示された値に基づいて使用したい機構を選択し、選択した機構が gss_OID_set の適切なメンバーになるようにアプリケーションで設定できます。あるいは、希望する機構と使用できる機構のリストをアプリケーションで比較することも可能です。

付録 D

Sun 固有の機能

この付録では、GSS-API の Sun の実装に固有な機能について説明します。

実装に固有な機能

GSS-API の実装によっては、いくつかの動作が若干異なる場合もあります。ほとんどの場合、実装による違いはプログラムに最小限の影響しか与えません。どのような場合でも、実装に固有な動作 (Sun の実装も含む) に依存しなければ、移植性を最大限にすることができます。

Sun 固有の関数

Sun の実装に固有な GSS-API 関数は存在しません。

人が読める名前についての構文

GSS-API の実装によっては、名前の出力可能な形式についての構文が異なる場合があります。移植性を最大限にしたいアプリケーションでは、名前を比較するときに、人が読める (つまり、出力可能な) 形式で比較するのではなく、`gss_compare_name()` を使用して一致するかどうかを内部形式で比較するべきです。

Sun の 実装の gss_display_name() は名前を次のように表示します。*input_name* 引数がユーザープリンシパルを指す場合、gss_display_name() は user_principal@realm を *output_name_buffer* として、gss_OID 値を *output_name_type* として戻します。Kerberos v5 が実際の機構である場合、gss_OID は 1.2.840.11354.1.2.2 になります。

gss_display_name() に指定した名前が gss_import_name() への呼び出しで作成されていた場合、GSS_C_NO_OID を名前型として指定すると、gss_display_name() は GSS_C_NO_OID を *output_name_type* パラメータ経由で戻します。

匿名の形式

gss_display_name() 関数は、匿名の GSS-API プリンシパルを示すとき、文字列 <anonymous> を出力します。この名前に関連する名前型 OID は GSS_C_NT_ANONYMOUS です。Sun の実装で有効な印刷可能な名前の中では、これ以外に < で始まり > で終わるものは存在しません。

選択されたデータ型の実装

Sun の実装では、gss_cred_t、gss_ctx_id_t、gss_name_t のデータ型はポインタです。他の実装の中には、算術型として指定されるものもあります。

コンテキストの削除と格納されたデータの解放

コンテキストの確立が失敗した場合、Sun の実装では「構築中」のコンテキストを自動的に削除しません。したがって、アプリケーションがこの事態を処理する、つまり、gss_delete_sec_context() でコンテキストを削除する必要があります。

Sun の実装では、格納されたデータ (内部名など) を、メモリー管理を通じて自動的に解放します。しかし、用心のため、データ要素が必要でなくなったときには、アプリケーションで適切な関数 (gss_release_name() など) を呼び出すべきです。

チャネルバインディング情報の保護

Sun の実装では、チャネルバインディングに含まれる情報を暗号化しません。したがって、プログラマは、この情報へ攻撃不可能であるとは考えるべきではありません。

コンテキストのエクスポートとプロセス間トークン

Sun の実装では、コンテキストのエクスポートをサポートします。しかし、GSS-API の他の実装の中には、サポートしないものもあります。コンテキストのエクスポートで使用されるプロセス間トークンには、オリジナルのセキュリティコンテキストからの重要なデータ (暗号化鍵など) が含まれている可能性があります。GSS-API の Sun の実装では、プロセス間トークンを暗号化しません。したがって、セキュリティコンテキストをエクスポートするアプリケーションは、転送中のこのようなトークンをラップして保護する必要があります。

Sun の実装では、同じコンテキストに対する複数のインポートの試みを検出および拒否します。

サポートされる資格の型

GSS-API の Sun の実装では、`gss_acquire_cred()` による、`GSS_C_INITIATE`、`GSS_C_ACCEPT`、および `GSS_C_BOTH` の資格の獲得をサポートしています。

資格の有効期間の設定

GSS-API の Sun の実装では、資格の有効期間の設定をサポートします。したがって、プログラマは `gss_acquire_cred()` や `gss_add_cred()` などの関数で、資格の有効期間に関連するパラメータを使用できます。

コンテキストの有効期間の設定

GSS-API の Sun の実装では、コンテキストの有効期間の設定をサポートします。したがって、プログラマは `gss_init_sec_context()` や `gss_inquire_context()` などの関数で、コンテキストの有効期間に関連するパラメータを使用できます。

ラップサイズの制限と QOP 値

GSS-API の Sun の実装では、実際の機構とは異なり、`gss_wrap()` で処理するメッセージの最大サイズに制限を課しません。アプリケーションは `gss_wrap_size_limit()` でメッセージの最大サイズを決定できます。

GSS-API の Sun の実装では、`gss_wrap_size_limit()` を呼び出すとき、無効な QOP 値を検出します。

minor_status パラメータの使用

GSS-API の Sun の実装では、関数が *minor_status* パラメータで戻すのは、機構に固有な情報だけです。他の実装では、戻されたマイナー状態コードの一部として実装に固有な戻り値が含まれることもあります。

付録 **E**

Kerberos v5 状態コード

Kerberos v5 状態コードの表

各 GSS-API 関数は 2 つの状態コードを戻します。メジャー状態コードとマイナー状態コードです。メジャー状態コードは GSS-API 自身の動作に関連します。たとえば、セキュリティコンテキストの有効期間が終了した後で、アプリケーションがメッセージを転送しようとした場合、GSS-API は `GSS_S_CONTEXT_EXPIRED` というメジャー状態コードを戻します。メジャー状態コードのリストについては、139ページの「GSS-API 状態コード」を参照してください。

マイナー状態コードを戻すのは、GSS-API の実装でサポートされる実際のセキュリティ機構です。現在のところ、GSS-API の Sun の実装でサポートされる唯一のセキュリティ機構は Kerberos v5 です。Sun が実装する Kerberos v5 のことを SEAM (Sun Enterprise Authentication Mechanism) と呼びます。このマニュアルでは同じであると考えてかまいません。すべての GSS-API 関数は最初の引数として *minor_status* (または *minor_stat*) パラメータを受け取ります。関数が戻ったときにこのパラメータを調べることによって、アプリケーションは、関数が成功したかどうかに関わらず、実際の機構が戻した状態を知ることができます。

次の表に、Kerberos v5 が *minor_status* 引数に戻す状態メッセージのリストを示します。

GSS-API 状態コードについての詳細は、29ページの「状態コード」を参照してください。

表 E–1 Kerberos v5 状態コード 1

マイナー状態	値	意味
KRB5KDC_ERR_NONE	-1765328384L	エラーなし
KRB5KDC_ERR_NAME_EXP	-1765328383L	データベース内のクライアントのエントリの有効期間が終了している。
KRB5KDC_ERR_SERVICE_EXP	-1765328382L	データベース内のサーバーのエントリの有効期間が終了している。
KRB5KDC_ERR_BAD_PVNO	-1765328381L	要求されたプロトコルのバージョンはサポートされていない。
KRB5KDC_ERR_C_OLD_MAST_KVNO	-1765328380L	クライアントの鍵が古いマスター鍵で暗号化されている。
KRB5KDC_ERR_S_OLD_MAST_KVNO	-1765328379L	サーバーの鍵が古いマスター鍵で暗号化されている。
KRB5KDC_ERR_C_PRINCIPAL_UNKNOWN	-1765328378L	クライアントが Kerberos データベースに見つからない。
KRB5KDC_ERR_S_PRINCIPAL_UNKNOWN	-1765328377L	サーバーが Kerberos データベースに見つからない。
KRB5KDC_ERR_PRINCIPAL_NOT_UNIQUE	-1765328376L	プリンシパルが Kerberos データベースに複数のエントリを持っている。

マイナー状態	値	意味
KRB5KDC_ERR_NULL_KEY	-1765328375L	クライアントまたはサーバーの鍵が空である。
KRB5KDC_ERR_CANNOT_POSTDATE	-1765328374L	チケットが遅延のために無効である。
KRB5KDC_ERR_NEVER_VALID	-1765328373L	要求された有効期間が負であるか、短すぎる。
KRB5KDC_ERR_POLICY	-1765328372L	KDC ポリシーが要求を拒否した。
KRB5KDC_ERR_BADOPTION	-1765328371L	KDC が要求されたオプションを実行できない。
KRB5KDC_ERR_ETYPE_NOSUPP	-1765328370L	KDC が暗号化型をサポートしていない。
KRB5KDC_ERR_SUMTYPE_NOSUPP	-1765328369L	KDC がチェックサム型をサポートしていない。
KRB5KDC_ERR_PADATA_TYPE_NOSUPP	-1765328368L	KDC が `padata` 型をサポートしていない。
KRB5KDC_ERR_TRTYPE_NOSUPP	-1765328367L	KDC が `transited` 型をサポートしていない。
KRB5KDC_ERR_CLIENT_REVOKED	-1765328366L	クライアントの資格が取り消された。
KRB5KDC_ERR_SERVICE_REVOKED	-1765328365L	サーバーの資格が取り消された。

表 **E–2** Kerberos v5 状態コード 2

マイナー状態	値	意味
KRB5KDC_ERR_TGT_REVOKED	-1765328364L	TGT が取り消された。
KRB5KDC_ERR_CLIENT_NOTYET	-1765328363L	クライアントがまだ有効でない。後程再試行してください。
KRB5KDC_ERR_SERVICE_NOTYET	-1765328362L	サーバーがまだ有効でない。後程再試行してください。
KRB5KDC_ERR_KEY_EXP	-1765328361L	パスワードの有効期間が終了している。
KRB5KDC_ERR_PREAUTH_FAILED	-1765328360L	事前認証が失敗した。
KRB5KDC_ERR_PREAUTH_REQUIRED	-1765328359L	追加の事前認証が要求された。
KRB5KDC_ERR_SERVER_NOMATCH	-1765328358L	要求されたサーバーとチケットが一致しない。
KRB5PLACEHOLD_27 ～ KRB5PLACEHOLD_30	-1765328357L ～ -1765328354L	KRB5 エラーコード (27 ～ 30。予約済み)
KRB5KRB_AP_ERR_BAD_INTEGRITY	-1765328353L	復号化整合性チェックが失敗した。
KRB5KRB_AP_ERR_TKT_EXPIRED	-1765328352L	チケットの有効期間が終了している。
KRB5KRB_AP_ERR_TKT_NYV	-1765328351L	チケットがまだ有効でない。
KRB5KRB_AP_ERR_REPEAT	-1765328350L	リプレイされた要求。

マイナー状態	値	意味
KRB5KRB_AP_ERR_NOT_US	-1765328349L	チケットが Kerberos v5 用でない。
KRB5KRB_AP_ERR_BADMATCH	-1765328348L	チケットと認証用データが一致しない。
KRB5KRB_AP_ERR_SKEW	-1765328347L	クロックスキューが大きすぎる。
KRB5KRB_AP_ERR_BADADDR	-1765328346L	ネットアドレスが間違っている。
KRB5KRB_AP_ERR_BADVERSION	-1765328345L	プロトコルのバージョンが一致しない。
KRB5KRB_AP_ERR_MSG_TYPE	-1765328344L	メッセージの型が無効である。
KRB5KRB_AP_ERR_MODIFIED	-1765328343L	メッセージのストリームが変更された。
KRB5KRB_AP_ERR_BADORDER	-1765328342L	メッセージの順番が間違っている。
KRB5KRB_AP_ERR_ILL_CR_TKT	-1765328341L	レルム間チケットが無効である。
KRB5KRB_AP_ERR_BADKEYVER	-1765328340L	キーのバージョンが使用できない。

表 E–3 Kerberos v5 状態コード 3

マイナー状態	値	意味
KRB5KRB_AP_ERR_NOKEY	-1765328339L	サービス鍵が使用できない。
KRB5KRB_AP_ERR_MUT_FAIL	-1765328338L	相互認証が失敗した。

マイナー状態	値	意味
KRB5KRB_AP_ERR_BADDIRECTION	-1765328337L	メッセージの方向が間違っている。
KRB5KRB_AP_ERR_METHOD	-1765328336L	代替の認証方法が要求された。
KRB5KRB_AP_ERR_BADSEQ	-1765328335L	メッセージ内のシーケンス番号が間違っている。
KRB5KRB_AP_ERR_INAPP_CKSUM	-1765328334L	メッセージ内のチェックサム型が不適切である。
KRB5PLACEHOLD_51 ～ KRB5PLACEHOLD_59	-1765328333L ～ -1765328325L	KRB5 エラーコード (51 ～ 59。予約済み)
KRB5KRB_ERR_GENERIC	-1765328324L	一般的なエラー
KRB5KRB_ERR_FIELD_TOOLONG	-1765328323L	フィールドがこの実装には長すぎる。
KRB5PLACEHOLD_62 ～ KRB5PLACEHOLD_127	-1765328322L ～ -1765328257L	KRB5 エラーコード (62 ～ 127。予約済み)
(値は戻されない)	-1765328256L	内部使用のみ
KRB5_LIBOS_BADLOCKFLAG	-1765328255L	ファイルロックモードのフラグが無効である。
KRB5_LIBOS_CANTREADPWD	-1765328254L	パスワードを読み取れない。
KRB5_LIBOS_BADPWDMATCH	-1765328253L	パスワードが一致しない。
KRB5_LIBOS_PWDINTR	-1765328252L	パスワードの読み取りが中断された。

マイナー状態	値	意味
KRB5_PARSE_ILLCHAR	-1765328251L	構成要素名の文字が無効である。
KRB5_PARSE_MALFORMED	-1765328250L	プリンシパルの表現形式が間違っている。
KRB5_CONFIG_CANTOPEN	-1765328249L	Kerberos 構成ファイル `/etc/krb5/krb5` が開けない (または、見つからない)。
KRB5_CONFIG_BADFORMAT	-1765328248L	Kerberos 構成ファイル `/etc/krb5/krb5` の形式が不適切である。
KRB5_CONFIG_NOTENUFSPACE	-1765328247L	完全な情報を戻すには領域が不足している。
KRB5_BADMSGTYPE	-1765328246L	符号化用に指定したメッセージ型が無効である。
KRB5_CC_BADNAME	-1765328245L	資格キャッシュ名の形式が間違っている。

表 E–4 Kerberos v5 状態コード 4

マイナー状態	値	意味
KRB5_CC_UNKNOWN_TYPE	-1765328244L	資格キャッシュ型が不明である。
KRB5_CC_NOTFOUND	-1765328243L	一致する資格が見つからない。

マイナー状態	値	意味
KRB5_CC_END	-1765328242L	資格キャッシュの終わりに到達した。
KRB5_NO_TKT_SUPPLIED	-1765328241L	要求がチケットを提供していない。
KRB5KRB_AP_WRONG_PRINC	-1765328240L	要求のプリンシパルが間違っている。
KRB5KRB_AP_ERR_TKT_INVALID	-1765328239L	チケットが設定したフラグが無効である。
KRB5_PRINC_NOMATCH	-1765328238L	要求されたプリンシパルとチケットが一致しない。
KRB5_KDCREP_MODIFIED	-1765328237L	KDC 返信が期待したものと一致しない。
KRB5_KDCREP_SKEW	-1765328236L	クロックスキューが KDC 返信には大きすぎる
KRB5_IN_TKT_REALM_MISMATCH	-1765328235L	初期チケット要求でクライアントとサーバーの領域が一致しない。
KRB5_PROG_ETYPE_NOSUPP	-1765328234L	プログラムが暗号化型をサポートしていない。
KRB5_PROG_KEYTYPE_NOSUPP	-1765328233L	プログラムが鍵型をサポートしていない。
KRB5_WRONG_ETYPE	-1765328232L	要求された暗号化型がメッセージで使用されていない。

マイナー状態	値	意味
KRB5_PROG_SUMTYPE_NOSUPP	-1765328231L	プログラムがチェックサム型をサポートしていない。
KRB5_REALM_UNKNOWN	-1765328230L	要求されたレルムの KDC が見つからない。
KRB5_SERVICE_UNKNOWN	-1765328229L	Kerberos サービスは不明である。
KRB5_KDC_UNREACH	-1765328228L	要求されたレルムの KDC に到達できない。
KRB5_NO_LOCALNAME	-1765328227L	プリンシパル名のローカル名が見つからない。
KRB5_MUTUAL_FAILED	-1765328226L	相互認証が失敗した。
KRB5_RC_TYPE_EXISTS	-1765328225L	リプレイのキャッシュ型がすでに登録されている。
KRB5_RC_MALLOC	-1765328224L	これ以上メモリーを割り当てられない (リプレイのキャッシュコードで)。
KRB5_RC_TYPE_NOTFOUND	-1765328223L	リプレイのキャッシュ型が不明である。

表 **E–5** Kerberos v5 状態コード 5

マイナー状態	値	意味
KRB5_RC_UNKNOWN	-1765328222L	一般的な不明な RC エラー
KRB5_RC_REPLAY	-1765328221L	リプレイされたメッセージ。
KRB5_RC_IO	-1765328220L	リプレイの入出力操作が失敗した。
KRB5_RC_NOIO	-1765328219L	リプレイのキャッシュ型が非揮発性記憶装置をサポートしない。
KRB5_RC_PARSE	-1765328218L	リプレイのキャッシュ名の解析/形式エラー
KRB5_RC_IO_EOF	-1765328217L	リプレイのキャッシュ入出力でファイルの終わりに到達した。
KRB5_RC_IO_MALLOC	-1765328216L	これ以上メモリーを割り当てられない (リプレイのキャッシュ入出力コードで)。
KRB5_RC_IO_PERM	-1765328215L	アクセス権がない (リプレイのキャッシュコードで)
KRB5_RC_IO_IO	-1765328214L	入出力エラー (リプレイのキャッシュ入出力コードで)
KRB5_RC_IO_UNKNOWN	-1765328213L	一般的な不明な RC/入出力エラー
KRB5_RC_IO_SPACE	-1765328212L	リプレイの情報を格納するにはシステム領域が不足している。

マイナー状態	値	意味
KRB5_TRANS_CANTOPEN	-1765328211L	レルム変換ファイルが開けない (または、見つからない)。
KRB5_TRANS_BADFORMAT	-1765328210L	レルム変換ファイルの形式が不適切である。
KRB5_LNAME_CANTOPEN	-1765328209L	lname 変換データベースが開けない (または、見つからない)。
KRB5_LNAME_NOTRANS	-1765328208L	要求されたプリンシパルで使用できる変換が存在しない。
KRB5_LNAME_BADFORMAT	-1765328207L	変換データベースエントリの形式が不適切である。
KRB5_CRYPTO_INTERNAL	-1765328206L	暗号システム内部エラー
KRB5_KT_BADNAME	-1765328205L	鍵テーブル名の形式が間違っている。
KRB5_KT_UNKNOWN_TYPE	-1765328204L	鍵テーブル型が不明である。
KRB5_KT_NOTFOUND	-1765328203L	鍵テーブルエントリが見つからない。
KRB5_KT_END	-1765328202L	鍵テーブルの終わりに到達した。
KRB5_KT_NOWRITE	-1765328201L	指定された鍵テーブルに書き込めない。

表 **E–6** Kerberos v5 状態コード 6

マイナー状態	値	意味
KRB5_KT_IOERR	-1765328200L	鍵テーブルへの書き込み中にエラーが発生した。
KRB5_NO_TKT_IN_RLM	-1765328199L	要求されたレルムのチケットが見つからない。
KRB5DES_BAD_KEYPAR	-1765328198L	DES 鍵のパリティが不良である。
KRB5DES_WEAK_KEY	-1765328197L	DES 鍵が弱い鍵である。
KRB5_BAD_ENCTYPE	-1765328196L	暗号化型が不良である。
KRB5_BAD_KEYSIZE	-1765328195L	鍵サイズが暗号化型と互換性がない。
KRB5_BAD_MSIZE	-1765328194L	メッセージサイズが暗号化型と互換性がない。
KRB5_CC_TYPE_EXISTS	-1765328193L	資格キャッシュ型がすでに登録されている。
KRB5_KT_TYPE_EXISTS	-1765328192L	鍵テーブル型がすでに登録されている。
KRB5_CC_IO	-1765328191L	資格キャッシュ入出力操作が失敗した。
KRB5_FCC_PERM	-1765328190L	資格キャッシュファイルのアクセス権が間違っている。
KRB5_FCC_NOFILE	-1765328189L	資格キャッシュファイルが見つからない。

マイナー状態	値	意味
KRB5_FCC_INTERNAL	-1765328188L	内部ファイル資格キャッシュエラー
KRB5_CC_WRITE	-1765328187L	資格キャッシュファイルの書き込み中にエラーが発生した。
KRB5_CC_NOMEM	-1765328186L	これ以上メモリーを割り当てられない (資格キャッシュコードで)。
KRB5_CC_FORMAT	-1765328185L	資格キャッシュの形式が不良である。
KRB5_INVALID_FLAGS	-1765328184L	KDC オプションの組み合わせが無効である (ライブラリ内部エラー)。
KRB5_NO_2ND_TKT	-1765328183L	要求に 2 番目のチケットが指定されていない。
KRB5_NOCREDS_SUPPLIED	-1765328182L	ライブラリルーチンに資格が提供されていない。
KRB5_SENDAUTH_BADAUTHVERS	-1765328181L	送信された `sendauth` のバージョンが不良である。
KRB5_SENDAUTH_BADAPPLVERS	-1765328180L	(`sendauth` により) 送信されたアプリケーションのバージョンが不良である。

マイナー状態	値	意味
KRB5_SENDAUTH_BADRESPONSE	-1765328179L	(sendauth の交換中) 応答が不良である。
KRB5_SENDAUTH_REJECTED	-1765328178L	(sendauth の交換中) サーバーが認証を拒否した。

表 E–7 Kerberos v5 状態コード 7

マイナー状態	値	意味
KRB5_PREAUTH_BAD_TYPE	-1765328177L	事前認証型がサポートされていない。
KRB5_PREAUTH_NO_KEY	-1765328176L	要求された事前認証鍵が提供されていない。
KRB5_PREAUTH_FAILED	-1765328175L	事前認証が失敗した (一般的なエラー)。
KRB5_RCACHE_BADVNO	-1765328174L	リプレイのキャッシュの形式のバージョン番号がサポートされていない。
KRB5_CCACHE_BADVNO	-1765328173L	資格キャッシュの形式のバージョン番号がサポートされていない。
KRB5_KEYTAB_BADVNO	-1765328172L	鍵テーブルの形式のバージョン番号がサポートされていない。
KRB5_PROG_ATYPE_NOSUPP	-1765328171L	プログラムがアドレス型をサポートしていない。
KRB5_RC_REQUIRED	-1765328170L	メッセージのリプレイ検出が rcache パラメータを要求した。

マイナー状態	値	意味
KRB5_ERR_BAD_HOSTNAME	-1765328169L	ホスト名を標準化できない。
KRB5_ERR_HOST_REALM_UNKNOWN	-1765328168L	ホスト用のレルムを決定できない。
KRB5_SNAME_UNSUPP_NAMETYPE	-1765328167L	名前型におけるサービスプリンシパルへの変換が定義されていない。
KRB5KRB_AP_ERR_V4_REPLY	-1765328166L	初期チケットの応答が Version 4 のエラーを示している。
KRB5_REALM_CANT_RESOLVE	-1765328165L	要求されたレルムに対して KDC を解釈処理できない。
KRB5_TKT_NOT_FORWARDABLE	-1765328164L	要求しているチケットは転送可能なチケットを取得できない。
KRB5_FWD_BAD_PRINCIPAL	-1765328163L	(資格の転送中) プリンシパル名が不良である。
KRB5_GET_IN_TKT_LOOP	-1765328162L	`krb5_get_in_tkt` 内でループが検出された。
KRB5_CONFIG_NODEFREALM	-1765328161L	Kerberos 構成ファイル `/etc/krb5/krb5.conf` がデフォルトのレルムを指定していない。
KRB5_SAM_UNSUPPORTED	-1765328160L	`obtain_sam_padata` の SAM フラグが不良である。
KRB5_KT_NAME_TOOLONG	-1765328159L	鍵タブ名が長すぎる。

マイナー状態	値	意味
KRB5_KT_KVNONOTFOUND	-1765328158L	鍵テーブル内のプリンシパルの鍵バージョン番号が間違っている。
KRB5_CONF_NOT_CONFIGURED	-1765328157L	Kerberos 構成ファイル `/etc/krb5/krb5.conf` が構成されていない。
gERROR_TABLE_BASE_krb5	-1765328384L	デフォルト

用語集

ACL
「アクセス制御リスト (ACL)」の項を参照してください。

GSS-API
Generic Security Service Application Programming Interface の略。さまざまなモジュール方式のセキュリティサービスのサポートを提供するネットワーク層です。GSS-API はセキュリティ認証、整合性、および機密性のサービスを提供します。さらに、セキュリティに関連して、アプリケーションの移植性を最大限にすることを可能にします。「認証」、「機密性」、「整合性」の項も参照してください。

MIC
「メッセージ整合性コード (MIC)」の項を参照してください。

MN
「機構名 (MN)」の項を参照してください。

QOP
「保護品質 (QOP)」の項を参照してください。

アクセス制御リスト **(ACL)**
特定のアクセス権を持つプリンシパルのリストが格納されているファイル。通常、サーバーはアクセス制御リストを調べて、クライアントがサービスを使用するための権限を持っているかどうかを判断します。GSS-API で認証されていても ACL で許可されていなければ、プリンシパルはサービスを拒否される可能性があることに注意してください。

委託
実際のセキュリティ機構で許可されている場合、プリンシパル (通常はコンテキスト起動側) は、自分の資格とピアとなるプリンシパル (通常はコンテキスト受け入れ側) に「委託」することで、ピアプリンシパルをプロキシに指定できます。「委託」された資格を使用すると、ピアプリンシパルはオリジナルプリンシパルの代わりに

要求を行うことができます。たとえば、プリンシパルが rlogin を使用して、あるマシンから別のマシンにリモートログインする場合などです。

エクスポート名　gss_export_name() で GSS-API 内部形式 (特に機構名) から GSS-API エクスポート形式に変換された名前。エクスポート名は memcmp() で GSS-API 以外の文字列形式と比較できます。「機構名 (MN)」と「名前」の項も参照してください。

機構　データの認証や機密性を実現するための暗号化技術を指定するソフトウェアパッケージ。たとえば、Kerberos v5 や Diffie-Hellman 公開鍵など。

機構名 **(MN)**　GSS-API 内部形式名の特別なインスタンス。通常の GSS-API 内部形式名では 1 つの名前に対して複数のインスタンス (それぞれが実際の機構の形式での) を持つことができますが、機構名は特定の機構に一意です。機構名は gss_canonicalize_name() で生成されます。

機密性　データを暗号化するセキュリティサービス。機密性には整合性と認証のサービスも含まれます。「認証」、「整合性」、「サービス」の項も参照してください。

クライアント　狭義では、ユーザーの代わりにネットワークサービスを使用するプロセスを指します。たとえば、rlogin を使用するアプリケーションなどです。サーバー自身が他のサーバーやサービスのクライアントになる場合もあります。広義では、サービスを使用するプリンシパルを指します。

誤順序の検出　多くのセキュリティ機構では、メッセージストリーム中のメッセージが不適切な順序で受信されたことを検出できます。メッセージの誤順序の検出は、(利用できる場合は) コンテキスト確立時に要求する必要があります。

コンテキスト　2 つのアプリケーション間の「信用の状態」。2 つのピア間でコンテキストが正常に確立されると、コンテキスト受け入れ側はコンテキスト起動側が本当に主張しているとおりのアプリケーションであることを認識して、コンテキスト受け入れ側に送信されたメッセージを検証および復号化できます。コンテキストに相互認証が含まれている場合、起動側は受け入れ側の ID が有効であると認識して、

受け入れ側から送信されたメッセージを検証および復号化できます (復号化は任意)。

コンテキストレベルトークン　「トークン」を参照してください。

サーバー　ネットワーククライアントにリソースを提供するプリンシパル。たとえば、boston.eng.acme.com というマシンに rlogin する場合、そのマシンは rlogin サービスを提供するサーバーです。

サービス
1. (ネットワークサービスと同意)。ネットワーククライアントに提供されるリソース。複数のサーバーによって提供されることもあります。たとえば、boston.eng.acme.com というマシンに rlogin する場合、そのマシンは rlogin サービスを提供するサーバーです。
2. セキュリティサービスは整合性または機密性のサービスであり、認証以上の保護レベルを提供します。「認証」、「整合性」、「機密性」の項も参照してください。

資格　プリンシパルを識別する情報パッケージ。プリンシパルの「識別バッジ」であり、プリンシパルが誰であるか (そして、多くの場合、プリンシパルがどの特権を持っているか) を示します。資格はセキュリティ機構によって生成されます。

資格キャッシュ　指定された機構によって保存された資格を保持するための保存領域 (通常はファイル)。

承認　プリンシパルがサービスを使用できるかどうか、プリンシパルがどのオブジェクトにアクセスできるか、および、各オブジェクトにどのようなアクセスの種類が許可されているかを決定するプロセス。

整合性　ユーザー認証に加えて、転送されたデータの有効性を暗号タグで証明するセキュリティサービス。「認証」、「機密性」、「メッセージ整合性コード (MIC)」の項も参照してください。

セキュリティサービス　「サービス」の項を参照してください。

セキュリティフレーバ　「フレーバ」の項を参照してください。

セキュリティ機構　「機構」の項を参照してください。

相互認証

コンテキストが確立されたとき、コンテキスト起動側は自分自身をコンテキスト受け入れ側に認証する必要があります。また、それに加えてコンテキスト起動側が受け入れ側の認証を要求する場合もあります。受け入れ側にも認証が必要な場合、両者は相互認証されていると言います。

データ型

データの形式。たとえば、int、string、gss_name_t 構造体、gss_OID_set 構造体など。

データリプレイ

データリプレイは、メッセージストリーム内の単一のメッセージが複数回受信された場合を指します。多くのセキュリティ機構でデータリプレイの検出をサポートしています。リプレイの検出は、(利用できる場合は) コンテキスト確立時に要求する必要があります。

トークン

GSS-API 構造体 gss_buffer_t の形式であるデータパケット。トークンは、ピアとなるアプリケーションへの転送用に、GSS-API 関数で生成されます。

トークンには 2 種類あります。コンテキストレベルトークンには、セキュリティコンテキストを確立または管理するために使用される情報が格納されます。たとえば、gss_init_sec_context() は、コンテキストの受け入れ側に送信するための、コンテキスト起動側の資格ハンドル、ターゲットマシンの名前、要求されるさまざまなサービスのフラグなどをトークンに格納します。

メッセージトークン (メッセージ毎トークンやメッセージレベルトークンとも呼びます) には、ピアとなるアプリケーションに送信されるメッセージから GSS-API 関数によって生成された情報が格納されます。たとえば、gss_get_mic() は、指定されたメッセージから識別用の暗号タグを生成し、ピアに送信されるトークンに (メッセージと一緒に) 格納します。技術的には、トークンはメッセージとは別であると考えられています。このため、gss_wrap() は *output_token* ではなく *output_message* を生成すると言われます。

「メッセージ」の項も参照してください。

不透明性

「不透明」の項を参照してください。

不透明 **(**参照できない**)**

データの値や形式がそれを使用する関数には見えない場合、そのデータは不透明であると言います。たとえ

ば、gss_init_sec_context() への *input_token* パラメータはアプリケーションには不透明ですが、GSS-API にとっては重要です。同様に、gss_wrap() への *input_message* パラメータは GSS-API には不透明ですが、ラップを行うアプリケーションにとっては重要です。

名前

プリンシパルの名前。たとえば、「joe@machine」などです。GSS-API の名前は gss_name_t 構造体を通じて処理されます。このような名前はアプリケーションには不透明です。「エクスポート名」、「機構名 (MN)」、「名前型」、「プリンシパル」の項も参照してください。

名前型

名前の形式。名前型は gss_OID 型として格納され、名前に使用されている形式を示します。たとえば、「joe@machine」という名前の名前型は GSS_C_NT_HOSTBASED_SERVICE であるなどです。「エクスポート名」、「機構名 (MN)」、「名前」の項も参照してください。

認証

要求されたプリンシパルの ID を確認するセキュリティサービス。

プライバシ

「機密性」の項を参照してください。

プリンシパル

ネットワーク通信に参加する、一意な名前を持つ「クライアント/ユーザー」または「サーバー/サービス」のインスタンス。GSS-API ベースのトランザクションにはプリンシパル間の対話が含まれます。次に、プリンシパル名の例を示します。

- joe
- joe@machine
- nfs@machine
- 123.45.678.9
- ftp://ftp.company.com

「名前」と「名前型」の項も参照してください。

フレーバ

従来、フレーバは認証の種類 (AUTH_UNIX、AUTH_DES、AUTH_KERB) を示していたため、セキュリティフレーバと認証フレーバは同じ意味です。RPCSEC_GSS もセキュリティフレーバですが、これは認証に加えて、整合性と機密性のサービスも提供します。

保護品質 **(QOP)**
: QOP は Quality of Protection の略で、整合性や機密性のサービスと一緒に使用される暗号化アルゴリズムを選択するときに使用されるパラメータ。整合性と一緒に使用する場合、QOP はメッセージ整合性コード (MIC) を生成するアルゴリズムを指定します。機密性と一緒に使用する場合、QOP は MIC の生成とメッセージの暗号化の両方に対するアルゴリズムを指定します。

ホスト
: ネットワークを通じてアクセス可能なマシン。

メッセージ
: GSS-API ベースのアプリケーションからそのピアとなるアプリケーションに送信される `gss_buffer_t` オブジェクト形式のデータ。たとえば、「`ls`」はリモートの `ftp` サーバーにメッセージとして送信されます。

  メッセージには、ユーザーが提供するデータ以外の情報が格納されることもあります。たとえば、`gss_wrap()` はラップされていないメッセージを受け取り、そのメッセージを送信用にラップします。このとき、ラップされたメッセージには、オリジナル (ユーザーが提供した) メッセージとともにその MIC が格納されます。メッセージは格納しない、GSS-API が生成した情報は「トークン」と呼ばれます。詳細は「トークン」の項を参照してください。

メッセージ整合性コード **(MIC)**
: Message Integrity Code。データの有効性を保証するために、転送されるデータに添付される暗号タグ。データ受信側は独自の MIC を生成し、送信された MIC と比較します。両者が同じ場合、メッセージは有効です。`gss_get_mic()` で生成される MIC などはアプリケーションからも見えますが、`gss_wrap()` や `gss_init_sec_context()` で生成される MIC などはアプリケーションからは見えません。

メッセージ毎トークン
: 「トークン」の項を参照してください。

メッセージレベルトークン
: 「トークン」の項を参照してください。

リプレイの検出
: 多くのセキュリティ機構は、メッセージストリーム中のメッセージが不正に繰り返されたことを検出できます。メッセージリプレイの検出は、(利用できる場合は) コンテキスト確立時に要求する必要があります。

索引

K

M

O

Q

R

S

あ

い