GUMP
Grand Unified Meta-Protocols
Recipes for Simple, Standards-based Financial Cryptography

Barbara Fox
Brian Beckman
Appendix by Dan Simon

Microsoft Corporation
February 1997

Abstract. In this paper, we present a set of simple, all-parties-authenticated application protocol frameworks appropriate for a wide variety of financial applications running on the Internet. Collectively, we call these frameworks "GUMP", for Grand Unified Meta-Protocols. The driving goal of the design is simplicity, so as to reduce dramatically the cost of engineering and deployment of application protocols. The simplicity of GUMP follows directly from a number of business-level premises, chief of which is that the client must digitally sign all transactions.

One builds an application protocol from GUMP by "filling in the blanks" with custom business data types and logic. In that sense, GUMP is a set of frameworks, templates, or meta-protocols. The goal of this paper is not to engineer protocols, but to describe abstractly how they might be straightforwardly engineered, concentrating on the *authentication* phases common to most, if not all, financial protocols. The applications may include home banking, purchasing, bill payment, securities trading—any application that requires client-server mutual authentication and integration with legacy systems.

While many of the points in this paper may seem embarrassingly simple and obvious, that is, in fact, the point. In the design of public-key protocols each design team inexorably ends up inventing nearly the same primitive notions. Since no team can afford the time to abstract general frameworks, these protocols end up being virtual collections of special cases. Furthermore, the written specifications, again due to time pressure, frequently do not carefully distinguish between requirements, high-level design, and deep details, mixing them all together in one, swirling description. The really hard problem then falls to the implementors whose job it becomes to translate complex protocol design into simple working and interoperable code.

GUMP is our attempt to provide a greatly simplified abstract toolkit for the protocol

engineer. We present three application protocol prototypes—*Registration*, *Transaction*, and *Delegation*—based on the pending IETF TLS (Transport-Layer Security) Protocol, which is based on Netscape's widely deployed SSL (Secure Sockets Layer). The GUMP Registration meta-protocol assumes the password (shared-key) extensions to TLS as proposed to the IETF working group and documented in the Appendix. These extensions protect a GUMP one-time shared secret that the server uses to authenticate a certification request. The rest of the protocols make minimal usage of cryptography beyond digital signatures. All leverage the client-authentication feature of SSL version 3.

The contributions of this paper include:

1. Reduction of multiple financial account relationships to a single *unsecret*, which, when certified along with a public key, supports authentication without secrecy.
2. A new class of Internet-safe transactions with *delegation*, where a member of an access group may give permission to an agent to initiate a transaction on his behalf.

1 Background

A growing number of financial applications are migrating to the Internet, where their designers are attempting not only to duplicate existing functions, but also to improve them along the way. Some applications, like home banking and bill payment, are already commonplace in other electronic embodiments. Others are newer, like brokerage account management and business-to-business purchasing. They all, however, have certain requirements in common:

- Privacy
- Mutual authentication
- Integrity
- Non-repudiation
- Coexistence with legacy systems

Fortunately, at this stage of Internet evolution, public-key cryptography is becoming ubiquitous. Virtually all web servers and browsers support secure-channel protocols and public-key authentication, at least of the server. With the recent deployment of SSL (Secure Sockets Layer protocol) version 3 [1], public-key authentication of the client is now also widely available.

In this paper, we acknowledge that SSL is a *de facto* standard for secure HTTP connections—there are approximately 40 million browsers and servers capable of running it. We also assume that SSL's base technology will continue to evolve as TLS (Transport Layer Security) [2] within the IETF. As a natural progression, standardized, interoperable application protocols built on TLS will follow. GUMP is a first attempt to propose a framework for these application protocols.

Specifically, we describe:

1. Authenticating to a legacy authenticator with passwords or PINS without exposing them over a 40-bit encryption channel.
2. A generic, fill-in-the-blanks financial application suite that leverages TLS client authentication to accommodate *financial-institution relationship certificates*, which should be distinguished from generic *identity certificates*. The latter have no binding to a relationship between a client and a financial institution.
3. A non-cryptographic means for "Internet-safe" transactions with *delegation*, building on the relationship certificate.

It is a primary design goal that an entire GUMP application suite be buildable with "off-the-shelf" components. For financial applications especially, starting with well understood, mature, standards-based secure protocols such as SSL/TLS has the obvious advantages of:

- Interoperability and broad, cross-platform installed base
- Reduced cost to understand, analyze, and approve
- Reduced cost to design, implement, integrate, and deploy
- Built-in evolution path through the IETF's TLS standards process
- Presumption that the cryptographic bedrock like random number generation, key exchange, and authentication are done correctly

Our main message is that the application protocol frameworks illustrated here involve only small, incremental changes to what is already a ubiquitous platform. For most application purposes, high-quality cryptographic solutions are now cookbook-simple.

More formally, the following premises drive our requirements analysis and design:

1. HTTP is, by far, the most important transport for Internet financial applications for the foreseeable future. TLS, in turn, with the proposed extensions and with embedded signed documents can service virtually all security requirements of financial applications, when those requirements are sufficiently simplified. SSLv3 provides mutual authentication and moderate privacy. Embedded signed documents provide non-repudiation for transactions and support store-and-forward backends. The proposed extensions to TLS provide secure shared-secret authentication to support registration.
2. Transactions are fundamentally two-party affairs. Multi-phase, multiplexed, staged, forwarded, and nested transactions can all be designed out of two-party elements.
3. Public-key authentication protocols are easier to implement and deploy than are shared-secret authentication protocols. Use of shared-secret protocols should be minimized and eliminated where possible. Nevertheless, integration with legacy systems will require their use in some circumstances.

4. Clients should sign all transactions digitally. With signatures, a transaction protocol has non-repudiation and freedom from managing secret authentication data. Account numbers, PINs, passwords, and so on become worthless to an adversary since s/he cannot use them without also creating a valid digital signature, assuming replay protection. The adversary cannot create a valid signature without controlling the client's private key. This premise implies that clients must have certified signature keys.
5. Financial institutions will (or should) insist on managing their own client name spaces. In a public-key setting, this implies that they will issue their own certificates for client signature keys, since they need to bind their own attributes—like account numbers—to principals they recognize.

2 Financial Applications: Basics

Assume, as a baseline, that clients of any financial application demand a private channel, insist on mutual authentication, and agree to sign *every* transaction. Of these, authentication is the least understood. In the paper-and-ink world, possession of a State document—presumed uncopyable—often suffices, but this is not possible in the electronic world where anything can be copied. In the electronic world, Alice authenticates Bob by getting him to prove *current possession* of one or more secrets: a password, PIN, shared secret key, or the private half of a public-private key pair. In financial protocols, and especially on the Internet, requirements for each of these types often coexist within the same session. We use the term ***certification*** to mean establishing the means of this proof so that authentication can proceed later on in a manner mutually acceptable to client and financial institution. First, we describe a traditional, non-electronic certification protocol as a prelude to describing GUMP's meta-protocols.

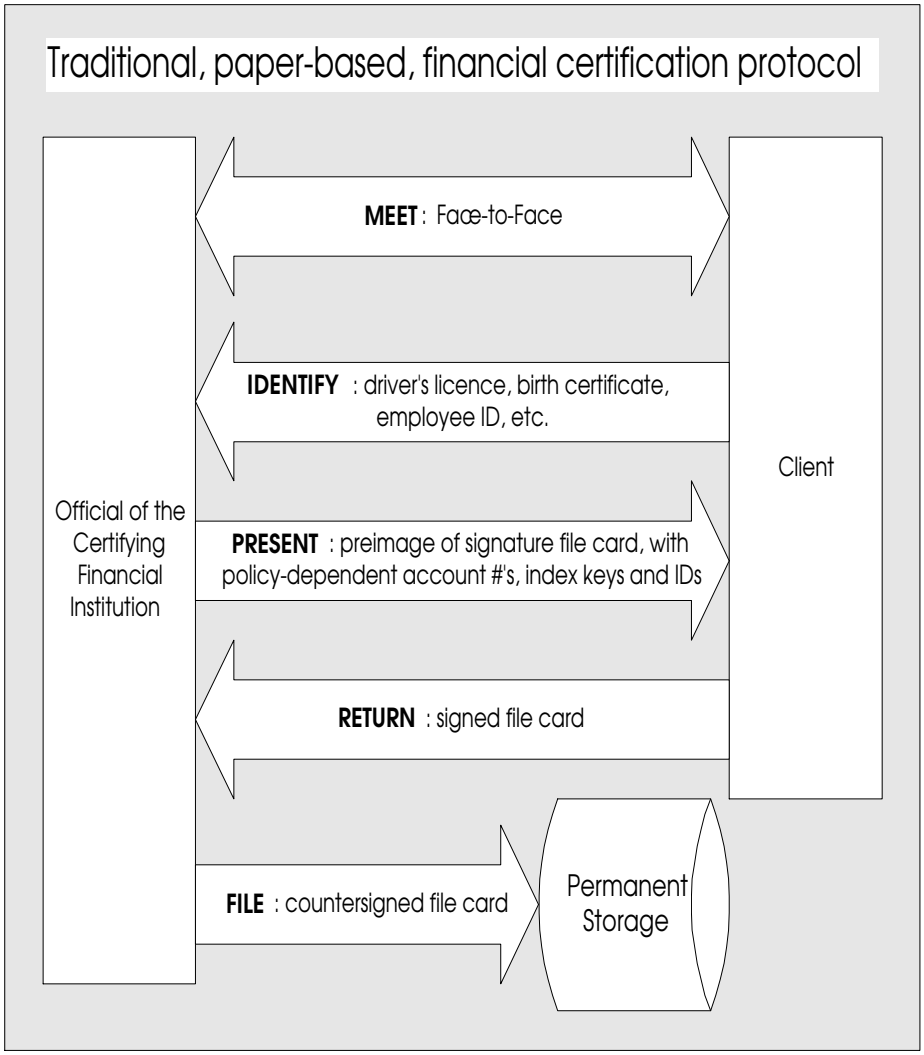
2.1 Traditional certification protocol

A typical client consumes a number of products and services provided by a financial institution. The client may have demand-deposit accounts, mortgages, unsecured loans, annuities, insurance accounts, credit cards, debit cards, check cards, ATM cards, brokerage accounts, and so on at a single institution. The institution and client agree on an account number for each distinguishable product or service, but the set of account numbers are bound to a single client identity—indeed, they *constitute* the client identity from the point of view of the institution. No mere name and address, etc, would suffice without the account numbers. This definition of identity benefits both the client and the institution. The client will have freedom of movement among the accounts and the institution can track the client's activities for legal and business purposes.

The traditional certification protocol goes roughly as follows:

1. **MEET**: The client and an official of the financial institution meet face-to-face.
2. **IDENTIFY**: The client provides satisfactory proof of identity—*at least* as defined by the State—to the official of the institution. This proof may be in the form of a birth certificate, driver's license, employer picture ID, and so on.
3. **PRESENT**: The official copies some of the identity information plus some information of his own to a paper signature card and presents the card to the client for signature (we call the card a *preimage* at the moment, since it is not signed, in anticipation of the digital analog of this protocol). The information on the card is a matter of institution policy, but must contain a master account number or some other kind of primary index key into the institution's client database.
4. **RETURN**: The client signs the paper signature card with ink and gives it back to the official. Perhaps the client signs one card for each account. More interestingly from our perspective, *the client may sign a single card that contains all the client's account numbers*.
5. **FILE**: The official of the institution countersigns the card and files it, making it a ***signature-card-on-file (SOF)***

The following figure illustrates this protocol



After this certification protocol, the financial institution accepts transaction orders only when accompanied by a fresh paper-and-ink client signature that an official of the institution can compare against the SOF. The institution uses one SOF for a whole collection of accounts, that is, for a whole collection of products and services. The SOF becomes the institution's permanent record of ***evidence of prior authentication.***

We also use the term 'authentication' to refer to the later comparison of signatures on transaction orders against the SOF, noting that authentication, in this latter usage, is authentication of the transaction, not of the client him- or herself.

The security of the traditional protocols follows from the following assumptions:

1. Paper-and-ink signatures are hard to forge. Only professional criminals can even attempt forgery. Casually counterfeit transactions are not feasible.
2. The official of the financial institution can be trained to detect forgeries.
3. The client is not liable for signatures obtained 'at gun-point', and so on. Business case law applies myriad other restrictions and remedies to signed transactions.
4. No transactions are accepted without a signature and a signature check.

The most important is the last: it means that *account numbers are worthless to an adversary who cannot forge a signature*.

2.2 Foundations for GUMP

This observation was the foundation for GUMP. We asked ourselves whether we could avoid the cryptographic, infrastructural, and engineering complexities that follow from attempting to hide numbers from adversaries on the Internet. The answer is only if the numbers have no value. For example, to a first approximation, the only reason a credit card number is valuable to an adversary is that s/he may use the number to buy things without a signature via MOTO (Mail-Order Telephone-Order). If a digital signature and signature check were required on every transaction, then the card number alone would have no value. Now, to a second approximation, there are other good reasons to keep a credit card number secret against other protocol hazards. Forgery might be attempted, or the bank might never check the signature, might not require the merchant to pass signed draft slips back for signature check, and so on. However, none of these hazards exist with public-key cryptography. It is mathematically infeasible for anyone to forge a digital signature, and signatures may be cheaply checked by anyone, not just the financial institution

The latter point is another fundamental difference between paper-and-ink signatures and digital signatures. The financial institution must keep the SOF secure against substitution or disclosure, so the institution becomes a central bottleneck for signature checking. Since it is not feasible to check every transaction at the institution, various systems have various ways of bypassing the check. A credit card, for example, has a tamper-resistant signature block on the back, and merchants are supposed to check it

against the signed draft. Essentially, the cardholder's bank delegates the signature check to the merchant in this case, at the risk of the back-of-card being modified by clever hackers. However, in the case of public-key digital signatures, anyone who trusts a given CA root key can perform foolproof verification of any signature. These principles underly the design of the Visa/MasterCard Secure Electronic Transaction (SET) protocol [5] which is specifically tailored for encrypted credit card transactions between a consumer, an untrusted merchant, and a trusted gateway. GUMP, unlike SET, assumes a generic, two-party transaction where no valuable account number is required and authentication rather than encryption insures integrity. While not appropriate for credit card transactions where the consumer-merchant relationship is spontaneous and transient, GUMP could apply to the large class of financial transactions that depend on a trusted and on-going relationship; for example, between a customer and his bank.

In fact, the following GUMP protocol examples are intended to be electronic equivalents of common bank practices. The only twist is the use of a unique, but not secret number, in every signed transaction.

3 GUMP Registration Meta Protocol

The Internet analog of a SOF is a ***certified public signature key (CPSK)***. ***The GUMP Registration Meta-protocol (GRMP)*** is a framework for designing and implementing a financial institution's certification policies that result in a client's CPSK, packaged as a ***GUMP relationship certificate (GRC)***. The GRC, of course, is public information that can be sent with transaction packets, stored in online directories, cached on distributed machines, and so on.

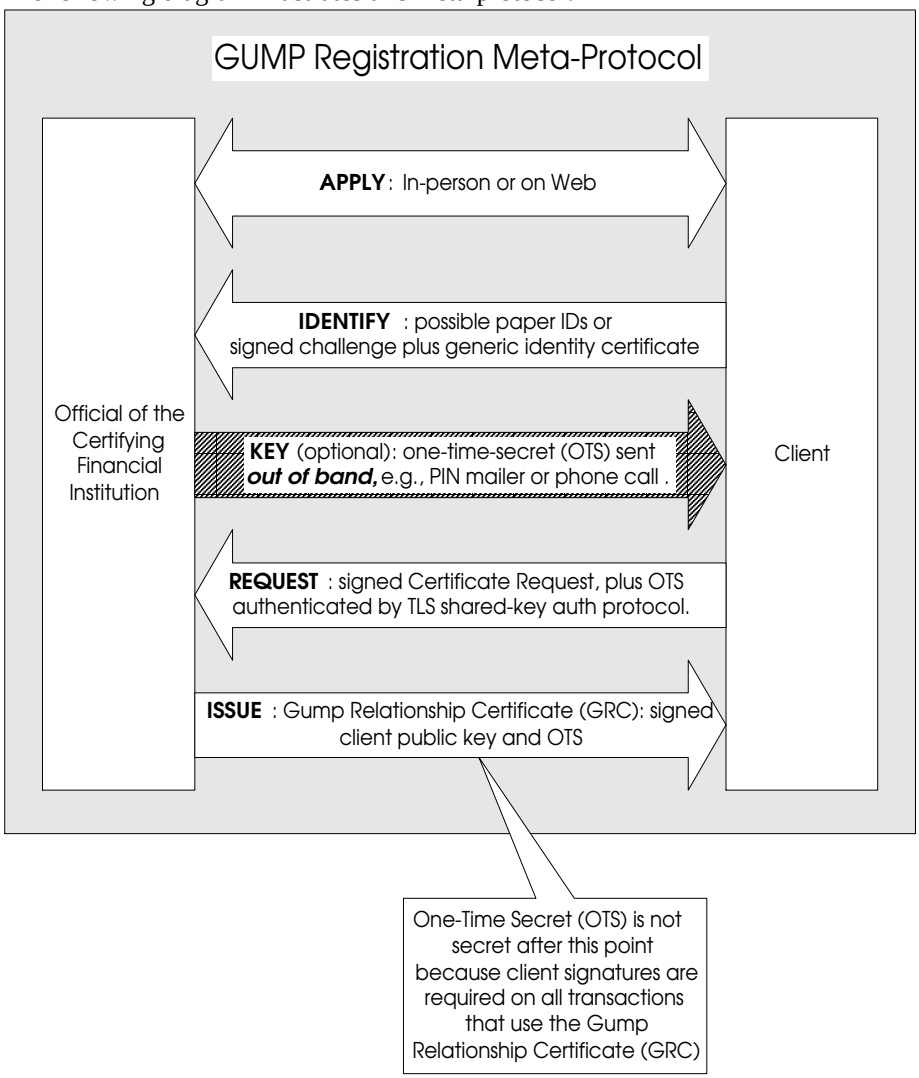
The GRMP has the following steps:

1. **APPLY**: The client applies for a service either in person on through the financial institution's web site.
2. **IDENTIFY**: The client provides satisfactory proof of identity to the official of the institution. GUMP allows maximal freedom for policy choices by the institution:
   - 2.1. In the case of face-to-face certification, identification might be showing State documents, as in the traditional protocol.
   - 2.2. In the case of electronic identification, the institution might require a signature on a challenge with a verification key from a generic identity certificate, such as one from Verisign or the US Postal Service.
3. **KEY (optional step)**: The official of the institution gives the client a ***one-time secret (OTS)*** out-of-band, for example, in a PIN mailer. We propose, here, that this secret be the hash of all the account numbers that will be serviced by the GRC. This proposal allows the institution and client to check one another: each can independently compute the hash. However, the exact form of the OTS is not very important. From the protocol standpoint, it must simply be unique to the institution issuing it. From the system design standpoint, it will most likely be an

index key that the institution will use to lookup client information. This step can be skipped, as a matter of policy, if the client is separately authenticated as in 2.2; the institution may simply issue the OTS in the GRC in step 5.

4. **REQUEST**: This is the first step that is always online. The client digitally signs and submits a PKCS#10 (or other suitable form of a…) ***Request for Certification (RFCert)***, which contains a proposed public signature key, and securely proves possession of the OTS via the TLS shared-key authentication protocol in the Appendix. The digital signature, at this point, proves only that the client controls the as-yet-uncertified private signature key.
5. **ISSUE**: The institution digitally signs and sends back a GRC binding the client's public signature key to the OTS. From this point on, the OTS is *no longer secret*. It is simply a public attribute, certified by the institution, of the client's public signature key. This is the reason we call it the ***unsecret.*** An adversary cannot use the OTS without also taking control of the client's private signature key, which we make as difficult as possible.

The following diagram illustrates this meta-protocol**:**



The GRC is the analog of the SOF—Signature-card-on-file. The institution accepts no transactions without a digital signature, which an official of the institution (or his computer programs) can compare against the SOF.

We note, with emphasis, that *step 3 is the only part of the entire proposal that requires modification of standard, off-the-shelf technology*. The proposed modification of TLS [3] can be used not only for the GUMP unsecret, but much more broadly for passwords, PINs and other secrets. These modifications avoid sending passwords, PINs, or other secret authentication data over weakly encrypted channels, like those approved for export from the US. Instead, they pass a quantity that any party possessing the shared secret can recompute, but that no one else can. The quantity is essentially the hash of the shared secret, though other data linking the quantity to the current session and to established names of both parties is included in the hash. Details may be found in the appendix

3.1 Mutual Authentication without Secrecy

One innovation of this paper is the unsecret number, which ties the public key in the GRC to all client relationships with the financial institution. The number is useless to an adversary who does not control the client's private key. We may now see why encryption of this number is not needed so long as all transactions are signed. Before certification, the client satisfies the institution's certification policy. After certification, the client satisfies GUMP's authentication policy, which is simply to sign, digitally, a transaction instrument containing a freshness challenge, proving current control of the private signature key corresponding to the public key in the GRC. This public key is bound to the unsecret by the institution's signature on the relationship certificate.

The cryptographically savvy have already recognized that the pubic key alone is a perfectly good "unsecret". The whole exercise of hashing account numbers is to give the client and the institution a non-random number they can independently control,and to reduce the unsecret to 20 bytes or less. For example, the unsecret might be a user or group denoting access permissions in an ACL. More likely, it will be a primary database key, so some thousand-odd bits of composite bignum would not be helpful in this regard. To be fair, the substitution of any agreed-upon datum would make an equally good unsecret. It is simply a local name.

3.2 The Identity and Relationship Certificates

SSLv3.0 includes both server and client authentication. A browser client routinely authenticates a server as a part of the Server Hello. With version 3.0, the server may require the client to sign a challenge and present a certified public signature key that verifies the signature on the challenge. Typically, to use this feature, the client gets an identity certificate from a commercial or government certificate authority, such as

Verisign or the Postal Service, and stores it someplace convenient, like a browser database or a directory service. The financial institution certification policy (in steps 2 and 4 above) may accept the identity certificate in place of traditional, show-your-face-and-papers identification. So, why still issue a relationship certificate binding the client's public signature key to the unsecret? The answer is for *local control of names*. If the institution were to rely on someone else's identification and naming conventions to conduct its business or index its databases, it would have to internalize or map those naming conventions to its own legacy systems. If, however, the institution issues its own certificates, then it can bind the client's public key to the most convenient identifier. The unsecret does not need to be globally unique, it does not need to be checked by anyone else, it does not have any external requirements whatsoever. It is simply the institution's own local "name" for the client.

The GUMP relationship certificate then, is a proxy for a business relationship that can be resolved to a single person and any number of accounts.

4 GUMP Transaction Meta Protocol

Transactions in GUMP are incredibly simple and freely configurable. A client may conduct a GUMP transaction with any entity that trusts the root key of the GRC signature chain. Let us call that entity the *merchant*, just for the sake of discussion. It may be the institution itself or it may be any number of delegates or associates who trust a signature verification chain that includes the GRC and its signature keys.

The client and the merchant may conduct arbitrary negotiations prior to the client's signing a ***transaction instrument (TI)***. We advocate that the client authenticate the merchant via SSL/TLS. We further advocate that the merchant authenticate the client via SSL/TLS client authentication. In the latter, the merchant may request the GRC rather than the prosaic identity certificate, because the financial institution either issued or understands the unsecret number. However, GUMP does not require SSL/TLS client auth nor that client auth, if used, employ the GRC rather than the identity cert.

When negotiations are closed and the client is ready to commit the transaction, GUMP requires the client to construct an instrument signed with the (private half of the) client key, and to embed the signed instrument with a valid GRC that the merchant trusts into the SSL/TLS stream. Embedded S/MIME might be a suitable format for such an embedded capsule. However, encryption should be avoided as it is not needed for GUMP, it is already provided by SSL/TLS, and embedded encryption creates ITAR product export clearance problems. The reason for the embedded, signed instrument is that TLS mutual authentication does not provide the merchant with a permanent, non-repudiable record of client commitment.

GUMP requires only that the instrument be replay proof, meaning that it contain a secure timestamp, a unique transaction identifier, or a large (128-bit, at least) random

freshness challenge that the client signs. It is a good idea for a transaction instrument to contain all three replay protections. Otherwise, structure and content of the instrument are free.

5 GUMP Delegation Protocol: Memberships

Again, Secure Electronic Transactions—SET—the Visa-Mastercard standard [5], addresses the problem of spontaneous credit card transactions on the Internet. The model assumes that the buyer and seller prefer to remain anonymous and that a virtual session between the client and the merchant's bank is necessary to protect the credit card number from malicious misuse by the merchant.

While this degree of caution is appropriate for the Internet in general and especially for transactions where the card number alone can be used in fraudulent, unsigned, no-card-present transactions, there exists a different category of transactions where the parties trust each other and, to some extent, transact frequently. Examples include business-to-business purchasing, frequent flier programs, and even sports club memberships. The common thread is that there is a contract—"of sorts" or formal—between the partners that explicitly lays out the rules, which are almost always:

> *Alice (the member) authorizes Bob (the club) to do something on a case-by-case basis*

A trivial example might be that Alice authorizes Bob to send her company the goods described on her purchase order—but only upon Bob's presenting her with a detailed invoice. The purchase is completed only when Alice signs and returns the invoice, acknowledging the terms of sale and completing the commitment to buy.

The relationship between Alice and Bob has the following characteristics:

1. They trust each other, or, at least they did once when they created their shared unsecret, meaning they have already shared a secret.
2. Because Bob already possesses Alice's secret, he can use it **on her behalf** without having her share it again; but,
3. The contract between them requires that Bob get Alice's (signed!) permission each time.

The essence of the matter is that Bob can negotiate for Alice, but only Alice can commit. So, Bob may have a ***GUMP Delegation Certificate (GDC)*** that binds his public key to Alice's unsecret, and Bob may use the GDC to establish mutually authenticated TLS negotiation sessions with Alice and others. But only Alice can sign a transaction instrument with her GRC.

The GDC may simply be a certificate issued by Alice for Bob, containing her

unsecret, his public key, and verifiable with her GRC. We see that expediency mitigates toward every client being a CA of sorts. This observation has been made by the authors of SDSI [4]. Indeed, we speculate that GUMP—in pure meta form (especially if the rest of Lisp or Scheme were added to SDSI) or in some concrete application—can be very easily implemented on top of SDSI.

In terms of the protocol, requesting and granting permission adds another step – but also has the advantage of **transparently** adding a third party to the transaction. The underlying concept is that this trust need not be transitive:

Alice authorizes Bob (whom she trusts) to use her unsecret to negotiate a transaction with Chuck (whom only Bob knows and trusts). Each time, Bob must present the permission and Alice must sign it.

6 Design and deployment : advantages of simplicity

High on the list of advantages of standards-based financial protocols is quick and painless deployment. Channel security (TLS) is quickly becoming requisite for business on the Internet, and the only change required to use all of the GUMP protocols is shared-key authentication—and this is already destined to become a part of the TLS standard. Better yet, secure-channel protocols and APIs to use them are being bundled with platform components like operating systems and browsers. In most cases, writing security protocols "from scratch" has no payback.

The rest of the problem then is design, and unfortunately, that's where most of the really serious mistakes are made.

Encryption vs. authentication: Confusion of these two operations tracks cryptography's relatively recent emergence as the darling of the business press. Encrypting is chic, but in many cases not at all necessary to accomplish the security objective. A combination of Hash functions and digital signatures can provide access control, message authentication, and data integrity --- all without secrecy --- as illustrated in this paper. Encryption should be avoided when not necessary, because it makes governments nervous.

Trust hierarchies: We advocate only two at most—the issuers of GUMP Relationship Certificates and the issuers of the identity certificates. These, in turn, should be either self-signed or two-deep. Users can easily install trusted keys of certificate authorities in their browsers, and with GUMP, multiple accounts with each entity are represented by a single relationship certificate containing the unsecret. The advantage of "flat" rather than "fat" trust trees is obvious: people - not computers - have to maintain them.

Key Management and security policy: Central to the success of any security design is how well keys are managed: generated, stored, refreshed and replaced upon loss

or compromise. While stringency of individual policies may vary based on requirements, their design and **test** is no less important than that of the software itself. In practice, security systems most often fail from the simple combination of carelessness or inadequate disaster planning. By contrast, the cryptography is the simple part.

Finally, the original objective of this paper was to stimulate good design and low-cost deployment of financial security solutions by supplying frameworks based on "off-the-shelf" standards. While we anticipate creative extensions, permutations, and of course, corrections as the GUMP set is actually implemented, our objective is achieved if a single GUMP --- either meta-protocol or concept --- finds its way into real financial applications.

7 Acknowledgements

Thanks to Daniel Simon, Yacov Yacobi, Rick Johnson, Mike Daly and Dipan Dewan for their valuable contributions to our work.

References

[1] A.O. Freier, P. Karlton, and P.C. Kocher, *The SSL Protocol: Version 3.0*, Mar. 1996. Internet Draft, ftp://ietf.cnri.reston.va.us/ internet-drafts/draft-freier-ssl-version3-%01.txt.

[2] A.O. Freier, P. Karlton, and P.C. Kocher, T Dierks. *The TLS Protocol Version 1.0*, Nov 1996. Internet Draft, ftp: ://ietf.cnri.reston.va.us/ internet drafts/draft-freier--tls-protocol-00.txt.

[3] D Simon, *Addition of Shared Key Authentication to Transport Layer Security (TLS)*,Nov 1996. Internet Draft, ftp: ://ietf.cnri.reston.va.us/ internet drafts/draft-simon-tls-passauth-00.txt

[4] Rivest, R. L. and Lampson, B., *"SDSI—A Simple Distributed Security Infrastructure"* http://theory.lcs.mit.edu/~rivest/sdsi10.html

[5] Visa and MasterCard, *Secure Electronic Transactions Protocol* (SET), August 1996, http://www.visa.com/ cgi-bin/vee/sf/set/intro.html?2+0, www.mastercard.com/set

APPENDIX

Shared Key Authentication for the TLS Protocol

1. Introduction

This document presents a shared-key authentication mechanism for the TLS protocol. It is intended to allow TLS clients to authenticate using a secret key (such as a password) shared with either the server or a third-party authentication service. The security of the secret authentication key is augmented by its integration into the normal SSL/TLS server authentication/key exchange mechanism.

2. Why Shared Key Authentication?

Recent transport-layer security protocols for the Internet, such as SSL versions 2.0 and 3.0 [1, 2] and PCT version 1 [3], have effected challenge-response authentication using strictly public-key (asymmetric) cryptographic methods, with no use of out-of-band shared secrets. This choice has both benefits and drawbacks. The primary benefit is improved security: an asymmetric private key used for authentication is only stored in one location, and the out-of-band identification necessary for public key certification need only be reliable, not secret(as an out-of-band shared key exchange must be). In addition, the difficult task of out-of-band shared-key exchange in shared-key authentication systems often leads implementers to resort to human-friendly shared keys (manually typed passwords, for instance), which may be vulnerable to discovery by brute force search or "social engineering".

However, shared-key authentication has certain advantages as well. These are, chiefly:

- Portability: Precisely because shared keys are often human-remembered passwords or passphrases, they can be transported from (trusted) machine to (trusted) machine with ease--unlike asymmetric private keys, which must be transported using some physical medium, such as a diskette or "smart card", to be available for use on any machine.

- Backward Compatibility: Shared-key authentication is in very wide use today, and the cost of conversion to its public-key counterpart may not be worth the extra security, to some installations.

- Established Practice: Shared-key authentication has been in use for quite a while, and a valuable body of

tools, techniques and expertise has grown up around it. In contrast, public-key authentication is very new, its associated tools and methods are either untested or non-existent, and experience with possible implementation or operation pitfalls simply doesn't exist.

These reasons are particularly relevant when individual human users of a service are being authenticated over the Internet, and as a result, virtually all authentication of (human) clients of such services is currently performed using shared passwords. Typically, servers implementing one of the aforementioned transport-layer security protocols, and needing client authentication, simply accept secure (i.e., encrypted and server-authenticated) connections from each client, who then provides a password (or engages in a challenge-response authentication protocol based on a password) over the secure connection to authenticate to the server.

Unfortunately, such "secure" connections are often not secure enough to protect passwords, because of the various international legal restrictions that have been placed on the use of encryption. Obviously, secret keys such as passwords should not be sent over weakly encrypted connections. In fact, even a challenge-response protocol which never reveals the password is vulnerable, if a poorly chosen, guessable password is used; an attacker can obtain the (weakly protected) transcript of the challenge-response protocol, then attempt to guess the password, verifying each guess against the transcript.

However, it is possible to protect even badly-chosen passwords against such attacks by incorporating shared-key authentication into the transport-layer security protocol itself. These protocols already involve the exchange of long keys for message authentication, and those same keys can be used (without the legal restraints associated with encryption) to provide very strong protection for shared-key-based challenge-response authentications, provided that the mechanism used cannot be diverted for use as a strong encryption method. This latter requirement makes it essential that the shared-key-based authentication occur at the protocol level, rather than above it (as is normally the case today), so that the implementation can carefully control use of the long authentication key.

3. Protocol Additions

Starting from SSL version 3.0 notation and formats, the following three new HandshakeTypes are added, and included in the Handshake message definition:

shared_keys(30),shared_key_request(31),
shared_key_verify(32)

A new CipherSuite is also included, to allow the client to signal support for shared-key authentication to the server:

```
TLS_AUTH_SHARED_KEY = {x01, x01};
```

The client's inclusion of this CipherSuite is independent of other listed CipherSuites, and simply indicates to the server the client's support for shared-key authentication.

3.1 SharedKeys message

The SharedKeys message has the following structure:

```
struct {
     DistinguishedName auth_services_client<1..65535>;
} SharedKeys;
```

This optional message may be sent by the client immediately following the ClientHello message; in fact, if sent, it is actually enclosed within the ClientHello message, immediately following the last defined field of the ClientHello message. (For forward compatibility reasons, the SSL 3.0 ClientHello message is allowed to contain data beyond its defined fields, and because there is no ClientHelloDone message, the server cannot know that an extra message follows the ClientHello unless it is actually included in the ClientHello message itself. A server that does not support shared-key authentication will simply ignore the extra data in the ClientHello message.) Although enclosed within the ClientHello, the SharedKeys message retains the normal structure and headers of a Handshake message.

The SharedKeys message contains a list of distinguished names of authentication services to which the client is willing to authenticate. This list need not be exhaustive; if the server cannot find an acceptable authentication service from the list in the SharedKeys message, then the server is free to reply with a list of acceptable services in a subsequent SharedKeyRequest message.

In cases where pass-through authentication is used, this message allows clients to be able to notify servers in advance of one or more authentication services sharing a key with the client, so that the server need only fetch (or use up) a challenge from a single service for that client. This message may also be useful in non-pass-through situations; for example, the client may share several keys with the server, associated with identities on different systems (corresponding to different "authentication services" residing on the same server). If a server receives a SharedKeys message, then any

subsequent SharedKeyRequest message can contain a single authentication service selected from the client's list.

Note that sending a SharedKeys message does not in itself normally reveal significant information about the client's as-yet-unspecified identity or identities. However, if information about the set of authentication services supported by a particular client is at all sensitive, then the client should not send this message.

3.2 SharedKeyRequest message

The SharedKeyRequest message has the following structure:

```
struct {
         DistinguishedName auth_service_name;
         opaque display_string<0..65535>;
         opaque challenge<0..255>;
} AuthService;

struct {
     AuthService auth_services_server<1..65535>;
} SharedKeyRequest;
```

This optional message may be sent immediately following the server's first set of consecutive messsages, which includes the ServerHello and (possibly) the Certificate, CertificateRequest and ServerKeyExchange messages, but before the ServerHelloDone message. The auth_services_server field contains a list of distinguished names of shared-key authentication services by which the client can authenticate. The challenge field accompanying each authentication service name contains an optional extra authentication challenge, in case the server needs to obtain one from an authentication service for pass-through authentication. If none is required, then it would simply be an empty (zero-length) field. Similarly, the display_string field may contain information to be used (displayed to the user, for example) during authentication, if needed; its interpretation is left to the implementation.

3.3 SharedKeyVerify message

The SharedKeyVerify message is sent in response to a SharedKeyRequest message from the server, at the same point at which a CertificateVerify message would be sent in response to a CertificateRequest message. (If both a CertificateRequest and a SharedKeyRequest are sent by the server, then the client may respond with either a CertificateVerify message or a SharedKeyVerify message. Only one of the two messages is ever sent in the same

handshake, however.) The SharedKeyVerify message has the following structure:

```
struct {
     AuthService auth_service;
     opaque identity<1..65535>;
     opaque shared_key_response<1..255>;
} SharedKeyVerify;
```

The value of auth_service must be identical to one of the AuthService values on the list in SharedKeyRequest.auth_services_server. If the client does not share a key with any of the authentication services listed in the SharedKeyRequest message (and cannot supply a certificate matching the requirements specified in the accompanying CertificateRequest message, if one was sent), then the client returns a "no certificate" alert message (in its normal place in the protocol).

The format of the identity field is left to the implementation, and must be inferable from the accompanying value of auth_service. The value of shared_key_response is defined as

```
SharedKeyVerify.shared_key_response
  hash (auth_write_secret + pad_2 +
    hash (auth_write_secret + pad_1
          + hash (handshake_messages)
          + SharedKeyVerify.auth_service.auth_service_name
          + SharedKeyVerify.auth_service.display_string
          + SharedKeyVerify.auth_service.challenge
          + SharedKeyVerify.identity + shared_key) )
```

Here "+" denotes concatenation. The hash function used (hash) is taken from the pending cipher spec. The client_auth_write_secret and server_auth_write_secret values are obtained by extending the key_block by CipherSpec.hash_size bytes beyond the server_write_key (or the server_write_IV, if it is derived from key_block as well), and using this extended portion as the client_auth_write_secret value. (Only the client_auth_write_secret is used, since only the client ever sends a SharedKeyVerify message.) The value of handshake_messages is the concatenation of all handshake messages from the first one sent up to (but not including) the shared_key_verify message. The pad_1 and pad_2 values correspond to the ones used for MAC computation in the application_data message. The fields from the SharedKeyVerify message are input with their length prefixes included.

4. Normal Authentication

A shared-key-based client authentication may proceed as

follows: the client includes the TLS_AUTH_SHARED_KEY CipherSuite in its list of CipherSuites in its ClientHello message. It also may or may not send a SharedKeys message along with the ClientHello message, listing the authentication services with which the client shared a key for authentication purposes. In any event, the server sends a SharedKeyRequest handshake message following the ServerHello and accompanying messages containing a list of names of one or more authentication services; if a SharedKeys message was sent, then this list will contain a single choice from the client's SharedKeys message. The client, on receiving the SharedKeyRequest message, selects an authentication service from the server's list (if more than one is offered) and constructs the appropriate authentication response as described above, sending it back, along with its identity and choice of authentication service, in a SharedKeyVerify handshake message. The server itself also constructs the correct authentication response using the known shared key, and checks it against the one provided by the client. The authentication is successful if the two match exactly. Note that if the shared key is password-based, then it would typically be derived from the password using a one-way cryptographic hash function, rather than being the password itself, so that the original password need not be remembered by anyone but the client.

5. Pass-through Authentication

In some circumstances, it is preferable for shared keys to be stored in one place (a central, well-protected site, for instance) while servers that actually communicate with clients are elsewhere (possibly widely distributed, but maintaining secure connections to the central shared-key server). One of the advantages of the shared-key authentication method proposed here is that it allows "pass-through" authentication by a third party, if the server accepting the public-key key exchange and the server sharing the key with the client happen to be different. (The use of a separately derived authentication key in the response computation makes this possible.)

Pass-through authentication might work as follows: The server would either collect random challenges in advance from its authentication services, or request them as needed. (If the client sends a SharedKeys message, then the server can select an authentication service from the client's list, and obtain a challenge from that service alone.) Assuming that the client indicates support for shared-key authentication by including the TLS_AUTH_SHARED_KEY CipherSuite in its list, the server would then send a list of one or more authentication services and associated challenges in a SharedKeyRequest

message. The client would then select an authentication service (if more than one is offered), compute the correct authentication response using the above proposed formula, and send it to the server in a SharedKeyVerify message.

The server, on receiving a response from a client, would pass it through to the authentication service, along with the values necessary to recalculate it: the client_auth_write_key, the hash of all the handshake messages and the identity field from the certificate verify message. The authentication service would then use the values provided, along with the secret key it shares with the client and the challenge it supplied, to reconstruct the correct value of the response. If this value exactly matches the one provided by the server, then the authentication would succeed; otherwise it would fail.

References

[1] K. Hickman and T. Elgamal, "The SSL Protocol", Internet Draft <draft-hickman-netscape-ssl-01.txt> (deleted), February 1995.

[2] A. Freier, P. Karlton and P. Kocher, "The SSL Protocol Version 3.0", Internet Draft <draft-freier-ssl-version3-01.txt>, March 1996.

[3] J. Benaloh, B. Lampson, D. Simon, T. Spies and B. Yee, The PCT Protocol", Internet Draft <draft-benaloh-pct-00.txt>, November 1995.