Non-intrusive authentication*

Daniele Alberto Galliano
Politecnico di Torino - Dip. Automatica e Informatica
corso Duca degli Abruzzi 24 - 10129 Torino, Italy, phone: +39-11-5647072, fax: +39-11-5647099, e-mail: galliano@athena.polito.it

Antonio Lioy
Politecnico di Torino - Dip. Automatica e Informatica
corso Duca degli Abruzzi 24 - 10129 Torino, Italy, phone: +39-11-5647021, fax: +39-11-5647099, e-mail: lioy@polito.it

Fabio Maino
Politecnico di Torino - Dip. Automatica e Informatica
corso Duca degli Abruzzi 24 - 10129 Torino, Italy, phone: +39-11-5647072, fax: +39-11-5647099, e-mail: maino@polito.it

Abstract

Available security solutions often are not widely used because the associated secure applications are awkward to use or they lack functionality when compared to standard insecure tools.

To avoid this dicothomy, we developed a non-intrusive (or external) client-server authentication framework which requires no modification to both the clients and the servers. In this way, full featured clients can be used to the satisfaction of the user community, and off-the-shelf servers can be used with augmented security to the happiness of the system administrators.

Our approach relies on software agents which use private keys and a challenge-response protocol to authenticate TCP/IP connection setup. The paper discusses the general framework as well as a sample implementation. Attacks and countermeasures are also outlined. The approach explicitly doesn't address data privacy during transmission, as we would rather see it placed at application level.

Keywords

Authentication in distributed systems, Trusted third-party, Network security

---

*This work was partially supported by project MURST 40% 'Metodologie e Strumenti di Progetto per Sistemi Distribuiti e Paralleli'

1 INTRODUCTION

Nowadays client-server is the dominant computing paradigm. It has brought many benefits, mostly in the form of flexibility and scalability, but it has also given rise to severe security problems because most client-server applications have retained the old centralized authentication schema: passwords. While passwords are adequate when authentication is performed over a private channel, they are completely inadequate when used in a open networked environment such as today's LANs and Internet. In fact, in TCP/IP all the data - passwords included - are sent in clear over the network, and hence they can be easily captured by eavesdroppers.

The problem is so general that several solutions have been proposed to overcome it.

OTP (One Time Password) methods (Haller 1994, McDonald *et al.* 1995) do not care about passwords being sent in clear because they are never reused: for every service request users have to provide a new password, usually extracted from a pre-stored list or generated on the fly by a crypto calculator. This solution is acceptable for those services that require authentication only few times a day (*e.g.* main session login), but it is completely unusable for short-life services (*e.g.* incoming e-mail checking) which require frequent authentication. Moreover, this technique is not easily applied to authenticate software agents acting on behalf of humans.

The most widely known and used network authentication system is surely Kerberos (Steiner *et al.* 1988). This is an excellent trusted third-party solution to provide user authentication, but it is rather complex. In fact, if it is to be applied to a network application, both the client and the server side need to be heavily modified (*kerberized*). As a consequence, a lot of maintenance is needed to upgrade the application to new versions of Kerberos or of the operating system. For certain services (such as the Network File System) support for Kerberos must be provided right in the kernel, and so it can be done only by the OS vendor. Another drawback is that, if a service has been protected with Kerberos, only kerberized clients can be used, and the user is forced to choose from a limited set of client applications. Last but not least, Kerberos requires the client host to be secure because the *tickets* are stored on the local disks, and thus can be easily stolen by anybody with system privileges; this can be a problem in open environments with public clients, as often used in an academic environment.

Client-server authentication can also be achieved by using public-key solutions, as is currently done when the SSL protocol (Freier *et al.* 1996) is adopted to secure WWW transactions. However, also this solution requires a safe storage for the private key of the user and the servers. In turn, this calls for use of smart cards, which are still expensive and require special hardware that is not always readily available.

All these solutions suffer from a common drawback: they require specially modified clients and servers to achieve their functionality. A quick look at the major USENET security newsgroups makes clear that people are eagerly looking for security-enabled applications, but they want them right on their platform and with the same user interface they are accustomed to. In other

words, users want security to be added to their preferred applications, and they are unwilling (with few exceptions) to trade functionality for security.

2 NON-INTRUSIVE AUTHENTICATION

Based on the analysis of the problems encountered by the current distributed authentication systems, we propose a new one (GIANO) which complies with the following requirements:

- users must authenticate themselves explicitly only once when a new session is started (afterwards, it is the duty of a proper authentication agent to maintain a list of authenticated users for each client, and to transparently provide evidence of their identity to the servers queried by the user applications)
- standard client queries to servers will implicitly trigger the authentication agent (this allows the use of this authentication framework with standard vanilla client applications)
- user service keys are never transmitted in clear over the network (they are stored locally in encrypted format by the client authentication agent, which then uses a challenge-response protocol to prove the users' identity to the servers)

To satisfy these requirements, each client participating in the GIANO authentication framework runs a daemon, `gianod`, on a well known TCP port. This daemon talks both to local and remote processes to perform separate tasks.

Processes running on the client will make a local connection (normally during login) to the local `gianod` in order to subscribe the user to the authentication framework. Subscription requires `gianod` to get the user's key and to store it in a protected local memory area. To achieve an higher level of security, a different key for each service can be managed by the authentication agent, but it is also possible for a user to share keys across services.

When a server receives an operation request, it first contacts the `gianod` of the client. If the calling process (or one of its ancestors) is enrolled into the authentication framework, `gianod` provides to the server the proper authentication data, otherwise authentication fails and consequently access to the service is denied.

The proof of the user identity is provided by `gianod` to the authentication server via a challenge-response protocol, to prevent key transmission over the network. Basically, `gianod` acts as an electronic safe to which the user commits his service key upon login. Later, `gianod` will provide evidence that the keys are in the safe by answering to the challenges posed by the servers.

Since authentication takes place only upon request from the server, GIANO requires no modification at all of the standard client applications. Client machines have just to run the `gianod` authentication agent, which is the only piece of code that needs to be protected.

On the server side, few modifications to the application server code are needed to query the distributed authentication agent before providing the service. Even better, if we are allowed to manipulate the network tables (such

as `inetd.conf` in UNIX) we can totally avoid to modify the server code by simply intercepting calls to the protected server ports. Services requests will be forwarded to the standard server behind the port only if the client is properly authenticated.

It is easy to see that in the most general case GIANO is able to easily add authentication to connection-oriented services which were not developed with security in mind.

3 IMPLEMENTATION

This section refers to the three fundamental kinds of interaction between the user and the authentication framework.

The GIANO daemon, as shown in figure 1, acts just as a safe meant to keep the ID data of each user subscribed: these data are the encrypted key, the expiration of the subscription, and the PID the user subscribed for. Comparing this approach to the Kerberos one, that uses file as the ticket repository, shows that GIANO, that maintain the authentication instances in main memory, provide an higher level of secrecy.

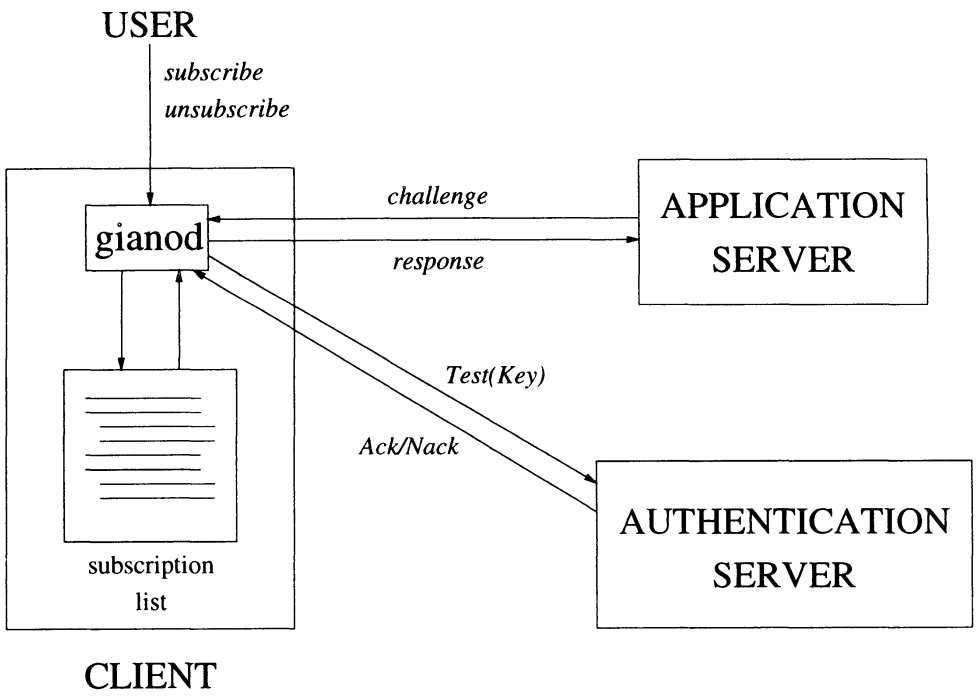

Figure 1 GIANO, a general view.

We assume that every server is physically secure and we assign a key to each service and to each user in the system. In this way we have a bunch of GIANO actors, that are not always real users, but just authenticated entities. Every key is known to the authentication server, so that it is able to check the response provided to any challenge.

3.1 User authentication

When a user wishes to get the GIANO service of authentication brokering, he should commit his keys to the `gianod` daemon. To do this, he must send to the well known port of the daemon a special packet, composed of the following fields:

```
Subscr:USERNAME:KEY:PID
```

`Subscr` identifies the message as a subscription;
`USERNAME` is the name by which the user is known to the authentication system;
`KEY` is an hex representation of the encrypted key made from the user's key;
`PID` is the PID the user wants to subscribe: the only values allowed are the PID of the calling process or its parent.

When the daemon receives the request, and recognizes it as a subscription request, it checks the key with the authentication server. If the user supplied the correct key, the authentication server will reply with an acknowledgment and the number of the seconds the subscription will be valid. Then the daemon detects from the kernel the PID of the process calling, hence only local processes are allowed to ask subscription. If all the conditions are satisfied, the daemon will add a new entry in its table for the process. The table will hold this data:

`pid` the pid of the process subscribed;
`login` the name of the user;
`key` the encrypted key;
`expiration` the last second this subscription will give a valid authentication.

Of course, this chain of events should take place at login, to be sure that every process the user will activate will be authenticated. It means that we shall have to modify only these two clients: login and xlogin; alternatively, a special application can be developed to perform GIANO subscription explicitly after a standard login.

3.2 Subscription of a new process

When starting a new application, the authenticated user will not have normally to subscribe the corresponding process to GIANO, because it will inherit the authentication from its chain of parents. However there are cases when an authenticated new process must be explicitly subscribed for itself:

- when the parent will die before the child, *i.e.* the child is launched as a background process, and will not end at user logout;

- when the new process should be authenticated with a new, or anyway different, authority (this applies either to a different user, or to the need of a longer lifetime).

3.3 Client and server authentication

As shown in figure 2 there are four processes involved in this task:

P, is the process the user executes: it is unaware of the authentication activity;
A, authentication agent (*i.e.* `gianod`) on the user machine, the same of **P**;
G, `cerere`, the master authentication server: it can be implemented by more than one server for redundancy, but it is represented as a single entity for sake of simplicity;
S, the application server process activated to answer **P**.

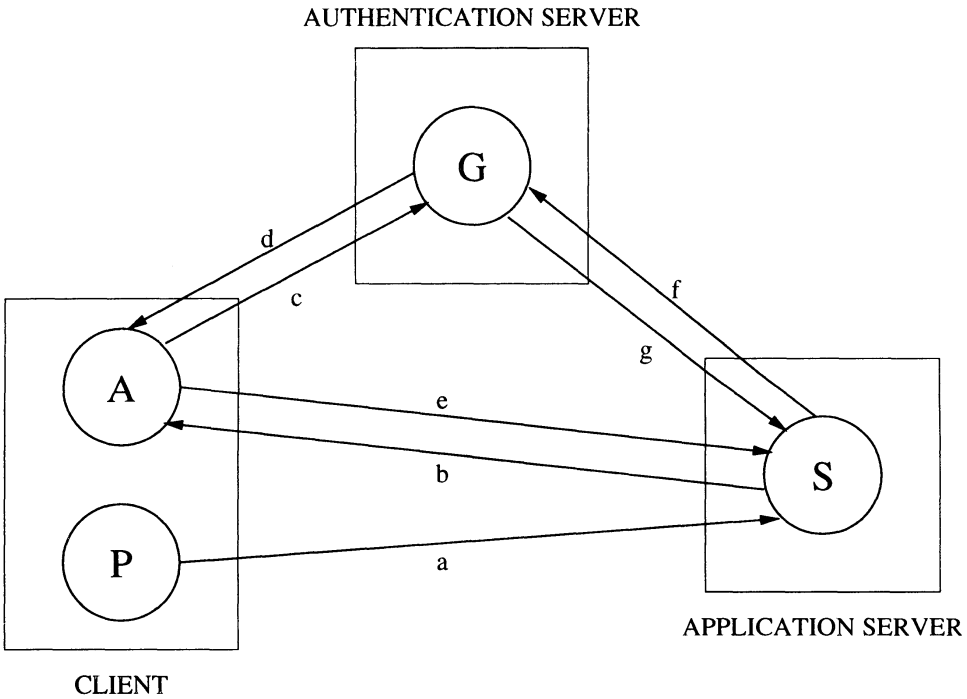

Figure 2 Service validation sequence: a) Service request; b) ID request; c) Server authentication by the broker for the user; d) Server authentication produced by the authentication server; e) User ID; f) User authentication; g) User authentication produced by the authentication server.

The packet that the actors involved into the process use to exchange messages has the following structure:

```
CRYPT:AUTHOR:TEXT:PEER:EDATA
```

Where:

`CRYPT` is the encryption of `TEXT` by the `AUTHOR`'s key;
`AUTHOR` is the user issuing the authentication request;
`TEXT` is the challenge created by the server process;
`PEER` is the user, which `AUTHOR` wants to authenticate;
`EDATA` that contains a timestamp, the length of the message, and the message itself, all this fields are encrypted with a key owned by the `AUTHOR`.

The user starts the session by launching the client process **P**. This process is the execution of a program not modified in any way to accomplish authentication tasks. This allows us to use standard commercial products that may have optimizations we cannot reproduce easily.

Process **P** will contact the server host and try to talk to the server process **S** (Step a). This is a modified version of the program, which will be able to perform authentication issues. As a matter of fact we developed a general purpose server to authenticate the client, before starting server process itself: it avoids loss of performance and reduces the amount of work necessary to introduce this authentication.

The server process can easily discover the host who submitted the service request, and from which port: this is what the authentication is based upon. It now produces a challenge which, the user has to solve to have its request accomplished.

Challenge format is fixed, to provide further exchange of informations during the authentication procedure. Its structure is:

```
TSTAMP:PID$IP:PORT
```

Where:

`TSTAMP` is the UNIX standard time of creation of the challenge, this prevents both replay and cut-and-paste attacks to the challenge;
`PID` is the PID of the server process itself, and is introduced to increase the unpredictability of the challenge;
`IP` is the IP address of the client host;
`PORT` is the network port of the client host, from which the request comes. This field provide to `gianod` the clue to identify the process requesting the service.

The challenge is transmitted into a standard packet (Step b) to process **A**; the packet contains this information:

`CRYPT` the challenge encrypted;
`AUTHOR` the entity which represents the service;
`TEXT` the challenge in clear that the service proposes to the user;
`PEER` the string *unknown*, usually, unless an earlier transaction stated the identity of the user;
`EDATA` a brief description for logging purposes.

In case the user is already known to the service handler, it can proceed to request the authentication without waiting for the user machine's answer.

Once **A** receives the request from the service process, it proceeds to search in the kernel itself which process holds the port from which the server machine was contacted.

The table of subscription is scanned for the process and all its ancestors. If none of them is found, the request is refused: **S** learns that **P** has no right for the service, and it should not be provided.

Otherwise, if it is found in the table, the authentication daemon **A** will have the ID data and the key, received when the family of processes was subscribed. Using these, it can produce its own solution of the challenge, and issue it to the authentication server, to check the server identity (Step c). The packet for process **G** will be composed of these fields:

`CRYPT` the user solution of the challenge;
`AUTHOR` the user;
`TEXT` the challenge received by the user;
`PEER` the server requesting the authentication;
`EDATA` a brief description for logging purposes.

G has two possible answers for this kind of requests: if `AUTHOR` authentication fails, an error message will be returned, encrypted with the `AUTHOR` key; otherwise, **G** will provide the challenge solution `PEER` should have computed, encrypted a second time with the `AUTHOR` key.

A will then receive the confirmation of **S** identity (Step d), so it will reply its own version of the challenge, in order to authenticate with the now trusted server **S** (Step e). The packet for process **S** will be:

`CRYPT` the user solution of the challenge;
`AUTHOR` the user;
`TEXT` the challenge received by the user;
`PEER` the server requesting the authentication;
`EDATA` a brief description for logging purposes.

G will then receive a symmetrical request from **S**, in order to authenticate the answer received from **A** (Step f); and symmetrically it will answer (Step g).

At this point all the actors are sure of the identity of each other, and **S** can carry on the normal transactions with **P**.

3.4 Status of the implementation

In order to test the validity of our authentication framework, we developed a GIANO environment under DIGITAL UNIX and HP/UX. The two main components of GIANO are:

`cerere`, the GIANO authentication server, which maintains all the knowledge about the authentication framework;

`gianod`, the daemon which performs all the authentication operations on user's behalf.

Some services that use the GIANO infrastructure have also been developed:

`glogin`, a secure login that subscribes a user to the authentication framework;
`gexec`, a secure exec to spawn a process that doesn't inherit the authentication state from the parent process, but gets a new one;
`limen`, a general purpose spawner for net services. It is a configurable general purpose interface for the usual net server `inetd`.

The functionality of these programs is detailed in the next sections.

Cerere

This application maintains the database of users, with all the related informations, and offers two services.

The main service, which covers the largest part of the activity, is the third-party authority, that confirms the mutual authentications between clients and servers. It can be easily replicated on several machines, using read-only copies of the database. Every transaction is logged, and is stateless. Since each request contains all the information needed, only one answer is needed to accomplish the task.

The secondary service that `cerere` must provide is related to administrative tasks: `cerere` is responsible for adding, deleting, and retrieving users from the authentication data-base, handled with `ndbm` routines.

Every entry has the key encrypted with the database master key, chosen at installation time. There is also a checksum, calculated with the same key: the database is, by these means, tamper-proof. To improve this security feature, `cerere` can be started in a conversion mode, that will ask for the old master key and a new master key, and then it will translate the database from one key to the other.

Giano daemon (`gianod`)

This daemon should be started at boot time to provide two main services: authentication and administration. The authentication service is the core project, and has been illustrated in the previous sections.

By administration, we mean process subscription. This occurs, when the daemon receives a packet starting with the string `Subscr` which cannot be the result of an encryption.

Glogin

This program covers two needs: user subscription and key change.

At login time, this program will generate the user subscription packet, send it to the well known port of `gianod`, and wait for the answer. The user subscription may fail for a few reasons:

- `cerere` installed in the system didn't accept the confirmation request issued by `gianod`

- `cerere` server replied saying that the user is expired
- `cerere` server replied saying that the key has expired

In this latter case, the program will ask for new key and its confirmation, and then it will issue an update request to the authentication server to change the key.

The user subscription can operate at two levels: it can be used to subscribe the parent process, asking in this way the brokering service to `gianod`; but it can also establish an interactive session in the host, updating the log files.

Gexec

To accomplish the task of spawning a new process that doesn't inherit the authentication state from its parent, a command named `gexec` has been developed.

Some special processes, such as the ones that perform a background computation, need an authentication instance different from that of their parent.

In its simplest form, `gexec` executes the new process, giving it a new authority in force of a precedent subscription. It does not request the key, since it comes from an already validated chain of processes. The result is that the subscription will be considered valid for a number of seconds specified in the authority database. This number is a data related to the user entry and retrieved at the moment of the subscription. It cannot be bigger than the amount specified at installation time. In this way a process can be executed for a very long time, without renewing its subscription. It can be useful for long unattended calculations, typically those launched at night or in the weekends.

Another use of `gexec` is the execution of a process authenticate by the means of a different user. This way an administrator can create a shell, in which he can issue commands to perform administration tasks, without interfering with his usual activity. We think this obviously better than having multiple authority files available on the system, as it was, for instance, in a Kerberos environment.

Limen

Since our major concern was the development of an authentication system to improve security in a standard distributed environment, we tried to reduce at a minimum the need to patch existing software. And since servers need to start the gianod authentication process, it was necessary to produce a general purpose shell to encapsulate server programs: this will provide authentication before executing the server program itself.

Born to be used with the standard UNIX network daemon `inetd`, `limen` is a configurable general shield for network servers.

Its use is very simple, and can be illustrated referring to the format of `inetd.conf` file:

```
service-name stream/dgram protocol mode user pathname argv
```

The complete pathname of `limen` will be put instead of the server's one, whose

name will be replaced by the pathname itself. The usual arguments will follow; after those a string will follow, to introduce the `limen` parameters.

This way, `limen` will have in its argument vector all the informations needed to execute the program that will perform the service.

Currently the options available for `limen` are:

`0:` it precedes the name to be used as argument 0, if different from the last part of the pathname indicated as argument 0 for `limen`;
`a:` it precedes an option of the program to be executed, that in its syntax will precede the name of the user, which issued the service request; it should be left as the last, if no option is needed but the username itself;
`b:` this option enforces security, preventing *limen* to execute the server program, unless the authentication was accomplished (it is disabled for default, but implied in the previous);
`c:` it means that the server must be executed as a login session, so `limen` will update the log files, and open the pseudo-tty with the child;
`d:` it introduces the name chosen for the log file; if absent, a default chosen at installation time will be used.

A practical example of the use of `limen` is the following one, by which an authenticated login was realized with minimum effort:

```
alogin stream tcp nowait root /Giano/bin/limen
    /bin/login -p #G# -a -f -c -d /Giano/logs/alogin.log
```

4 CONCLUSIONS AND FURTHER WORK

We believe to have designed an authentication system which is both simple and effective. Its main novelty lays in the fact that it doesn't require nearly any modification to the existing software, and this is really a big benefit for both users and system administrators.

So far there is only one problem we didn't cope with: shadow servers. It is possible for a malicious host to spoof the real server's IP address, and then the client will probably establish the connection with someone obviously not eager to trigger the authentication session.

Since the client itself is not aware that an authentication session should be performed, it will interact with the fake server as it would do with the real one.

Our solution to this problem is to rely upon the adoption of IPv6 with its packet authentication features.

Extension of the authentication mechanism to DOS/Windows based system is really easy. At boot or start time, a simple application will ask user's key, and then will act as a GIANO daemon, waiting for authentication requests. The system will be really simplified by the single-user environment, since all the requests will be for the same user. A trivial TSR or a WINSOCK application will fit this task.

ACKNOWLEDGEMENT

We want to thank Vic Abell, of the Purdue University Computing Center. His analysis of the kernel data structures (used in the development of `lsof`) was extremely valuable in the realization of `gianod`.

REFERENCES

Freier, A.O. and Karlton, P. and Kocher, P.C. (1996) The SSL Protocol Version 3.0. *IETF Internet draft.*

Haller, N. (1994) The S/key one-time password system. *The ISOC Symposium on Network and Distributed System Security*, 151-7.

McDonald, D.L. and Atkinson, R.J. and Metz, C. (1995) One Time Passwords In Everything (OPIE): Experiences with Building and Using Stronger Authentication. Proc. of *the Fifth USENIX UNIX Security Symposium*, Salt Lake City, UT, 177-86.

Steiner, J.G. and Neuman, C. and Schiller, J.I. (1988) Kerberos: An Authentication Service for Open Network Systems. Proc. of *Winter 1988 Usenix Conference*, Dallas, TX, 191-202.

BIOGRAPHY

Daniele Alberto Galliano is a young registered professional engineer, who took degree in Electronical Engineering from Politecnico di Torino. A former administrator of Athena based distributed system of that University, he is working now as a professional consultant in UNIX systems for universities and public administration. Strongly interested in multivendor distributed systems, he developes administration tools, with a massive advanced use of Tcl/Tk. From his interest in this topic comes the involvement in the field of security.

Antonio Lioy is an associate professor of computer engineering at the Politecnico di Torino. Professor Lioy holds a *laurea* (aka master) degree in Electronic Engineering *summa cum laude* and a Ph.D. in Computer Engineering from Politecnico di Torino. He is a registered professional engineer and a member of the IEEE and the IEEE Computer Society. His research interests are in the fields of computer security and CAD of digital systems.

Fabio Maino is a Ph.D. student in computer engineering at the Politecnico di Torino. He holds a *laurea* (aka master) degree in Electronic Engineering and he is a registered professional engineer. His research interests are in the fields of computer and network security.