4 Designing

4.1 Overview

Although the topic of this project may lead to think that only two programs and a web application have to be designed, it is wrong to assume that the design can be done without experiencing troubles. The headline of the project was about portability. Thus, the application (agent) that had to be written had to adapt to a well known architecture, should it be Linux, Solaris or Windows. Developing a typical application like one of the inventory tools that was tested appears to be inconvenient because here our need is quite different. Four out of five tools embed mainly a PHP application for administrating and management purposes only. Here, there is no need to administrate any database. The goal of the agent is to retrieve information that will be used later to facilitate the management of vulnerabilities. PHP applications are more designed to provide interfaces for users that can help them view, check and modify their choices. Consequently a PHP application does not fit with our exigencies. What is required is a feature that can receive automatically reports from agents and later create CMSI reports. Some user interaction might appear only if a user wants to manually change his configuration. For this last case, a PHP application seems to be the most convenient service, since everybody could connect to through a web browser.

The requirements soon led to think about using OpenSSL to provide cryptographic functions. This choice has revealed relatively relevant because OpenSSL has a real portable code. Its designers have worked a lot to offer a portable application compatible with lots of systems.

From a personal point of view, I was not very familiar with PHP. Although this study showed me that it was possible to create complete applications with this language, I felt surer with C/C++. I had already worked on OpenSSL source code as a former project theme, so I was quite convinced I could take advantage a second time of its functionalities.

4.2 Agent

4.2.1 Programming language

To design the agent, several possibilities can be chosen. First it can be contemplated to realize a single application whose source code will be compatible with every distribution. To do such a trick, C/C++ offers everything needed. OpenSSL provides portable code that works for lots of distribution, so there would not be any problem to adapt it to both UNIX and Windows. The generation of the report might take more time to think. On Windows, retrieving information on Windows is realized by several Windows C functions that do not exist on UNIX (there is no registry on UNIX, for example). Nevertheless, pre-compiling directives like #ifdef WIN32 represent a good solution to isolate specific Windows portions of the code. The main problem comes actually from the compilation
step. C++ also presents the advantage of implementing safe strings and their use are much more convenient than the old type char. In C, strings exist with the <char> type, which can reveal to be insecure if it is not well used. Lots of dangerous functions exits like strcpy, strcat... that handle character strings and might in certain conditions lead to memory corruptions, or even security problems. In C++, strings are defined using the <string> class. It provides lots of functions that help the developer to use strings in its program in a secure way. Moreover, the class provides methods like append that avoid manually reallocating memory. In C, appending a string should have be done using a malloc or a realloc call, so the programmer has to carefully program its software paying a lot of attention to the memory usage. The code is then more complicated and it becomes difficult to write a safe code.

Writing a C/C++ tool for UNIX is something common yes, but when it is about to distribute the release version, problems might appear. In a Linux environment, C++ code can be compiled using g++. Like gcc, this tool can be encountered in a lot of packages but unlike gcc, few applications need it by default. Administrators usually have to compile applications using gcc, so this tool usually exists on a system. It is a little rarer to find g++. So, as it was the case for the OpenSSL library development, a second package should be downloaded to perform the compilation. Who can guarantee that final users will have the correct packages to compile the source? Our goal is to provide a stand alone agent that would require as few dependencies as possible, that is to say the most independent possible. If users are asked to download packets such as libssldev or g++ to compile, this exigency will not be satisfied. Besides the name of these libraries change according to the distribution, forcing us to provide additional details during the installation phases. So a dilemma appears: releasing the source and forcing administrators to download an arbitrary package, or simply releasing a hard coded executable for each distribution. Although this last option can be considered, it requires a lot of time to compile a program for each distribution. (For example, five different versions of Ubuntu exist). And it must not be forgotten that the agent has to run on newer distributions. This assumes a subsequent work for people who want to support the application. Building as many versions as possible would make versions skyrocket as lots of different distributions exist. So, a universal agent may not be the right solution, and so does a C/C++ agent for Linux.

On the other hand, for Windows, if we want to avoid any nuisance related to using ActiveX objects that might not have been installed yet, a complete C++ program offers such a workaround. Javascript and VBscript might depend on supplementary controls to use the file system, the registry or even transfer data, that is why I preferred C++ to them. Then, data transfer would be hold through raw sockets. The Winsock library provides all we need to implement socket connections. Windows has been built around stand alone executables, that is why it appears so common to create a new one.

On UNIX, the choice remains open. Java has proved in zCI Computer Inventory System that it could be a powerful tool to carry the agent task. Its code is easy to understand, and most of UNIX distributions own a java compiler. On the reverse, Windows cannot run java code without a Java VM installed so there is no point in trying to develop a java code for Windows since most servers are bound to refuse to run it.
Scripts represent perhaps the most significant way to achieve the agent. Every single UNIX distribution is compatible with Shell Script. Despite some of them may interpret differently some commands than others, Shell Script is definitely a good idea to create a UNIX agent. Lots of UNIX services are run by scripts and they do not require any additional packages, nor need them to be compiled. Associated with a telnet client, it is possible to send every type of data to any remote host through the Internet.

Perl and Python scripts likewise represent a good alternative to Shell Script. But their dependencies to Perl and Python packages make them less attractive compared to Shell Script. However, in spite of my initial preference for Shell Script, I had later to shift this choice to Perl, due to incompatibility of some commands with certain distributions (telnet removed from SuSE distribution).

### 4.2.2 Fingerprinting the system

The inventory tool analysis has illustrated how to perform a complete system review. This review most certainly depends on the nature of the platform and on the distribution (Windows, Linux, Solaris, BSD...). Obviously, Windows systems have nothing in common with UNIX ones, so two categories must be distinguished. In a UNIX environment, the `uname` command appears to be perfect to determine the nature of the system. Here are showed different results given by the command on my test machine:

```bash
uname -a
Linux pc 2.6.22-14-generic #1 SMP Sun Oct 14 23:05:12 GMT 2007 i686 GNU/Linux
```

**On Solaris:**

```bash
uname -a
SunOS solaris-test 5.10 Generic_118833-36 sun4u sparc SUNW,Sun-Fire-V890
```

`uname -s` prints the name of the system, `uname -r` its version, and `uname -p` the architecture (although this last one might not work on certain systems: `uname -p = unknown on Ubuntu`). This first step can then distinguished at least three different categories.

For Linux systems, retrieving a list of installed components can be achieved by certain applications such as:
- `dpkg` for Ubuntu and Debian
- `rpm` for Mandriva, SuSE, Red Hat, Fedora
- `equery` for Gentoo

But consequently the name of the OS is needed. Linux systems have however the characteristic to save the name and version of the OS in a file stored in `/etc/`. Though its name changes according the OS, it can be grabbed by a simple regular expression: `/etc/*release`. Scripts have a full support of those kinds of expression, determining which OS is running can be done by:

```bash
cat /etc/*release | grep OSNAME
```
Therefore replacing `OSNAME` by known distributions, the current distribution is deduced, and so becomes the name of the package retriever. Basically they work like this: `packageretriever -list` gives a full list including for each package its name, its version, some additional descriptions:

```
dpkg -l
...  
i  xwininfo  1:1.0.2-0ubuntu X client - xwininfo
ii xwud  1:1.0.1-0ubuntu X client - xwud
ii yelp  2.20.0-0ubuntu Help browser for GNOME 2
ii zenity  2.20.0-0ubuntu Display graphical dialog boxes from shell scripts
ii zip  2.32-1  Archiver for .zip files
ii zlib1g  1:1.2.3.3.dfsg compression library - runtime
ii zlib1g-dev  1:1.2.3.3.dfsg compression library - development
``` 

### 4.2.3 Interesting data to collect

The server will be able to identify agents according to their IP address. But since some of them might share the same network and be NATed with the same IP, the machine name is also interesting. The report will then be saved in such a way that it could be possible to differentiate each machine at first glance. And later, one could use these data to categorically identify the machine. UNIX environments present the advantage that whatever the system is, the command `hostname` gives the full name of the computer. Language is also interesting to grab but in the case of UNIX, English has always been the main language. Only language in applications differs, such as graphical interfaces (Gnome, KDE) where it is possible to set it). Architecture likewise does matter, because nowadays OS configuration depends a lot on the kind of architecture. `uname -p` satisfies the need. And last, comes the strict identification of the OS: its name, sometimes its codename, its version, the name of the actual kernel and its version. This information constitutes the best way to identify with high accuracy an OS.

Results on my test machine:

| host: pc  |
| osname: Ubuntu |
| codename: gutsy |
| osversion: 7.10 |
| archi: i686 |
| kernel: Linux |
| kernelversion: 2.6.22-14-generic |

Results on Solaris:

| host: solaris-test  |
| osname: Solaris |
| codename: Solaris |
| osversion: 10 3/05 |
| archi: sparc |
| kernel: SunOS |
| kernelversion: 5.10 |
I thought the best way to make easy the header parsing on the server was to define these keywords. Each time a keyword would be noticed, the corresponding field would be filled, making the task purely mechanical. A star was employed to dissociate the header from the software list. Stars cannot appear in machine names, and there is no distribution that takes a star in its name, codename or version. A star is then the perfect character to act as a separator.

Given that the system architecture of Windows has nothing in common with UNIX architecture, so goes for the method to retrieve interesting system information. Applications are not considered like packages, they have their own installer that is free to set up the system in accordance with the needs of the application. Therefore, no standard has been developed to define a correct installation, so data that can be grabbed may differ from UNIX. Using environment commands is not as common as on UNIX: there is no hostname command that can get us the name of the machine. However, programming with Windows API can save us a lot of time. Lots of functions have been designed for a developer to interact with the system and take its best advantage. Therefore, the most common and obvious commands in UNIX have their equivalent in Windows as C functions. Instead of calling a 'hostname' command, the C function GetComputerName can be used to get this so important name. Thus, every single piece of information listed above is accessible through a complete system information C function GetVersionExA. Once called, this function will fill a structure where the programmer will later be able to distinguish which version of Windows is currently running, the name of the Windows distribution, if it has any service pack and the machine architecture. So even if it seems to be more complicated than on UNIX, C functions are powerful and let us retrieve the most precise details.

Windows OS have also the characteristic to be released in a certain language. This detail is important in the sense that for instance, when updates are released, it is asked to precise the language of the system for compatibility purpose. Consequently it is very interesting for us to obtain this piece of information.

GetSystemDefaultUILanguage has been defined to return an integer code corresponding in a table to the language currently set. It then provides the default language of the system.

Here is the result of a successful system audit:

<table>
<thead>
<tr>
<th>host: NICO-QF111Q52Y7</th>
</tr>
</thead>
<tbody>
<tr>
<td>osname: Windows Server 2003</td>
</tr>
<tr>
<td>codename: Enterprise Edition</td>
</tr>
<tr>
<td>osversion: 5.2.3790</td>
</tr>
<tr>
<td>archi: IA32</td>
</tr>
<tr>
<td>servicepack: Service Pack 2</td>
</tr>
<tr>
<td>language: fr</td>
</tr>
</tbody>
</table>

Results on Windows Vista:

<table>
<thead>
<tr>
<th>host: Vista-test</th>
</tr>
</thead>
<tbody>
<tr>
<td>osname: Windows Vista</td>
</tr>
<tr>
<td>codename: Home Premium</td>
</tr>
<tr>
<td>osversion: 6.0.6000</td>
</tr>
</tbody>
</table>
Unlike UNIX, no package lister is installed by default on Windows, and there is also no way to obtain with accuracy the details of an application. For example, with dkg, it is possible to view the version of the package. On Windows, nothing ensures that the version of an application will be available for the user. Typically, applications are installed by an installer. They are normally configured to add an entry in the list displayed in Add or Remove Software in the Configuration Panel, but nothing forces them to do so. Applications might be installed without telling explicitly the system they have been installed. Problems might then occur if something has been installed without registering its tracks. In such a case it is absolutely impossible to determine whether it exists. It is as if someone had compiled its sources on Linux and was using them. There is no registration of the program in the package list. However one can assume that applications running on a company server have been properly installed, so this case should not often occur. As the inventory tools study showed it, the best way to list application is to query the system registry for the name and the version of applications that can be uninstalled.

Software entries are stored in the registry under the root key \HKEY_LOCAL_MACHINE at: Software\Microsoft\Windows\CurrentVersion\Uninstall.

Among the different kind of information, the name and the version of the program can be easily obtained through RegOpenKeyExA and RegQueryValueExA. The list is then integrally browsed and finally a full list of application name and version is retrieved.

4.2.4 Installation (Cron and Task scheduler)

As the analysis has revealed it, most of the agents issued from the tools do not have automatic start feature. They require to be launched by hand by a user, which does not suit with our demands. Our agent must be left independent, and the user might not even record it is on its system. Which answer could then be chosen to satisfy those conditions? OCS Inventory perhaps provides acceptable ideas. On UNIX, auto start is realised by a Cron job. Given a time, the agent will be executed as a daemon without even notifying the user it has been activated. This solution therefore seems ideal. The agent should be run on a server, and in normal conditions, servers are started once for all. Some of them might run for days, weeks and months without being restarted. So starting a service during the start up of the machine could be contemplated if the agent has an internal loop, for example, that waits for some time and automatically restart the main functionalities. But on UNIX, a Cron job avoids to do such a complicated task. Cron entries are given under this form:

```
# DO NOT EDIT THIS FILE - edit the master and reinstall.
# (cron.txt installed on Thu Jun 8 12:04:33 2006)
# (Cron version -- $Id: crontab,c,v 2.13 1994/01/17 03:20:37 vixie Exp $)
00 12 ** 5 /home/test/agent/agent.pl
```
where it is possible to define specifically each part of the time. Our agent could afterwards be executed once a week, every Tuesday, twice a month, or every day at 10h43.

Windows case is relatively similar. By default, Windows has a task scheduler accessible from the configuration panel. This feature can be considered like a Cron application, but as it has a graphical user interface, it can take much time to configure than a simple Cron entry, but its results are the same. This method presents also the asset to invite the administrator to precise on which account he wants the agent to run. Should he think its account has too much privilege, he can freely choose the best account to use to start the task. And since the agent functionalities do not need any root account, it can consequently be run by local system, or any user able to read the registry.

![Task property panel]

Figure 7 - Task property panel

Another possibility would have been to create a new registry entry to automatically start the program during Windows start up, and then register a timer function with Windows C functions that would have been called by the system to check whether the current time corresponds to the trigger time. The main trouble with such a system is that in case of any crash or abnormal termination, the agent would not keep running and would need a manual restart. Solutions exist to solve this problem. Windows offers to system developers the ability to design drivers and system services. A services pretty looks like a standard application, save that it needs to be first registered as a Windows service. Once done, it is automatically started by the service manager each time the system boots, and it is possible to register it in such a way that in case of failure the service would be automatically restarted.
4.3 Server

4.3.1 Overview

The conception of the server is directly related to the development of the agent. Because both interact, at least the network part needs to be realised at the same time. Socket calls in C/C++ are quite symmetric, so what can be done in one side is easily translatable to the other. The server should be able to listen on a certain port, accept agent connections, retrieve their reports, analyse them, create a resulting CMSI report and eventually store those results into a database. It is right to think that a script might be able to realize all these tasks. By adding the MySQL Perl module to Perl, complete applications with database interactions become imaginable. But like PHP, I was not very familiar with Perl to build such an application. And like OpenSSL, I already had an experience with MySQL++, a C++ driver for MySQL. Interaction with C++ is really easy, that is why I preferred this option. Last, each time an agent connects a new thread would be started, so that others could connect without being refused.

For the user interface, PHP was indeed the most significant solution, and its programming would be easier given that the inventory tools are issued as open source software, so one can freely look and inspire himself of them. This application should work on a new database containing information towards administrators, their machines, and software installed on their machines.

4.3.2 Writing CMSI reports

As it is described in the German CMSI model, products are organized as a family tree where their attributes like version, vendor etc. are given under a form of a tree. Each of its leaves represent a string later used to categorically identify the application. From the server point of view two categories should be distinguished:

1. an OS
2. a piece of software

Because of the implementation of the agent, only two characteristics of an application are sent to the server: its name and its version. But more information is provided in OS entries. They contain architecture, language, version, service pack and codename details besides of the name and version of the OS. This leads then to proceed differently according to the attribute.

Once the header of the file has been parsed, an internal HOST object is filled and later used to query CMSI information. The request has the following form:

```
select family.familiy_tag, family.name, family.descr, family.parents.node_tag, product.product_tag, 
product.name, product.descr, product.id from family, family.parents, product 
product.family=family.id and family.name='FAMILYNAME' and product.name='PRODUCTNAME' and 
family.parents.family=family.id
```
These records are then converted to an XML structure, and for any additional attributes a new XML record is generated so that no information is lost.

For software, the report consists of entries like

```
APPLICATION, VERSION
APPLICATION, VERSION
...
```

Likewise, for each entry in the report, a preliminary query to the database is done to obtain information on the product:

```
select family.family_tag, family.name, family.descr, family_parents.node_tag, product.product_tag, product.name, product.descr, product.id from family, family_parents, product where product.family=family.id and family_parents.family=family.id and product.name='APPLICATION'
```

This query lets us know which family the product belongs to, its description and its identifier in the database. Sometimes difficulties may happen during this search. For example, if the agent noticed: Microsoft Office Professional Edition 2003, this name will not always be found in the database. If a first search does not find anything, a next search with Microsoft Office Professional Edition is done, and so on until a result is found. If only one result is found, then the node is elected. If two or more results appear, it means that the product had not been entered yet in the database, hence the request is dropped.

All these information are then filled in an internal CMSI_data object which is later passed to a function that will create the XML report.

4.4 Configuration

Both server and agent can be configured to fit to the user's requisites. The Perl script lets the ability to define variables that will be used during the normal behaviour:

```
$server="127.0.0.1";
$port="6667";
$privatecertificate="agent.pem";
$publiccertificate="pub.pem";
```

and a config.ini files is available for Windows users:

```
servename=10.0.2.2
serverport=6667
privatecertificate=agent.pem
publiccertificate=pub.pem
```

Moreover, it is also possible to set up a repository for saving reports. In such a case, reports will be stored on the hard drive and later manually sent through the PHP application. No socket connection will be performed. In both cases, the variable is reportpath. If defined, reports are saved locally, otherwise reports are sent through the network.
5 Implementation

In this part only the main and complicated parts of the code are explained. For further details, please refer directly to the source code.

5.1 Agent

5.1.1 Overview of the Agent

Once the agent has been started it will execute the following steps:
- Get the host name.
- Identify the system.
- Identify the way to retrieve the software list.
- Sign a MD5 hash of the file using a certificate.
- Encrypt the signature and the list.
- Send it to the server.

Each of those steps does not require specific privileges, so the program should be run with low privileges.

5.1.2 Getting system information

5.1.2.1 GetHostName()

In order to identify the machine, the report has to contain its name. On Linux system, this name is found in the variable ’hostname’. On Windows system, a call to this kernel32 function is needed to get this name.

```c
BOOL WINAPI GetComputerName(_out LPTSTR lpBuffer, _inout LPDWORD lpnSize
```

This function copies up to lpnSize characters in lpBuffer, and return true in case of success.

5.1.2.2 GetDistrib()

The purpose of this function is to retrieve the system data. Its behaviour differs according to the Linux distribution. Most of the Linux distribution creates a file named /etc/lsb-release or similar where the name of distribution can be found. Some of them add other information, like the version of the distribution, or its underlying name.

For instance on an Ubuntu Gutsy distribution the file contains the following entries:

```
DISTRIB_ID=Ubuntu
```
DISTRICT_RELEASE=7.10
DISTRICT_CODENAME=gutsy
DISTRICT_DESCRIPTION="Ubuntu 7.10"

On an OpenSuSE system, specific contents are found in /etc/SuSE-release:
openSUSE 10.3 (i586)
VERSION = 10.3

On Mandriva, those data are found in /etc/mandriva-release:
Mandriva Linux release 2008.0 (Official) for i586

Therefore the easiest way to proceed is to read /etc/*release and distinguish according to the content of these files.

Finally, the command 'uname -r' can be performed to get the Linux version.

The method employed on Windows system has nothing to do with the one used on Linux. It takes advantage on several system functions that return detailed information on the running system.

In <windows.h>, the following structure is declared:

```c
typedef struct _OSVERSIONINFOEX {
    DWORD dwOSVersionInfoSize;
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
    DWORD dwBuildNumber;
    DWORD dwPlatformId;
    TCHAR szCSDVersion[128];
    WORD wServicePackMajor;
    WORD wServicePackMinor;
    WORD wSuiteMask;
    BYTE wProductType;
    BYTE wReserved;
} OSVERSIONINFOEX,
```

This structure can be filled by a call to:

```c
BOOL WINAPI GetVersionEx(_inout LPOSVERSIONINFO lpVersionInfo)
```

Actually, only six of those fields are interesting. The values dwMajorVersion and dwMinorVersion are used to determine the version of Windows. For example dwMajorVersion = 5 and dwMinorVersion = 2 indicate that Windows 2003 is currently running. dwMajorVersion = 4 and dwMinorVersion = 90 indicate that Windows Millenium Edition is the current system. dwBuildNumber determines the compilation figure. For an up to date Windows XP system, this value equals to 2600. szCSDVersion is used to retrieve the service pack information, and finally wSuiteMask indicates the label of the system. This value can be used to determine if the system come from a Professional Edition or a Home Edition etc.
For example:
```c
if(osvi.dwMajorVersion==5 && osvi.dwMinorVersion==1 &&
osvi.dwPlatformId==VER_PLATFORM_WIN32_NT) {
    if (osvi.wProductType == VER_NT_WORKSTATION) {
        if (osvi.wSuiteMask & VER_SUITE_PERSONAL) {
            o << "Windows XP\ncodename: Home Edition";
        }
    } else {
        o << "Windows XP\ncodename: Professional";
    }
}
```

### 5.1.2.3 GetLanguage()

This function is specific to Windows system, provided no language distinction is performed in a Linux environment. On Windows, it is possible to retrieve the language of the system via the function defined in Kernel32:

```c
LANGID GetSystemDefaultUILanguage(void)
```

It returns a number used to determine which language is currently set on the system and so a simple case statement transforms this number to a human readable string.

### 5.1.3 Retrieving the list

This information is then used to obviously identify the running system, but also the tool that will be used to retrieve the list of installed software. The method to retrieve software information is then deduced once the system has been identified. Those tools can be divided in three main parts:

For Debian based system, dpkg is used to get the list.
For Mandrake, SuSE or Red Hat, rpm is used.
Gentoo typically uses equery.
Other distributions like Solaris, AIX, FreeBSD... have their own way to retrieve the list. For example, on Solaris, pkginfo is used, whereas HP-UX uses swlist and FreeBSD uses pkg_info.
The output can then be piped through awk or sed, or matched with any regular expression to obtain only the name of the software and its version.

The strategy used on Windows system is significantly different. There is no default installation of Windows that contains a tool that can create a list of existing applications. The best way to proceed is to act as Windows application Add or Remove software do. Whenever a piece of software is installed, a new key is created in the registry under the key HKEY_LOCAL_MACHINE at this location:

```plaintext
Software\Microsoft\Windows\CurrentVersion\Uninstall
```

A piece of software usually sets up its entries using its name, its version and its access path. Calls to the next Windows Advapi32.dll defined functions let a program to access a key and read its contents.
LONG WINAPI RegOpenKeyEx(
    __in HKEY hKey,
    __in_opt LPCTSTR lpSubKey,
    __reserved DWORD uOptions,
    __in REGSAM samDesired,
    __out PHKEY phkResult
)

LONG WINAPI RegQueryValueEx(
    __in HKEY hKey,
    __in_opt LPCTSTR lpValueName,
    __reserved LPDWORD lpReserved,
    __out_opt LPDWORD lpType,
    __out_opt LPBYTE lpData,
    __inout_opt LPDWORD lpcbData
)

LONG WINAPI RegCloseKey(
    __in HKEY hKey
)

The agent opens a key, and for each entry belonging to this key, scans for the name and version fields.
Once done, whatever the system is, the report contains two distinct parts:
   i. A header part, where system information are explained
   ii. A list of "software, version". To make parsing easier, these parts are separated by a star.

5.1.4 Signing the report

In the requirements, security is perhaps the most important thing. The report must be sent in a secure way, so in other words, data sent on the network must be encrypted and signed, to protect them for a potential interception or modification. Each server has its own certificate for authenticating purposes. This certificate will therefore be used to sign the report. OpenSSL's source code provides several functions to sign using a certificate. Signing is usually done with the command:

openssl dgst -binary -out <file_where_checksum_is_stored> -sign <certificate> <file_to_sign>

The prefix 'binary' indicates that the checksum will be displayed in binary format. A prefix 'hex' encodes the binary in base64 format, which is human readable.

<OpenSSL/dgst.c> contains the code related to this feature. The most important calls are at lines
271: bmd=BIO_new(BIO_f_md());
326: sigkey = load_key(bio_err, keyfile, keyform, 0, NULL, e, "key file");
360: if (BIO_set_md(bmd,md))
445: i=BIO_read(bp,(char *)buf,_BUFSIZE);
478: if(EVP_SignFinal(ctx, buf, (unsigned int *)&len, key))
load_key is defined in <OpenSSL/apps/apps.c>. Since this function is not exported by default, it is necessary to redefine it. Here is the prototype described in this file:

```c
EVP_PKEY *load_key(BIO *err, const char *file, int format, int maybe_stdin, 
    const char *pass, ENGINE *e, const char *key_descrp)
```

EVP_PKEY is an internal OpenSSL’s structure representing the key. Each of its fields is quickly accessible and consequently this object is very easy to use. The function prototype carries a lot of parameters that can be ignored due to the specifications of the agents. Machines where the agent will be run possess a certificate in PEM format that does not take any password to protect the certificate. Consequently this function can be shortened to:

```c
EVP_PKEY *load_key( char * filename ) with a main call to 
    PEM_read_bio_PrivateKey( ... ), a macro defined in 
    <OpenSSL/crypto/pem.h>:

#define PEM_read_bio_PrivateKey(bp,x,cb,u) (EVP_PKEY *)PEM_ASN1_read_bio( 
    (char *(*)(char **))d2i_PrivateKey,PEM_STRING_EVP_PKEY,bp,(char **)x,cb,u)
```

A call to this function sets up an EVP_PKEY object that will be used later during the digest process.

Several algorithms can be used to do a checksum. OpenSSL provides mainly MD5, SHA-1, SHA-256 or SHA-512. MD5 has been chosen, but it can be replaced by any of these algorithms. In OpenSSL, the MD5 algorithm is represented by a structure EVP_MD and initialised by const EVP_MD *EVP_md5() defined in <OpenSSL/evp.h>.

The message digest algorithm also has to be initialised before any use. To do this, a call to BIO_f_md() must be completed. It returns the message digest BIO (Basic I/O) method. This is a filter BIO that digests any data passed through it, and basically acts as a BIO wrapper for the digest routines EVP_DigestInit(), EVP_DigestUpdate() and EVP_DigestFinal().

Thus, any data written or read through a digest BIO using BIO_read() and BIO_write() is digested "on the fly".

Eventually BIO_set_md() sets the message digest to the MD5 algorithm previously set up. This must be called to initialize a digest BIO before any data is passed through it. It acts as a BIO control macro. At this point, each data passed through an I/O function is digested.

File management functions are also provided by OpenSSL and are defined in <OpenSSL/bio.h>:

```c
int BIO_read(BIO *b, void *data, int len);
int BIO_write(BIO *b, const void *data, int len);
BIO *BIO_new_file(const char *filename, const char *mode);
BIO *BIO_read_filename(BIO *b, const char *name);
```

Finally, the digest is computed with the key to form the final signature. This is realized by EVP_SignFinal defined in <OpenSSL/evp.h>.

```c
int EVP_SignFinal(EVP_MD_CTX *ctx, unsigned char *sig, unsigned int *s, EVP_PKEY *pkey)
```
This function takes the data in the context ctx, and signs them using pkey. The result signature is then placed into the buffer pointed by sig.

Once the list has been dropped to a file, the function int SignFile(string Filename, char *Checksum) coordinates all the previously explained method to digest the list, and creates a MD5 checksum computed with the private key.

The report is then concatenated to the signature and is ready to be encrypted. For more convenience, the signature is converted to a human readable format. So the first 256 bytes of the report form the signature that will be later checked by the server to authenticate the report.

5.1.5 Encrypting the report

Manually, this step can be done with the following command:

```bash
openssl smime -encrypt -aes128 -in <file_to_encrypt> -out <encrypted_file> <certificate>
```

OpenSSL provides lots of encryption algorithms, such as for instance, Blowfish, RC4, RC5, AES, DES, DES-3... and some of them are compatible with some additional block chaining module (mainly Cipher Block Chaining, Cipher Feedback or Output Feedback). AES is relatively strong, and no flaw has been discovered in its implementation, so it was decided that this algorithm would encrypt the data. For more accuracy, AES-128-CBC was chosen. As a result, encryption is performed using 128 bits blocks, each block xored with the previous one and the first data block is actually generated by the OpenSSL random number generator. This generator is set up by this function:

```c
const char *RAND_file_name(char *file, size_t num) defined in <OpenSSL/rand.h>.
```

AES-128-CBC is described as an EVP_CIPHER object defined in <OpenSSL/evp.h>:

```c
EVP_CIPHER *EVP_aes_128_cbc()
```

Once the cipher has been set up, the encryption process is mainly achieved by the function PKCS7_encrypt.

```c
PKCS7 *PKCS7_encrypt(STACK_OF(X509) *certs, BIO *in, const EVP_CIPHER *cipher, int flags) defined in <OpenSSL/pkcs7.h>
```

This function takes in parameters the list of certificates that will be used during the process, the data to encrypt and the cipher. On success, this function returns a Public Key Cryptography Standard #7 (PKCS7) enveloped data structure (used in Secured Multipurpose Internet Mail Extension – S/MIME for example).

Unfortunately, once data have been encrypted, they are directly encoded in Base64 which cannot be avoided. One of the main advantages of such a method is that it provides a great portability. It is then possible to send an encrypted email with ASCII content. In this case, an administrator who, for some reason, wants to manually send its software list just needs to select the file and sends it. On the other hand using Base64 to encode data increases the size of the message. However, since the list of software usually does not contain more than 50Kb, this process does not have a significant impact on the network load.
Encrypted data are then stored in a file with this function also defined in `<OpenSSL/pkcs7.h>`:

```c
int SMIME_write_PKCS7(BIO *bio, PKCS7 *p7, BIO *data, int flags)
```

At this step, the report is ready to be sent. Hereunder is showed a sample of such a report:

MIME-Version: 1.0
Content-Disposition: attachment; filename="smime.p7m"
Content-Type: application/x-pkcs7-mime; smime-type=enveloped-data; name="smime.p7m"
Content-Transfer-Encoding: base64
MlKOPQYJKoZIhvNQAqCDolKOLjCCjioCAQAxggEXmlIuBLQIBADCBITCBhvELMAkG
A1UEBhMCRAVMxExAQBgNVDABgTCUJhcmmNlbG9uYTESMBAGA1UEBxMJsBQyYz2Vsb25h
MQwwCgYDVQQKEwNVUEMxDDAKBgNVBAAsTA0JQjEEMWMBQGA1UEAxMNMTkyLjE2OC41
...

5.1.6 Sending the report

This part is basically the same for Windows and Linux systems. Only a few function calls differ, so only the Windows version is explained here.

On Windows, socket calls are made through the Winsock API. All the following functions are defined in `<winsock.h>`
The Winsock library is initiated by

```c
int WSASStartup(WORD wVersionRequired, LPWSADATA lpWSADATA).
```

A new socket is initiated by SOCKET socket(int af, int type, int protocol). In this case, the type SOCK_STREAM is used because the connection will be made through the TCP protocol. This function returns a handle to the socket that has been created.

The connection is created by:

```c
int connect(SOCKET s, const struct sock_addr* name, int namelen).
```

Thus, an object sock_addr has to be defined first. Its fields are filled with the following data:

```c
sin_family = PF_INET
sin_port = htons(port)
sin_addr = *(struct in_addr *)he->h_addr
```

where `he` is a pointer to an hostent object that has been initialised by struct hostent *gethostbyname(const char *name), name representing the IP address of the server.

In case of success, connect(...) returns a null value indicating that data are allowed to be sent through the socket.

Data are then sent by:

```c
int send(SOCKET s, const char *buf, int len, int flags)
```

And finally the connection is terminated with a call to:

```c
int closesocket(SOCKET sock).
```
5.1.7 Compilation and dependencies

On Linux, no compilation is needed since the agent is written in Perl. Having installed Perl and OpenSSL is sufficient to execute the program. Both applications are generally found by default in UNIX distributions.

On Windows, several versions of Visual Studio can be used. They all need OpenSSL to be previously installed and compiled, with its library and include path set in the Visual Studio’s paths setting, but further program’s dependencies can appear according to the version used to compile.

Visual Studio 6.0 can be used, if the SDK platform has been previously installed. This comes from the fact that wSuiteMask of the OSVERSIONINFO object is not defined by default in winbase.h. Platform SDK updates the libraries so that it is then possible to take advantage of these new properties. In such a case, no additional dependencies are needed. However since Visual Studio 6.0 is not anymore supported, the SDK platform cannot be directly downloaded and has to be ordered by CD on MSDN website.

Visual Studio .Net 2003 and 2005 are by default compatible with the updated OSVERSIONINFO object, so both can be used to compile. It is nevertheless easier to compile with the 2003 release because Visual Studio 2005 generates files with dependencies. Unless the Visual Studio 2005 exportable libraries are installed on a system, the program will not work properly. Therefore, an additional package whose size is around 2.5 MB should be downloaded by the administrator before installing anything.

2003’s version does not require a lot of specific libraries after compilation. A single library MSVCR70.DLL is necessary so it can easily be provided with the downloadable package. Provided this library, the executable is totally autonomous.

Finally, the agent must be distributed with OpenSSL’s shared libraries sasleay32.dll and libeay32.dll because otherwise the application will not initialize properly.
5.2 Server

5.2.1 Overview of the Server

The server does the following tasks, step by step:
- Listen for a new connection.
- Accept it.
- Dispatch the handling of the connection to a new thread.
- Listen again.

The thread created processes the report:
- Retrieve all packets.
- Decrypt the report.
- Verify the signature.
- For each entry of the report:
  - Get the corresponding CMSI data.
  - Save them in a CMSI XML file.
  - Update the admincmsi database with these data.

The server should be started by a Cron job or as a scheduled task. It only requires the right to listen to a port, and the right to connect to the database.

5.2.2 Behaviour

5.2.2.1 Main procedure

The main procedure of the server acts as a loop waiting for connection and dispatching the handling of the connection to a new thread. As a result, multiple connections can be processed at the same time, and agents do not have to wait to send their report. Hereunder are the main parts of this procedure.

A socket is firstly created and asked to accept remote TCP connections. This is completed by the following functions, defined in <sys/socket.h>:

```c
int bind( int socket, const struct sockaddr *address, socklen_t address_len )
int listen( int socket, int backlog )
```

When an agent connects, the connection is accepted by the ‘accept’ function:

```c
int accept( int socket, struct sockaddr *restrict address, socklen_t *restrict address_len)
```

The connection is then handled by a thread named routine. The next function is defined in <pthread.h>.

```c
int pthread_create( pthread_t *thread, pthread_attr_t *attr, void *(*start Routine)(void *), void * arg);
```

5.2.2.2 The thread

The thread routine can be detailed into 2 main steps:
• Reading packets and storing them in a file until all packets have been sent
  ssize_t read(int fd, void *buf, size_t count) (defined in sys/types.h)
  AppendToFile( const char * Filename, const char * Data, unsigned long DataSize )

• Decrypting and extracting file information
  int ProcessFile( string * ipparam, string filename = "" )

This last function realises the main part of the server. It analyses the report, extracts its information and queries database to obtain and register CMSI information.

This job is shared between different functions:
  • DecryptFile
  • ExtractSignature
  • VerifySignature
  • ExtractOS
  • ParseHostInfo
  • CMSIHostQuery
  • CNSIQuery

A CMSI XML report is then generated and contains every piece of information in a CMSI based format.

5.2.2.3 Decrypting the report

DecryptFile and VerifySignature behave in the same way than the corresponding agent’s functions. They indeed act in a symmetrical way.

For the decryption function, the certificates are first loaded:
  cert = load_cert( Public_Certificate )
  key = load_key( Private_Key )

cert represents the certificate the agent has used to encrypt the list and key represents the key to decrypt. They both call functions in <openssl/pem.h>:

EVP_PKEY *PEM_read_PrivateKey(FILE *fp, EVP_PKEY **x, pem_password_cb *cb, void *u)
X509 X509 *PEM_read_bio_X509_AUX(BIO *bp, X509 **x, pem_password_cb *cb, void *u)

The rest of the function is extremely similar to the agent’s encryption function. Two main calls ensure the decryption of the contents. On the agent side the encryption was done using:

PKCS7 *PKCS7_encrypt(STACK_OF(X509) *certs, BIO *in, const EVP_CIPHER *cipher, int flags);
int SMIME_write_PKCS7(BIO *out, PKCS7 *p7, BIO *data, int flags)

And on the server side:

PKCS7 *SMIME_read_PKCS7(BIO *in, BIO **bcont)
int PKCS7_decrypt(PKCS7 *p7, EVP_PKEY *pkey, X509 *cert, BIO *data, int flags)

On the server side, the PKCS7 structure is defined while the contents of the list have not been deciphered yet. Decryption is then performed by the call to PKCS7_decrypt.
5.2.2.4 Verifying the signature

Here, the first main call is the call to `EVP_KEY *load_pubkey (const char *file)`. It loads the public key associated to the certificate used to sign the list. The server determines the key that has to be employed according to the IP address of the agent.

The signature provided by the agent is then loaded into memory using the following call:

```c
int BIO_read(BIO *b, void *buf, int len) (defined in <openssl/bio.h>)
```

And finally, int `EVP_VerifyFinal(EVP_MD_CTX *ctx, unsigned char *sigbuf, unsigned int siglen, EVP_PKEY *pkey) (in <openssl/evp.h>)` computes a signature for the list of applications and checks it with the provided signature. A positive return value indicates that the file has been authenticated, a null value indicates a signature mismatch, and a negative value indicates an internal error. The last case is usually raised because of memory access or file access troubles.

5.2.2.5 Extracting and parsing host data

This part is completed by the int `ParseHostInfo( string *HostInfo , Host *host)` function. It first creates a Host object:

```c
typedef struct _Host {
    string osname;
    string label;
    string osversion;
    string hostname;
    string archi;
    string servicepack;
    string language;
    string version;
    string kernel;
    string kernelversion;
    string ip;
    string DBindex;
} Host;
```

and once done, fills each of its fields with data extracted from the header of the list. Once done, the hostname field is passed to void `WriteCMSGHeader( ostream &CMSI_file, string ip, string hostname )`, which writes the header of the CMSI XML file. This header actually contains the name of the machine, its IP address and the current time. Thus, the machine which sent the report can be fully identified.

So as to obtain CMSI data associated to the OS name and its version, osname and osversion are concatenated and passed to int `CMSGHostQuery( ostream &CMSI_file, Host* host)`. Thereby, there is no difference from the database point of view between a system and an application. It is a consequence of the CMSI contents description: both are registered the same way into the database. There is no formal difference.
The rest of the list is processed in the same way as the system identifiers and passed through void CMSIQuery( ostream &CMSIfile, string applist, Host *host).

5.2.2.6 Querying the database

Since the database is driven by MySQL, MySQL++, a C++ wrapper for MySQL’s C API, will help a lot to provide access.

The connection to the database is achieved through the mysql++ function Connection::Connection(const char* db, const char* server, const char* user, const char* password, unsigned int port).
It takes into parameters the name of the database, its location, and the credentials required to access it. An object Connection is then created, and will be used during all the transactions.

Requests to the database are made by the function Connection::Query query(const std::string& qstr). A typical sql request for CMSI information takes the following representation:
select family_tag from family, family_parents where family.id in ( select family from appname where app = 'application_name' ) and family.id = family_parents.family

Thus, for each entry in the software list, CMSI information is requested. In case of no result, it means that the database does not contain information regarding the application. This occurs for instance in Linux systems. Lots of applications are identified, but however only a few are really interesting.

```cpp
int RetreiveCMSIData( string appname, mysqlpp::Connection* conn, CMSI_data * cmsidata )
```

is aimed at querying the database and retrieving the corresponding CMSI tags.

Those results are finally written to the report by the function:

```cpp
void WriteCMSI( ostream &CMSIfile, CMSI_data* CMSIdata )
```

It writes both OS information and software information. Below are two examples of an XML representation of a software entry:

An OS entry:

```
<Family tag="w2k">
<Parents>
  <Parent tag="win"/>
</Parents>
<Names>
  <Name>Windows 2000</Name>
</Names>
<Description>The Windows 2000 product family</Description>
<Products>
  <Product tag="server">
    <Names>
      <Name>Server</Name>
    </Names>
  </Product>
</Products>
```

54
A Software entry:

```xml
<Family tag="ms_office">
  <Parents>
    <Parent tag="office"/>
  </Parents>
  <Names>
    <Name>Microsoft Office</Name>
  </Names>
  <Description>The Microsoft Office product family</Description>
  <Products>
    <Product tag="ms_office_xp">
      <Names>
        <Name>Microsoft Office XP</Name>
      </Names>
    </Product>
  </Products>
</Family>
```

All those events are coordinated in void CMSIQuery( ostream &CMSIFile, string applist, Host *host ). This function takes the whole list of application in argument, initiates the connection to the database and afterwards extracts each line to eventually call:

```c
int RetrieveCMSIData( string appname, mysqlpp::Connection* conn, CMSI_data * cmsidata )
```

### 5.2.2.7 Updating database

Each time a report is sent by an agent, the adminncmsi database is automatically updating so that administrators will later be able to consult their data. In the code this part is performed by two functions:

```c
int UpdateServerDB( mysqlpp::Connection *conn2, Host *host )
int UpdateServerInfoDB( mysqlpp::Connection *conn, CMSI_data *cmsidata, Host *host ).
```

The first one is aimed at adding or retrieving an identifier for the machine that has sent its report. If the machine is already registered in the database, its identifier is returned; otherwise, a new machine record is added. Administrators should later register this machine through the PHP web application for example.
The second one registers or updates software information with CMSI identifiers to the database. Each time a new application is inventoried a new entry is added or updated under the table serverinfo.

### 5.2.2.8 Compilation

This application basically works only on a Linux system. To compile, MySQL++, MySQL-dev and OpenSSL development tools and headers must be installed. Otherwise the compilation fails.

```
g++ server.cpp -o server -g -Wall -I/usr/include/mysql -I/usr/local/include/mysql++ -lmysqlpp -lssl -lpthread -lmysqldclient
```

### 5.3 A secondary database

Since an administrator must have the right to consult or change its machine configurations, each time the server receives a report, he has to convert it into a CMSI report, but also has to keep all these information into a database, for later use. Afterwards, the administrator can connect to the web application and check its active configurations. Here creating a new database seem to be the better choice because PHP and SQL are definitely compatible, but one could object that given that all the CMSI reports are accessible, filling a new database would be only a redundant operation. It is however much easier to access information using SQL requests than implementing a new XML parser in PHP for these reports.

#### 5.3.1 Structure

The database does not need to contain lots of section. First, a user must login to the web application. In no way a guest could be able to modify a machine he does not administrate, besides if he is not an administrator but only a lambda user. So as a login step is necessary, a table containing admin credentials like his name and surname should be defined, and so does its password. Leaving a password in plaintext in a database can appear insecure since in case a data theft anyone would know its password. Nevertheless, registering a MD5 sum of this password appears more secure given that for the moment the algorithm has not been cracked. Dictionary attacks exist, but they definitely turn to be brute force attacks. Finally, this administrator could be identified by a unique number (the key of this table, ID number).

Servers behave the same way than administrators. Instead of registering their names, they can be identified by their IP address and their machine name. To determine which administrator manages which machine, they should be attributed its ID. This scheme implies that a machine can be managed by only one administrator. A unique ID number should finally identify them.

To conclude, software could be identified by their CMSI identifier (found in the CMSI database, table product), and a version attribute. The table should
Additionally contain the ID of the server software are installed on, so that they can be associated to them.

![Database Scheme](image)

**Figure 8 - Scheme of the Database**

### 5.3.2 Access

Once the data have been converted to CMSI data, the server can update the database with the new information. This update consists in ensuring it is the first time the machine sends its report. If it is, then a new entry with machine information is inserted. Else, the ID of the machine is retrieved:

```sql
select Id from admincmsi.server where hostname='HOSTNAME' and ip='IP'
```

Or, if no result:

```sql
select max(Id)+1 from admincmsi.server
insert into admincmsi.server values (SERVERID, 'HOSTNAME','IP','-1')
```

As there is no way to deduce who is the administrator the machine belongs to, -1 is inserted as an ADMINID. This entry should be manually corrected later. A preliminary check is afterwards done to ensure whether it is the first time the application is going to be registered. If the result of the query is empty then a new entry is inserted:

```sql
insert into admincmsi.serverinfo values (SERVERID, 'PRODUCT_ID','VERSION')
```

Else the database is updated:

```sql
update admincmsi.serverinfo set version='VERSION' where ServerId=ID and cmsid in (select id from cmsi.product where product_tag='PRODUCT_TAG')
```
A final commit is realised, letting the administrator the ability to check its most recent configuration via the PHP application.
6 Tests

6.1 On client side

Because of its ability to work on multiple platforms, the agent has been written in multiple steps, stepping from one system to another. On UNIX platform, it currently works on Ubuntu, Debian, SuSE, Gentoo, Fedora, Red Hat, Mandriva, Solaris, FreeBSD and OpenBSD.

On Windows platform, it has been successfully tested on Windows XP SP2 and SP3, Windows 2000 Server and Advanced Server, Windows 2003 Enterprise Edition SP0 and SP2 and Windows Vista.

It was obviously not possible to conduct those tests on a single machine alternatively running a different system. Such a hypothesis would have consumed a lot of hard drive space but also it would have forced to create early lots of empty partitions. Therefore, a powerful alternative was selected: using virtual machines.

Typically, a virtual machine is created on a host and takes advantage of all of the resources of the system without being aware it is in fact emulated. It is then possible to create virtual machines that share the same disk space than the host, and a part of its memory. They can coexist without any problems and are very easy to manage. Most of the software that emulate virtual machines leave the possibility to take a snapshot of the machine, so that, in case of any problem, users can return later to this previous save. Consequently, there is no need to do any kind of system restoration since previous safe copies can be instantly reloaded.

But their main interest is to offer a share of different resources, such as network or memory, at the same time. Therefore, running five different servers on the same machine becomes possible due to this technique. An administrator can install as many servers as the system resources allow to and run multiple services on those machines. So a web server, a mail server and a dns server can live together in the same host but with different characteristics (ip address, machine name, memory amount...). This is a real opportunity for testing. As it is always possible to step back, one can install a piece of software, try it or check it, do some tests, and later step back as if nothing had ever happened.

Among the different products that provide virtualization capacities, four are really interesting. Microsoft VirtualPC, not free, has been designed for Windows and is relatively easy to use. But since the system that has been used for designing the server works in a Linux environment, this option was soon set aside. In a Unix environment, three other tools provide virtualization:

- VMware
- Xen
- VirtualBox

Xen is highly powerful because the virtualized machine is not really virtualized. This tool actually makes several machines share the same
environment, as the other, but instead of being emulated, the code is executed natively. It means that two systems can coexist and can really use the same resources. In such a system, none of the machine can be considered as a host, since they all offer the same privileges. Consequently the main advantage appears in the performance. A native code is much faster than an emulated one. So for servers that require lots of resources this solution is bound to be privileged. On the other hand, its complexity makes it less accessible for private OS like Windows, because the source code of the OS has to be modified in order to work correctly. As a result it appears that this tool is not really recommended for our purposes, and moreover because OS like Windows Vista are not correctly handled.

VMware offers standard possibilities to make virtual machines. It declines in two offers, VMware Workstation (free) and VMware Server (not free). The second one is more destined to companies with its server oriented mode. Installed as a service, it offers the possibility to create accounts for administrators so that they can later manage their servers using Remote Desktop Connections. VMWare Workstation is more aimed at testing due its backup functionalities. Its different options and the aspect of its graphical interface make it a good opportunity for common users. It is besides compatible with lots of systems and thus appears as a strong competitor in the virtualization market.

VirtualBox is perhaps the most convenient virtualization application to use. It is under a GPL licence, therefore free, and offers a highly user friendly interface. In terms of performance, it is approximately equivalent to VMWare Workstation but its ease of use is unquestionably better. On the other hand it might suffer from less compatibility with systems, making it on this point less competitive than VMWare Workstation. However compatibility with systems was enhanced in the last version (1.6.0).

I opted for VirtualBox 1.5.2 at the beginning because I was already familiar with this application. I had already used it to make several Windows and Linux machines virtual, so I thought it could be the best way for testing. Unluckily, the lack of compatibility with FreeBSD and Solaris led me to choose also VMWare Workstation to work with those systems. But when version 1.6.0 was issued I decided to leave VMware and stay with VirtualBox.

Basically, the agent was designed this way. First, a new hard drive was created on VirtualBox and a new system was then installed on. Once fully configured, the home directory on the host was shared, so that it was possible to work freely with the files associated to the project. In fact, it was as if a new computer had been set up and was using the network share of another machine. There is no real difference between UNIX systems; they more or less behave the same way, so designing the agent on them was like repeating the same procedure. Those tests were however useful because they soon illustrated problems with the first designing choice (developing the UNIX agent in Shell Script). Indeed, it uncovered an unexpected problem relying on the use of telnet. Since I opted to use raw sockets to transfer a report, I contemplated telnet a good opportunity to satisfy my purposes. But problems started to appear with Mandriva. Due to its default mode, the telnet client did not behave like it was expected to, sending lines instead of characters, and dropping connection after a
random amount of byte had been transferred. This could surely have been solved with a closer look to its default configuration, but a major telnet related issue was uncovered on SuSE OS. Actually, for security reasons, the telnet client had been removed from default installation, therefore meaning that this definitely could not be used to transfer data. This part underlines the importance of testing an application at each step of its process, because sometimes, like in this case, design errors might appear requiring a complete review of the implementation step. Fortunately, Shell Script was not so different from Perl so re designing the code was quite easy. A more complete application might have triggered a more complicated situation! Anyway, the telnet solution had never been elected in inventory software, leading us to think that it was not a so good solution.

### 6.2 On Server side

Several tests have been conducted to ensure the behaviour of the server with uncommon and weird reports. Servers are always privileged targets from the attacker point of view, because they may use unsecure applications that can suffer from a vulnerability remotely exploitable. But without even thinking to this catastrophic case, it is interesting to try to audit an application to see if it contains bugs easy to find, that may crash the server. A server that often crashes is very embarrassing for an administrator. Here is a few list of the tests that were conducted to find bugs:

- long application name (50 kb)
- long report (2 Mb)
- empty report
- high rate report sending
- SQL injection
- incorrect application name (null size)
- incorrect report header (lots of carriage returns, lost of blank, comas, long fields, etc.)

In the specific case of the server, it is pointless to generate and send random reports because of the cryptographic mechanism. Each time a report is sent, the decrypt function in OpenSSL is called to decipher the report. An incorrectly ciphered report is instantly reported with an error raised by a PKCS7 routine. And so does a report containing an invalid checksum. If the checksum does not match with the report sent, the main thread stops right away so the report is dropped. However some weird cases exposed later might regardless trigger some bugs in the program. Consequently only intelligent testing should be used here: a report correctly ciphered and a valid corresponding checksum.

Many of these tests have contributed to uncover bugs in the piece of software and as a result improve its security. The main example of those bugs was a problem in the allocation of memory. In some cases, memory was allocated with a new operator, and was not freed before the thread exits, and a bug related to the free of this part of memory could be triggered after lots of reports have been processed. This bug was uncovered using a high rate report sending method. Assuming that an agent sends lots of reports in a few time, it may triggers some
random bugs. This code made the problem surface after about 300 memory allocations.

Another measure that was taken was to limit the size of the report. During the loop to retrieve data in the application, a threshold was set for the report size. Typically a report may contain a maximum of 100KB, in case for example of a workstation where lots of different software would be installed. These install several dependencies that may figure in the final package list and therefore may contribute to a large sized report. On a standard Ubuntu distribution, with common packages installed, this list may not reach 36KB. Then it can be extended up to 100KB in case it is used for other purposes. But for servers, this size may never be reached. Consequently rejected any reports that might contain more bytes that actually needed is a powerful security measure. Someone will not be able to make it busy sending overly long reports.

Lots of other bugs were uncovered using fake or wrong formatted reports. During the development, the string class has often been used, mostly with its method `substr()`. This method returns a new string containing a part of the original string. In normal conditions, a previous call to the method `find()` is realized to retrieve an index of the first position of the string searched. But in case this string does not exist (for instance, the star delimiting the header and the list), the function will return `-1`. If this number is not checked before using with `substr()`, `substr()` will raise an exception when called, attempting to read more memory than existing. Actually `-1` is converted as an unsigned long and so `substr()` tries to read $2^{32}-1$ bytes from the memory leading to a crash. Those bugs were mostly uncovered by using empty entries like for example in a report:

```
codename: Enterprise Edition
osversion: 5.2.3790
servicepack: Service Pack 2*
```

Here, the code assumes at least one carriage return will be used between the header and the star (the star delimits the application part and the header part). A lack of carriage return would lead the following code to crash while calling `substr()`:

```
j = HostInfo->find_first_of("\n", i+1);
val = HostInfo->substr(i+lenkeyword,j-i-lenkeyword);
```

`lenkeyword` represents the size of the key word used to parse the information, so it equals 11 in the 'servicepack' case. A missing carriage return would lead `j` to equal `-1`, which means $2^{32}-1$ as an unsigned long and therefore crash the `substr()` method. A workaround was to put:

```
if ((j-i) > len) return EXIT_FAIL;
```

with `len` referring to the size of the header, so that the report would be considered corrupted in case of a wrong header:

```
j = HostInfo->find_first_of("\n", i+1);
if ((j-i) > len) return EXIT_FAIL;
val = HostInfo->substr(i+lenkeyword,j-i-lenkeyword);
```

The same problem was found in the management of the signature. A report containing less than 256 bytes would trigger an error because of:
if (OpenFile( (Filename+".txt").c_str(), &content) == EXIT_FAIL) {
    return EXIT_FAIL;
}
signature = content.substr( 0, 256 ); <- crash due to lack of data

A simple test is enough to remove this bug:
if (content.size() <= 260) return EXIT_FAIL;

Some time was also dedicated to avoid SQL problems related to the names of the programs in the report. As the server requests for CMSI information regarding the nature of a program, its name was first directly used in a SQL request:

requeststart = "select family.family_tag, family.name, family.descr, family.parents.node_tag, product.product_tag, product.name, product.descr, product.id from family, family.parents, product where product.family = family.Id and family.parents.family = family.Id and product.name";
requestmid = "=";
requestend = ""
...
request = requeststart + requestmid + appname + requestend;
mysqlpp::Query query = conn->query( request );

This part of the code works perfectly in normal conditions. The name of the program that needs CMSI information is appended to the SQL request leading the server to query the database for those additional details. In case the name simply does not exist, then the database drops the requests and returns no result found. Therefore, if this name was not correct, because too long for example, nothing would occur because of the secure data handling in mysql++.

Typically a common request has this form:
"select family.family_tag, family.name, family.descr, family.parents.node_tag, product.product_tag, product.name, product.descr, product.id from family, family.parents, product where product.family = family.Id and family.parents.family = family.Id and product.name = 'APPLICATION_NAME'

where APPLICATION_NAME equals for example 'mysql-server'.
Thus it seems to be possible to manipulate this string to do other things. Assuming the entry found in the report was "mysql-server"; drop table cmsi_product", people with malicious attentions can alter the good behaviour of the database leading to just a simple altered entry to a complete data compromising. Such an attack is called a SQL injection, and it is quite difficult to protect from. But in our case, since APPLICATION_NAME is delimited by quotes, the only way to escape from those quotes is to directly print one in the string like the example above. Therefore it is quite easy in this case to defend the code from malicious intentions by filtering any request containing a single quote. From the attacker point of view, this piece of code might appear vulnerable, but finally is impossible to exploit.

Such a vulnerability also exists in this part:

requeststart = "select Id from admincmsi.server where hostname=";
requestmid = "" and ip="";
requestend = "";
request = requeststart + host->hostname + requestmid + host->ip + requestend;

Here host->hostname refers to the name of the station that sends the report. The string can easily be manipulated to let malicious people corrupt the database. In normal conditions, the request looks like:

```
select Id from admincmsi.server where hostname='MACHINE_NAME' and ip='192.168.0.5'
```

In this case, host->ip cannot be controlled because the ip is taken from the socket properties. But a hostname field like "machine_name"; drop table admincmsi.server; select Id from admincmsi.server where hostname='boom' could lead to querying the database for:

```
select Id from admincmsi.server where hostname='machine_name' ; drop table admincmsi.server; select Id from admincmsi.server where hostname='boom' and ip='192.168.0.5'
```

Obviously this is extremely critical but can be avoided by a simple function before doing any request:

```
int VerifyQuery( string * query ) {
  if (query->find("'") >= 0)
    return EXIT_FAIL;
  return EXIT_SUCCESS;
}
```

If any single quote is found, the function returns EXIT_FAIL showing an error occurred and no request is forged.

The flaw also existed in:

```
requeststart = "update admincmsi.serverinfo set version=";
requestmid = " where ServerId=";
requestmid2 = " and cmsid=";
request = requeststart + (cmsidata->osversion=="" ? cmsidata->version : cmsidata->osversion) + requestmid + host->DBindex + requestmid2 + cmsidata->product_tag + ((string)"");
query <<= request;
query.execute();
```

and

```
requeststart = "select family.family_tag, family.name, family.descr,
family_parents.node_tag, product.product_tag, product.name, product.descr,
product.id from family, family_parents, product where product.family=family.Id
and family.name="";
requestmid = " and product.name="";
requestend = " and family_parents.family=family.Id";
```

if ( conn.connect( "cmsi", "localhost", "root", "root" ) ) {
  request = requeststart + host->osname + requestmid + host->label + requestend;
}

64
mysqlpp::Query query = conn.query( request );

In the first case, cmsiddata->version or cmsiddata->osversion are taken directly from the report, and in the second case, the dangerous strings are host->osname and host->label also coming form the file. Here again the same corruption would be possible with crafted entries.

Note that it has been quite simple to protect smb share the code from a single quote injection, but it could have been much harder in other circumstances. Most of the time, it appears difficult to secure the code from those kinds of attack, but luckily this case was easy.
7 Conclusion

Six months ago, two main reasons led me choose this project. The first one was that the project was related to computer security, a theme I really liked. I was very interested to be concerned with a security topic for this last project, and designing a complete system related to was definitely a good opportunity for beginning. I thought I could discover this world with the help of the esCERT security team, and they did contribute to help me in. The second reason was that the project was held abroad. I had already spent a few months in Barcelona, and I was extremely motivated to renew for a second semester this experience.

Although there were several conditions that had to be respected in the specifications, I was very free to conduct this project according to my feelings. I was able to define how the applications would behave, how they would be designed and how they would interact. That is why this project was quite attractive. Nothing was already done, excepted the CMSI data base, and I had to design everything. The following diagram shows how the project has been conducted through four months:

![Gantt diagram of the project](image)

Some difference might appear between the first diagram realised at the beginning of the project and this one, due to for example a long testing phases. But as a rule, steps were correctly foreseen. Moreover this diagram is sometimes wrong because some parts of the code had to be designed at the same time. For instance, it was necessary to design both connection modules at the same moment to ensure its working.

The analyse part was also really interesting. I chose to test five open source applications with approximately the same characteristics, but each one was conceived without having any common point with another. They presented so many ways to transfer a report for example, that I had never heard about some of them. Studying such applications makes us discover things we completely ignored until then and so it contributes to make the project attractive.

I have moreover been forced to discover a new language: PHP. It appeared that finally it was very user friendly, and that complete applications could be developed within a few days. Without any real notion of PHP, I managed to create most of the part of the web formulary in little more than a week. Likewise, I had never been involved in writing Perl scripts. But I seen realized they were
powerful and could replace C/C++ programs in lots of case. Where 15 lines are needed to set a socket in C, only a couple is enough in Perl. I am actually wondering whether I should have tried to design the server itself in this language, or directly integrate it to the web application. But at the beginning, when I these two languages did absolutely not sound familiar to me, that is why I did not decide to go further in my investigation. If I had a little more knowledge in these technologies, chances are I would have dropped C++ for them.

However, I do not regret this choice, because I could explore once more the fascinating code of OpenSSL. I had already worked on its algorithms, how they were implemented and how a new one could be added. This time I explored how the code was built so that they could be called to sign or to encrypt data. Obviously that was not so easy, but spending time to understand how things work did definitely not appear as a waste of time.

Lots of time was dedicated to tests. I was already familiar with VirtualBox and its virtualization facilities. But I only experienced it on Windows machines. This time I received the opportunity to discover Linux guests. And fortunately, because the 1.5.2 release did not support neither FreeBSD 7.0 nor Solaris, I was forced to discover VMware Workstation, which appeared to be also highly competitive, but perhaps less user friendly. I did not try Xen because of it is said much more complicated, but my results with the previous ones have convinced me to try it.

The whole inventory system should be firstly installed in the network of the university. Chances are that I help administrators to configure and to solve problems during the last two weeks. The code I provided can be upgradable due to its simplicity. Perl scripts are easily changeable, so if someone decides to port the agent to a new system he should not experiment a lot of problems. The Windows agent works up to Windows 2008 Server, but in case new Windows are released, one might look to the MSDN support to receive help to upgrade the code. Because system recognition is based on system constants, they should be manually added to provide more recognition. It could however be assumed that the current system would work without needed to be updated for at least 2 or 3 years.
Products are organized according to their family. When the server asks for CMSI information for an application, the request is directed to the product name attribute. Consequently, if they correspond, the CMSI name is retrieved after the query and can then be exported to the XML report.

The model is very efficient because of its design. A machine that interprets the CMSI reports can then identify without any error its contents. Therefore, this model shows all its importance when it is about to determine which machine may suffer from a flaw. If the flaw has been described using CMSI data, a machine can instantly determine the vulnerable machines.
Annexe 2 – Creating certificates

Here is the complete process to generate two certificates, a private and a public one. First, it is necessary to generate a new certificate authority. Usually those are created by entities like Verisign. For example, when a web browser receives a certificate, it first tries to ensure that it has been issued by a trusted authority. Those organisms certify that they have provided this certificate therefore it can be considered as authentic. In this project, it was no use to ask such an authority to provide an authentic certificate. This generates costs that are not necessary for testing purposes. OpenSSL let a user create its own Certification Authority:

```bash
openssl req -new -x509 -keyout ca_key.pem -out ca_cert.pem -days 1024
```

This command generates a private key and the associated certificate for the CA. Note that this certificate will be valid during 1024 days.

This process asks for several details:

```
Enter PEM pass phrase: < password that will be used to encrypt the key
-----
You are about to be asked to enter information that will be incorporated into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter "." the field will be left blank.
-----
Country Name (2 letter code) [AU]:ES
State or Province Name (full name) [Some-State]:Barcelona
Locality Name (eg, city) []:Barcelona
Organization Name (eg, company) [Internet Widgits Pty Ltd]:UPC
Organizational Unit Name (eg, section) []:FIB
Common Name (eg, YOUR name) []:My CA Authority
Email Address []:email@somemail.com
```

Then, the certificate has to be signed by itself. It is then called an auto signed certificate:

```bash
openssl x509 -in ca_cert.pem -days 1024 -out ca_cert.crt -signkey ca_key.pem
```

Once done, this certificate seems to have been issued from a known organization. It can therefore be trusted.

**Generation of the certificate for the server**

It is first necessary to generate a pair of keys. One will be used to decrypt, the other to encrypt.

```bash
openssl genrsa -out myKey.pem 1024
```

The public key is then exported to be distributed:

```bash
openssl rsa -in myKey.pem -pubout -out myPublicKey.pem
```

At this point, two keys have been generated. It is now necessary to generate a certificate request for those keys:
openssl req -new -key myKey.pem -out MyRequest.pem

You are about to be asked to enter information that will be incorporated into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter ":", the field will be left blank.
-----
Country Name (2 letter code) [AU]:ES
State or Province Name (full name) [Some-State]:Barcelona
Locality Name (eg, city) []:Barcelona
Organization Name (eg, company) [Internet Widgets Pty Ltd]:UPC
Organizational Unit Name (eg, section) []:FIB
Common Name (eg, YOUR name) []:127.0.0.1 <- Name of the server. As it is used on a test machine, it is possible to put both 127.0.0.1 or localhost
Email Address []:mail@somemai.com

Please enter the following 'extra' attributes to be sent with your certificate request
A challenge password []:password
An optional company name []:UPC-FIB

Hereunder are the contents of this certificate, given by the following command:
openssl req -in MyRequest.pem -text -noout

Data:
Version: 0 (0x0)
Subject: C=ES, ST=Barcelona, L=Barcelona, O=UPC, OU=FIB,
CN=127.0.0.1/emailAddress@mail@somemai.com
Subject Public Key Info:
Public Key Algorithm: rsaEncryption
RSA Public Key: (1024 bit)
Modulus (1024 bit):
 38:69:ef:ff:10:c9:01:02:2f:9a:e7:ba:3a:df:76:
e4:5e:0e:ed:b4:50:bd:9a:0e:b4:d1:c0:be:6be9:
Exponent: 65537 (0x10001)
Attributes:
  challengePassword :password
  unstructuredName :UPC-FIB
Signature Algorithm: sha1WithRSAEncryption
The request is then submitted to the Certificate Authority that will eventually issue a valid certificate:

```
openssl x509 -days 1024 -CAserial ca_cert.srl -CA ca_cert.crt -CAkey ca_key.pem -in myRequest.pem -req -out myCertificate.pem
```

Now the certificate is digitally signed by our Certificate Authority. It means it can then be used to sign documents. It will attest of our authenticity. To verify the success of the operation, this command shows whether or not the certificate has been approved.

```
openssl verify -CAfile ca_cert.crt myCertificate.pem
```

As a result, it is possible to sign and encrypt documents. Assuming the file `temp.txt` contains the string "Hello world",

```
openssl dgst -hex temp.txt
```

will return a simple hash of the document:

```
f0ef7081e1539ac00ef5b761b4fb01b3
```

and:

```
openssl dgst -hex -sign myCertificate.pem temp.txt
```

will return a signed hash for the document:

```
MD5(temp.txt)=
868aa1f06a237385ef96ea06874bef15c48c4ed6d0fb181fe82e09726b7792c2dd2b62208f1f5f5eb70b380
4406c580909e3ee305889942a9c9d0ce3693e95a5676515aab73e608eecf78213108d3b73245f21181aac
9aa906a4c5649a7c03f2eb22f04a36028385ca600ec159330e578509a2a4e6ecfeb3537ffece4c0b5538
```

To verify (assuming the hash has been written to hash.sig):

```
openssl dgst -signature hash.sig -verify myPublicKey.pem temp.txt
```

Verified OK

If the string "Hello world" is replaced by "Hello works", a signature checks would uncover the corruption:

```
openssl dgst -signature hash.sig -verify myPublicKey.pem temp.txt
```

Verification Failure

We can also encrypt a document that only the owner `myKey.pem` will be able to read:

```
openssl smime -encrypt -aes128 -in temp.txt myCertificate.pem
```

```
MIME-Version: 1.0
Content-Disposition: attachment; filename="smime.p7m"
Content-Type: application/x-pkcs7-mime; smime-type=enveloped-data; name="smime.p7m"
```
This command encrypt a document using the AES 128 bits cipher, writes a SMIME header, and code in Base64 the encrypted data. Actually this is used to send ciphered emails over the network. One can for example use:

```bash
openssl smime -encrypt -in temp.txt -text -from emitter@mail.com -to destination@mail.com -subject "Ciphered email" -des3 myCertificate.pem | sendmail
```

to send to destination@mail.com a mail encrypted via Triple DES.

This seems to be the easiest way to sign a document with OpenSSL. Besides, as no user interaction is required, using this method in a script is regarded as highly advisable.

However it should also be noted that the certificate must not hold any password encrypting the private key. Otherwise, calling OpenSSL in a script would require asking the user his password before encrypting the report. And obviously, as the agent must be run without any user interaction, there is no possibility to let him precise his secret.
Annexe 3 – CMSI XML-based report

<System_list>
  <header>
    <ip>127.0.0.1</ip>
    <machine>BILOU-F10A357A6</machine>
    <current_time>05Jun2008</current_time>
  </header>
  <Families>
    <Family tag="wxp">
      <Parents>
        <Parent tag="win" />
      </Parents>
      <Names>
        <Name>Windows XP</Name>
      </Names>
      <Description>The Windows XP product family</Description>
    </Family>
    <Family tag="prof">
      <Names>
        <Name>Professional</Name>
      </Names>
      <Description>Windows XP Professional</Description>
    </Family>
    <Family tag="ms_office">
      <Parents>
        <Parent tag="office" />
      </Parents>
      <Names>
        <Name>Microsoft Office</Name>
      </Names>
      <Description>The Microsoft Office product family</Description>
    </Family>
  </Families>
</System_list>
This report has been generated from a machine running Windows XP and Office XP.
Annexe 4 - PHP Application

This application has been designed to let an administrator change and consult himself its machines' configurations.

After having logged on (page index.php), he can browse on info.php (main menu).

![Main menu](image1)

**Figure 8 - Main menu**

If he chooses to select a machine, a new screen appears showing him the contents of its machine. He can then change, remove or modify its software. If he needs to manually send its report, uploadconf.php provides the necessary:

![Upload a report](image2)

**Figure 9 - Upload a report**
Adding a new application is also very easy. After having selected the machine and choosing Add Software, a new form appears (addsoft.php):

![Figure 10 - Add Software](image)

Or:

![Figure 11 - Add Software](image)

One can then choose its software and version and add it as a new application.
References

The wikimedia foundation. Wikipedia, [online], http://www.wikipedia.org/ (consulted all over the project)

Securityfocus Symantec Corporation. Securityfocus, [online], http://www.securityfocus.com/ (consulted all over the project)

CERT-Verbund. Common Model of System Information (CMSI), [online], http://www.cert-verbund.de/cmsi/en.html (consulted in March and April)


The Debian Community. SSL keys, [online], http://wiki.debian.org/ (consulted in May)

The Ubuntu Community. Documentation francophone pour Ubuntu, [online], http://doc.ubuntu-fr.org/ (consulted all over the project)

Chebli, Charles. Signature et Chiffrement (report (DEA) - Saint Joseph Faculty), 2003 [online], www.lb.refer.org/memoires/583025CharlesCHEBLI.pdf (consulted in February and March)

Microsoft, The Microsoft Developer Network, [online], http://msdn.microsoft.com/ (consulted all over the project)

OCS Inventory Team. OCS Inventory NG (1.02 RC2) [Application]. 05/01/20008. http://www.ocsinventory-ng.org/ (consulted all over the project)

Zamroni, Aris. zCI Computer Inventory System (3.1) [Application]. 25/04/2007, http://zci.sourceforge.net/ (consulted all over the project)

The Open-AudIT Crew. Open-AudIT (20071003v2) [Application]. 10/02/2007, http://www.open-audit.org/ (consulted all over the project)

Williams, Matt. ManagePC (2.5.0.13) [Application]. 05/29/2008, http://managepc.3ee.co.uk/ (consulted all over the project)

Breton, Thomas. H-Inventory (1.0.0) [Application]. 04/06/2007, http://www.h-inventory.com/ (consulted all over the project)