INSTITUTE FOR SECURITY AND OPEN METHODOLOGIES

SPSMM

Secure Programming Standards Methodology Manual

Created by Victor A. Rodriguez

CURRENT VERSION:	v0.5.1
NOTES:	English
CHANGES:	-
DATE OF CURRENT VERSION:	May 2002
DATE OF ORIGINAL VERSION:	-

THE SECURE PROGRAMMING STANDARDS METHODOLOGY MANUAL

When an interface is exposed (programming, user, etc.) it can be used or abused. It is abuse that should be avoided (depending on what side of the coin you are). Since we plan to standardize this methodology from the programming side and as so you may be a programmer, system architect, team leader, etc. you want to keep your interface clean, usable, and with no component capable of being abused.

As Internet applications get more and more complex and the code generation becomes more and more automatically generated through the use of wizards, programming environments, programming frameworks, etc. we are at a distinct disadvantage when you play the odds. Techniques for secure programming is something that has been tackled in many forms and for many languages. We are attempting to meet this with a single methodology regardless of programming language, application environment and development tools.

The idea of making this an open standard again means that it can benefit anyone but also can give the bad guys an insider's view to the process. However, if the methodology has been followed, the weaknesses should have already been addressed. The other side of the coin is that if more programmers read and understand this manual, more action can be taken to develop applications safely and to further develop this methodology.

Acknowledgments

I wish to give my thanks to David A. Wheeler for writing the "Secure Programming for Linux and Unix HOWTO" [1] that have inspired me. Also many thanks to the people that wrote the "SECPROG@SecurityFocus.com working document" [9], a kind of "My First Dictionary" in Secure Programming.

Those who have been contributed to this manual in valuable ways have been listed at the top of this document. Each person receives recognition for the type of contribution not as to what has contributed. The use of contribution obscurity in this document is for prevention of biases. Only in translations of this document are persons actually noted since the translator becomes the contact of the translated document that includes not only keeping the document up to date but also FAQs, submissions and comments in that language to be translated into English for the original when appropriate.

Terms

- **DoS (Denial of Service):**attack whose purpose is to disrupt the system accessibility. It doesn't affect the data contained in the system.
- **out-of-the-box:** standard installation, using the factory defaults.
- **super-user:** a special user (also known as root, administrator, etc.) that has no restrictions inside the system.

Intended audience

This manual has been written for programming professionals that want to establish guidelines and/or procedures for secure programming in any IT project, regardless of programming language, application environment and development tools.

This manual does not examine the proper way to use a particular development tool or how to program in a given language.

Developers will find this manual useful in making better programs, more resistant to hostile environments such as those who need to perform well in Internet.

Scope

The ultimate goal is to set a standard in secure programming methodology which when used in either manual or automated security testing results in meeting operational security requirements for interface usage and avoid its abuse. The indirect result is creating a discipline which can act as a central point in all security tests regardless of programming language, application environment and development tools.

Exceptions

The exceptions not covered, yet, in this paper are :

- Programs whose main purpose is security management
- Programs that must run as super-user

Preparing to work in a hostile environment

This preparation is not an easy task, and have been done previously in different ways, each one covering its point of view, from different starting points, etc. We will state our one in this work.

Taken a look at Secure Programming for Linux and Unix HOWTO [1] (section 2.2, Security principles) secure programming can be interpreted as the accomplishment of these goals: confidentiality, integrity and availability. That is right and can be changed or molded according to any particular environment, but is a very good and generic answer. Here the problem starts in how to reach these ones. What we try to emphasize is not the fixing per se, but an open source methodology that provides the basis for method or procedure development.

A system task is to implement functional requirements which, properly agreed, can be tested. In a more detailed approach. Coding is done to achieve this goal but in an atomic scale (at the subroutine or programming procedure level). What are not part of functional requirements is the system availability, which establishes system usability conditions such as which days it should be used (labor days, seven days a week, etc.), daily time frame usage (9:00 AM to 5:00 PM, etc.), maximum time with no system access (2 hours per moth, etc.) and a lot of requirements that define the so called Service Level Agreement (SLA).

Is in this point where, when running in hostile environments like Internet, to provide the required SLA levels (integrity), or the impossibility of information theft (security) or provide information to the proper owner (confidentiality) is for particular interest.

Once we know about these and its negative impact, there's a big impact in how to adopt a systematic approach to solve them. To begin with, let us look at the factors that give rise to these problems :

- **what not to do:** it is very difficult to build for "what must not be done"; protecting something that you don't know exactly what is became a heavy duty work (a very good article about this can be read at November 1999 Crypto-gram newsletter [2]). It is similar to the microbial defense, you can't protect from them unless you know that they exist and how can be defeated.
- **complexity:** this gives raise to one of the oldest and hardest problems of humanity. It is called communication, and as complexity is added to a system its better handled if the whole task is divided into sub-tasks, that belongs to different entities or teams, that must communicate between them to reach the goal; the more teams you have the more communication links must be established and, as you know, this is an exponential function which converts the system build in an exponentially growing complex (take a look at software essential difficulties in "No silver bullet" [3]). This allows the existence of hidden difficulties covered by the "there's too much places to look at" syndrome.

Methodology

As this work is part of the "The Open-Source Security Testing Methodology Manual" (OSSTMM for short), the same methodology will be used, stating the Parameters and Tasks oriented to Secure Programming, but with a slightly different approach. In OSSTMM we have discovered that there is a flaw, and now we want to know where exactly it is located, how to fix it and avoid it in future coding.

This section will work in the first and second goals (discovering and fixing) while avoidance is a topic for the "Secure Programming reinforcing" section (to be developed).

Input Data

Must be noticed that interfaces rely on the implementation of some technology, and that the security exploits in the implementation must be fixed to obtain a complete problem solution; it's not enough to fix the interface implementation on the program under testing.

One point to have in mind is that input data is not only the one acquired through the user/other program interface but also the one obtained from the system (date, environment variables, etc.) and other software that conforms the basis for the system. In short every piece of data that feeds the program/system being analyzed.

Input data limits validation

Once the input data is matched against good patterns, it can be not quite good. For example if an account identifier is numeric, not all combinations are valid and not all account sizes are available.

Please note that these constraints aren't related to the real world meaning of the variable, but with the security implicacies of the variables length, that is to say when the technology implementation (languages, operating systems, etc.) are stressed and take note on its behavior (one common test is to call functions with parameters that are in the Kilobytes order and that are supposed to be no more than a few bytes).

Expected results:

- System behavior to unexpected and validated inputs

Tasks to perform:

System monitoring:

- CPU usage (percentage, time, peaks, etc.)
- Memory usage (real, virtual, cache, etc.)
- I/O usage (disk space, burst access, etc.)
- Network usage (packet dropping, generation, etc.)

Code working:

- Code segment/s identification that handle each input and its validation
- Code fixing

In this topic the analysis and code working is a kind of art, because there is more important the system behavior instead of the data being returned. This requires a broad knowledge of the technology being used, mainly the language implementation nuts and bolts such a memory management, process/thread generation, locking, IPC, etc.

An example of the bugs discovered with this test can be seen in "Appendix A" using the "Format String bug". It effect can be minimized if the proper checking on data limits and format are done.

Process

Once a protection has been established for the outside world, we must protect from our own system coding and inner practices.

An analysis of how to protect the process is basically the protection of all the resources used by your system: memory, disk, network, etc. to avoid its misuse by itself or by another person.

Resource consumption

Every program makes resource consumption for its own purposes (CPU time for calculations, memory for internal and intermediate storage, files for miscellaneous processing, etc.). Under certain conditions its possible to make the program began a consumption cycle that can end in system

thrashing or program dumping. These are called denial of service, commonly known by its acronym DoS.

Commonly this is done only with the accepted data patterns, but is a good practice to work with the rejected ones also, and variations of the latter ones (also known as *dirty test*).

Expected results:

- Resource consumption and usage for each input data pattern and state (referred to the implemented state machine).

Tasks to perform:

System monitoring: For each resource class (CPU, memory, etc.)

- Proper resource acquisition
- Proper resource usage
- Proper resource liberation
- Deadlock monitoring/analysis

Code working:

- Code segment/s identification that cause the DoS
- Code fixing

To avoid such DoS conditions is a good practice to limit the amount of resources used, and reject petitions when an specified level of resource consumption has been reached.

An example of the bugs discovered with this test can be seen in "Appendix A" under the title "resource retention".

Input data patterns validation

Test on various data patterns for success or failure on acceptance. This can be split among different screens, scripts, URLs, interface type, etc. to handle it according to the available resources (time, budget) and people. Is recommended to test all available interfaces (GUI, API, etc.) and not only those exposed to the user.

In this point the tester must focus in how to elaborate data sets that not only doesn't agree with the data pattern established, but that also try to exploit some hidden and obscure architectural element in which the program is based (operating system, language, database, etc.).

Expected results:

- a list of accepted and rejected data patterns

Tasks to perform:

Data rejection/acceptance:

- Check for accepted data that must be rejected
- Check for rejected data that must be accepted (false positive)
- Check for accepted data that must be accepted (false negative)
- Check for rejected data that must be rejected

Code working:

- Code segment/s identification that handle each input and its validation
- Code fixing

In code fixing, for this one, the key part is not just to add code to reject the discovered bad data, but also to extrapolate it to more generic patterns. At this point the method used for filtering and rejecting the bad data must be discussed. The chosen method depends on the environment limits as explained in "Input data" in Appendix C, but there is something about filtering that we discuss here: the accepting/rejecting method.

There are two basic methods :

- **Accept pattern**: the input data is checked against accepted patterns, narrowing it as new vulnerabilities are discovered.
- **Reject pattern**: the input data is checked against rejection patterns, adding new patterns as they are discovered.

If you pay attention each method has its drawbacks. The second one is easier for adding new discovered patterns, but requires additional tests for each rejection pattern. The first one is cleaner and more generic because you work on reducing the acceptor pattern, but you must be careful not disturb the actual pattern, which requires a bit more thinking and testing.

A typical example of the bugs found with this test can be found at "Appendix A" under the title "Remote compromise"

Resource usage prediction

Sometimes being so predictive in usage (that means to have a usage pattern) it is not good, because helps to the prediction of the next move to be done by the system. This can be used for an attacker to impersonate the system (giving the same network answer and fake the system, knowing the next filename to use and redirect it to scratch a sensitive file, predicting session identifiers, etc.)

Expected results:

- Resource usage trace

Tasks to perform:

System monitoring: For each resource class (CPU, memory, etc.)
Resource analysis: Search for patterns in the resource usage.

Code working:

- Code segment/s identification that cause the DoS
- Code fixing

Attack prevention

When we are processing the data obtained we tend to think that the available data is safe, but ... has the actual data passed the security controls if the program is not being executed under the specified environment ?? What if an attacker has found the way to execute the processing part bypassing the security checking ??

Some programs make code security checking to control if they are working in the proper environment and/or if the code has not been modified.

Expected results:

- Detailed system response under abnormal operating conditions

Tasks to perform:

System execution:

- Using other entry points than the normalized ones
- With code injection/modification
- Modifying the operating conditions

Code working:

- Code segment/s identification that cause the DoS
- Code fixing

The most common fix to think about is to include a similar code that checks and/or cleans the data entered, but this is a task reserved for the input checking module and these should not be duplicated. The task to be carried out here is a different one: detection of abnormal running conditions and **not** data health checking.

Output

You have filtered all the input, have been cautious about resource usage, but ... that is not enough. What about of telling to your competitor the keys for success ?? Sometimes is not such drastical as this, but giving it some clues only gives a time difference between giving all the information at once or some clues.

Something must be noted, and is that some output errors can be seen as process errors because they produce some subtle information, product as an unexpected (spurious) output. What we will try to depict here are information passing in the normal output channels, such as HTML hidden tags, default errors that can bring internal structure information, and problems in the style.

Data clearance levels

This concept is pretty clear in Military use. Briefly, you have some categories (top secret, secret, public, etc.) that tags each object (piece of information, people, etc.) and a person can access any resource tagged with the same or minor level that its own (e.g. a person with clearance level "secret" can access "secret" and "public" resources but not "top secret" ones). Apart from this, a resource has the same security clearance as the maximum level that any of the objects that it contains (e.g. a document containing "secret" and "public" data is classified as "secret"; if a top secret piece is added it becomes a "top secret" document. The same applies to communication channels, you can't transmit a "top secret" document through a "secret" channel (the channel has less protection mechanisms than the transmitted document can accept).

This same concept should be applied to the output of our systems.

Expected results:

- Clearance levels for each output object (data, persons, channels, etc.)

Tasks to perform:

- Code segment/s identification that cause output
- Output intended subject classification
- Output channels classification
- Output data classification

Code working:

- Data tagging with clearance levels
- Code fixing

FIX ME: Think about something similar to the UML use cases and diagrams !!!!!!!!!!!!

Sometimes you want to give a lot of information, and this is not always a good practice as you have seen, and the information included contains the errors often given by the OS that contains file paths, etc. that can give some clue of information to people not intended to have it.

A clear example of this are web servers. A default configured MS IIS (Internet Information Server) shows an error message stating that the page doesn't exist AND showing the supposed file path in the file system (if the file doesn't exist) or the output of the script in case of error (that can contain internal data structures or debugging info not intended for the general public audience). Its counterpart (Apache) shows a message indicating that an error has occurred, the administrator should be notified AND the juicy info is stored in the error_log.

Appendix A - Most common faults

In this section the faults will be depicted, sometimes in detail and sometimes a slight idea will be shown, mostly in the case where they have been broad studies or papers discussing it.

This appendix is not intended as an exhaustive list of faults, but the most common and representative ones, just to put some common background in the kind of information, tools, and related results to be found.

Stack smashing

Probably you know what a smash is and probably what a stack means in computer jargon. The term "buffer overflow" is perhaps best known but both refer to the same problem.

I will describe it only slightly, because there is some very good articles that completely describes it ([5] and [6]).

When you have two variables defined in a high level language and both are adjacent in the source code, then there is a high probability that both use adjacent memory areas. So if you write one of them and trespass its limits , then you are writing in the other one memory area. Let us say that the first one has 30 bytes, when you access the 31st byte indeed you are accessing the 1st byte of the second variable.

These seems of not too much use, but let us do a bit of digging. When you are using a high level language mostly it uses subroutines (also called functions, procedures, etc.) that uses its own local variables. It is a common practice that the compilers use a memory portion called stack where the address of the next instruction to be executed, called return address, when the subroutine ends and also the local variables are stored there. Now we can exploit this locality because if we access these local variables we can go beyond its limits and access the return address. You cannot make magic knowing the return address but if you can inject a new piece of code and point the return address to it, the you can exploit a program and make it work for you.

This can be avoided using some tricks like inserting some health checks for the return address and act upon it (you can end the program, try to restore the damaged return address, etc.). Also this checking can be defeated, but there is some minor probabilities of occurrence for these problems. A good article on this topic can be read at [7].

Format String bug

There is functions and functions, some more flexible than others ... but flexibility has its costs and limits, one of such is security. There is a common set of functions in C (called the printf family) that one of its parameters is a string containing the format, and order, in which the subsequent parameters will be fed to the function. For example the function call :

```
printf( "Hello, %s. You have visited us %u times", name, counter );
```

says that the parameters passed are three (the string, and the variables name and counter), and the information of quantity and type of them can be read from the first one (also called format string) : one string (%s) and one unsigned integer (%u).

Now this "format string" is hard coded in the application, what about given the control on the format string to the outside world ?? Take as an example the next code snippet :

```
void flex_printf( char *format_string, name;
                      unsigned int counter  ) {
        printf( format_string, name, counter );
    }
```

where format_string is a string whose input was provided by an unverified / untrusted user. This could allow it to view some internal info, useful for data gathering and subsequent use in exploits (such as a stack smashing) through string manipulation such as changing the %s or %u for a more convenient way to gather memory content (e.g. passing an hex in counter and declared in the format_string that its a pointer to a string or char *). For a complete description, and "in the wild" examples, take a look at reference [10]

Remote compromise

This is the explanation of an exploit that combines two common mistakes : no filtering (or a defective one) and the running of a program with excessive privilege level.

Follows the "Brief Summary" extracted from the advisory at [11] :

"MS Jet database engine (which runs Access databases) allows an individual to embed VBA in string expressions, which may allow the individual to run command line NT commands. This, combined with the flaw of IIS running ODBC commands as system_local allow a remote attacker to have full control of the system. Other web servers may be affected. Many MS Jet engines are affected, but may not lead to elevated privileges."

Basically you can embed VBA (Visual basic for Applications) commands inside an SQL sentence and let the ODBC to execute them (no proper filtering), such as system commands through a shell escape. This could be minimized if it runs with a username that have the minimal privileges to execute its task in an efficient way. This was not the case.

No further explanation will be done. The cited advisory is a masterpiece that deserves to be read "in detail".

Resource retention

When you initiate a TCP connection the stated machine establishes that three single steps must be followed :

- Request the connection to the remote end
- The remote end answers with an acknowledge (else the connection is rejected)
- You must answer with an acknowledge

Regarding it simple mechanism it lacks of a control mechanism to avoid unnecessary resource contention. Let's suppose that when your computer request a connection to the remote computer it crashes, then the acknowledge is sent by the remote end but yours one doesn't receives it so the remote computer waits until some time-out is reached and considers the connection closed. Because TCP/IP was designed to work in non-reliable high latency environmental conditions it will wait too much time until that it considers the connection closed (or failed). This wastes resources in the remote computer (CPU and memory mostly) that can be exhausted if a lot of connections remain in this state.

A very complete report can be found at CERT, in reference [12]

Appendix B - Tools

What comes to your mind when I say tools. If the answer is programs to use to enforce secure programming you are almost right, if and only if, it doesn't means only ready to use. You must be prepared for some in-house work or some program adaptation to your own needs.

Log everything

After all of these problems to be sorted one thing must be clear, and is that despite all our efforts the programs are not perfect, and imperfections can, and will, appear. To allow these holes discovering the proper tools must be used such as debuggers and dump analyzers, but sometimes this tools are not useful, and in particular when you don't know where to start your search.

Do you remember the "Hansel and Gretel" tale ? They dropped bread through the path and it could be undone later. We will use the same trick, leave a trace of each important code snippet being visited with the most important data obtained, and under an attack (or a simple common failure) the program can be backtracked.

FIX ME: What should and shouldn't (DoS attacks, special chars, passwords, ...)

Ready to use

This is simply a list of known tools to enforce secure programming, whose preferred ones are the Open Source children.

Name:	Flawfinder
Briefly:	a program that examines source code looking for security weaknesses ('flaws') in C/C++ code.
License:	General Public License (GPL).
Link:	http://www.dwheeler.com/flawfinder/
Name:	ITS4
Briefly:	It is a simple tool that statistically scans C and C++ source code for potentially security vulnerabilities.
License:	ITS4 NON-COMMERCIAL LICENSE for source code
Link:	http://www.cigital.com/its4
Name:	LCLint
Briefly:	It is a tool for statically checking C programs
License:	General Public License (GPL).
Link:	http://lclint.cs.virginia.edu/
Name:	StackGuard
Briefly:	StackGuard is a compiler that emits programs hardened against "stack smashing" attacks. Stack smashing attacks are the most common form of penetration attack.
License:	StackGuard is Free Software: it is an enhancement to GCC, and is distributed under the GPL in both source and binary forms
Link:	http://www.immunix.org/stackguard.html
Name:	FormatGuard
Briefly:	FormatGuard protects C and C++ programs against the format bug, most commonly used in printf family functions that use variable arguments passing.
License:	FormatGuard is Free Software: it is an enhancement to GCC, and is distributed under the GPL in both source and binary forms
Link:	http://www.immunix.org/formatguard.html
Name:	RSX
Briefly:	RSX is a Runtime addressSpace eXtender providing on-the fly code remapping of existing Linux binaries, to implement a non-executable stack and short/long heap areas. It targets common buffer-overflow problems preventing code execution in mapped data-only areas.
License:	Proprietary License with Source
Link:	http://freshmeat.net/projects/rsx
Name:	PageExec
Briefly:	Implementation of non-executable pages for IA-32 processors
License:	???
Link:	http://pageexec.virtualave.net/

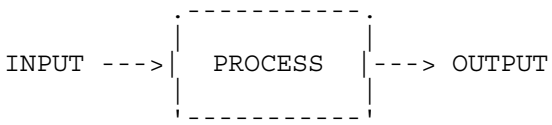
Name:	Libsafe
Briefly:	Buffer overflow and format string avoiding tool, packaged as a library that intercepts all function calls made to library functions that are known to be vulnerable
License:	Libsafe version 2.0 source code licensed under the GNU Lesser General Public License.
Link:	http://www.avayalabs.com/project/libsafe/index.html

Appendix C - Faults

Once that we have navigated to the old way, sailing near the coast, we need to trace a more detailed map. It is my personal preference going from the generic aspects to the particular ones, so let us start in this fashion.

Input data

Lets start at the basics about programs or information manipulation. The typical schema for information manipulation is to obtain the data, process (transforms) it and offers the results: input-process-output



Not big deal, but to avoid malfunctions and side effects (some with system dependencies) it must be filtered to obtain the proper data.

Suppose we are dealing with a banking system where the must be entered an account number and that the account balance should be shown, then the input (account number) must have a proper format (lets say three numbers, five letters and eight numbers), which must be validated (checked against a template) and enforced (rejected if not in the proper format or the account does not exist).

In this example the choices for formatting and reinforcing are fairly simple, just you cannot give an account balance on a non existent account, but there's more subtle problems (which will be shown further FIX ME: add them when they appear such as shell commands) that are not functional ones as the described items. This rule can be taken: all that is not used must be avoided.

There is a lot of choices to do to reinforce this, and each one depends on the particular system goals:

- **Abnormal end:** this is the most radical, because in case an invalid data is detected the system interpret that is under attack (someone searching for improper data formats) and terminates.
- **Rejection:** the halfway, when an invalid data is detected it is asked again for it proper format and a new input is waited. This has a hot point for attackers, and is that it can be asked any number of times until a valid data is detected.

- **Usable:** the bad parts of the data are stripped off and only the data that complains to the proper format is used. Typically used for variable length data such as e-mail addresses, URLs and postal addresses.

One language that has embedded capabilities for secure programming is Perl. It can be run in the called "tainted mode" where, essentially, each piece of data that is gathered from outside the actual program is marked as dirty or "tainted" and cannot be used until it is cleaned. Usually the laundry machine for this tainted data is made of filters that implement the reinforcement, and if the tainted data is used without laundering the Perl program stops its execution (look at [4] for a deeper understanding)

FIX ME: if other languages have similar characteristics mention them here.

Processing (Pandora's box)

Sometimes you take a lot of care about the data taken outside the program but, I hate to say it, it is not enough. Because we are not perfect people and make mistakes we must protect the program, and the system, against us. Remember that in any program there is a lot of factors to be accounted for (this is so true that in software reliability analysis the program behavior is studied through the use of statistics) and not all of them can be taken care for at once, so the probability to introduce bugs is really high (this probability decreases as the program is tested over the time).

This is not uncommon in all well established software factories the use of testing, running environments, etc. that simulate the conditions in the field (running in its final destination) and runs it in a protected environment, allowing it to fail, thrown errors, or whatever it wishes to do. The same is done when installed in its final destination, it is run with a lot of precautions until it runs as a productive process, and the monitoring is a critical process in the first weeks.

Most modern programs such as C++, Java, Perl and Phyton contain a programmer protection mechanism. Basically they run some code parts in a protected environment, and trap any error (or exception) that could be generated by the program.

For example, you can surround your code with a block that isolates it from its surrounding environment in case of failure, and returns the control to a code portion that handles the error:

```
try() {
    Code_under_suspicion();
} catch (Exception e) {
    System.println "Code_under_suspicion(),
                Exception generated : " + e;
}
```

Now suppose that we are developing critical programs such as in medicine, aeronautics, rocketry, etc. where they must run yet if the program fails. Imagine that during a flight the program that tracks the route to the final airport, in the console can be read this message :

"Exception thrown at 0x3F745DE9. The system is going down"

So these programs are build with this in mind, and let the program continue its work under special conditions (let us say that the isolate the code that caused the problem, perform some checking on hardware and software, etc.).

To read a well founded revision on exceptions applied to Java, please take a look at [8], in the Links section.

Sometimes it is not a good choice to do it this way, but put fences around the whole program sometimes is easier: they require no extra programming time, no recompiling and installing needed for fence activation or deactivation. When this occurs a series of tools can be used that are pretty standard such as dump analyzers (post-mortem analysis) or debuggers that wraps the program and when an error occurs it is trapped and managed by the debugger. These can simply print a stack trace, variables contents and exit or offer the programmer/operator a kind of shell to allow the program inspection, resuming, restart, halting, dumping, etc.

Output data

And speaking about side effect, did you have in mind that your output is another process input ? And worst yet ... what about your garbage ?

The first stuff goes first. The same problem that you have with your input data is transferred to the process that will handle your output data and process it. The process is not so crude because your program/system is acting as a filter but, anyway, your program can do bad things (not on purpose, of course) that can damage or worsen a good work.

Let's take an example where you process your data and, as part of the output process, a file is generated. Nothing seems to be wrong, all is in the specs and we are following the games rules. Are you sure ? If we generate a fixed file name for the output data, let as say output.data, simply we :

```
filehandle = open( "output.data", w );
 write( filehandle, data );
 close( filehandle );
```

Of course you surely added some checking for the file existence, asked the operator/user for the file erase in case it existed, make a backup, and all sort of things, but ... what if the underlying layers are compromised and you are not writing in the place you are thinking of ? Have you a "clean path" to the file ?

FIX ME: Work more on the previous topic

Now, as you surely know, one important information source for spying is the use of garbage bin collection and analysis (that is why the paper shredder exists after all). The most common example of this is to answer the question "what do you do with your warning and error messages ?" Surely you have noticed that in some web servers when an error occurs, such as a missing page or a CGI failure, the error data is sent to the client browser, and messages like this can appear :

The requested CGI program in '/home/httpd/cgi-bin/script' failed to

execute. The next lines contain the error:

```
Couldn't connect to database 'Customers' in server 'internal_db'.
Send a message to hostmaster@this_domain.com
```

What a great source of information we have in these lines !!! This SHOULD be redirected to the error log or any other place that can be read by the administrator/operator. After all this is a nonsense for most of the users and a rich information source for crackers, don't you ?? And the sad part of this story is that most web servers are configured insecurely out-of-the-box.

Appendix D - Links

[1] **Title:** Secure Programming for Linux and Unix HOWTO
Author: David A. Wheeler
Location: http://www.dwheeler.com/secure-programs

[2] **Title:** Why Computers are Insecure
Author: Bruce Shneier
Location: http://www.counterpane.com/crypto-gram-9911.html#WhyComputersareInsecure

[3] **Title:** No silver bullet
Author: Frederick P. Brooks, Jr.
Location: http://www.undergrad.math.uwaterloo.ca/~cs212/resource/Articles/SilverBullet.html

[4] **Title:** Perl security
Location: http://www.perl.com/pub/doc/manual/html/pod/perlsec.html

[5] **Title:** Smashing The Stack For Fun And Profit
Author: Aleph One
Location: http://www.phrack.com/search.phtml?view&article=p49-14

[6] **Title:** Avoiding security holes when developing an application
Author: Frederic Raynal, Christophe Blaess, Christophe Grenier
Location: http://www.linuxfocus.org/English/March2001/article183.meta.shtml

[7] **Title:** Bypassing StackGuard and StackShield
Author: Bulba and Kil3r
Location: http://www.phrack.com/search.phtml?view&article=p56-5

[8] **Title:** What's an Exception and Why Do I Care?
Author: Sun Microsystems
Location: http://java.sun.com/docs/books/tutorial/essential/exceptions/definition.html

[9] **Title:** SECPROG@SecurityFocus.com working document
Author: Secure Programming list contributors
Location: http://www.securityfocus.com/forums/secprog/secure-programming.html

[10] **Title:** Analysis of format string bugs
Author: Andreas Thuemmel
Location: http://www.securityfocus.com:80/data/library/format-bug-analysis.pdf

[11] **Title:** NT ODBC Remote Compromise
Author: Matthew Astley & Rain Forest Puppy
Location: http://www.securityfocus.com/archive/1/13882

[12] **Title:** CERT Advisory CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks
Author: CERT
Location: http://www.cert.org/advisories/CA-1996-21.html

Open Methodology License (OML)

PREAMBLE

A methodology is a tool that details WHO, WHAT, WHICH, and WHEN. A methodology is intellectual capital that is often protected strongly by commercial institutions. Open methodologies are community activities which bring all ideas into one documented piece of intellectual property
which is freely available to everyone.

With respect the GNU General Public License (GPL), this license is similar with the exception for the right for software developers to include the open methodologies which are under this license in commercial software. This makes this license incompatible with the GPL.

The main concern this license covers for open methodology developers is that they will receive proper credit for contribution and development as well as reserving the right to allow only free publication and distribution where the open methodology is not used in any commercially printed material of which any monies are derived from whether in publication or distribution.
Special considerations to the Free Software Foundation and the GNU General Public License for legal concepts and wording.

TERMS AND CONDITIONS

1. The license applies to any methodology or other intellectual tool (i.e. matrix, checklist, etc.) which contains a notice placed by the copyright holder saying it is protected under the terms of this Open Methodology License.

2. The Methodology refers to any such methodology or intellectual tool or any such work based on the Methodology. A "work based on the Methodology" means either the Methodology or any derivative work by copyright law which applies to a work containing the Methodology or a portion of it, either verbatim or with modifications and/or translated
into another language.

3. All persons may copy and distribute verbatim copies of the Methodology as are received, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and creator or creators of the Methodology; keep intact all the notices that refer to this License and to the absence of any warranty; give any other recipients of the Methodology a copy of this License along with the Methodology, and the location as to where they can receive an original copy of the Methodology from the copyright holder.

4. No persons may sell this Methodology, charge for the distribution of this Methodology, or any medium of which this Methodology is apart of without explicit consent from the copyright holder.

5. All persons may include this Methodology in part or in whole in commercial service offerings, private or internal (non-commercial) use, or for educational purposes without explicit consent from the copyright holder providing the service offerings or personal or internal use comply to points 3 and 4 of this License.

6. No persons may modify or change this Methodology for republication without explicit consent from the copyright holder.

7. All persons may utilize the Methodology or any portion of it to create or enhance commercial or free software, and copy and distribute such software under any terms, provided that they also meet all of these
conditions:

a) Points 3, 4, 5, and 6 of this License are strictly adhered to.

b) Any reduction to or incomplete usage of the Methodology in the software must strictly and explicitly state what parts of the Methodology were utilized in the software and which parts were not.

c) When the software is run, all software using the Methodology must either cause the software, when started running, to print or display an announcement of use of the Methodology including an appropriate copyright notice and a notice of warranty how to view a copy of this License or make clear provisions in another form such as in documentation or delivered open source code.

8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on any person (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If said person cannot satisfy simultaneously his obligations under this License and any other pertinent obligations, then as a consequence said person may not use, copy, modify, or distribute the Methodology at all. If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

9. If the distribution and/or use of the Methodology is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10. The Institute for Security and Open Methodologies may publish revised and/or new versions of the Open Methodology License. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.
NO WARRANTY

11. BECAUSE THE METHODOLOGY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE METHODOLOGY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE METHODOLOGY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE IN USE OF THE METHODOLOGY IS WITH YOU. SHOULD THE METHODOLOGY PROVE INCOMPLETE OR INCOMPATIBLE YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY USE AND/OR REDISTRIBUTE THE METHODOLOGY UNMODIFIED AS PERMITTED HEREIN, BE LIABLE TO ANY PERSONS FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE METHODOLOGY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY ANY PESONS OR THIRD PARTIES OR A FAILURE OF THE METHODOLOGY TO OPERATE WITH ANY OTHER METHODOLOGIES), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.