Whitepaper

Secure Programming in C/C++

(c) 2005, EUROSEC GmbH Chiffriertechnik & Sicherheit

Document History

Version-Number	Date	Editor	Reviewer	Status	Remarks
1.0	July, 2005	EUROSEC			

created by

EUROSEC GmbH Chiffriertechnik & Sicherheit

Sodener Straße 82 A, D-61476 Kronberg Germany

www.eurosec.com   mail: kontakt @ eurosec. com

Phone: +49 (0) 6173 60850

Table of Content

1 Introduction 5

2 Specific Threats for C/C++ Programs 6

3 Secure Programming Guidelines....8

3.1 General Guidelines....8

3.1.1 Coding style guidelines 8

3.1.2 Input data validation 8

3.1.3 Performance optimization 10

3.1.4 Assertions and debug traces 10

3.2 Enforcing security checks at compile time....10

3.3 C++ specific Guidelines 11

3.3.1 Access of variables and methods 11

3.3.2 Mutable objects....11

3.3.3 Inheritance....11

3.4 C/C++ specific Guidelines 12

3.4.1 Buffer overflows....12

3.4.2 Programs running with root privileges (UNIX specific) 14

3.4.3 TOCTOU attacks....14

3.5 Security critical modules and tasks 15

3.5.1 Separation of security critical modules....15

3.5.2 Cryptographic and security relevant libraries....15

3.5.3 Sensitive data....15

3.5.4 Pseudo random data 16

4 Resources....17

4.1 Bibliography 17

4.1.1 Guidelines on secure programming 17

4.1.2 Buffer overflows....17

4.1.3 Format string attacks 17

4.1.4 Setuid programming....18

4.1.5 Race conditions....18

1 Introduction

The C/C++ programming languages have been the most important higher programming languages for years, and they are still indispensable, when it comes to program applications which have to perform extremely efficient or that interoperate closely with the operating system (which itself is usually largely written in C). All this is possible with C/C++ by direct access to memory manipulating functions (using pointers to memory locations). On the other hand this direct access to memory makes C/C++ inherent unsafe languages, and makes buffer overflow and format string attacks possible. If the planed purpose of an application would allow the program to be coded in an inherent safer language like Java, this alternative should be evaluated. (But note that even a programming language like Java, that comes with a lot of build−in security features, does not guarantee by itself the fulfillment of security relevant requirements).

While there are particular security relevant threats when programming in C/C++ (discussed in the next section), security issues should be considered throughout the whole software development process, independent of the particular programming language chosen.

Therefore the section containing the programming guidelines is divided into two parts. The first part treats general principles that should be obeyed independent of the particular programming language chosen. The second part concentrates on specific guidelines that should be considered when programming in C/C++. Also some specific Unix system functions are discussed in this context.

2 Specific Threats for C/C++ Programs

Many critical problems of programs written in C/C++ have their reason in incorrect or careless access of memory. Incorrect access of memory is easily possible, as running C/C++ programs usually do not perform any checks whether accessed memory has been allocated or initialized. In particular no memory boundary checks are performed. Also allocation and freeing of memory is an error prone task, where mistakes may easily introduce memory leaks or even crash a program.

Errors in conjunction with memory access and management are often hard to detect and their complete elimination for some software product (before shipping) does usually represent an enormous effort including thorough testing phases. While such testing and debugging phases usually allow to achieve some level of assurance that a program does function as specified and preserves integrity and privacy of data under normal operation (in particular as long as the input values are in a well specified range), testing the behavior under malicious input conditions is often neglected or just not completely feasible.

While it may be just a nuisance or else it may undermine the reliability of a program, when it is possible to crash a process by malicious input (DoS attacks), the most severe security problems arise, when an attacker can trigger a program that has high privileges (run by root, or being a setuid program owned by root) to execute some function of his choice. Usually an attacker tries to overflow some buffer in such a way that the return address of some function call is overwritten with the address of another function that returns a shell. If the program runs with root privileges, this is a root shell. The code for the function that opens a shell is often also part of the input string. To overwrite some memory (in particular return addresses of function calls) an buffer overflow attack or a format string attack may succeed.

A buffer overflow is possible when the length of some input data (usually some input string) is not correctly checked, before it is copied to some buffer on the stack. If the target buffer is smaller than the input data, the bytes on the stack above the target buffer get overwritten. As the stack grows downwards, the return address of the calling function lies somewhere above the target buffer, and hence it may be corrupted by the input data. The change of the relevant stack area caused by a buffer overflow attack is visualized in Fig.1 and Fig.2. The code to open a new shell is sketched in Fig.3 (if its binary form has to be integrated into some string input variable, it has to be adopted to not contain any zero bytes).

Format string attacks are possible if some formatting function like printf() is incorrectly used, i.e. if some input string becomes part of the format string. Using parameters like "%x" and "%n" in a format string allows an attacker to read from and to write to the stack

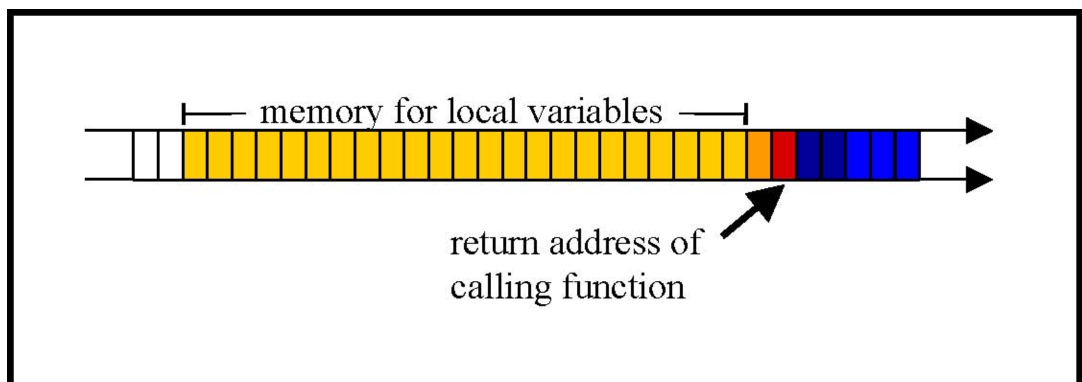

Fig.1 (Stack after some function has been called)

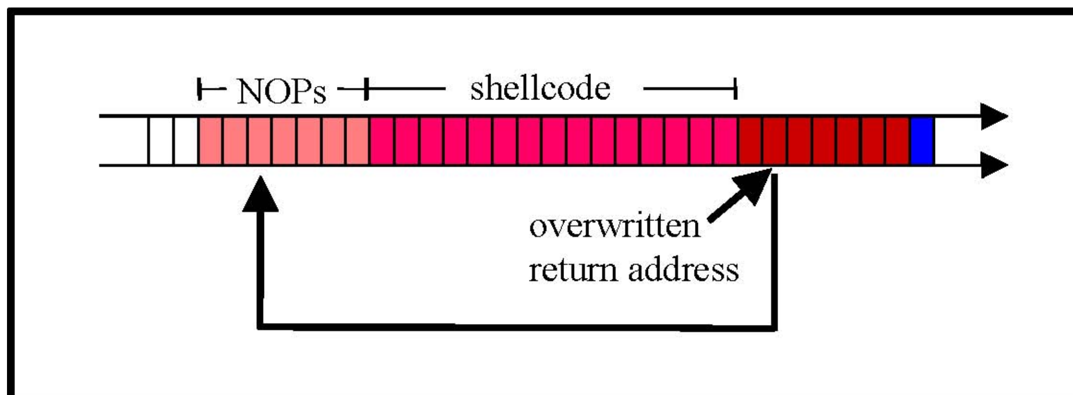

Fig.2 (Stack after buffer overflow)

```
int main() {
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
   return 0;
}
```

```
int main() {char *name[2];name[0] = "/bin/sh";name[1] =
  NULL;execve(name[0], name, NULL);return 0;
}
```

Fig.3 (Shellcode)

3 Secure Programming Guidelines

As mentioned in the introduction, secure programming is not possible without obeying some general good programming practices. Therefore this chapter is divided into two parts. The first part contains general rules that should be followed to write secure programs, while the second part concentrates on C/C++ specific topics. Although the guidelines given in the first part are of a somewhat general nature and similar rules can be formulated for other programming languages as well, this paper concentrates on the specific implications for programming applications in C/C++.

3.1 General Guidelines

3.1.1 Coding style guidelines

- Establish and enforce company wide coding style guidelines.

Having a well established software development process provides the necessary framework to produce robust, stable and secure software. Part of such a development process should be the implementation (and enforcement) of a set of coding style guidelines. Only if all software developers adhere to the same best programming practices, the final code will be in a consistent and simple state. This is the basis for maintainable and robust code, that can be evaluated (reviewed) and trusted to function as specified. Of course, experienced software engineers should be involved, when best practices are established.

3.1.2 Input data validation

- Always validate input to public methods.

It is estimated that more than 50% of all security vulnerabilities are caused bybuffer overflow attacks. To avoid such weaknesses the enforcement of definiterules how to check arguments of public methods is most important.In particular string buffer arguments must always be checked for length's.Furthermore, the characters in an input string have to be checked against a list ofallowable characters (white list checking).

- Never use gets() to read from standard input. Use fgets() instead.

Lines of text can be read from standard input using the functions gets() or fgets().While fgets() accepts a parameter to limit the number of bytes to read, gets()provides no such possibility for limiting the input length.As gets() does not check the length of an input string, before copying the string tomemory, a buffer overflow attack can always be successfully mounted when gets()is used.

- Never use input data as input to a format string

Format strings are input arguments to functions like fprintf() that transform some further input data into a string whose format is determined by the format string. The usage of input data as part of format strings may be exploited by format string attacks. Hence, to guard a program against format string attacks, it has to be verified that no input data goes into some format string. For example, if stringptis a pointer to some input data string, fprintf has not to be used in the form printf(fpt,stringpt) but the following expression is safe:

printf(fpt,"%s",stringpt)

- Avoid the usage of environment variables.

Values of environment variables that are used by a program may also be considered as input data, although they are often neglected and not documented in public interface specifications. As environment variables may easily be manipulated, it is good programming practice to avoid the usage of environment variables. If values from environment variables have to be used, very strict validations should be performed. In particular the lengths should be checked and the characters should be tested against some white list of allowed characters.

- Do not call a shell to invoke another program from within a C/C++ program.

From within a C/C++ program it is possible to call a shell to invoke another program. This can be accomplished with function calls like system() or popen(). Using such function calls opens a program to attacks by modification of the environment (particularly by changing the PATH and IFS variables). To avoid dependencies on the environment, usage of execve() is recommended instead.

- To call another program from within a process use execve() or execle().

To call another program from within a process one of the exec functions may be used. The functions execl(), execv(), execlp() and execvp() do use the current environment when calling a new program, whereas execve() and execle() do have an input argument for the new environment to be used. As it is safe programming practice not to depend on the environment, the execve() or execle() function calls should be used. Moreover execlp() and execvp() evaluate the PATH environment variable to find the called executable. This makes these function calls vulnerable to attacks that change the PATH variable.

- Explicitly set parameters and clean signals that are inherited from parent processes and that might interfere with the actual process.

There are quite some properties that a process inherits from its parent process. Among those things are the umask, signals and resource limits. These data has to be considered as input to the child process. To get a well defined state at startup and even to prevent DoS attacks (by setting inappropriate resource limits) the child process has to validate and clear these data accordingly.

- Provide utility methods for input data validation and transformation purposes.

To support a stable and uniform mechanism for input parameter checking (simplifying implementation and code auditing as well), appropriate methods for such purposes should be provided in a dedicated package. For example, a task that occurs frequently in input validation is the transformation of some string representing a number into some elementary data type (int, float, ...) or to just check if a given string represents a number in a given range (e.g. an integer with 16 digits). Providing some globally reusable methods for such purposes, avoids having multiple different implementations scattered throughout the code, some of them possibly having flaws.

- Consider to automatically generate input data validation methods.

If the functionality for a formally specified API (for example given in some XML format) has to be implemented, it should be considered to generate the code for input validating methods directly from the specification (parsing the parameter types and specified ranges for parameter values from the API specification and generating the code). Using such an automated code generation mechanism will contribute to a very high level of assurance, that the input validation methods are correct, complete and uniformly implemented. Furthermore, this approach may help to keep the code synchronized with the specification. (The code can be updated to reflect some small changes in the specification simply by recompilation.)

3.1.3 Performance optimization

- Optimize performance only after profiling.

If code is not profiled for performance critical sections, performance issues may be overemphasized during the initial programming phase. This may encourage developers to write quite obscure code that may be difficult to maintain and to review and that finally may include hidden security flaws. To the contrary, coding style guidelines should be enforced, that emphasis rules to write simple code that is easy to understand and maintain and code should only be optimized with respect to performance after it has been profiled.

3.1.4 Assertions and debug traces

- Add debug traces to your code.

During software development and testing phases assertions and debug traces may provide valuable hints about the location and reason for bugs occurring in the running program. Also statistics and profiling information may be included in debug traces. Debug traces not only help a lot during the development phase, but for example also do provide means for quality engineers to follow the program execution path and allow a first analysis when it comes to narrow down a bug. In general thorough testing, which is an integral part in the production of secure software, will not be possible without sufficient trace information.

3.2 Enforcing security checks at compile time

- Enable all compiler warnings and pay attention to these warnings.

Compiler warnings may dispose problematic lines of code, that may led to security flaws. In particular, many C/C++ compilers can detect inaccurate format strings.

- When writing code for large projects prefer C++ over C.

Through the use of an object oriented language there are quite some security mechanisms that can be enforced at compile time. In particular, the possibility to encapsulate variables and to restrict access to variables and methods allow for a cleaner design and implementation and finally for an easier review.

3.3 C++ specific Guidelines

3.3.1 Access of variables and methods

- Declare variables and methods to be private whenever possible.

As it is good (safe) programming practice to always limit access as much as possible (principle of least privileges), everything should be declared to be private by default.

- Make public access to variables only possible via accessor methods (get/set methods).

Making variables private and limit access only via accessor methods allows to differentiate between reading (get) and writing (set) access rights. Furthermore this defines a central point, where checks can be implemented, before allowing access/modification of the variable's value.

3.3.2 Mutable objects

- Make objects immutable whenever possible.

Immutable objects are objects whose state (value of member variables) can not be changed after construction. It is a good (safe) coding rule to reduce the number of modifiable variables as much as possible. Obviously, immutable classes are optimal with respect to this principle. Furthermore, following the principle of designing and using immutable objects, also is of advantage with respect to synchronization issues.

- Never save references to mutable objects in some member variable, if such a reference is a parameter of a public constructor or method.

A necessary condition for programming secure software is the implementation of clear interfaces and strict data encapsulation. The possibility to change member variables of an object from the outside does represent a severe violation of this basic rule. Hence, if references to mutable objects are given as parameters to a public constructor or method, the constructor/method should never directly copy any such reference to its member variables. If the value represented by the referenced object has to be stored, a newly constructed copy (a clone) of this object has to be constructed first, to finally store a reference to this clone.

- Never return references to mutable member objects from public methods.

Again, the possibility to change member variables of an object from the outside does represent a severe violation of secure programming principles. If the value of a member object has to be returned, a deep copy (clone) of this object has to be constructed first, to finally return a reference to this clone.

3.3.3 Inheritance

- Program by contract using the template method design pattern.

When implementing some public method, you should "program by contract", that is you should first check all preconditions (input validation), then implement the essential routines, and finally check postconditions (if necessary) before returning. To guarantee that all implementations of a overridden method perform identical precondition and postcondition checks, the following template method design pattern should be used: Declare your public methods as final methods in your base class. Implement the precondition and postcondition checks in your base class. Then, to do the essential routines, let your method call some protected auxiliary method. It is this auxiliary method that can then be overridden in some derived classes.

3.4 C/C++ specific Guidelines

3.4.1 Buffer overflows

Buffer overflows account for most security breaches in today's software. They often occur when input data is not checked sufficiently. But they may as well occur when inherent unsafe string handling functions are used. Guidelines for input data validation are summarized in 3.1.2. This section complements those guidelines with further C/C++ specific rules.

- Avoid the usage of strcpy() and strcat(), use strncpy() and strncat() instead.

As the functions strcpy() or strcat() do not allow to limit the length of the input string to be copied or appended to the output buffer, their usage bears the potential risk that buffer overflows may occur. Therefore it is strongly recommended to use the safer function calls strncpy() and strncat() instead.

Similarly, if the functions snprintf() and vsnprintf() are available in the environment, their usage is preferable to the usage of the functions sprintf() and vsprintf().

- Avoid the usage of strcpy() and strcat(), use strncpy() and strncat() instead.

As the functions strcpy() or strcat() do not allow to limit the length of the input string to be copied or appended to the output buffer, their usage bears the potential risk that buffer overflows may occur. Therefore it is strongly recommended to use the safer function calls strncpy() and strncat() instead.

- Use precision specifiers in format strings for string arguments in formatting functions.

Calls to one of the formatting functions sprintf(), vsprintf(), scanf(), sscanf(), fscanf(), vscanf(), vsscanf() or vfscanf() may have string arguments in its format string. Most implementations of such functions allow the specification of the maximal length of such string arguments as in the following example (here at most 100 bytes are copied from the associated variable argv[0]):

```
sprintf( buffer, "Usage: %.100s argument\n", argv[0] );
```

If the length of such a string argument is not restricted by a precision specifier such calls are a potential source for buffer overflows. If the used implementation of the formatting functions does not allow the usage of precision specifiers for string arguments, all variable string arguments have to be checked for length before such a formatting function is invoked.

- If streadd() or strecpy() are used, allocate a destination buffer that is four times as long as the input buffer.

Calls to one of the functions streadd() or strecpy() require some precautions as the resulting expanded string written to the destination buffer may be up to four times as long as the input string. In order to prohibit buffer overflows to occur under all circumstances, in particular if the input string may be chosen by an attacker, output buffers to the function calls streadd() and strecpy() should be generally at least four times as long as the input buffers.

- Check output buffer sizes when calling system functions like realpath() or getpass().

Function calls like realpath() or getpass() return a string to a buffer that has to be supplied by the caller. In order to prohibit buffer overflows to occur under all circumstances, output buffers supplied to function calls have to be always of such a size as to accommodate the largest possible string returned by the function under consideration. Make sure that all such buffers sizes are determined accordingly.

- Check output buffer sizes when calling system functions like realpath() or getpass().

Function calls like realpath() or getpass() return a string to a buffer that has to be supplied by the caller. In order to prohibit buffer overflows to occur under all circumstances, output buffers supplied to function calls have to be always of such a size as to accommodate the largest possible string returned by the function under consideration. Make sure that all such buffers sizes are determined accordingly. (For realpath() and getpass() buffers of size PATH_MAX and PASS_MAX have to be supplied.)

- Check input buffer sizes when calling system functions like realpath(), syslog() or getopt().

Functions like realpath(), syslog() or getopt() take some string as input. Depending on the particular implementation of these functions it may be possible to produce a buffer overflow when invoking such functions with an unreasonable large input buffer. Therefore, it is safe programming practice to determine the maximal allowable length of such input strings for a particular application and truncate input strings accordingly before invoking functions like the one's mentioned above.

- Use auditing tools to scan source code for potential problem spots.

To scan source code for potentially dangerous uses of unsafe functions some tools may be used. These generate evaluation reports that may help in identifying security−related implementation flaws, describe the potentially problems and suggest remedies.

- Compile programs for testing purposes with compilers that produce code with extensive memory integrity checks.

To perform memory bounds checking you may use specialized compilers that produce code with checks your running program for any memory boundary violations. Such tools may also check for memory leaks and do further auditing. Usually, programs compiled with thorough generalized boundary checks do not perform efficiently, but they allow the detection of severe bugs during testing phases.

3.4.2 Programs running with root privileges (UNIX specific)

A very important programming rule is to always adhere to the principle of least privileges and to make sure to run programs only with the least privileges necessary. In particular, programs under the Unix/Linux operating system should not run with root privileges if this can be avoided.

- Do setuid/setgid programming if a program needs root privileges.

When a program does need root privileges, do setuid/setgid programming and drop root privileges as soon as possible.

- In setuid/setgid programs keep the code running with EUID/EGID set to root as short as possible and drop root privileges completely as soon as possible.

If a program that is invoked by an unprivileged user needs root privileges to perform some of its tasks, such a program must be owned by root and has its setuid bit set. As long as the effective user ID (EUID) of such a setuid program is not reset to the UID of the invoking user (using setreuid()), the program runs with root privileges. Furthermore, child processes do inherit the EUID of such setuid programs. As long as the effective user ID (EUID) of a setuid program is set to root, the program runs with root privileges and a successful attack may result in a root shell. Therefore, the bulk of a program should run as an unprivileged user. If this is really not possible, make use of the technique of compartmentalization, i.e. isolate the part of your program that has to run as root and make this part as small and simple as possible and let the remaining part run as an unprivileged program.

3.4.3 TOCTOU attacks

- Always avoid to close and reopen files that live in directories that may be susceptible to access by another process (e.g. temporary files).

In particular programs running with root privileges that have to access the file system may be vulnerable to security critical race conditions. To avoid "Time−of− check, time−of−use" flaws (TOCTOU) it is important to take measures that prohibit an attacker to change a file (or a link to a file) after such a file has been checked but before it is used. For example, vulnerable files should be kept open as long as it may be necessary to have access to them.

- Avoid file system calls that take a filename for an input argument and prefer those calls that take file handles or file descriptors.

For many purposes it is possible to either use file system calls that take filenames as input parameter or to use file system calls that reference files by file handles or file descriptors. If file system calls are used that take a filename for an input argument programs are vulnerable to attacks that change a file (a link to a file) in between of two such calls. Hence, after a file has been opened, it should only be accessed via its file handle or file descriptor whenever possible.

- Do not use the access() function in setuid programs.

(c) 2005, EUROSEC GmbH Chiffriertechnik & Sicherheit

For security critical file operations a setuid program should make sure to operate on the intended files and not on files that may have been replaced by an attacker by changing symbolic names. As the access() function takes a filename as an an input argument, its use is not safe against TOCTOU attacks. Therefore, placing all files used by the program in a directory that is only accessible by the UID of the program is a safer alternative.

3.5 Security critical modules and tasks

3.5.1 Separation of security critical modules

- Identify security critical modules/tasks during the specification and design phase.

Identifying security relevant tasks, such as the generation of PIN numbers, in early phases of the software development process is mandatory in order to define clean interfaces to such functionalities. Furthermore, having identified such tasks/modules, implementation and review of such modules can be delegated to software engineers with appropriate skills.

- Regularly review all code to check whether security critical methods are only implemented in distinguished modules.

To assure some security standard of the overall product it is mandatory that all security critical tasks are delegated to dedicated modules. Some poor ad hoc implementation of such a task may ruin the overall security goals completely.

3.5.2 Cryptographic and security relevant libraries

- Use well established or evaluated libraries for security critical algorithms and services.

Taking special precautions when developing such modules is mandatory to assure some level of security. Therefore, you might consider whether the usage of third party software libraries might be an alternative. Using such third party software that has been certified or is open source and has been reviewed thoroughly, might increase the level of assurance that the modules satisfy your security requirements considerably.

- Do not implement some proprietary cryptographic or security relevant routines.

It is generally good practice to only use well established and investigated algorithms for cryptographic purposes. Proprietary algorithms have very often been the reason for embarrassing security holes.

3.5.3 Sensitive data

- Do not store some secrets in your code (like pins or secret keys for encrypting user input).

For anyone who may obtain a copy of the code, secrets compiled into the code are NO secrets! A skilled attacker will be able to find such secrets, for example by disassembling and decompiling the code. If you can take the risk that some skilled attackers may find such data, you may nevertheless take measures that keep less skilled persons from fulfilling this task such as stripping the code, implementing antidebugger measures or using obuscation techniques.

- Take special precautions, when processing sensitive user input data like PINs or passwords to clear such data as soon as possible.

To avoid heap inspection attacks and unnecessary swapping of secret data to the disk, it is good programming practice to clear the data (overwriting the corresponding memory), as soon as the sensitive data has been processed.

- Take care not to echo sensitive data to the UI.

When users have to input some sensitive data, it is good practice to not echo the keystrokes, either by not displaying any characters or by displaying some fixed char for every keystroke (often a '*').

3.5.4 Pseudo random data

- Never use standard function calls like rand() or random() for the generation of random numbers in security relevant contexts.

Random numbers are needed in many security relevant contexts, e.g. for the generation of initial passwords, PINs, TANs or SessionIDs. Function calls like random() do not provide a source of cryptographically secure randomness. To the contrary the sequence of output data is predictable, as soon as some consecutive values have been observed. If the usage of a cryptographically secure source of randomness is needed, an appropriate implementation should be provided globally in a dedicated module that has to be used for such purposes. Remember that the implementation of a well seeded pseudo random number generator is a non trivial task.

4 Resources

4.1 Bibliography

4.1.1 Guidelines on secure programming

- John Viega, Gary McGraw, *Building secure software: How to avoid security problems the right way,* Addison−Wesley, 2002. (http://www.buildingsecuresoftware.com)
- David A. Wheeler, *Secure Programming for Linux and Unix HOWTO,* (http://www.dwheeler.com/secure-programs/)
- Michael Howard, David LeBlanc, *Writing secure code,* Microsoft Press, 2002.
- Lawrence C. Paulson, *Software Engineering II*, Lawrence C Paulson Computer Laboratory University of Cambridge, Lecture Notes, 1999. (http://www.cl.cam.ac.uk/Teaching/2001/SWEng2/notes.pdf)
- Steve Bellovin (smb@research.att.com), *Shifting the Odds, Writing (More) Secure Software*, 1996, 36 Seiten
- M. Bishop, Robust Programming, 1998.
- M. Bishop, UNIX Security: Security in Programming, SANS Tutorial, 1998.

4.1.2 Buffer overflows

- Aleph One . Smashing The Stack For Fun And Profit. Phrack, 7(49), November 1996.(http://www.phrack.org/)
- DilDog . The Tao of Windows Buffer Overflow, April 1998(http://www.cultdeadcow.com/cDc_files/cDc-351/)

4.1.3 Format string attacks

- Kalou/Pascal Bouchareine, Format string vulnerability, The Hacker Emergency Response Team (HERT), first published in Bugtraq, Tue, 18 Jul 2000. (http://www.hert.org/papers/format.html)
- scut/team teso, Exploiting Format String Vulnerabilities, March 2001. (http://www.team-teso.net/)

- C. Blaess, C. Grenier, F. Raynal, Avoiding security holes when developing an application – 4: format strings (http://www.security-labs.org/full-page.php3?page=120)

4.1.4 Setuid programming

- M. Bishop, How to Write a Setuid Program, :login; 12(1), Jan/Feb 1986. (http://nob.cs.ucdavis.edu/~bishop/papers/Pdf/sproglogin.pdf)

4.1.5 Race conditions

- M. Bishop and M. Dilger, Checking for Race Conditions in File Accesses, Computing Systems 9(2) pp. 131–152, 1996. (http://nob.cs.ucdavis.edu/~bishop/papers/)

(c) 2005, EUROSEC GmbH Chiffriertechnik & Sicherheit