On the Importance of Secure Coding

Hagai Bar-El
info@hbarel.com

Abstract

Secure coding (secure programming) is a field that is gaining a lot of attention. Flaws are constantly discovered in a wide range of known server applications. These flaws are not flaws emerging from an insecure high-level design of the applications but are flaws that were introduced at the source code level and that are a result of careless programming. Such flaws can be exploits of buffer overflows or the result of lacking input validation routines. In this document I will provide a brief definition of secure coding and of secure programs and will try to assess the reasons for the need to focus efforts on this aspect of information security.

Definition of Secure Coding

Secure coding can be most intuitively defined as the act of writing *secure programs*. Secure programs are programs that cannot be manipulated (other than by changing the software program object itself) into performing *illegal operations*. Illegal operations, for the purpose of this discussion, are operations that compromise security and that the program was not intended to perform according to its design, or was not intended to perform in the circumstances in which it was manipulated into performing. It is worth mentioning that programs that have bugs in them are not necessarily insecure programs according to this definition. Buggy programs may be insecure if their bugs allow an adversary to perform illegal operations according to their abovementioned definition. If a bug can cause the program to perform some unpredicted operation but this operation has no significance in terms of an information security compromise, then the program is not necessarily considered as insecure.

Why is Secure Coding Important?

Application design is multi-layered of its nature. Every application that is of a minimal complexity and above requires design to be done in multiple layers. A protocol implementation, for example, requires a design for the protocol itself, a design of top-level functions and interfaces, a detailed design of the various functions implementation, and the design of the program code itself. The coding itself can be considered as the bottom layer of the application design.

For security to be complete it requires attention in each of the design stages. A secure protocol implementation first needs to be designed securely as a protocol in the protocol definition level, its implementation is then required to be designed with security in mind so not to introduce implementation-specific vulnerabilities. Within the implementation design, the functions need to be defined so security issues do not fall in between the cracks and so the security roles are well defined and well assigned among the components of the implementation. Further down the implementation chain, the source code itself must be designed securely so not to introduce flaws resulting from bad assumptions being made by the programmer or from careless use of functions and resources. Secure coding is nothing but a link in the chain of design security, which is the lowest one in the implementation's top-down model. As often said, a chain is as secure as its weakest link. As a result, the security of this low-level link shall not fall below the security of any link above it so not to compromise the security level of the entire application.

One parameter that hardens the keeping of the lowest level as secure as the higher levels is that security is harder to assure on the lower levels than it is on the higher ones. Put in different words, security flaws are harder to spot when they are in lower levels of the design. The closer we get to the bit level the lower is the probability of flaw discovery. This is for several reasons: First, the skill of most security analysts is based heavily on design-levels, somewhat justified by the wrong perception that security is an issue of the "top layers". Indeed, security design needs to start from the very top, but it shall definitely not end there. Security analysts often feel closer to Data Flow Diagrams and to high-level interface definitions than to C language instructions or to op-codes. The second reason is that properly assuring the security of source code requires a more-or-less manual review of high portions of the source code. Some code scanners exist to make this task less time consuming by flagging problematic function calls. Still, no matter how one looks at it, this is very time consuming task, and this time is often far from pleasant. A complete security review on the source code, which may require a lot of man-hours, is likely to eventually add to the overall cost of the product. Therefore, source code review is seldom done and even more seldom done correctly. These facts naturally lead to flaws that are introduced in the lowest layer of the implementation and that are never discovered by the developer.

Unfortunately, too often are these flaws discovered by the wrong people leading to the publication of exploit codes. Exploit codes may lead almost any sort of application on which they run to perform almost any operation. Web server applications got the most attention lately and exploits were written for manipulating them into displaying sensitive files, into invoking shells and into running arbitrary code.

Causes Of Exploitable Flaws

Regular programming bugs can come from many sources and can be caused by a large set of common mistakes. Exploitable security flaws in programs, on the other hand, usually originate from one of several common habits of short-seeing. The purpose of this article is not to present the causes of software flaws and therefore this chapter is brief and shall be seen as background or introductory information only.

Most commonly, assumptions are made regarding the type of input that the program receives from an external source and often also regarding its length. *Buffer overflows* are the result of many programming flaws and are caused when the application attempts to store information it received from an external source in pre-allocated memory space that is not large enough. In web programming environments web servers are often lead into performing illegal operations by being sent well-crafted strings that were unpredicted by the programmer but yet interpretable by the server. *Race condition* flaws result from the programmer's assumption that a state that was verified at some point still holds at a later moment when operations based on that state are performed.

Many of the flaws originate from the lack of safety nets provided by modern computing platforms. The compilers are providing the programmer with complete flexibility in use of memory and other resources, trusting him to use this capability with care. The programmer, on the other hand, is assuming the compiler and the environment to provide invisible *safety nets* that will eliminate problems caused when this flexibility is applied in circumstances that are unexpected at the stage of programming. As a brief example: The C language compilers provide the running application with flexible access to the memory space, including access to memory addresses that were not formally allocated for the specific purpose beforehand. The programmer writing a function that receives a string from the keyboard and placing it in a limited memory buffer assumes the environment will not allow contents of the string to overflow the allocated space if too long. This assumption, in different flavors, is the main cause of buffer overflows.

Changing programming habits and verifying that the source code makes no assumptions on the inputs it receives is the only way to solve this problem. Compilers cannot embed automatic runtime checks without a notable loss of performance. Similarly, interpreter modules cannot be programmed to know and react to special malicious strings of data that they may get at runtime, also because "malicious" is a context-sensitive property. Moreover, some flaws result from the interaction between independent components. In these cases an automated "filtering" of risky situations is completely inapplicable.

Security Through Obscurity

"Security Through Obscurity" is the approach by which an applications security level is considered to be increased due to the non-disclosure of its internals. In common wording it means that an application is secure if the attacker does not have the necessary information about how it works. A lot was said and written about this approach and mainly about the reasons it is so wrong.

No application may rely its security on the fact that the adversary does not know how the application runs. Briefly, the hidden nature of the applications code cannot be trusted for assuring the security of the application. The first reason for that is that the fact that the source code is not provided with the application does not mean the internal structure of the application cannot be discovered. The source code may be revealed at some future time for this or that reason, or leaked. Even if the source code is forever kept safe, reverse engineering is possible and detailed analysis of the machine code can teach an adversary a lot about the functions, structure, input validation schemes and other components of the program. One must remember that in order for an adversary to be able to attack an implementation he/she does not need to have a complete and accurate dump of the source code. Often, a short glimpse at a small block of the machine code can reveal enough information for the launch of a successful attack. Generally speaking, according to a known lemma the secrecy of any software component cannot be assured against the owner of the machine on which the software is run (true for common computing platforms).

Security Through Obscurity can provide an additional layer of security for system that are secure otherwise, but should never be trusted as the only level of defense.

Summary

Secure coding is nothing but an additional link in the chain of security measures that are to be taken in the various stages of application development. However, it does have several unique properties that make it slightly different than the other security considerations that are taken in the higher levels of the design. Security of source code is harder to assure, as source code is harder to inspect, for its being written in a non-human language, and as the vulnerabilities are harder to detect and their detection requires different skills than the skills most information security analysts have. In order to assure the security of this level of the development process, the most effective way is through the adoption of secure programming practices and through the adequate training of programmers. Source code inspection is also necessary as an additional measure of flaw detection. Without both the initial awareness of the source code programmers and the code review by skillful analysts, the task of assuring the robustness of an application code against skilled attackers with a high motivation and a lot of patience is close to impossible.