Writing Secure Software Is Just Difficult Or Plainly Impossible ?

Writing about writing software is also a difficult ordeal. Too much has been written, said and discussed about writing software that picking up again the subject makes the reader automatically skip to the next article. So, for the few readers that managed to get to this point we immediately state that we will not discuss about one approach / method / theory of writing software with respect to another, nor we will propose our approach to manage small, medium or large projects. We will instead try to understand where we stand today from the point of view of someone who has been involved in small and medium projects of software development.

But let us start from a more general point of view. Let us assume that we would be able to write *secure* software, that is software that:

1. given any input, always produce the expected output
2. never allows the user to input wrong data or to misuse the program.

If software would behave according to 1 and 2, there would not exist bugs (1) nor vulnerabilities and exploits (2) from viruses to phishing and so on. We would not need reactive ICT security as it is security today, but ICT security would be needed only as one of the guiding principles of the designing phase of the software and of the overall applications.

Obviously this is pure utopia.

There are many competing aspects which make it impossible to get close to these two points. For example:

I. even medium size project produce huge amount of code

II. projects are always late and there is a big pressure from the management on the developers to cut corners, simplify, omit something to finish not too late

III. for clear marketing and commercial reasons, software should be monolithic, that is a single object which does everything, accepting all kinds of inputs and doing all possible kinds of processing on it

IV. requested features keep changing, usually by the time that the first prototype of the program is ready, the final user has requested to add some new features and remove some others, and often these modifications require structural changes in the program.

But what is more important from our point of view is that the approach to writing software is almost exclusively based on the *functional* aspects of it, and it cannot be otherwise!

This is because there is a basic dichotomy which we have not yet been able to overcome: *humans are not machines*. The way we think is, at least at the moment, quite different from how machines execute code. Machines execute structural and sequential instructions whereas we prefer to associate and jump between concepts by similarity. Thus the way a user expects an IT system to behave is fundamentally different from how an IT system behaves since we are not able to think as machines.

This leads to the unsolved User Interface (UI) problem. The User Interface is that software component which should allow the human user to understand and utilize the software. Thus it should translate human *intentions* into machine instructions and code execution. Here we find our first critical security issue.

From the user point of view we would like the User Interface to be able to understand the human requests. This should be accomplished by guiding the user through a set of choices and inputs which would follow a human reasoning sequence. But we humans are not all equals and what is intuitive for one of us, could be counter-intuitive for many others. This depends both on basic human skills (down to the genetic level) and also on education, experience, culture etc. So guiding a user through a *reasoning* which would make the application understand what the user wants, is not that simple: one has to accommodate for many different possible users, reasoning and approaches. If we bring this to the extreme, we get to a UI that wither asks too many questions and rapidly becomes unusable or tries to guess what to do. In either cases the major risk is to accept input from the user which actually should not be accepted. Indeed the more freedom we allow the user to give almost arbitrary input to the program, the more difficult it becomes to parse, control the input and take the correct decision on what to do.

From the security point of view, there are two kinds of risks:

1. if the UI is not clear enough or the user does not understand it, the user can be lead to give wrong inputs and, without knowing and understanding, to do the *wrong* thing; this is where all kinds of swindle come in, from phishing on
2. a malicious user or attacker can exploit the freedom of possible inputs and lack of strict controls, to subverse the application and make it do what it was not supposed to do.

We often hear from developers that problem 1. is not an IT and UI problem: “If people are stupid

and fall for it, it is their problem, not a problem of my application." This is obviously partially true: there will always be people who would "fall for it" and this because they have not understood what they should do and how it works. But, as we said, the UI must aim to make the highest possible number of users understand and use correctly the program, and this number should be close as possible to 100% and not for example only 30%.

For what concerns 2. a developer would tell you that he cannot allow a user to input some kind of data for usability and at the same time deny the input of the same kind of data for security reasons. This is again partially true: if from a technical point of view and for security reasons you should not allow the user to input some kind of data, but you are required to allow it anyway, then you should guarantee that you fully parse this data to prevent all possible kinds of abuse. As we will see later on, this has basically two kinds of problems: In complex applications it is practically impossible to control all possible flows of the programs and their income and outcome; Strict controls on all inputs and outputs could slow down the application to the point of becoming unusable.

So for usability we need to give the user reasonable freedom on the interaction with the program and to make the program guess what the user wants. From the security point of view we need to severely limit what the user can input and, even better, make the user follow the computer logic instead of the human reasoning.

This dichotomy is often seen in meetings between code developers and UI designers: the first think in computer logic, the latter tries to adhere to human reasoning. The latter make requests that the first say are not possible to satisfy or implement, (i.e. they are not able to write code which does that). Then, after a long discussion a compromise is found and agreed upon which often

1. makes coding difficult and prone to security risks
2. introduces in the UI features which confuse the user and make it less intuitive.

Somehow this is the major problem that we do not know really how to solve. We are able to write secure code following the computer logic, but this cannot be easily used by general users and often does not have all the features which are requested. Vice-versa, we are not able to write applications following the human logic that work and are secure. So we are stuck in between trying to realize something usable and at the same time without too many bugs and security holes. From a security perspective this is in practice equivalent to ignoring or violating security principles and best practices. Good intentions are usually not enough!

On the other side, when one is managing a software development project, one knows that the main problems he has to face are

1. keep the project in schedule
2. assure that the requested features work as expected.

These two have top priority because otherwise one will suffer direct economical losses. These problem are big and serious enough (companies can go broke on them) that nothing else is really considered unless *there will be time* (and this will never be).

Thus even if we always start with the best intentions about writing secure code, already at day one of the project we postpone till an unclear future the security issues, since we do not really understand the functionality issues and we definitively need to solve these first!

But of course, if we would apply a *secure* approach to the development of software, we would automatically solve most of the ICT security issues. The *secure* approach must be pervasive, at every level and stage of the project, from the highest and most general specifications to the lowest level code.

Even if this has been said, written and repeated many times, it is extremely rare that a software development project is managed at all levels with a secure approach. A secure approach usually means adding some extra difficulties to the two main problems mentioned before (schedule and features) and unfortunately its importance in practice is too often underestimated.

There are two main issues that if correctly considered at all levels and stages of a software development project, do add a good security stance to the final product:

1. control of the data
2. control of the flow of execution.

Let us discuss again these two issues but from a different point of view. Controlling data is done in two points: the first when the user inputs data or the program reads data from a source, the second when data is passed between different components of the program itself. When data enters an application it must be always parsed and normalized. The best approach is to identify for each possible inputs the set of characters (and the possible character-sets), parse each input, transform it in a fixed character-set, and if there are characters not allowed, reject the input. After that, one should check if the input makes sense and only if it does it should be processed, otherwise again it should be rejected.

Unfortunately it is often difficult to make a list of allowed characters, in particular for direct user input where this is often seen as a limitation to the UI. So too often the opposite approach is taken, that is to make a list of forbidden characters. The main problem with this approach is that the list is never complete and that it usually changes every time new features and components are added to the program. At the end in this way we cannot be sure if the input is safe to be processed or not.

Data must be checked also when it is passed between different components of the program. The problem here is to decide down to which level we should check the data. In principle we should do it at the level of each routine, but this usually is too much, in the sense that it will add too much code which slows down the execution and increase the possibility of adding bugs. So usually extra checks on passed data are rare, and it ends up that each routine trusts that the data that receives from the other routines of the program (written in different times by different people) is safe for its own processing. Unfortunately this trust is often misplaced and the principal causes are the many changes or additions done during the life of the program.

Concerning the flow of execution of the program, the main issue here is the well known problem of the *'if-then-else'* where the '*else*' is missing. Writing software is often a very big exercise in logic and the basic building block of logic is the '*if-then-else*' construct. This pervades the life of a program at all level, from the high level specifications and the main features and flow of data, to the smallest routine. The missing '*else*' is a logical killer, and if it is easy to deal with it in the low level code, it is often difficult even to realized that it exists at high level.

Indeed, when one writes code, instead of relying on the default values for variables and statements, it is more safe to impose that each '*if*', with its possible '*else-if's'*, is always closed by an '*else'*. In this way, at the expense of a couple of extra lines of code, the low level code will never end up in undefined territory, since all possible situations are always considered.

When you consider how different components of the program interacts and the full flow of the execution, again usually there are many '*if*' but they are often more difficult to understand since they are not written explicitly in the code. If it is more difficult to find the '*if*', it is even more difficult to understand if the related '*else*' is present or not. The larger the program, the number of components, the number of features etc., the more difficult is to understand if all possible flows of execution are correctly handled and that there are not situations where a missing logical tree could lead to unexpected results, that is bugs, data mishandling and security risks.

Andrea Pasquinucci

PhD CISA CISSP

http://www.ucci.it/