

Secure Software

John D. McGregor, Clemson University and Luminary Software LLC, U.S.A.

Abstract

Granting access to those who should have it and denying access to those who shouldn't is a basic feature in many software products. Security is of strategic importance in many markets and types of products. In this month's issue of Strategic Software Engineering, I will explore some issues about the strategic importance of security. I will discuss the influence of other product qualities such as correctness on the security of the product.

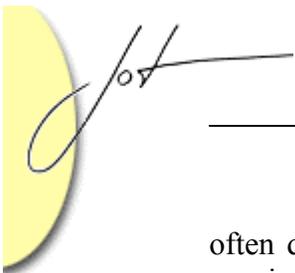
1 INTRODUCTION

In January 2005, George Mason University found that hackers had gained access to a database containing information on 32,000 people including their social security numbers. In February 2005, ChoicePoint, a data collection service, announced that a security breach threatened the personal information of over 145,000 people. And the list could go on. We want the software that manages our personal and professional data to be secure.

I recently attended the First Annual Cyber Security and Information Infrastructure Research (CSIIR) Workshop on Software Security held at Oak Ridge National Laboratory. As part of that workshop I made a presentation based on the following premise: *Poorly written software will have more security vulnerabilities than well written software*. In this issue I will expand on that topic with emphasis of the strategic importance of security.

Notice that the title of this column is “secure software” as opposed to “software security.” That’s intentional. I am viewing this as the quality attribute of “being secure” rather than considering security features such as access control and data encryption. Taking this approach brings engineering processes to bear on the problem of how to achieve that or any other quality.

Software is secure when those who have authorization can use its functions and when those who do not have authorization can not. The secure quality attribute extends this definition to the data managed by the software. It is difficult to confine the achievement of security to a single application since its security usually depends upon the security of the operating system and utilities that provide essential services. Therefore, security is



often defined as a quality of an entire environment – the platform and all applications running on that platform.

Obviously being secure, like any quality attribute, is more important in some products than others. Software that controls significant hardware, such as an airplane, or that manages significant data, such as banking information, will require much higher levels of security than software that controls an advertising sign or plays a game. However, a security vulnerability in a game running on a platform shared with software that performs secure business transactions may endanger those transactions.

McGraw makes the very good point that much of the software security work deals with operational level fixes because they were designed by operational level people [McGraw 04]. That is too late. I will discuss how to incorporate security as a quality consideration early in the development life cycle.

2 MOTIVATION

There is growing support for developing secure software by focusing on software engineering best practices. I will review some of the literature so that you can understand the range of responses to the security problem and so that I can discuss several aspects of the software engineering approach.

The Security Across the Software Development Lifecycle Task Force led by co-chairs Ron Moritz of Computer Associates, and Scott Charney of Microsoft made a number of recommendations about improving software development techniques that will in turn improve the security of the software being produced [Moritz 04]. Included in those recommendations are:

- Adopt software development processes that can measurably reduce software specification, design and implementation defects.
- Software producers should adopt practices for developing secure software.
- Software producers, where appropriate, should conduct measured trials of available approaches and publish their results.

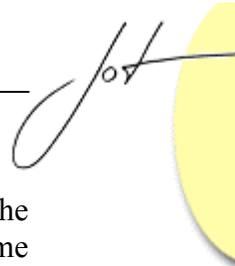
I will discuss some actions that follow the first two recommendations.

Gary McGraw in his “Building Security In” department in IEEE’s Security and Privacy and describes the “trinity of trouble. [McGraw 04]” These are three problems that contribute to increasing security problems. They are:

- Ubiquitous network connections
- Easily extensible systems
- Increasingly complex systems

The latter two problems are clearly software engineering issues and I will address these shortly.

What makes secure software strategic to a company? It’s the risk of losing the trust of your customers as well as the risk of litigation. There are several ways to mitigate these



risks. In the next section I will discuss qualities of software products that reduce the probability of security problems and in the following section I will discuss some techniques for achieving those qualities.

3 QUALITIES RELATED TO SECURITY

Talking about “well written” software, as I did in my premise, is too vague for engineering analysis. In this section I want to examine specific qualities and their relationship to being secure. Almost any specified quality that is not achieved by the product could degrade the secure quality and introduce a security vulnerability. However, certain qualities speak to the resistance of a product to attack.

Correct

Correctness is a quality that is often implicitly required rather than explicitly specified. For our purposes, correctness is the ability of a software product to satisfy its functional requirements. Security is often compromised by the mistaken idea that a formal proof of the specification results in correctness. A proof is only a first step. In fact some of the most prevalent security vulnerabilities arise from either an incomplete specification or a failure to implement the specification exactly as stated.

Buffer overflow errors account for a large percentage of vulnerabilities. So called “complete” specifications often consider only static qualities and do not specify operational characteristics such as maximum size of a data structure or how overflows will be handled.

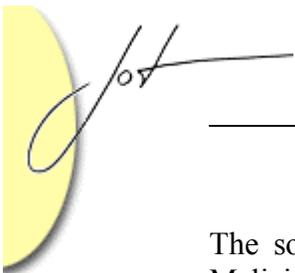
Even if the specification was complete and correct, it will not matter if the implementation of that specification is created by a human. Automatic program generation is similarly flawed unless the generator has been proven correct.

If the program is not correct then it becomes difficult to know whether the program’s failure to meet expectations is due to a security breach or just built-in incorrectness.

Robust

The degree to which a product is robust is the percentage of time it can continue to function in the face of unusual conditions. Notice that I did not say function correctly. The specification often does not indicate what happens in the case of unexpected errors in which case there is no definition of “correct.”

Embedded systems are often defined to be robust by having an error state in which the system is specified to perform some function that will do the least harm to the hardware or the environment. This often is a reset back to its initial state but it may be a transition to some other intermediate state. Any input that is not covered by the specification results in the system entering that error state.



The software is vulnerable in those regions of input where there is no specification. Malicious attacks are often probes to find the boundaries of a specification. Once this limit is established, data is supplied that stresses the system looking for areas in which the software is not robust.

Robustness is not only affected by malicious attacks. It also is affected by accidental data errors.

Reliable

The reliability of a product is measured by the percentage of the operating time that the product performs requested functions correctly. Software is vulnerable when there are specified inputs for which the product does not produce correct results. The more unreliable the software the more vulnerable the software.

Reliability is a property of individual components and it is an emergent property of an assembly of components. When the faults causing the software to fail is a result of the composition the vulnerability is particularly difficult to recognize. It can not be found during unit testing and may be so narrow that finding it during system testing is also unlikely.

Quality assurance activities are needed at each of the three traditional levels: unit, integration, and system.

4 BUILDING SECURE SOFTWARE

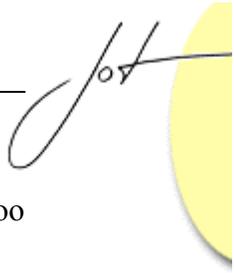
Consistent error handling

A large number of vulnerabilities are exploited by causing an error and taking advantage of how the error is handled. The best strategy is to bullet-proof the software so that errors don't happen but it is doubtful that will happen any time soon. Therefore, the alternative strategy is to provide a consistent error handling scheme. The expectation is engineers will be less likely to make mistakes in the presence of a consistent error handling scheme

I will not go into a comparison of returning error codes versus exceptions here. The point to be made here is that the error handling needs to be visible at the appropriate level. Error mechanisms that will be propagated between functions but within the component must be visible in the function-level specifications within the component. Error mechanisms that will be propagated between components need to be explicit and public in the component's specification.

Robust data structures

As I said above the best defense is to bullet proof the software. Buffer overflows are a leading source of vulnerability. One of the participants in the CSIIR workshop made the excellent point that you can't overflow a hardware buffer. Why should it be different with



a software buffer? There are widely-used practices that can prevent overflows but too often they are not followed.

What is the acceptable behavior when new data is available and there is no room for it in the existing buffer? The possible answers are:

- Standard approach – continue as usual, runoff the end, reference random memory, cause wild and crazy things to happen in your program
- Not so standard approach – do nothing, don't write the data, it will eventually be lost
- Throw a specified exception – allow others to handle
- Expand buffer to accommodate, after checking that there really is more memory

Obviously, the first two approaches are not acceptable but the first one is widely used. The last two approaches are not mutually exclusive. Taken together they form an implementation pattern (different from a language idiom and more detailed than a design pattern). Figure 1 shows the decision tree for the implementation. Different languages will require different language idioms.

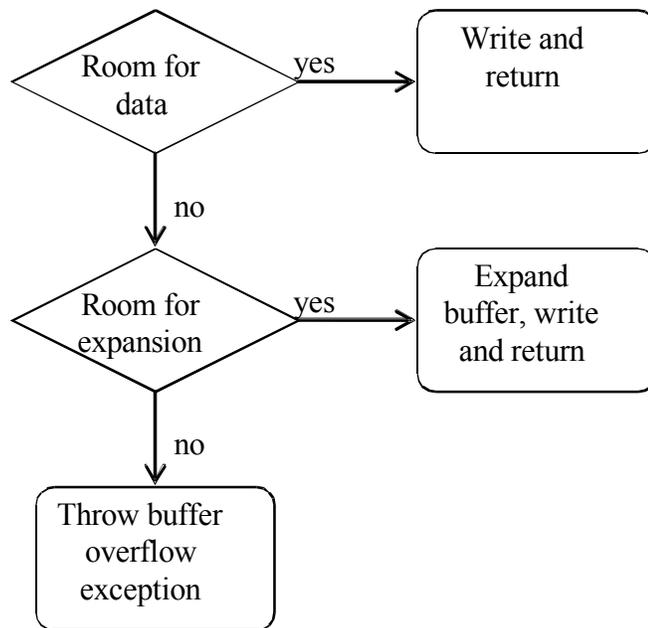
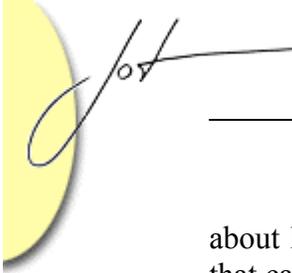


Figure 1 - Buffer overflow implementation pattern

While I have focused on the buffer overflow problem for obvious reasons, the same detailed analysis should be done for every state that is maintained in a product.

Misuse and Abuse cases

Software engineering provides techniques to build a product to a purpose. The use case technique has proven an effective technique for capturing the thinking of stakeholders



about how the product will be used [Jacobson 92]. Change cases are a type of use case that capture how stakeholders think the product will change [Ecklund 96].

Several authors describe misuse and abuse cases as an approach to helping stakeholders think about possible scenarios that need to be defended against [Hope 04]. This includes defining actors that model attackers and brain storming how the attackers would “use” the system. These abuse cases can be built on known attack patterns. The report of the Security Across the Software Development Lifecycle Task Force included a list of 49 such patterns. Table 1 shows a few of their attack patterns.

Table 1 - Attack patterns

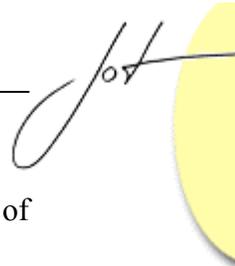
Use a User-Supplied Configuration File to Run Commands That Elevate Privilege
Make Use of Configuration File Search Paths
Direct Access to Executable Files
Embedding Scripts within Scripts
Leverage Executable Code in Non-executable Files
Argument Injection

Our use case template includes multiple scenarios that describe how the actor uses the product. We include “sunny day,” alternative, and exceptional scenarios. Misuse scenarios can also be included in the standard use cases. These differ from the ones that accompany an attack actor in that these may be accidental situations initiated by an innocent, careless user.

5 SUMMARY

I have discussed some of the things that are being done to engineer secure software. I believe there is much more that can be done. At the workshop I proposed an agenda for for research to expand the range of techniques available for engineering security. The items on the agenda are:

- Develop method engineering tactics and guidelines that enhance the security quality of the software through improved processes.
- Structure architecture evaluation techniques to focus on security by searching for static security patterns.
- Discover and capture test patterns that correspond to dynamic security patterns.
- Develop focused test techniques to effectively explore security test patterns while reducing the test suite size.



-
- Create a defect model for security that can be used to predict types and number of security vulnerabilities in scientific codes.

Execution of these actions would add to the set of tactics that are currently available for engineering secure software.

Security is a quality like any other non-functional requirement. It must be engineered into the product rather than being added on at the last minute. It can also be subject to tradeoff with other more important qualities – security versus testability for example. It can also be a point of variation in a product line architecture - products that are secure and products that are not.

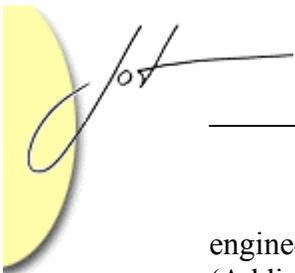
Security becomes more important as more of our personal and business data is computerized. The secure quality attribute has to be as carefully engineered as every other quality upon which our strategic goals depend.

REFERENCES

- [Ecklund 96] Earl F. Ecklund, Lois M.L. Delcambre, and Michael J. Freiling. *Change Cases: Use cases that identify future requirements*. **Proceedings of the Eleventh Conference on Object-Oriented Programming Systems, Languages, and Applications**, Association for Computing Machinery, 1996.
- [Hope 04] Paco Hope, Gary McGraw, and Annie I. Anton. *Misuse and Abuse Cases: Getting Past the Positive*, **IEEE Security and Privacy**, IEEE Computer Society, 2004.
- [Jacobson 92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. **Object-Oriented Software Engineering**, Addison-Wesley, reading, Massachusetts, 1992.
- [McGraw 04] Gary McGraw. *Software Security*, **IEEE Security & Privacy**, IEEE Computer Society, 2004.
- [Moritz 04] Ron Moritz and Scott Charney. **Improving Security Across the Software Development Life Cycle**, Security Across the Software Development Lifecycle Task Force, 2004.

About the author

Dr. John D. McGregor is an associate professor of computer science at Clemson University and a partner in Luminary Software, a software engineering consulting firm. His research interests include software product lines and component-base software



engineering. His latest book is *A Practical Guide to Testing Object-Oriented Software* (Addison-Wesley 2001). Contact him at johnmc@lumsoft.com.