Rules of Thumb for Developing Secure Software: Analyzing and consolidating two proposed sets of rules

Dr. Holger Peine
Fraunhofer Institute for Experimental Software Engineering, Germany
Holger.Peine@iese.fraunhofer.de

Abstract

This paper presents guidelines to develop secure applications in the form of "Do's and Don'ts" applying mostly to the software design level, but also to the implementation level. It builds on two collections of similar rules published in two seminal books in the area of secure software development, criticizes and improves those earlier rules and extends them by several new ones. The paper does not cover how to apply such rules in general. The main direction of improvement is making the rules more constructive, less ambiguous, and removing aspects not related to security.

1. Introduction

An email client executing script code of unknown origin with full access to your files? A privileged program letting itself be tricked into misusing its privileges by feeding it unexpected parameter values? A system program leaving traces of secret information in an unprotected temporary disk file? All these are examples of computer security breaches caused by careless programming. Such software security blunders are the root cause of a large share of computer security failures (the other two major causes being configuration errors — such as leaving network services open for global, unauthenticated access — and usage errors such as blindly clicking on links in unsolicited email messages).

Software security blunders abound because security knowledge is not very wide-spread among software developers — not to mention the all too-common attitude of "let's get the features first, we'll add security later (well, if there's some time left then...)". University courses on computer security usually cover abstract security goals, encryption techniques, network security, and maybe some security management procedures. However, *software* security knowledge, that is, how to develop software that cannot be tricked into performing any insecure function, has been largely ignored until a few years ago.

The recently growing atttention to software security can be arguably pinpointed to the publication of two ground-breaking books on secure software development: "Building Secure Software" by John Viega and Gary McGraw [1] and "Writing Secure Code" by Michael Howard and David LeBlanc [2]. Both books contain highly useful information on software security at the general, conceptual level as well as concrete coding tips at the implementation level, without actually overlapping too much in detail[1]. Another commonality is that both books try to condense their advice into 10 (resp. 13) easily remembered rules, a sort of "10 commandments of secure programming". These two sets of nicely succinct rules (see the resepective tables below) have been publicized widely, and serious scientific work has started to build on them.

However, I find both sets of rules profiting from a closer analysis and an overhaul that weeds out irrelevant aspects, directs the formulations towards concrete advice, and recognizes duplicates. In particular, I deem it important to intercept advice which is perfectly sensible for good software development in general, but not specific to *secure* software development — the danger with such general advice is that a superficial reader might get the impression "I've heard all this before so many times, I don't need to read further". Additionally, I suggest to add a few new rules to the consolidated set, arriving at a set of 20 rules after all, which I would like to propose as an improved and unified version of all the

1. Not to the least because [1] focuses on Unix software, while [2] does the same for Windows.

0-7695-3102-4/08 $25.00 © 2008 IEEE
DOI 10.1109/ARES.2008.142

rules proposed so far. This paper will proceed now to criticize the two referenced rule sets rule by rule, and then present the suggested consolidated set, elaborating a bit further on the new rules. Note that possible ways to apply such rules are not discussed in this paper.

2. Rules proposed by Viega and McGraw

Turning first to the rule set proposed in "Building Secure Software" and listed in table 1, I find nothing wrong in principle with rules VM1 to VM4; however, I suggest spending a few more words on rule VM1 "Secure the weakest link" by adding "... not the easiest or most obvious one", which points out the most common error this rule is intended to guard against. Rule VM2, "Practice defense in depth" could be rephrased as "Use several layers of defence", which expresses the same intention more clearly, and VM3 "Fail securely" as "Stay secure in case of failures" for the same reason.

Table 1. Rules by Viega and McGraw

VM1	Secure the weakest link
VM2	Practice defense in depth
VM3	Fail securely
VM4	Follow the principle of least privilege
VM5	Compartmentalize
VM6	Keep it simple
VM7	Promote privacy
VM8	Remember that hiding secrets is hard
VM9	Be reluctant to trust
VM10	Use your community resources

A rule that clearly profits from some elaboration is VM5, "Compartmentalize". This formulation only evokes a vague conception that there should be components in an application that are somehow isolated. This could refer to information hiding, or to controlling the data flow, or to different levels of privilege. Since information hiding is a general software engineering principle not specific to security, and since data flow control will be treated by its own rule (C4/HL7, see below), I suggest to concentrate on the privilege aspect, as this is what isolation with respect to security ultimately boils down to. Accordingly, rephrase VM5 as "Make components with differing privileges", a formulation that tells *how* the components should be ultimately different in the technical respect.

I find some problems with rule VM6 "Keep it simple", too. Certainly simplicity is a worthy goal for any software design; however, there is no specific relevance to security when expressed in such a general form — complexity invites errors in any respect, be it functional errors or security errors. Therefore I suggest to refine this rule into three rules calling for simplicity in function, programming interface, and user interface, each one with respect to security: "Don't be more general than necessary" (a new rule C10),"Minimize your attack surface" (one of Howard/LeBlanc's rules, see HL2 below), and "Make the secure way the easy way" (another new rule C13). (The rationale of all newly introduced rules will be explained in the respective section below.)

I find rule VM7, "Promote privacy" not optimally clear since—as it becomes apparent in its discussion in [1]— it refers to both personal information about humans and to guarding information about systems which could help launching attacks on those systems. However, these two issues are rather different in their level of abstraction, practical implications, and technical treatment. The latter issue of caution in publicizing system information is quite adequately covered by a specific rule C18 "Don't reveal more than necessary" applying to systems only. In contrast to that, privacy—that is, the confidentiality of personal information—is a much more abstract and comprehensive issue with manifold implications (e.g. legal ones). However, at the technical level we are dealing with here, there is no difference regarding the software measures used to protect personal information compared to protecting other confidential data such as passwords. Accordingly, I don't see a specific rule for privacy in this context.

Rule VM8 "Remember that hiding secrets is hard" again mixes two unrelated aspects when discussed in [1]: That of hiding secrets buried in a piece of software from the users of that software trying to reverse-engineer it, and that of insider attacks, in particular on networks. The latter aspect has little to do with software security, and I therefore suggest to concentrate on the first aspect only, expressed more clearly as rule C12, "Be careful when storing secrets".

Rule VM9 "Be reluctant to trust" again mixes two aspects of which only one refers to building software: On the one hand, reluctance to trust other software components, such as components of a distributed application running on untrusted machines, or third-party software whose security properties cannot be verified;

on the other hand, reluctance to trust other people's claims. The former aspect, however, is already covered by C4 (HL7) "Validate all data from lower-privileged sources".

Rule VM10, "Use your community resources" is certainly good advice, but does not seem to exhibit any aspects specific to building secure software.

3. Rules proposed by Howard and LeBlanc

The first rule proposed in "Writing Secure Code", "Learn from mistakes", is very widely applicable, far beyond secure software development, and although the authors elaborate somewhat on useful instantiations of this rule in development, I still find the advised procedures very useful (such as determining the root cause of an error and adding a test to prevent any future incarnation of the same error), but not really specific to security errors, and so my suggestion is once more to leave out such general advice.

Table 2. Rules by Howard and LeBlanc

HL1	Learn from mistakes
HL2	Minimize your attack surface
HL3	Use defense in depth
HL4	Use least privilege
HL5	Employ secure defaults
HL6	Remember that backward compatibility will always give you grief
HL7	Assume external systems are insecure
HL8	Plan on failure
HL9	Fail to a secure mode
HL10	Remember that security features != secure features
HL11	Never depend on security through obscurity alone
HL12	Don't mix code and data
HL13	Fix security issues correctly

I suggest only minor formulation improvements, if any, for rules HL2 to HL6, which should be obvious when comparing them with the consolidated rules formulations. HL2 recommends to limit the number of different means of interaction with an application. HL6 refers to the problem of retaining vulnerabilities from earlier versions of an application in newer ones for the sake of compatibility.

I find the advice of HL7 "Assume external systems are insecure" should be more comprehensive and should be expressed in a constructive form: Avoid the conception evoked by the word "external system" that only input coming from foreign code is suspect, since data from your own code executing on a foreign machine is equally suspect. Actually, all data flowing from a lower-privileged to a higher-privileged component must be validated, whether as input data (e.g., parameters) or as output data (e.g. return values, replies). Finally, to include the human user in the set of untrusted "components" (a word strongly associated with software or hardware), "source" rather than "component" is used in the final formulation of rule C4: "Validate all data from lower-privileged sources".

Rule HL8, "Plan on failure" is another instance of sensible, but not security-specific advice — any software should catch and handle failures. Rules HL9, HL11, HL12 are basically fine as they are and go into the consolidated set unchanged, again except for some reformulations I find clearer. HL11 warns against the fallacy that secret application designs will remain secret (a hope that was disappointed many times in history). The reasoning behind HL12 is that code can be much easier abused for attacks than "passive" data, so reduced security must be weighted against the increased versatility of code.

Rule HL10 "Remember that security features != secure features" means that one should not get a false feeling of security only from unspecifically adding some "security features" such as encryption to the target application. On closer analysis, it turns out that this rule is nothing but a negative formulation of rule VM1 "Secure the weakest link" (meaning that security measures should be applied at places where most needed, not where easiest to apply) which appears as C2 in the consolidated rule set. Rule HL13 "Fix security issues correctly" is a final instance of a very sensible advice which is, however, not specific to security and therefore omitted from the consolidated rule set.

4. The consolidated rule set

The consolidated rule set consists of rules from [1] and [2], often rephrased as discussed above, plus some additional rules I would like to suggest. Only those additional suggestions will be explained any further now. None of the advice contained in the two earlier rule sets should be completely new to a seasoned professional, and the same holds for the additional rules suggested next. However, it seems useful to summarize all rules in an explicit, constructive, strictly relevant, and non-redundant form.

Table 3. The consolidated rule set

C1	Assess your threats
C2	Secure the weakest link, not the easiest or most obvious one
C3	Secure the weakest link, not the easiest or most obvious one
C4	Validate all data from lower-privileged sources
C5	Beware of components with conflicting security assumptions
C6	Use several layers of defence
C7	Minimize your attack surface
C8	Use the least possible privilege
C9	Stay secure in case of failures
C10	Don't be more general than necessary
C11	Employ secure defaults
C12	Be careful when storing secrets
C13	Make the secure way the easy way
C14	Beware of backward compatibility
C15	Don't depend on an attacker's ignorance
C16	Recognize and answer attacks
C17	Separate code and data
C18	Don't reveal more than necessary
C19	Use only publicly scrutinized cryptography
C20	Use a truly random source to create secrets

The ordering of the consolidated rules is roughly from the more general or abstract to the more specific and concrete advice — that is, the ordering does not imply any temporal sequence of application, or any ranking of importance.

Rule C1 "Assess your threats" lays the ground for any security-aware development — systematically determine the potential threats of the target application. This is certainly not a new idea and is even somewhat out of line with the other rules, which apply to the design and implementation phases of software development, while "Assess your threats" applies to the analysis phase. However, I find it so fundamental to secure software development that I cannot help but include it in such a vademecum for the secure software developer. [2] contains still one of the best discussions of such "threat modeling" that I am aware of.

Rule C5 "Beware of components with conflicting security assumptions" seems in order because software components usually make (often unwittingly) some assumptions about their context of use that would breach security if violated, while the component is perfectly secure in the context it was intended for. If such a component is later combined with another component with different implicit assumptions, their combined effect may result in an unexpected security breach. Here is a historic example of such an unfortunate combination: One component is a generator for 32 bit random numbers that produces numbers with only the upper 24 bits being truly random; the implicit assumption here seems that *all* 32 bits of output would be used, and that 24 bits of true randomness would be sufficient in the intended context of use. The second component now uses the first one to generate password strings of many bytes length by chaining the output of repeated calls to the first component—unfortunately, it uses only the lower 8 bits of the "random" output, since it is a little bit easier to generate byte strings of any length this way:

```
for (i = 0; i < length; i++)
    password[i] = random() & 0xFF;
```

The combined effect was thus that the generated password strings are not random, but easily predictable[1]. It is a general problem with component-based systems that the implicit assumptions of components are often not fully documented, and this is especially true for security-relevant assumptions.

1. This vulnerability was contained in the generation of "magic cookies" in the X-Window system on older variants of Unix.

Rule C10, "Don't be more general than necessary"advises against offering more general mechanisms than strictly required — the more and sharper tools you carry with you, the easier they can be used against you. Examples include using a full-fledged programming language to add a macro facility to a relatively simple application, or starting a subprocess through the highly implicit and versatile system() function on Unix systems. Such dangerous generality is often the result of saving effort (if the general mechanism is already available, whereas a specific one would need to be built from scratch), or of a good-faith effort of "provident programming" to incorporate functionality "just in case", or of striving for simplicity, since the general mechanism may actually be the simpler one as well (note that this aspect discerns rule C10 from the old "Keep it simple" advice).

Rule C13, "Make the secure way the easy way", is derived from the observation that security mechanisms should be easy to use, or users will bypass them (obviously, here is a conflict with C11 "Employ secure defaults"). For example, the protected objects should be well-understandable to the users, and granting and revoking privileges on objects should be explicit and (ideally) reversible.

Rule C16 "Recognize and answer attacks" advises not to give away the option of recognizing obvious patterns of attacks and blocking them where possible — the application could react on observations such as repeatedly failing access attempts or overly long and strange input data by temporarily blocking accesses from the suspicious source.

Rule C19 "Use only publicly scrutinized cryptography" seems in order when considering the long history of faulty encryption algorithms and cryptographic protocols. Since naive attempts at creating proprietary cryptography happen again and again, an explicit advice against this does not appear redundant.

Rule C20 "Use a truly random source to create secrets" may seem like a technical detail at first glance. However, most computer security ultimately builds on cryptography, and cryptographic keys and secrets ultimately depend on obtaining truly random data. Therefore following this rule, while applying to only a tiny fraction of an application's code, is absolutely crucial. Nevertheless, history is full of examples for badly chosen "random" numbers used to generate cryptographic keys.

Finally, I do not want to let it go unmentioned that before [1] and [2], there has been a much earlier paper by Saltzer and Schroeder [3] , which both of the former presumably have drawn upon. The latter had suggested eight "design principles" of "information protection", most of which have found their way into the various rule sets discussed so far: Economy of mechanism (asking for general simplicity, like VM6), fail-safe defaults (HL9/C9 — *not* HL5/C11!), complete mediation (of all accesses — goes without saying today), open design (C15), separation of privilege (requiring two principals for a sensitive action, similar to the "four-eyes principle": *not* C3, but an infrequent, special case of C6), least privilege (VM4/HL4/C8), least common mechanism (C7/HL2, C10), and psychological acceptability (C13).

5. Concluding remarks

Rules or principles as discussed in [1, 2, 3] and here have been known in the (software) security research community for quite a while now, but have not really made their way into the consciousness of the general software developer. Part of the reason for this is certainly that such rules are only guidelines to consider, but not recipes to follow in order to arrive at a secure application. Some rules can contradict each other, and many rules contradict other goals of software development such as performance, functionality, simplicity, usability, interoperability, and ease of development. While developing secure applications remains an engineering challenge and may even be infeasible depending on one's definition, such rules of thumb could nevertheless prevent many security programming errors, provided that software developers understand how to map them to concrete programming concepts of their respective programming platform and application domain. While methods how to perform this mapping are out of the scope of this paper, the rules consolidation in concept and wording suggested in this paper provides a moderate step on this long way to go. At any rate, compared to the "old-school" approach of checking software against lists of known vulnerabilities from the past, following general principles of secure software design such as those presented here will raise the bar even for many attacks unknown today. What remains to be done, of course, is the mapping to a concrete problem and a concrete programming platform: perhaps in the form of more specific "rules" (or whatever form such advice would take then) for specific APIs (e.g. an operating system, or some middleware), and for specific application domains. The concept of "security patterns" [4, 5] is a first step in this direction, defining software building blocks to solve various security requirements.

6. References

[1] John Viega, Gary McGraw, *Building Secure Software* 492 pp., Addison-Wesley, 2002 http://www.buildingsecuresoftware.com

[2] Michael Howard, David LeBlanc, *Writing Secure Code*, 2nd ed., 770 pp., Microsoft Press, 2003 http://www.microsoft.com/mspress/books/toc/5957.asp

[3] Jerome Saltzer, Michael Schroeder, *The Protection of Information in Computer Systems,* Proceedings of the IEEE, 63(9), pp. 1278-1308, September 1975.

[4] http://www.securitypatterns.org *A collections of links to many papers dealing with security patterns*

[5] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, Peter Sommerlad, *Security Patterns: Integrating Security and Systems Engineering,* 600 pp., J. Wiley & Sons, 2006