Appearing in: *Proceedings of the 8th IASTED International Conference on Software Engineering and Applications (SEA 2004)*, pages 269–274, Cambridge, MA, November, 2004.

Insecurity by Contract

Phillip L. Nico
Department of Computer Science
California Polytechnic State University
San Luis Obispo, CA 93407
pnico@acm.org

Clark S. Turner
Department of Computer Science
California Polytechnic State University
San Luis Obispo, CA 93407
csturner@calpoly.edu

Kimberly Knowles Nico
Computer Security Consultant
Morro Bay, CA 93442
Kimberly_Nico@yahoo.com

Abstract

Design by Contract is a design methodology that promotes software reliability and reusability by requiring each component module to have a well-specified interface and leaves a module's behavior undefined if its requirements are not met. The DBC methodology may well lead to software with fewer overall faults, but its lack of interface validation encourages the class of failures that, through error propagation, results in violation of security policy. In this paper we explore the interaction of the tenets of the design by contract methodology with the requirements of system security.

Keywords: Software Methodologies, Security, Reliability, Design by Contract

1 Introduction

Over the past decade, the principles of Design by Contract™ (DBC) have been promoted as a way to develop simple, correct software by clearly defining who is responsible for each component of that software. The claim is that by following these design principles the software produced will be more reliable,

Reliability, in this case, is defined in terms of *correctness*—the ability of software to perform its specified tasks—and *robustness*—the ability of software to respond appropriately to abnormal conditions [9, 11]. Software fails to be reliable through *faults*—departures from a system's intended behavior—which are caused by *defects* in the system, which are, in turn, caused by *errors* either of specification or of implementation [11].

We agree that the design by contract philosophy may indeed lead to software with fewer overall defects [14], but are concerned that this same philosophy—requiring interfaces to be precisely specified but allowing modules' behavior to be undefined if the specification is not met—may lead to a false sense of security.

When it comes to failures that lead to security violations—failures resulting from intentional attack—all faults are not created equal. What is of importance is not the number of potential faults, but how these errors in execution can propagate to cause damage—not in the presence of accident, but in presence of an intelligent and malicious adversary [13].

One principle of design by contract is that the behavior of a module whose input requirements are not met is undefined. This means that a module need not validate its inputs, and, further, should not, because the caller has already promised to meet the module's preconditions, and the introduction of (ostensibly) redundant code to validate inputs simply increases the number of places where defects can occur. This *Non-Redundancy Principle* and its interaction with system security is the focus of this work.

The rest of this paper is organized as follows. First, the following section will review the principles of Design by Contract, then section 3 will review the requirements of designing secure systems. Following that, section 4 looks at ways in which contracts can be breached to violate security principles. Finally, section 5 reflects on error responses, and section 6 draws some conclusions.

2 Design by Contract

Design by Contract [9, 11] is a software development methodology comprising two major principles. The first principle is that every component of the system must have a well-defined interface, a *contract,* defining its obligations and benefits in terms of preconditions, postconditions, and consistency requirements (invariants). A module whose inputs satisfy its preconditions is obligated to produce an output satisfying both its postconditions and its given invariants. A module whose inputs violate its preconditions is not obliged to meet its postconditions and is, in fact, free to do anything at all; its behavior is undefined.

Consider the example service contract, shown in Figure 1, used by Meyer [9] and revisited below in section 4.3.

The contract is one for a postal service, laid out in terms of *obligations* and *benefits.* A party meeting his obligations is entitled to the stated benefits from the other party. Generally these are reciprocals of one another as each party is obliged to provide the benefits to the other.

	Obligations	Benefits
Sender	• Provide a parcel within size and weight limits. • Pay postage.	• Parcel is delivered to addressee in a reasonable time.
Post Office	• Deliver parcel to addressee in a reasonable time.	• Need not deal with too large, too heavy, packages. • Gets paid.

Figure 1. An example contract for a postal service

2.1 Well Specified Interfaces

The first principle of DBC is that of the well-defined interface. Each module of a system must fully specify its pre-, post-, and invariant conditions. Furthermore, these specifications must be made explicit in the program itself via assertions. These assertions serve to inform the module's clients of their obligations and responsibilities, facilitating both correctness and re-use. Some languages, in particular Eiffel [10], provide embedded mechanisms for assertion checking that can be used during development and testing, but are customarily disabled when the final product is delivered.

We are whole-heartedly in favor of well-specified systems. DBC's insistence that every module be well-specified may indeed increase reliability, reusability, and security, to the extent that we can rely on specifications being complete, consistent, and correct.

2.2 Non-Redundancy Principle

The second principle of DBC, the Non-Redundancy Principle, [9, 11] states that modules should not verify their preconditions and that consumers should not verify modules' post-conditions. The arguments for this are: (1) that introducing redundant checks unnecessarily complicates the code, leading to more overall faults, and (2) that programmers will write better quality code if they know that they alone are responsible for its correctness because nobody else will be checking it.

Even if it is true that eliminating error checking code leads to fewer implementation defects, failure to test for error conditions will allow those errors that do exist to propagate unchecked. The choice to intentionally eliminate checks on error propagation is, on its face, disturbing to the security-conscious developer.

3 Designing for Security

Full specification and adherence to that specification is the way to create systems that function correctly under expected conditions. High assurance computing is the science of creating systems that do no harm when faced with unexpected conditions.

While security is sometimes a difficult concept to define, fundamentally it rests on three principles: confidentiality, integrity, and availability [4]. *Confidentiality* refers to a system's ability not to divulge information it shouldn't, *integrity* refers to a system's ability to produce true information, and *availability* refers to a system's ability to function for the system's intended users when required. Together, these three properties ensure that the system will do what it is specified to do (and nothing else) under both expected and unexpected conditions.

Security is a system property. When it comes to a security violation, it doesn't matter whose fault it is; the system as a whole has failed.

3.1 Types of threats

When we consider how a system will function in unexpected circumstances we need to consider how those circumstances would arise. In general, threats to a system can be broken up into several categories. In terms of DBC, these represent ways in which a contract can be violated. The threat categories are:

External attack: An attacker acting outside the system could violate an unverified system assumption and cause inputs to violate a module's contract.

Programmer errors: A programmer implementing one module in the system could unintentionally violate the input preconditions of another module. This is different from an external attack because the offending data is coming from inside the security perimeter. It may be possible for an attacker to act in ways that completely satisfy external interface requirements to trigger the internal violation.

Physical faults: The system's hardware may fail in such a way as to allow inputs in violation of a module's contract, or a value could change after it has been validated. This change could either be accidental or intentionally induced [5].

Insider misuse: A programmer or privileged user could use that privilege to intentionally violate a module's contract. In effect, this is similar to a programmer error, except that it is much less likely to be discovered during development and testing because the defect is both intentional and likely disguised.

Any of these threats could cause a contract to be violated and thus lead modules into the states where their postconditions are undefined. In addition, it is possible for several of the above mechanisms to be combined in an attack where several faults that are individually benign can be combined to violate a contract. In order to uphold the principles of confidentiality, integrity, and availability it is necessary to design systems first to limit both the number of such failures and the damage that can be done by those that exist.

3.2 Design Principles

In order to defend against the attacks described in the previous section, it is necessary to anticipate them in the system's design and limit exposure. Design principles intended to promote system security have been found in the literature for decades. In 1975 Saltzer and Schroeder [12] proposed a set of eight design principles intended to prevent security violations.

In practice, however, it has proved extremely difficult to both design and implement systems with no security vulnerabilities. The inherent complexity of software and the fallibility of its designers and implementors conspire to leave holes in the castle walls.

3.3 Defense in Depth

"No battle plan ever survived contact with the enemy."
— Field Marshall Helmuth Carl Bernard von Moltke

Experience has shown that no matter how carefully specified and implemented a system is, defects will still remain within it [8, 3]. Given this track record, we must assume that any piece of software will have some flaws remaining in it and respond accordingly.

Defense in Depth [1, 2] is the strategic principle that a series of interlocking defensive positions should be used both to support each other and to limit the scope the damage should one of the defensive positions fail. In the arena of military operations, where the strategy was originally defined, this can mean castles with moats surrounding walls, parapets overlooking the gates to protect them from attack, inner walls to which defenders can fall back when the outer walls fail, etc.

The model of physical defenses does not apply directly to software, but the principle still does. It is foolish for a system, no matter how well specified, to assume that its external perimeter is inviolable and have no contingency plan for what to do when it fails.

Note that failure-aware design does not mean that the system will necessarily be able to recover from the failure. A function that performs error checking and produces an error result is the antithesis of reliability. Once a module's precondition has been violated, it may very well be unable to fulfill its function[1]. Thus, the module is forced to fail, but all failures are not equal. By checking for the failure state and responding appropriately it may be able to mitigate the damage. When failure is the only option, choosing the nature of that failure can be of critical importance.

4 Breach of Contract

An attacker operates outside the bounds of the contractual framework. He is either unwilling or unable to satisfy the obligations of a contract that would achieve his goal. Furthermore, the benefit he seeks may be completely outside the bounds of the system in question. The attacker's approach is to violate a system's specification intentionally in order to cause it to do something outside of its intended behavior.

The attacker's breach of contract could arise from many possible sources. In this section we look first at the nature of these failings, then how they may be exploited, and finally an example of an attack using the post office example of section 2.

4.1 The Nature of Defects

It is generally accepted that any sufficiently complex piece of software will have within it some defects. These defects may either be *observable*, affecting the functionality of the system in the normal course of its use, or *latent*, not related to normal functionality and thus hidden during normal use. Reducing the number of defects in a piece of code generally leads to more reliable code. One of the motivations of DBC's non-redundancy principle is that adding code to modules to validate contractual assertions should be avoided because this practice increases overall code size and complexity and thereby increases the total number of defects in a given system [9]. Fewer defects should mean more reliable software. This point is well taken. What is not considered here, however, is that all faults are not created equal.

An automated teller machine might have a routine that controls the money dispenser, and it may have a contract that states that whoever calls it must check the user's balance before calling `dispense()`. Now consider the following two error cases:

1. `Dispense()` has code in it to independently verify the user's balance and there is an error in that code causing it always to fail. The result of this is that nobody will get any money. This is unfortunate, but remediable by walking into the bank to speak to a real teller. If there is an error that always determines the balance to be sufficient, that error is mostly harmless because the caller should have already checked.

2. `Dispense()` does not check, and there is a similar error in the caller's balance checking code that always determines the balance to be correct. Here, all withdrawal requests, valid or not, will be approved, and there is no remedy because the cash is already gone and untraceable.

Although this example is of a functional failure, the faults that lead to security violations are not necessary functional. That is, there is no relationship between visible effects when the system is used properly and security bugs, therefore there is no incentive to find these flaws because there are no bug reports [13].

[1] In fact, if it is able to fulfill its function in the presence of a violated precondition, the precondition must have been unnecessary.

Faults that could cause contract violations, and thereby security violations, fall into several categories:

Errors in specification: An error in the system's specification could lead to an erroneous contract that would cause a module to malfunction itself or to accept an input that would cause it to violate another module's preconditions.

In one sense this is not a defect at all, since the specification is the best definition of what the system is supposed to do. With a less narrow definition of defect, however, this is clearly an error in the system.

Defects in the code: An error in implementation can cause a module to violate either its own or another module's contractual obligations under certain circumstances. These errors may or may not manifest themselves in the normal course of use.

Insider abuse: An insider may insert code into a module that will either open a back door for use at a later date or that will intentionally violate another module's preconditions in order to produce a desired effect.

Environmental failure: Even if all of the software components are behaving responsibly it is possible for the situation to change between time of check and time of use. Consider situation described by Govindavajhala and Appel [5] where they demonstrated their ability to take control of a Java virtual machine, with all of its internal reference consistency checks, by inducing memory errors with a heatlamp!

Any of the above faults could lead to a situation where a module's preconditions are violated. In terms of DBC, this would free the module from its obligation to fulfill its postcondition and leave its behavior undefined. That undefined behavior provides an opening for attackers.

4.2 Undefined does not mean Unpredictable

Just because a system's behavior is *undefined* does not mean it is *unpredictable.* Computers are largely deterministic machines. Regardless of the specification leaving a module's response to a given input undefined, in general, a module's response to a given input is deterministic.

To illustrate this, consider the all-too-common buffer overflow attack. In this attack an attacker passes a program a string that is longer than the buffer intended to hold it and overwrites the neighboring memory. Typically the attack is perpetrated against a local array allocated on the program's runtime stack. The given string overflows the end of the buffer, overwriting the function's return address, and causing it to return to code embedded in the string itself rather than to the original caller. To effect this attack it is necessary for the attacker to know the hardware architecture, the size and layout of local variables on the stack, and the memory location of the stack itself in order to correctly determine the malicious return address. None of these things are typically part of the specification of the program, or, indeed, that of the language or operating system either. All of these things, however, do tend to be predictable, and that is what opens the door to the attack.

In the following example we look at what may happen when a system's behavior is undefined but can be anticipated.

4.3 A Bomb in the Mailbox

Meyer's example in [9] and the variant of it presented in section 2 involve a contract with the post office to carry a parcel. When applied to security, however, the model goes wrong in that an attacker is not interested in the offered product—conveying a parcel from here to there; he is interested in something else, a side-effect.

Consider the case where an attacker can place a bomb that does not conform to either the size or weight requirements in a mailbox without putting postage on it. Under the rules of design by contract, the attacker is permitted to do this and, by the non-redundancy principle, the post office is forbidden to check. The bomb is, however, in violation of the contract, and therefore the postal service is under no obligation to transport it anywhere in particular, but the attacker's goal isn't transportation, it is the destruction of other mail in the box.

In this example the attacker has been able to accomplish two things without violating the system's design principles:

1. He has accomplished the destruction of the mailbox preventing the post office from entering into any new delivery contracts, and,

2. He has destroyed all the mail in the box causing the post office to violate pre-existing contracts with those senders.

4.4 Analysis

DBC demonstrates a strong dependence on its contracts. We are interested in whether this involves necessary tradeoffs with some known security concerns. From the perspective of our adversary (the "black hat" hacker who seeks to control our system in some way), we wonder *whether the contract is sufficient to prevent a destructive exploit.* This central question may be broken down into some more basic security issues:

1. Is there a risk of incorrect, incomplete or inconsistent *allocation* (by system architect), or *derivation* (by software designer) of system security requirements to individual module contracts?

2. Is there a risk of incorrect, incomplete or inconsistent *implementation* of a given contract in code?

3. What is the risk that security may be compromised by design of the contracts themselves?

4.4.1 Specification Concerns

The state of software specification has not reached the point of high confidence in correctness, completeness and consistency, at least for software systems of nontrivial scope [7]. Thus, the security conscious programmer is not inclined to accept that preconditions or invariants should not be checked (when a call to his module is made) without some suspicion or defensive design measures.[2] Even if system specifications are correct, complete and consistent, there are two activities critical to module contracts that involve risk of error: allocation of security specifications to individual software modules as contracts, and derivation of the higher level security specifications down to individual module contracts. Again, the security conscious programmer will remain suspicious to protect the security of critical functions and data. Acceptance of a random call without a serious inquiry about whether preconditions have been met would be a mistake.

4.4.2 Implementation Concerns

Supposing that the first issues are apparently resolved, we have a set of contracts that specify a set of modules for the programmers to implement. The implementation process is not without its risk of error. Certainly, at the module level, the use of testing and proof techniques may prove useful to expose and correct defects. Testing, though indispensable, can expose defects but never show their absence [6]. This is where proof techniques may be used to fill the gap, and DBC embraces the possibility through use of pre- and post- and invariant conditions associated with the modules. Assertions may be used to check that no violations occur during runtime, but their efficacy depends on two things: 1) that assertions are "turned on" (not always true); and, 2) the (unanticipated) problematic state that violates the assertion must be reached. Thus, the exposure of critical errors for DBC works similarly to testing: there is not enough time or resources (in nontrivial cases) to completely cover the state space and expose all critical defects. The security conscious programmer must remain skeptical and defensive.

4.4.3 Inside Jobs

The last issue we examine is that of the malicious insider. It is a particularly difficult issue, to be sure, but not uncommon. Can the malicious insider design the contract to facilitate security violations and deter efforts to discover them? Of course, DBC does not address this issue directly. However, the non-redundancy principle appears to discourage checking of another module's call. If the other module is designed to violate system security by design of its own (and other) contracts, this philosophy of trust is misplaced. The security conscious programmer must look to defensive programming if such problems are anticipated at any level.

[2]Since security is an "emergent" property of a system, we realize that the system designer does indeed bear the lion's share of responsibility for proper specifications. However, security experience shows that security is a distributed responsibility where the system is to be protected regardless of the assignment of "fault" or responsibility (which *can* be useful during debugging or assessing liability).

5 Defensive Programming vs. Exception Handling

While we laud the goals of Design by Contract—perfection in both specification and implementation—we believe these goals to be unattainable.

Defensive programming does not—and cannot—prevent or recover from errors of either specification or implementation. Once an erroneous state has been reached where the specification does not specify behavior, the system has been damaged and there is no way for it to guess at what the designers would have wanted it to do. What it can do, however, is to limit the scope of damage caused by that failure. By preventing errors from propagating we hope to prevent the sorts of errors that allow confidentiality, integrity, or availability to be compromised.

Exception handling, as defined by the DBC community, still focuses on meeting the specification. Meyer [9] defines an exception as "a runtime event that may cause a routine to fail" where failure is defined a routine terminating in a state that violates its contract. A routine whose preconditions are not met is not obligated to honor its contract; therefore the undefined behavior of such a routine is not considered exceptional.

In deference to the DBC philosophy, we recognize that a tradeoff is being made to accept complexity, and the larger number of defects inherent in that complexity, in exchange for damage control. Certainly code should be made as simple as is practical, but no simpler.

6 Conclusion

The design by contract methodology has much to contribute toward the development of software that will function well in its intended environment. By requiring every module to be well-specified, it reduces the chance of error in both design and implementation. Even the non-redundancy principle, by not validating pre- and post-conditions, reduces the likelihood of defects in the final delivered product simply by reducing the amount of code to be produced. At the same time, however, not validating these conditions allows failures arising from the remaining defects to propagate silently through the system increasing the expected damage from those faults.

In any sufficiently complex piece of software we cannot guarantee that all contingencies will be addressed in a specification, nor can we guarantee that all modules will properly meet their specifications under all conditions. If software failures are a given, what remains to the system is the opportunity to choose the manner in which it fails. It can stand on the principle of a valid contract and refuse to bend, or it can take action to renegotiate that contract and limit the scope of the damage.

Caveat Emptor.

References

[1] *Information Assurance through Defense-in-Depth*, Feb. 2002. Directorate for Command, Control, Communications and Computer Systems, U.S. Department of Defense Joint Staff.

[2] BASS, T., AND ROBICHAUX, R. Defense-in-depth revisited: Qualitative risk analysis methodology for complex networkcentric operations. In *Proceedings of IEEE MILCOM* (Oct. 2001), pp. 28–31.

[3] BISHOP, M. Vulnerabilities analysis. In *Proceedings of the Second International Symposium on Recent Advances in Intrusion Detection (RAID'99)* (Sept. 1999), pp. 125–136.

[4] BISHOP, M. *Computer Security: Art and Science*. Addison Wesley, 2003.

[5] GOVINDAVAJHALA, S., AND APPEL, A. W. Using memory errors to attack a virtual machine. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (May 2003), pp. 154–165.

[6] KANER, C. The impossibility of complete testing. *Software QA 4*, 4 (1997), 28.

[7] LEVESON, N. *Safeware: System Safety and Computers*. AddisonWesley, 1995.

[8] MCCONNELL, S. Gauging software readiness with defect tracking. *IEEE Software* (May/June 1997).

[9] MEYER, B. Applying "design by contract". *IEEE Computer 25*, 10 (1992), 40–51.

[10] MEYER, B. *Eiffel: the language*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1992.

[11] MEYER, B. *Object-Oriented Software Construction*, second ed. Prentice Hall, Englewood Cliffs, N.J., 1997.

[12] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (Sept. 1975), 1278–1308.

[13] SCHNEIER, B. Why computers are insecure. *Crypto-Gram Newsletter* (Nov. 1999).

[14] TANTIVONGSATHAPORN, J. Design by contract—comparison to traditional practice with case studies. Master's thesis, California Polytechnic State University, San Luis Obispo, California, June 2004.