

Refactored Cognitive Dimensions and Secure Development – A Discussion Paper

Luke Church
Computer Laboratory, William Gates Building
University of Cambridge, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK
luke@church.name

Abstract

This discussion paper begins the process of using Refactored Cognitive Dimensions to build a cognitive model of some of the difficulties associated with secure coding. Some difficulties with both managed and unmanaged languages are considered. Possible applications are briefly discussed.

Introduction

Many security failures occur in the implementation of systems [1]. Traditionally these failures have been considered as technical problems. This paper considers some of the security concerns in modern software development from a Refactored Cognitive Dimensions perspective [2]. For brevity it is by no means exhaustive, but aims to provide some insights into the validity of the modelling approach.

This paper's domain of discourse is limited to code level security issues.

All security bugs are hidden dependencies

At the most simplistic level all security bugs can be considered to be hidden dependencies; some combination of input parameters can be chosen by an attacker to cause the system to behave in an unexpected and typically harmful manner. Hence the bug can be considered to be a hidden dependency of the behaviour of the system on its input. However this description could apply to any bug. To make progress some refinement is needed.

Semantic hidden dependencies

Hidden dependencies can be consumed and/or produced by a program. When a hidden dependency is consumed, a dependency that already existed within the system is opened to the user. For example a function may exhibit a dependency on the length of a string; this dependency is consumed by a program that allows the user to pass a string to the function without sanitising it properly. This contrasts to production where a new dependency is created, which isn't due to any underlying dependency in a library etc.

Common weaknesses

Semantic hidden dependencies have many direct effects on security. Amongst the most common security vulnerability is the buffer over-run attack. This is typically caused by allocating a fixed size buffer and copying more data into the buffer than it can hold. Frequently this happens due to the system failing to check the length of an input from a user. This can potentially allow an attacker to execute arbitrary code on the victim's machine.

This vulnerability can often be caused by misusing a library function, e.g. C's `strcpy`. This is due to the semantics of the function causing an unwanted behavioural dependency on its input data. This can also be considered to be a mis-match between the developer's view of the function's semantics (copy data from one string to another) and its actual semantics (copy a block of data from one memory location to another).

There is also a *premature commitment* element to this problem. The developer was forced to prematurely commit to the length of the buffer, before the length needed was known.

In some cases, premature commitment and hidden semantic dependencies can be combined. For example, in many languages, integers are implemented with fixed maximum values. If these values are exceeded they overflow from their maximum value to their minimum value.

This results in a developer having to prematurely commit to the maximum size of the number they will need. However the commitment is no longer explicitly declared, but is now implicitly due to the semantics of the language. It can therefore be viewed as a premature commitment to a hidden semantic dependency. This is also a semantics perception mismatch.

Hidden data dependencies

Hidden dependencies can be caused by data-flows. Typically the exploitable cases of these dependencies trace back to incorrectly sanitised user input. For example in a SQL insertion attack, the SQL statement that is executed against a database can be modified by the user inserting a carefully crafted string.

This can be viewed as a hidden semantic dependency on the semantics of SQL, however this only partially models the problem. The data-flow from the user to the execution of the SQL statement can be viewed as a further data hidden dependency, caused by directly trusting the users input.

This is an example of trust as a hidden data dependency. In a similar manner to semantic dependencies, not correctly modelling trust interactions can result in security vulnerabilities.

[In]Visibility through dependencies

Modern software projects have so many dependencies that they create a visibility problem. Tool support in various forms is now used to help to increase the visibility of such issues.

Some tools modify the code at compile time to change or eliminate some hidden dependencies (e.g. Microsoft Visual C++'s `/GS` option which inserts some buffer overflow checks [3]). Other tools are code analysers such as Microsoft's FxCop tool [4]. Whilst such tools are undoubtedly useful, they are not sufficient to solve the problem in themselves and can have usability problems due to the number of issues they report against a typical project.

Managed Languages – The Brave New World

Managed languages such as Java and C# have a slightly different set of security problems. They modify the dependencies and abstraction level by introducing a virtual machine that code is executed on. This gives numerous advantages and can reduce the mis-match of models between the developer and the compiler. Crucially strings in such languages may be of effectively arbitrary length and array bounds may be checked before access.

However the higher level of abstraction brings its own complexities, a couple of which are outlined below.

Can syntactic sugar cause your code to decay?

Syntactic sugar is added to many modern languages to make development sweeter. It involves adding a special syntax that is intended to ease development without affecting expressiveness.

Sweetened expressions may contain more sophisticated semantic meaning. This provides a number of advantages from a cognitive perspective, it decreases *cognitive viscosity*, by more closely aligning the language's operations to the developers' perceptions of these actions, it decreases *workstep* and it can increase the *visibility* of the programmers' intent.

However it can also introduce very subtle *hidden semantic dependencies*, especially as the syntactic sugar becomes more complex. For example, Java's generics allow the specialisation of a generic type to eliminate some casts:

Comparing the declaration and use of a list of elements of type `Cat` (Java psudeo-code)

Non-Generic	Generic
<pre>List cats = new ArrayList(); Cat cat1 = new Cat(); cats.add(cat1); //OK Dog dog1 = new Dog(); cats.add(dog1); //OK ... Cat cat = (Cat)cats.get(1); cat = (Cat)cats.get(2); //Runtime cast exception, it was a dog</pre>	<pre>List<Cat> cats = new ArrayList<Cat>(); Cat cat1 = new Cat(); cats.add(cat1); Dog dog1 = new Dog(); cats.add(dog1); //Compile time error ... Cat cat = cats.get(1);</pre>

The expected way of thinking about the generic collection is a typed 'Collection of cats' rather than a 'Collection of objects to which I have only added cats'. However due to complexities in the manner in which generics are implemented in Java it is sometimes possible to attack the internal representation and insert objects that aren't cats into a collection of cats, introducing a complex type-safety problem that only fails when the problematic item is read from the collection.

In this way generics are dangerous if an attacker can manipulate the data. The notation and typical use imply one behaviour, but due to a complex semantic issue the system can actually exhibit a different behaviour.

Security features increase cognitive viscosity

Frequently the security features of a platform are not used [3].

As with low privilege users on end-user systems a developer will have to do more work to enable their code to run with minimum permissions. Having to juggle security assertions, demands, impersonations etc. and a potentially complex security model generally results in *cognitive viscosity*. The developer's idea of a single task no longer aligns to that of the programming environment. A single task now involves not only the task itself but also the associated actions to acquire the necessary security permissions.

The response is to seek the path of least resistance. In a developers case this may well be demanding that their code runs with 'Full Trust', minimising any security management required, but removing any safety net that the runtime might have been able to provide.

Non-CDs Issues

There are some issues in the cognitive modelling of security that do not seem to fit easily into the cognitive dimensions framework.

Motivation

There is a motivation misalignment problem. Security is frequently considered to be a property of an application, not a primary feature. This may cause a series of problems, from managers not wishing to commit resources, to developers wanting to get on with 'developing the application' not writing security code, doing security reviews etc.

Inverted input-space problem

Typically when building an application, the space of the inputs that is considered is the space containing the inputs from a co-operative user. As the defect rate of software demonstrates, mapping this space to useful actions can be complex enough.

The security problem can be viewed as an additional requirement, that the input spaces from a hostile user are also mapped to appropriate actions. This requires thinking about all of the possible input space, even when it seemingly has no relevance to the task that the developer is concentrating on. This 'thinking of the universe outside the box' can be challenging.

Applications and extensions

A cognitive model of the difficulties faced during development of secure code would be useful for developers of new tools and languages to help to identify issues that occur at a cognitive level rather than a code level. Developers and code reviewers may benefit by being able to locate common problems faster and more reliably. Ultimately attackers may also benefit from an increased awareness of where mistakes are likely to be made.

Semantic hidden dependencies seem to be a particular problem area, suggesting that either eliminating them through further abstraction or making them visible might be helpful. The effect of abstraction levels seems complex, further work is needed in the development of the model in this area.

Conclusion

The Refactored Cognitive Dimensions framework can be used to build an approximate model of some of the cognitive complexities involved in writing secure code. There is some possibility that such a model may be useful in assisting with the design of language feature and tool support for secure development.

There seems to be much work still to be done in this area.

References

1. Anderson, R. (1993) Why Cryptosystems Fail, Proceedings of the 1st ACM conference on Computer and communications security
2. Green T. R. G, Blandford A. E., Church L. Roast C. R. What CDs did next (Pending)
3. Howard, M. LeBlanc, D. (2002) Writing Secure Code Second Edition. Microsoft Press
4. www.gotdotnet.com/team/fxcop/