A Practical Unsolvable Problem? Why Application Security is Hard: Some Ethico-logical Reflections

Introduction

It is well known - to the point of being a classroom example (Sedgewick and Wayne 2011) - that compiler design yields practical computationally unsolvable problems. For example, it is computationally unsolvable to detect all and only instances of unreachable code. It is also a long-lasting debate in the philosophy of mind (e.g., starting with Turing 1950) whether or not humans transcend these limitations. The latter debate often centers on the ability to prove theorems within formal systems and is certainly removed from the work-day experience of the working computer programmer, unlike the first topic I mentioned. However, neither of these areas have *direct* ethical impact. Nevertheless, there is a topic of concern to contemporary software developers which has ethical impact which is "multiplied" by it involving computationally unsolvable problems: this is the process (or activity) of application vulnerability assessment. As this topic is somewhat new to the philosophical literature (Douglas 2011 is one exception), it also bears introduction for the more philosophy-trained of the audience of the present paper; it is also where a lot of the action is these days in computing security (Sykora 2010) and hence also bears analysis here as it does not get enough computing professional discussion as well.

In this paper, I first introduce what application vulnerability assessment involves in broad outlines. In the second section I shall sketch a routine argument (familiar to some security professionals) that vulnerability assessment is a computationally unsolvable problem. In the third section I offer reflections (intended as areas of future investigation) on the seemingly banal result of the previous section and argue that it has profound consequences for the safe construction of (sufficiently complex?) software for any (publically available) purpose whatever. This is followed by a fourth section where I respond to critics, and the paper ends with a fifth section of brief summary conclusions.

Section 1 - What is Vulnerability Assessment?

In order to understand vulnerability assessment, first one must understand what software vulnerabilities are. Software vulnerabilities are actually remarkably difficult to characterize. To a first approximation, a vulnerability in a piece of software is a way to use it in a way that the software was not intended to be used (by its creators) such that something undesirable is then possible[1]. Note that this includes a value judgement immediately[2].

---

[1] The second clause is quite important. As pointed out by a comment on Douglas (2011) at IACAP 2011, sometimes unintended uses of a program are precisely the point, e.g., certain AI programs would be regarded as especially successful if they did something that their creators did not have in mind. It does not follow from this that all unintended "behaviours", even in an AI program, are worthwhile or free from security considerations.

[2] I believe this is unavoidable: I do not think one can talk about security or discuss artifact construction otherwise. However, I will not argue the point. See, e.g., Bunge (1998) for one reason why for the latter using his "rule based on law" principle; as for the former, I believe that the concept of security itself is value-laden. (Nothing stands on the precise characterization of the unavoidability; it seems safe enough as an assumption since it does very little work. If it is held by some that only some artefacts are subject to ethical evaluation, then the paper serves as a partial argument as to why at least some

To make this characterization more concrete, I will provide an example – a vulnerability type from database applications[3], called "SQL injection". I use this example because in its simplest form it is quite easy to understand - even for non-database developers, because injections in general are very common vulnerabilities and because in principle, SQL injection is completely remediable[4] (assuming one has source code to the application and a modern database access library).

Consider the following pseudocode:

```
input = GetUserInput
sqlString = "SELECT * FROM usertable WHERE username = '"
sqlString += input
sqlString += "'"
ExecSql (sqlString)
```

This pseudocode illustrates a quick-and-dirty way to get some rows back from a table matching the user's input on a column named "username". To see the problem, start with a request which will be a problem for this code. I call this the "`O'Brien`" test as it uses this common name to illustrate. Imagine, then, that GetUserInput returns[5] "`O'Brien`". Then one gets "`SELECT * FROM usertable WHERE username = 'O'Brien'`" as value for sqlString after the three assignment statements have executed. One can see the problem immediately. The ExecSql statement will fail with something like:

```
Incorrect syntax near 'Brien'.
Server: Msg 105, Level 15, State 1, Line 1
Unclosed quotation mark before the character string '
```

If this sort of message is presented to your users, a (not very, but somewhat) savvy viewer will know that your application is likely vulnerable to SQL injection[6]. Let's see how this works in the simplest case. Suppose then I arrange for `GetUserInput` to return "`' OR`

---

software does. Only those who deny artefacts (including in any extended sense, like via their users) are of any ethical import whatever would find this paper's ethical discussions useless. I hope such people would nevertheless find the epistemological lessons relevant.

[3] For more variety of examples across many different sorts of real software, an excellent resource is Dowd, McDonald and Schuh (2007).

[4] For those who remember the Sony Playstation Network exploit from a few years ago, the attack was reportedly done via SQL injection (no doubt of a more complicated nature). If true, this is nothing short of appalling as presumably Sony or their contractors built their logon mechanisms, etc. from scratch and should have known that SQL injection is completely preventable.

[5] Needless to say it doesn't matter if `GetUserInput` gets a result from a web form, a standalone application, a read from a file, etc.

[6] N.B.: Hiding the error, etc. is not enough as remediation. So-called "blind injection" where timing, subtle changes in the error handling display, other behaviour etc. gives the vulnerability away, is known.

`1=1; --`". (This can be, in principle, as simple as putting it in an input field or a query string on a web page.) Then the query is

```
SELECT * FROM usertable WHERE username = '' OR 1=1; --
```

This (by elementary logic) involves an always true WHERE clause; hence **all** rows will be returned from the table (-- simply comments any further clauses out; this is optional but useful in most situations for exploiting or analyzing the vulnerability). This could be information about other users, etc. A UNION keyword could combine the results with a system table, etc. The application is now compromised – it suffers from an application layer vulnerability. (How much so depends on other factors, of no concern to us presently.) Note that no firewall or simple intrusion detection system (IDS) can respond to this; nor can encryption (e.g.) SSL. Instead one needs a very clever IDS (which exist and raises other off-topic ethical questions), or, better, one should have built the application correctly in the first place.

I should also point out that this is the simplest form of SQL injection but all work by the principle of executing dynamic SQL statements without proper control of user supplied inputs as part of the query. This completes our discussion and illustration of "vulnerability".

Turn now to "assessment". This word is simply used in more or less its standard English sense, and so does not pose any problems *per se*, although understanding how to actually *perform* such an assessment is a matter of ingenuity and training (as well as the careful use of tools); I will not detail the process here.

Section 2 - Vulnerability Assessment is Computationally Unsolvable

Why is vulnerability assessment computationally unsolvable? Consider a program that contains code like the pseudocode from the previous section. One can represent this as some sort of complicated mathematical function mapping the user input on to various outputs. So, in the simplest case (like that above), one can imagine the vulnerability assessment as trying to show whether or not the function ever returns the string corresponding to the SQL Server error I quoted. By Rice's theorem (Kozen 1997), this is computationally unsolvable[7].

Section 3 – Brief Reflections

My reflections come in four categories. The first category concerns the responsibility of software developers to their bosses and managers. A second category is that of responsibility to the public. Third, I address the "personal integrity" considerations raised by our topic. Fourth, I allude to some potential areas of reflection within the philosophy of mind as well.

---

[7] Imagine coding the SQL Server error as the output "0" if necessary and so the problem is one of showing whether or not a recursive function (corresponding to `GetUserInput`, for example, or the whole procedure mentioned – it doesn't matter) ever returns the value "0". I also note in passing there are rare programming languages that are decidable. These have no "while" or equivalent constructs. Blaise, designed as a computational questionnaire implementation language for computer aided interviewing, for example, is such. However, its "sister language", Manipula (often used with Blaise programs), introduces "while" so in practice most programs written in with these tools use a computationally undecidable language.

Responsibility to Management

One is normally expected to be honest with one's management about the known problems with anything one builds. This is a banal truism of engineering ethics. However, software "construction" is not (yet?) regarded as a branch of engineering[8] at least to the same degree as other fields. Nevertheless, the truism would no doubt be part of many computing codes of ethics, as it is (for example) in the ACM code (ACM Council 1992, section 2.6 "Honor contracts, agreements and assigned responsibilities"). In the case of "unknownable unknowns" we are discussing, however, the honest answer to "is your program free of vulnerabilities?" will be in general "I don't know", which is, at first glance, "wishy-washy", even if it is true. (After all, this statement is also true if the developers simply did not investigate their program, the testers did not do their bit, etc.) A colleague (also a software developer) pointed out to me that she works under a "best effort" principle anyway; however, these considerations affect that as well; after all we do not know (and perhaps can never know) whether or not some system is sufficiently complex that vulnerability assessment exceeds human ability. Can we learn to state in precise terms the degree to which we have tested something?

Responsibility to Public

The problems associated with responsibility to management are magnified when dealing with the public, as then it is rather likely one will deal with people who have no knowledge of software construction and no real interest, time or motivation in learning[9]. Moreover, what should the legal liability be? Software warranties are notoriously noncommittal. In some jurisdictions the "no warranty" sort of remark has no legal standing (a license or use agreement cannot be used to sign away fundamental rights); but even in such places should we allow the public to bring suit or otherwise attempt to hold accountable the manufacturers of defective software if we know that the task of doing it perfectly is impossible? This shades into the philosophy of mind considerations and the "ought to implies can" principle debated in ethics. I shall deal with the first on its own later. The latter principle comes under attack due to determinism and the perennial "free will" debate. However one resolves that debate (or whether or not one has an opinion on it), even a libertarian free-willed agent may still be subject to undecidability. Gödel (see Wang 1987), for example, held that the limitative results would apply at any finite stage of an infinitely existing human soul's existence, but also thought that eventually each "level" of limitation would be transcended. This view is not widely shared, in any case the real contention here is that the programmers are unlikely to share the same metaphysical viewpoints as their users (if either party understands what is at stake!).

Personal Integrity

---

[8] In Canada, where the author is located, "engineer" is usually a protected title; one has to have appropriate academic and professional qualifications to use it (by analogy, say, with "physician"). However, there's no obligation to use an engineer in any way when constructing software, unlike, say, if one was building a public bridge, where one would presumably have to employ a civil or other appropriate engineer.

[9] Perhaps this point can be turned into an argument that the public should be educated (starting very young!) about the limitations of computing, but that is presently off topic.

A plausible feature of the personal integrity of a software developer (as with anyone) is the ability to take pride in the quality of one's work. If, however, the ultimate quality of one's work is unknowable, what then? Some people might find this result disheartening or the idea that one could never know when one has to stop working on something (assuming no external deadlines) very damaging to one's work-life balance. In practice, one can simply impose deadlines; fix one's schedule as one always has, etc. However, these impositions or personal decisions are then seemingly much more arbitrary.

Philosophy of Mind

As mentioned earlier, there has been close to 55 years or more debate on the consequences (if any) for the philosophy of mind from the limitative results proved by Turing etc. I suggest (and would like to address in further work if possible) that instead seemingly endless debate over whether or not formal systems results apply to brains, etc. that software development be taken as a concrete case. Similarly, rather than debates over AI, perhaps the debate could be refocused on the approaches of vulnerability assessment. For example, on the one hand, we know that automated tools are not nearly as good as reasonably well trained humans (Doupé *et al* 2010) On the other hand, the person claiming the human ingenuity must always be in principle capable of exceeding the machine tools has a good, clear case where "put up or shut up" can apply. On the "third hand", any human has a finite lifetime, so presumably can only analyze a finite amount of software and so investigations may prove fruitless after all[10].

Response to Critics

In this section I respond to four critics. I've called these objections "every case", "debugging", "despair" and "finite automaton vs. Turing machines". I deal with each in turn.

"Every Case"

A critic may rejoin to my reflections as follows. We can detect many vulnerabilities with tools and ingenuity - what makes you think that with greater tools and better developer education we might not improve to the point where the software complexity is such that we understand it completely, at least in principle? That is, we deal with only "solvable classes of the decision problem" (to re-use a famous book title).

There are two problems with this suggestion of my imagined critic, one practical, and one "theoretical". The practical consideration is that the complexity of computer programs is increasing drastically and the environment in which they run is becoming also increasingly

[10] This is where my earlier remarks about the idealizations might come into use. Is it more likely we'd have a well confirmed theory of the vulnerability assessment skills of humans then their prowess in deduction in a formal system? (Or any other mathematical context?) I don't know, but it might be worth considering, especially as it is a much more "concrete" activity and one with less traditional overtones (e.g., it would avoid debates over Platonism directly, presumably).

hard to control[11]. The second is that it is computationally undecidable what (mathematical) structures[12] are undecidable (Tarski *et al* 2010)!

"Debugging"

A critic might also ask, "but why bring in security into this? Isn't this just a special case of the computational unsolvability of general debugging?" The critic is correct that the argument presented in section 2, above, shows that debugging is computationally unsolvable. However, with increasingly complicated software, public facing applications (e.g., on web pages) the security risk is of special concern. This is for two principle reasons. One, as alluded to above in my response to the "every case" critic, the more complicated[13] a piece of software becomes, the more it seems likely we will never be able to convince ourselves it is sufficiently secure. In other words, it is one thing to have software which simply doesn't work; it is another to have software which (say on the public Internet) is dangerous to other users, their lives (virtual and real[14]) and data. The problem of "malware" is only increasing, and a lot of it gets into place because of vulnerabilities (and not simply failures in permissions or the like) in other applications. Two, a lot of developers would agree that long searches for how to explain slightly anomalous UIs (where they do not affect usability, etc.) are not worth while debugging tasks after a certain point, but would be hesitant to so categorize security problems. Both of these points are different ways to look at "not all bugs are the same risk or sort of problem". It is true that not all security risks are the same degree of severity either, but I do not know of any relationship between computational properties and security-related properties (other than the general one we have been discussing all along).

"Despair"

A critic says, "You've told us that this matter is pressing, important and generally an impossible situation. Is this a counsel of despair?" My answer comes in two parts: first, for the software developers and the public, an answer of realism (in the everyday, not any of the myriad philosophical senses). For the philosophically inclined, I also add another, more "optimistic" one. I discuss each in turn.

The "realism" answer is the dual to the answer of the "every case" objection. We can do a lot to improve our software, and we don't know how much we can do so. The dual to this is that we must get away from the idea that computers are like some sort of marvelous art medium

---

[11] One way this can be seen is to subscribe to the wix-users email mailing list. WiX is "Windows Installer XML", a free and open source toolkit for building Windows Installer (".MSI") packages. Even though this technology is Windows-only, one is overwhelmed at how complicated even installation of software is on this one platform.

[12] I assume (without proof) that it follows from this that determining which classes of computer programs exhibit undecidable features is itself undecidable.

[13] I do not have an appropriate metric of complexity to make this claim precise at the present time. The proverbial "lines of code" is very crude; however it is certainly true that the number of lines of code in software is increasing. Measures of slightly more current merit (e.g. cyclomatic complexity) are perhaps more relevant (since they explicitly measure branching) but I do not know of any directly security-related work in this area beyond the obvious that code with high cyclomatic complexity is hard to maintain in general.

[14] Certain sorts of web application vulnerabilities would, if present on your bank's pages, make it possible to have someone steal your very real money.

that we can bend to our will exactly. If this is depressing, so be it, but perhaps the Chomskyian point about being completely unconstrained has merit here too[15]. And this is optimistic, after a fashion.

"Finite Automata vs. Turing machines"

I regard this objection as the most philosophically interesting of the objections, but also the most difficult one to understand what practical consequences it has. My critic this time says, "Any real computer is actually in some way representable as a finite automaton: it has a finite (but enormous) number of states and events, and the Turing machine (hereafter, TM) model of having unbounded state is, strictly speaking, untrue of it. Hence, there are no real undecidable problems for it. Hence, you're tying yourself in knots because you're using an inappropriate idealization." I grant that the TM is an idealized model and that one uses different idealizations of a system for different purposes. However, one loses something important – programmability – without it. (As Douglas [2012] alludes, this is a fruitful way to understand the TM idealization.) This impact involves a fundamental change in how computing is conceived. While I am not completely opposed to revisionary computational philosophy (for example, as advocated by Douglas 2010), I nevertheless think that it should be done with great care and intellectual caution. I foresee three consequences, which I simply list to illustrate how drastic such a step would be.

1) The number of variables available to a program would not be potentially infinite any longer. While is certainly true that there is currently an upper limit to the size of arrays and such in any given programming language and target environment (e.g. Windows 32 bit .NET applications have a 2 gigabyte address space) the *syntax* of the language is not what prevents further memory usage and thus there would be seemingly ad hoc limitations imposed on it.
2) Similarly to point (1), how does one state the limitations in somewhat familiar programming idioms?
3) Not only memory allocation and variables would change, but uses of loops would change. I have mentioned that some languages are equivalent (suitably idealized) to primitive recursive functions but that these are very rare. It seems to be unknown whether or not there could be any useful intermediary classifications – i.e., between "for" and "while". They certainly would be very unfamiliar to programmers.

Summary and Conclusions

We have seen that a very real and pressing computational task is (perhaps) impossible in general and reflected on its practical and theoretical consequences. As a throw-away remark, I think the next steps have to be in analyzing the idealization of the Turing machine and seeing if one of more tractable character might allow us to get a better handle on vulnerability assessments and the reliability of software. Nevertheless, I think any such an analysis should involve a notion of "program", for the reasons canvassed. I realize this runs into cases of the "second level" undecidability mentioned, but I see no better way to proceed.

[15] Chomsky repeatedly says that it is a good thing that humans have real internal principles and mechanisms; for if they were completely unconstrained they could be manipulated arbitrarily by totalitarian social systems.

Bibliography

ACM Council. 1992. “ACM Code of Ethics and Professional Conduct”. Available at http://www.acm.org/about/code-of-ethics (Accessed December 12 2012).

Bunge, Mario. 1998. *Social Science Under Debate: A Philosophical Perspective*. Toronto: University of Toronto Press.

Douglas, Keith. 2010. “What Does a Computer Simulation Have to Reproduce? The Case of VMWare”. Unpublished presentation, IACAP 2010, Pittsburgh.

Douglas, Keith. 2011. “A Pseudoperipatetic Application Security Handbook for Virtuous Software”. Unpublished presentation, IACAP 2011, Aarhus.

Douglas, Keith. 2012. “Learning To Hypercompute? An Analysis of Siegelmann Networks”. Unpublished presentation, AISB-IACAP 2012, Birmingham.

Doupé, Adam., Cova, Marco. and Vigna, Giovanni. 2010. “Why Johnny Can’t Pentest: An Analysis of Black-Box Vulnerability Scanners”. In *Proceedings of the 7th international conference on Detection of intrusions and malware, and vulnerability assessment.* Berlin: Springer-Verlag.

Dowd, Mark, McDonald John and Schuh, Justin. 2007. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities.* New York: Addison-Wesley.

Kozen, Dexter. 1997. *Automata and Computability*. New York: Springer.

Sedgewick, Robert and Wayne, Kevin. 2011. *Introduction to Programming in Java*. Online at website: http://introcs.cs.princeton.edu/java/home/ . Accessed March 11, 2013.

Sykora, Boleslav. 2010. Unpublished remark during Learning Tree Course 940: Securing Web Applications, Services and Servers.

Tarski, Alfred. (with Mostowski, Andrzej and Robinison, Raphael.) 2010. *Undecidable Theories: Studies in Logic and the Foundations of Mathematics*. Mineola: Dover.

Turing, Alan. 1950. “Computing Machinery and Intelligence”. Mind Vol. 59, No. 236.

Wang, Hao. 1987. *Reflections on Kurt Gödel*. Cambridge: MIT Press.