The Dark Side of the Code (Transcript of Discussion)

Simon N. Foley[1(✉)] and Olgierd Pieczul[1,2]

[1] Department of Computer Science, University College Cork, Cork, Ireland
s.foley@cs.ucc.ie
[2] Ireland Lab, IBM Software Group, Dublin, Ireland
olgierdp@ie.ibm.com

Bruce Christianson: Right, I think it was Dijkstra[1] who said that if you don't formally specify a system it can never be insecure, it can only be surprising. The obvious course of action is for the European Commission to make formal specification illegal and then announce victory. But here to put the other side of that particular argument are Olgierd and Simon.

Simon Foley: Thanks Bruce. This is a joint talk. I'm going to give a brief introduction and some context, and then Olgierd will take over with more detail.

Following up on Bruce's comment, I remember Bob Morris would sometimes describe security as absence of surprise and we've been looking at the security surprises that result from programmer error. For a programmer, an implementation fails if there's a surprise in its execution where its actual behaviour is not as required or expected. Thinking of this in terms of security, the implementation is not secure if there's an attack whereby the system's actual behaviour is not consistent with its required or expected behaviour. For example, a replay attack in the actual behaviour of a security protocol is not what the programmer expected.

No surprise means that the actual behaviour of the program matches its expected behaviour. Of course this assumes that the programmer fully understands the expected and actual behaviour. In practice, the programmer is likely to take a flat view of the system, thinking only in terms of the APIs he uses, and he may not think about what happens under the hood. For example, a protocol developer thinks in terms of messages and crypto operations, and might not think too much about what happens in the low-level code behind the API. At the extreme, this developer lives in a two-dimensional world that we can think of as flatland.

Edwin Abbot, who was a Shakespearian scholar, wrote *"Flatland: A Romance of Many Dimensions"* in 1884 as a saritical science-fiction exploration of life in a two-dimensional world, how its citizens managed with just two-dimensions and the difficulties that they had in comprehending other dimensions. For example, there's an account of the author, a flatlander, who gets visited by a Sphere and can't comprehend how the Sphere can cause things to disappear in the

[1] Or perhaps Bob Morris, see LNCS 6615, p 120. However, I searched the Internet and found attributions to "apocryphal" and to "Brian Kernighan" as well as use without any attribution at all.

B. Christianson et al. (Eds.): Security Protocols 2015, LNCS 9379, pp. 12–21, 2015.
DOI: 10.1007/978-3-319-26096-9_2

flatlander's 2-D world and then re-appear at a different location. Often, when we study security, take a flatland view.

Contemporary software development is not flat; what might seem like a simple API call in flatland, results in complex interactions within the underlying system and its infrastructure. A programmer writes code at the abstract API level (flatland), may understand some of the lower levels of the code, but does not understand everything that happens behind the API. This is what we call the dark side of the code: low-level program behaviour that is not known or understood by the programmer and this can cause security surprises.

For example, a flatlander programmer didn't understand that a database management system was behind an API call and as a result an injection attack occurs and percolates back up into some surprise in the programmer's flatland. This is like the two dimensional flatlander being surprised by the behaviour of the three dimensional Sphere. Of course, programmers can still make mistakes at their level. Surprises can also occur from programmer blindspots in things they understand but overlook. We think that the usual software development techniques can help us deal with the complexity of the system and ensure that the actual behaviour of the system is consistent with expectation. However, security vulnerabilities still persist. For example, even with abstraction we can still have attacks.

Bruce Christianson: [Just go for it Sandy]

Sandy Clark: Well, I would argue that it is only going to get worse, because currently it is difficult to finish an undergrad degree and not get through the your programme without specialising, and without specialising early, whereas previous generations didn't specialise, you learned the entire system.

Simon Foley: Yes, certainly. I'm not suggesting that abstraction, frameworks and everything else are the solution. I think that they are often put forward as some sort of solution, but they create as many problems as they solve. If we're to have secure systems where we don't have surprises, and the actual behaviour corresponds to the expected behaviour. It seems we need a developer who is expected to understand everything and with the result that there's no dark side of the code. Taking that argument to its conclusion, what we seem to be looking for are, omniscient coders, such as the beings from the Q continuum in Star Trek, who understand all dimensions, not just the few dimensions of the 'puny' human race, or indeed the programmer race. The Q understand everything. Of course its security theatre for us to think that programmers can be omniscient and understand everything. There will always be a dark side of the code.

I hand you over to Olgierd who will to walk you through a concrete example of this dark side in the code. He'll show how it is really easy for programmers not to appreciate what's happening in the low level details. He'll also talk a little bit about how we've been using run-time verification techniques to look at this problem.

Olgierd Pieczul: I will be talking about real developers that I am here to represent. I will use an example, a bookmark sharing application. It is a web application that can be used to create, browse and manage web bookmarks.

Among other things, it can be used to save snapshot image of a bookmark, which then can be viewed by a user. In the past implementing such an application required a lot of work. But today, with the many of tools and frameworks available it is very easy to implement. For example, all the code required to process bookmark creation can be contained just in a few lines. This code creates snapshot images of the website addresses provided as the parameters in the request, and saves them along with a bookmark information using persistence framework. In only four lines of code the application gets user provided data, contacts and retrieves the website, generates snapshot images and returns operation status.

But what exactly is the code doing? Is there anything that can potentially have impact on security? At the fourth line we see something that may be related to database operations. Perhaps there is potential security problem of SQL injection. Actually there is not, because the persistence framework protects against that. The first two lines, where application calls a library to create snapshot images of a bookmark, may cause security problems. The code makes network connections to a user-provided location. Why is should that be a security risk? It is expected that the application will connect to the public web server, to download the website, and create an image, and report back to the user. But there is also an expected threat, that instead of specifying a public website, a malicious user will provide an address to an internal site. This will cause the application to connect to that internal server (that may hold sensitive information), create a snapshot images and present them back to the user.

So what's in the code again? The listing presents very, very high level code, and there is nothing in it that relates to internal/external addresses. The code responsible for creating the image snapshot, provided by a third-party library, is also at a very high level. It takes the website URL as a parameter, and it seems to verify whether this URL is correct, as it throws an exception for a "malformed URL". According to Java documentation, the URL is a "resource to the World Wide Web". Further in the code, you can see that there is a "connection" that is "opened" and an input stream being created from that connection. Also, there is a possibility of "socket timeout" or some network related problem. Although this gives some hints about the code operation, the snapshot library code remains very high level. This is an example of security gap: the code does not openly perform any of the TCP/IP operations and the threat is based on *expectation* regarding low level application behaviour. That this is possible is not apparent by simply inspecting the application code in isolation. It is a human task to correlate the high-level application behaviour with this low level threat.

The most obvious solution is also low level. The application may try to verify that the address of the bookmark is not a local address. But, as you can see in the listing, this pushes the code to lower level of abstraction. Now, the developer needs to consider concepts such as local address, and public address in, otherwise high-level, code.

Alastair Beresford: Isn't there also an alternative solution at a different level?

Olgierd Pieczul: There is, I will present some of them. In fact, this particular solution is wrong. It is based on incorrect assumption that the address from the URL is the address that application will create the snapshot from. It may not be expected, that after the address is verified, and application connects to the HTTP server, that server may perform an HTTP redirect to arbitrary location, such as internal network server.

This problem can be fixed if the application, while verifying the address, followed all HTTP redirects and verified the final location. When implementing such protection, the developer may not expect that it can also be bypassed. A DNS rebinding attack may be used to redirect the connection to internal location after it is verified.

In order to identify all of those problems, the developer needs to go down the low level. For what was at first a very simple, high-level application, the developer now has to consider HTTP protocol, redirects, DNS protocol and so forth.

Often, the developer will just document the issue and expect the user or administrator to provide external security controls, and in this case, control the application's network access. But that does not really solve the problem, it only shifts responsibility from the developer to the administrator. As a result, the gap widens even more and the administrator needs to expect that the application that is installed with this default configuration can potentially endanger their local network. It is very unlikely they would expect that.

Assuming this is the most likely solution, an administrator deploys a firewall, or Java policy that prevents the access of specific local addresses. However, this same code has yet another vulnerability. A malicious user may create a bookmark to a URL with a file protocol, such as `file:///etc/passwd`. Although the URL handling code seems to be focused on network-related operations, it is actually capable of processing URLs with non-network schemes, such as files. In this case, rather than opening a website, the application will open a local file, and create an image, and render that image to the user. In order to avoid this vulnerability, the developer needs to understand the low-level details of the URL handling in Java, even though they never actually directly call it in their code.

This is what we call "the dark side of the code". An application may be able to perform operations that may not be expected by its developers. Expecting that developers can fully comprehend the application to the lowest level, including all its interoperating components, is unrealistic and, in our opinion, security theatre.

So what we propose is that, rather than preventing any possible unexpected behaviour by the application, we verify if the actual behaviour of the application is consistent with what is expected. The developer knows what is expected, because that's how the application should work, and our goal is to verify the actual behaviour against the expectation. Now, the question is, how we can capture and model this expected behaviour? We propose using a model of behavioural norms, an abstraction of application execution traces. The norms represent how the application behaves under normal circumstances. They can be discovered automatically from the logs, and capture unknown patterns of interaction between system components.

On this slide you can see some norms for HTTP and HTTPS connections, successful and failed. We performed an experiment, whereby by capturing all permissions checks, the Java Security Manager, can be used to generate a trace of application activity. We then used the WebUtils library unit tests to 'exercise' the library and capture a trace of expected behaviour. This trace was used to generate a norms model of expected behavior and this, in turn, is used by the runtime verification mechanism to check that current execution corresponds with this model of expected behavior. In the case of a violation, the mechanism can alert or stop the application code execution. In short, the result of running this experiment was that bookmarks with HTTP URLs, and many other expected scenarios execute correctly, while bookmarks with file URLs are prevented by runtime verification.

To wrap up. In modern applications there is a gap between expected behaviour and the actual behaviour. This gap is unavoidable, and expecting developers to fill the gap with their own knowledge and skill is an unrealistic expectation. Rather than preventing all the known unexpected behaviour we propose to capture the expected behavior and verify the application against that behaviour. The behaviour norms that we have developed can be used for that verification. Thank you.

James Malcolm: How are you going to tell the difference between unusual behaviour and bad behaviour? Lots of things are unusual but perfectly OK?

Olgierd Pieczul: There are two answers. One is test coverage: how much the application is being tested. Another one is the abstraction of the expected behaviour. The behavioural norms are not really exact traces of code, and provide a level of abstraction and flexibility within themselves. But yes, it's a good question, this is the intrinsic problem of our research.

Simon Foley: To add to that, one thing to consider about the models of expected behaviour that we are building is that they're not just the Flatland view of the program. That is, they are not just in terms of the API the programmer uses but also includes events in the underlying infrastructure. For example, if it was a web application then the model could include all of the HTTP traffic generated during the unit tests, not just the traffic that is explicitly coded by the application. Such a model may be sufficient to detect anomalies in HTTP traffic as the application executes. Alternatively, the model of expected behavior may be extended to include all local file system accesses. This more detailed model can be used to detect the anomaly in the application Olgierd described. At some point a judgement must be made about how much detail should be included in this model. If we limit ourselves to direct API events, its likely we won't find any surprises. We want to include enough detail to be able to detect the interesting surprises. As Olgierd mentioned previously, finding the right level of abstraction is a challenge in anomaly detection in general.

Dong Changyu: Have we considered the possibility that attacks might arise at the lookalike expected activity.

Olgierd Pieczul: Yes. Some attacks may not represent themselves through different behaviour. This is mostly a matter of how the behavioural model is built. It may consider only "action" attributes, but also other that provide more context. It may also include execution parameters and context that may result in much more precise model. This may, however, reduce the flexibility.

Dong Changyu: So, are the mimicry attacks possible?

Olgierd Pieczul: Yes, though they are more difficult here than in similar solutions, as the model is much more precise.

Keith Irwin: Is the application you use in the presentation the only example?

Olgierd Pieczul: Yes, it is only a proof-of-concept application we developed for this experiment.

Keith Irwin: I was going to say, because I would have trouble believing that the connection to the internal server would look different from a connection to the external server, from the app's perspective.

Olgierd Pieczul: That was an expected threat, that was something we did not try to protect against using norms.

Keith Irwin: Although you would expect you would sometimes run into HTTP redirecting.

Olgierd Pieczul: Correct.

Mohammad Dashti: Ideally you would like to be able to execute these norms and get rid of the problem, right? These norms are somehow restricting your program to what you want it to actually do, and if you could execute them then you could forget about the program.

Olgierd Pieczul: But they are not the actual programme, they are only an abstraction.

Bruce Christianson: But the question is, what's the difference between a program and implementation, and an executable specification.

Olgierd Pieczul: It is not a specification but only an abstracted trace, covering only some portion of application activity, such as permission checks. It is much more abstract than the actual code.

Mohammad Dashti: Sure, but if I see this norm as a regular expression, for instance, then it can be executed. I'm not suggesting that your norms could be executed, but if in principle we want to have an executable norm?

Simon Foley: Yes, I think in principle you could say: I'll take a log from the unit testing and from that I'll infer some state machine that's in some sense equivalent to the code that I have written to some degree of approximation. Then I could take that state machine, and execute it. Yes, in principle, but in practice that's not we're doing. We're working to some level of abstraction of the events under the hood: we're not going all the way down to the lowest-level of system operation. For our current experiments we're looking at the network as the level of abstraction, the TCP/IP operations. We make sure that when the program runs then we're validating against those particular sequences of network operations, everything else is abstracted away. The idea is that if there's a deviation from what's expected (in the network events), then you'll get a slight re-ordering of these sequences, and then that's what gets detected by runtime verification.

Bruce Christianson: But to what extent are you just trying to enforce the implementation to respect the abstraction at the specification then? And to what extent are you actually trying to verify that the abstraction is correct? Or is that a non-goal? Or are you finding them in parallel together?

Simon Foley: If we go back to the beginning of our talk when I argued that that programmers are like Flatlanders, and they think only in terms of their own APIs. If you consider the application that Olgierd talked about at the beginning, then that's their view of the system, their Flatland view of the world. They have no idea of, or pay little attention to, the underlying sequences of network interactions. When they think of expected behaviour, they think in terms of the high program APIs. Thinking formally, we'd might argue that functional requirements are specified at this high level, and we don't need to think too much about what's happening down at these lower levels of abstraction, which we're not even trying to model in our specification. We write our high level program as usual. During unit testing of expected behaviour, the application runs and we log not just on the high-level APIs, but also on the low-level calls under the hood, such as network events. The model of expected behaviour is built from these events.

Bruce Christianson: So it's actually your implementer who is working at the higher level of abstraction, and your specification is at a lower level.

Simon Foley: Yes, what's being generated is the stuff from the dark side.

Olgierd Pieczul: So just to clarify, all of those low-level norms correspond with a single line of the application code: create a snapshot of a website. Various norms capture different types of low level behaviour that developer may not expect.

Max Spencer: So I propose that the developer still has to know about the dark side of the code, because they need to be able to anticipate the places they input in their unit tests that would generate enough activity for the base line behaviour. The application can still do lots of useful stuff, they need to anticipate. So it might just be shifting your understanding, where instead of having to understand enough about checks in your actual program to prevent the bad behaviour, you would just know enough about it to make sure that your norm generating tests for good behaviour. So it just seems like a bit of a shift from one to the other, but you are still requiring developers to know about the dark side.

Olgierd Pieczul: Using unit tests is just one of a number of possible ways to generate the base line behaviour. Another could be to monitor application operating in a test or production environment.

Frank Stajano: He makes the valid point that you cannot come up with a unit test that tests the entire network for some scenario if you don't think about it, right? You're not going to come up with that at random.

Ross Anderson: It may be more general than that. The application will only work in those sorts of cases that were initially tested, you cannot use it to innovate, you can't use it to do new stuff. If you test an automated car to drive on the left-hand side of the road and you take it to France it will just freak out and it would stop, and all the gears will crunch, and smoke will come from the parts and it will expire. Now that may be OK for an application, but it's not OK for a platform, because the whole point of a platform is that other people can then build stuff on top of it to do entirely new stuff. This would be no good for testing Windows 11.

Alastair Beresford: So maybe one way to fix this is to record the application activity for the first 10 days in production, assuming that there are no attackers. I record how it is used and then flip the switch.

Olgierd Pieczul: Of course, the expected behaviour may be gathered in multiple ways.

Ross Anderson: So it gives new zest to the application development life-cycle, all the programs are a teenager that will try all sorts of new stuff, but by the time it's 25 it will be set it its ways, and it will never change.

Tom Sutcliffe: Do you feel this may risk breaking the abstractions that your library is trying achieve in the first place?

Sandy Clark: That is OK. If you break them, you find out where they are broken and then you understand the assumptions that were made in the first place that do not work.

Tom Sutcliffe: Yes, but it may be that it was an implementation detail, which really doesn't actually matter for your use base, but have to spend two days debugging it to establish that. I'm just thinking of the case where, for example, you upgrade a library and suddenly it starts caching the DNS requests somewhere such that you don't get your resolve step happening. And then it suddenly breaks your norms.

Olgierd Pieczul: We have thought about it as well. It is a question when behaviour is being captured. It may happen during development cycle of a new version of the library, but also, when the application is upgraded to the new library.

Tom Sutcliffe: If all those libraries are under your control and are bundled with your app then I guess you could do that. But if you're potentially deploying against different versions, and you don't necessarily know, you don't have a well specified environment to start with, it could get quite tricky.

Olgierd Pieczul: Yes, this is a problem, and a possible way to solve it was to limit captured behaviour on both high and low level and only build the norms for the things you control, or you know that are not changing.

Tom Sutcliffe: I suppose it does lead onto the fact that if you do control more of the stack then you're less likely to get unexpected behaviours if you are deploying against fixed versions that you've run and tested with, so yes.

Ross Anderson: So if you're got a big red button in your app that says, training now, and then the attacker is just going to socially engineer the user into pressing the big red button.

Mark Lomas: Can I suggest that part of the problem is that your internal network is too open. What we tend to do for testing applications, is to put them on an isolated VLAN where they have no external connectivity at all. Now clearly the application will not function, so the developer has to decide, as part of application design, that it will communicate with these other entities within the organisation. Then they have to set up a firewall, set up logging and verify if there's at least one test case that exercise that firewall rule. If there isn't one then the developer made a mistake, if there is one, and it's inconsistent, then if you change control to make that permanent rule, and then gradually, gradually open up control to allow the expected behaviour.

Olgierd Pieczul: That is true, but the problem is that the application may legitimately contact that server for other purposes. For example, application may be expected to contact a database server but not to contact it in context of creating snapshot images of its interface. Also, unexpected access to files, will still work regardless what network protection is used.

Bruce Christianson: So this gives you access to external behaviour, but not to information flows that are internal? Can you enforce information flow using a mechanism like this?

For example, most frauds are done by people doing things that the rules allow them to do, but then using the information for a purpose other than the purpose for which they're supposed to have access.

Olgierd Pieczul: So in some cases it's a question of granularity of all those protections.

Bruce Christianson: Yes.

Olgierd Pieczul: So for example, the firewall rules can be very specific to say that this application is only allowed access to this very limited set of targets. Or it may open files but only some classes may open class can open some files.

Bruce Christianson: Could you augment this by actual instrument, put an instrument in the application code, and follow in control paths through the application code, tracing that, and seeing whether that complied with what you expected.

Alastair Beresford: And maybe it's the broader question about what level or levels you actually try and track.

Bruce Christianson: Which is kind of where we came in, yes, that's fair, yes, OK. Right, thank you very much…