Mismorphism: The Heart of the Weird Machine (Transcript of Discussion)

Prashant Anantharaman(✉)

Dartmouth College, Hanover, NH, USA
pa@cs.dartmouth.edu

As Jon mentioned, this is one of our two talks at this workshop. My colleagues Vijay and Michael, over here, will be presenting later in the morning. In this talk, I'll introduce what mismorphisms are, and some of the things that we work on, which are LangSec and weird machines. And I'll talk a bit more about some of the work we did in this paper. We want some insight from all of you about what we can do to improve our work. And we have some holes that we've identified that we want your help fixing.

What really are mismorphisms? Mismorphisms are, in essence, mismatches between how humans think about the same thing differently. A really nice example is the open source null character bug that was discovered by Dan Kandinsky and team. An attacker could use a domain name: the attacker owns the domain name, which is badguy.com, and the attacker crafts this certificate request and sends it to the certificate authority. The certificate authority reads this as, "Okay it's .Badguy.com"—it essentially means that it's a subdomain owned by the attacker. And then the certificate authority grants the certificate. So there's a null character over here. And a null character in a domain is not supposed to be allowed.

How the certificate authority handles this null character, and how the browser handles this null character, is different. And the exciting thing about this bug was that you could actually have something like a wild card in the beginning, which would mean that this domain gets cut. For example, the browser was actually reading this as the thing that is before the null character. This would necessarily mean that the attacker has a certificate that says that it's paypal.com, but it's actually not that. If it had a wildcard certificate, it would match any domain. The certificate is for any domain because the browser thinks the thing before the *null* character is what the certificate is for. Whereas the certificate authority thought the thing after the *null* character is what it was for.

This is a typical example of what a disconnect could be. How do we really go about cataloging these disconnects? What are mismorphisms? Mismorphism is a disconnect between what the reality is, what the person actually imagines it to be, and the symbol that is used to signify the reality. You can see that the three are distinctly different. The same can actually be applied to a security setting where there is a reality, and there is what we think (the mental model) and then how it's actually represented in the wild. And there is a disconnect between these three, and this is what we are actually getting to in this paper.

J. Anderson et al. (Eds.): Security Protocols 2019, LNCS 12287, pp. 125–131, 2020.
https://doi.org/10.1007/978-3-030-57043-9_12

This paper also deals with LangSec. Language-Theoretic Security (LangSec) says that the parser, which is the first part of the code that actually processes any input, is literally a decider for the input. The parser decides if the input is something that is a part of the language or not, and only if it is a part of the language it accepts the input. If it is not a part of the language, then you let it out.

Parsers actually are an essential part of weird machines. A weird machine is something that processes some input, and it runs some unintended computation. The computation that a weird machine runs is not intended, but it's accidentally present by design. It could be through poor design, or it could be through the lack of knowledge that this language is actually not decidable sometimes. Weird machines could stem from a bunch of these reasons. Our insight over here is this that mismorphisms lie at the heart of these weird machines.

What we are doing over here is that we are describing how different programmers might have thought differently, leading to various bugs. We are cataloguing some past bugs that we encountered and, in this paper, we are trying to come up with a formal notation for how we think these mismorphisms could have occurred.

We use three standard notations in this paper. The first one is the interpretation equivalence, the second one is interpretation uncertainty, and the third one is interpretation inequivalence. Basically, the intuition over here is that there is a predicate P and then there are interpreters A_1 through A_k. If it's an interpretation equivalence, then the interpreters A_1 through A_k all think the same about the predicate. So they arrive at the equal truth value for the predicate. In case of the interpretation uncertainty there is at least one interpreter A_1 through A_k who thinks that the value is uncertain and we are not really sure if it's going to be true or false for that predicate. And I think the inequivalence is intuitive over here. It is the case that there is at least one A_1 through A_k whose value doesn't match the other ones.

These are just three examples of mismorphisms that we came up with. There are many more in our paper. One specific thing is the Ethereum DAO disaster where the language was actually not decidable, the O and the D over here are the oracle and the designer. The designer assumes that the language is decidable, but then, in reality, the language is not decidable. We've categorized it as an interpretation uncertainty there. Shotgun parsers are one of the root causes for LangSec or input handling bugs. What we mean is that there's a buffer B that the implementer assumes doesn't change over time, whereas in reality the buffer actually changes over time because it's a shotgun parser. A shotgun parser is basically a parser where you perform input handling and processing interspersed, so the LangSec principle is this that you perform all input handling distinctly, separate from the rest of your code. You validate all your input and then only start processing, whereas in a shotgun parser you do input handling and processing interspersed in your code. A shotgun parser would be categorized in this way.

Let us see a couple of examples of how we arrived at these mismorphisms. One example is the Heartbleed bug. In a standard request, you say that "If you're alive then just reply back with this particular string in it." And there are two length fields involved over here. There is a length field included within the heartbeat request, and there is a length field for the overall packet. And obviously, both need to match in some way. And over here you could say, "Send me this forty thousand letter word called blah" when in reality the word 'blah' is just four characters. There is a mismatch over here, and the two length fields should have been validated. You could basically say that there is a sanity check where the oracle, the designer, and the implementer's interpretations don't really match over here, and the length field should have been validated.

Another example is what I mentioned earlier: the null character bug. There are two parsers, P_1 and P_2. These two parsers are meant to be equivalent since they are endpoints of the same protocol. In reality, they represent the view of the oracle, the designer, and the implementer. Hence, it's inequivalence between their interpretations.

That brings me to the discussion questions. We were wondering what sort of user interface could we actually build to help protocol developers or protocol implementers to actually catalog these vulnerabilities or submit these vulnerabilities to some central entity that we could build. And, we were wondering how we could actually go about making such a thing useful. I'll leave the floor to a discussion.

Jonathan Anderson: It seems like one of the interfaces that's most relevant to things like the Heartbleed attack, even leaving aside for a moment the interface for submitting examples—it's just the complexity of APIs, in the sense that Heartbleed didn't happen because people didn't have APIs for validating fields, it was because it was too annoying to use them so they hand-rolled something that sucked. I guess if you're studying language security, are you also coming up with ideas that are relevant for designing interfaces in ways that developers will align reality with the expression of what their code is trying to do?

Reply: Yep, that's totally relevant. Because one important thing that we're looking at through this work is to actually understand how these mistakes actually occur. And then, how you can actually build language principles to avoid them from occurring and how you can actually build tools that would help a developer not make the same mistakes again and again.

Frank Stajano: This classification of things depending on how the interpreters differ in their interpretation of the predicate seems to me to be very sensitive to the set of interpreters that you have. So, if you have a bland or a stupid interpreter, then we all agree that something may in fact be wrong. We have to have ingenious or devious or creatively written interpreters to actually uncover the discrepancies that your method would then classify as a mismorphism.

So, where do these interpreters come from? What's the general method for, given a new vulnerability, saying, "well, let me now feed it to a sensibly diverse group of interpreters to see if there is a mismorphism."

Reply: The way we actually went about it was to understand how protocols are implemented. For example, in Heartbleed, there are a group of protocol designers, and there are a group of people who actually implement these protocols. You have a group of people who design it and a group of people who actually implement it, and then there's the reality. We were actually looking at these three groups, we were not looking at any random interpreter. We were just looking at people who actually have something to do with these protocols and their implementations.

Frank Stajano: Thank you.

Alastair Beresford: So are humans an interpreter as well? For example, if you ask for a certificate for a domain such as paypal.com or badguy.com, that's a valid sub-domain but people misinterpret part of that as paypal.com in some cases. Is that part of your model or not? Is that a helpful viewpoint?

Reply: I don't really know. Vijay?

Vijay Kothari: We were thinking about that. We didn't really discuss that specific example in the current iteration of the paper, but yes, it's very interesting, thank you.

Fabio Massacci: I want to keep going on Frank's point. It's actually an interesting idea. From the software engineering perspective, you say, "Well, let's ignore all the complexity of the internal structure, just look at the parser." If the two parsers of the CA and the parser of Firefox interpret the string in the same way, then there will be fewer errors later on. You think that this would be a feasible approach?

Reply: What you just said is what the whole field that we are working on, LangSec, is about. We are trying to say, "How do you build parsers based on the formal specification of a protocol?" To be able to say to each other, these two protocols are equivalent.

Fabio Massacci: What I meant is you want to test the interoperability of the parser only, rather than proving that the parser is correct. Parsers are very difficult beasts to formalise correctly.

Reply: The thing is that it also depends on the language of the protocol. It needs to be within the equivalence boundary of the Chomsky hierarchy. It cannot be beyond the equivalence boundary to be able even to say that these two are interoperable. It's not something that you can just have a robust set of test cases or fuzzing cases for it. To be able to prove that two parsers are equivalent, it has to be within a specific boundary on the Chomsky hierarchy.

Michael Millian: Prashant touched on this briefly, but we do use fuzzing to some degree to get at this. But he is absolutely right that to prove it in a formal sense, for all inputs, that they're going to handle it the same way, it takes a little bit more than just fuzzing, but it takes exactly the formal proof.

Jovan Powar: Further to that point, the software design aspect of it is essentially like unit testing in API, right? You throw a lot of stuff at it and make sure, not that they're correct but that they do the same thing. And that's attacking the problem from the downstream end of it, where you've got the design system and you're checking that it behaves correctly. And obviously that is much harder to do because of the problems faced get much bigger.

What do you think of the idea of attacking from the language point of view and trying declarative languages, where you have to compare the declaration of the API correctly and lots of different things, and then leave it up to the implementation of the language to do it, which reaches a much smaller problem space to prove the correctness of the language?

Reply: I think, right now one of the more significant problems is that specifications themselves are inadequate or unclear, which is why the implementers end up implementing them differently or end up missing sanity checks, for example.

One of the overall goals of the entire field of security should be to come up with a better way of describing specifications of protocols so that at least such vulnerabilities would be fewer.

I do agree with what you said.

Michael Millian: To partially answer that question and to be more specific about some of the flaws in specifications themselves, something that we see more frequently than we would like, is a specification for a protocol that has a state machine or some kind of formal description of what a packet should look like and what fields should look like. But then, in prose, the specification offers more description on that or puts extra constraints. So, if you don't read all of the prose carefully, you're going to end up with different implementations based on how closely you read the document.

Reply: There has been work, quite recently, on studying these state machines of various OpenSSL implementations in operating systems, and they did end up quite different from each other. In practice we have seen that, although there is a specification, the specification is written in a way that the implementations end up being significantly different from each other.

Lydia Kraus: Thanks, this is an interesting topic and I'm wondering when I see your questions on the discussion slide, what is your conceptual idea of the whole thing? Is it something like a translator that translates the designer's mental model to the programmer's mental model?

Reply: That's an excellent question. One of our more important motivations in the field of LangSec is to have formal specifications written in a way that it can translate to code in a one-to-one mapping way. So, yes. The answer is yes. To have a way that you can have a one-to-one mapping from what the designers mental model is to what the reality of the code is.

Sasa Radomirovic: I've collected a few follow-ups. I'll start with Alastair. This was the "Is a human a weird machine," question. I think phishing attacks would be a case where the human has one idea of what's going on, whereas the attacker is creating a different mental model to match. Then, a more recent discussion on specifications and then annotating it with text. I think a couple or more years ago at this workshop there was a suggestion that we specify protocols in a way that, if you deviate from the specification, when implemented your implementation just wouldn't be correct any more, it wouldn't interact with our clients. But it seems to be a very expensive way of making interactions work. If you haven't specified thing carefully, many cases do not work, because you deviate from the specification. But then on the other side, if you wanted to do this formally, that seems very expensive too. And in the end it seems to be only with one code base that is being shared among everybody in order to ensure that everything is being implemented the same.

I feel like I would use it now too. I'm wondering: can you do more than this?

Reply: I agree with what you said. There's not much we can do, which is why we want to discuss this.

Michael Millian: I, for one, and my lab more broadly, are strongly for the position, closely related to what you said, which is reducing to a single code base. The way I like to think of this is moving towards a world where parser writers don't really roll their own parser, the same way that crypto-library writers don't really roll their own crypto. You really need to know what you're doing to get this right, and so for all the protocols we should have a couple standard libraries that have been proven correct, and that everybody can just call. We don't have that yet. We're working on that. But I think that's the ideal world we should be moving towards.

Simon Foley: I guess for me, this is really my interpretation which is different from formal verification of the codebase. A remark: a couple of years ago, we looked at struts, which is an MVC implemented in Java and widely used. We looked at the security controls in that, and I'm thinking that what we saw in the mistakes, and the way they were implemented in the security control were very likely mismorphisms. What we'd done is over a 12 year period, we've looked at how vulnerabilities have been set out against that particular security control, and then the developers were in turn, fixing those vulnerabilities. And they made the same mistakes over and over again along with a bunch of other issues; we gave a presentation a couple of years ago here. I was thinking, what we saw were people misinterpreting a graph so the implementors of struts was misinterpreting

this particular security control over and over again, and I'm trying to think, "Well, I wonder would it be useful, or how you might capture these things as mismorphisms."

It could also be useful for you if you wanted to do a study, to go back and look at something like struts that has a large number of vulnerabilities posted against it. A lot of them are the same kind of vulnerability as the one I spoke about yesterday about EquiFax is exactly the kind of vulnerability we'd been seeing five or ten years ago. And it might help you to identify patterns and mismorphisms that could be usable in a more general way.

Reply: What you describe is something that we can capture in our particular categorization. What you described is essentially a mismatch between what the designer intended with the API and what the implementer actually grasped from the API. It's something that we have looked at in specific instances, but we have not looked at one giant code base over a long period.

Simon Foley: Yeah, and in this case the designer was the implementer.

Reply: Oh, okay.

Simon Foley: A quick example would be as follows: the security control was doing whitelist checking against OGNL code that was being sent to the NBC in order to set internal settings and the designer would see a vulnerability posted and say, "oh, I must fix this", and as a sort of knee-jerk reaction would have said "oh, add some more checks," but they didn't think the problem through, and so they really only half-fixed it. You see this happening over and over again.

Now, describing each one, I think it could be described as a mismorphism. The bigger challenge is the fact that the designer, who designed it, didn't quite capture what they really intended. So, it's going back to the sort of social constructivism that they didn't engage in what was it that they were trying to describe and, just kind of getting into this half-baked fix.

Reply: Yeah, that's really interesting because, while we were looking through this, we were also looking at how a bug like Heartbleed originated, and how it was patched later on. So, the way the patch was implemented again was something we call a shotgun parser. It was patched in line and not in a separate place where you perform all the input validation first, and then go on to processing. I think it would be interesting to look at how bugs were patched later on and how the same mistakes were repeated.