(Some) security by construction through a LangSec approach

Erik Poll

Digital Security group, Radboud University Nijmegen, The Netherlands
erikpoll@cs.ru.nl

This talk discusses some good and bad experiences in applying formal methods to security and sketches directions for using formal methods to improve security using insights from the LangSec (language-based security) paradigm.

On the face of it, security looks like a promising application area for formal methods. Cyber security is a huge and still growing concern. It is widely recognized that security should be addressed *throughout* the software development life cycle, ideally by practising so-called Security-by-Design, and not bolted on later as an afterthought; this means that formal methods for security could be applied at any stage of the software development life cycle, from the earliest stages of requirements engineering to the final stages such as pen-testing or patching.

Still, all this is easier said than done. Security requirements can be tricky to formalise – or even to spot at all – and it can be difficult to say what it means for an application to be secure. It is often easier to say what may make an application insecure, as is done by lists of standard security flaws such as the OWASP Top Ten[1] or the CWE/SANS Top 25[2]. Such lists are very useful, but always incomplete, and lend themselves more naturally to testing for certain types of security flaws post-hoc than to guaranteeing their absence by construction.

A more constructive approach to security can be taken by realising that security problems typically arise in interactions and exploit the *languages* used in these interactions. The most obvious example is the interaction between an attacker and a system, where the attacker tries to abuse the interface the system exposes. This interface can be a network protocol, but it may also involve a file format, say JPEG, or a language such as HTML. Security problems can also arise in the interaction between two applications (or an application and an external service) even if neither of them is malicious. Classic examples here are the interaction between a web application and its back-end database, where SQL injection becomes a worry, or the interaction between a web application and the browser, where XSS becomes a worry.

The LangSec paradigm[3] highlights the central role played by the languages used for these interactions – e.g. file formats, protocols, or query languages – in causing security problems. Root causes of security problems identified are: the large number of these languages, their complexity, their expressivity, the lack of clear specifications, and finally the fact that parsers to process these languages are hand-written, and often mix parsing and processing of inputs.

[1] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[2] https://cwe.mitre.org/top25/
[3] See http://langsec.org, esp. http://langsec.org/bof-handout.pdf, or [5].

This also provides a clear way forward in using formal methods to improve security, namely by providing formal descriptions of the input languages involved and using these descriptions to generate parser code, thus getting at least some security by construction. Ironically, formalisms for describing languages are some of the best-known and most basic formal methods around, and parsing is one of the oldest and best understood parts of computer science, with plenty of tools for generating code. So it is a bit of an embarrassment to the computer science community that this is where modern IT screws up so badly, with so many security flaws. In addition to parsers, one would also like to generate unparsers (aka pretty-printers or serialisers), as interactions between systems typically involve an unparser at one end and a parser at the other end. Recent initiatives here include Hammer [2] and Nail [1]. Formal descriptions of input languages can also be used for testing, in test generation or as test oracles.

Even if we get rid of all (un)parser bugs, there remains the risk of *unintentionally* parsing some inputs [7], especially inputs coming from sources that an attacker can control. Here formal methods can also help, with data flow analysis to trace where data comes from and/or where it might end up. Ideally, such data flows can then be controlled by a type system, where different types explicitly distinguish the various languages that the application handles (e.g. to avoid the chance of accidentally processing a user name or a fragment of HTML as an SQL statement), the various trust levels associated with different input channels (e.g. to distinguish tainted inputs from untainted data), or both. As these types can be application-specific, it is natural to use extensible type systems for this, e.g. using type qualifiers [4] or type annotations [3], or to turn to domain-specific languages [6].

References

1. J. Bangert and N. Zeldovich. Nail: A practical interface generator for data formats. In *Security and Privacy Workshops (SPW), 2014*, pages 158–166. IEEE, 2014.
2. S. Bratus, A.J. Crain, S.M. Hallberg, D.P. Hirsch, M.L. Patterson, M. Koo, and S.W. Smith. Implementing a vertically hardened DNP3 control stack for power applications. In *Industrial Control System Security Workshop (ICSS'16)*, pages 45–53. ACM, 2016.
3. W. Dietl, S. Dietzel, M.D. Ernst, K. Muşlu, and T.W. Schiller. Building and using pluggable type-checkers. In *ICSE'11*, pages 681–690. ACM, 2011.
4. J.S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI'02*, volume 37 of *SIGPLAN Notices*, pages 1–12. ACM, 2002.
5. F. Momot, S. Bratus, S.M. Hallberg, and M.L. Patterson. The seven turrets of Babel: A taxonomy of LangSec errors and how to expunge them. In *Cybersecurity Development (SecDev)*, pages 45–52. IEEE, 2016.
6. C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In *ECOOP'14*, volume 8586 of *LNCS*, pages 105–130. Springer, 2014.
7. E. Poll. LangSec revisited: input security flaws of the second kind. In *Symposium on Security and Privacy Workshops (SPW)*. IEEE, 2018.