LangSec: Recognition, Validation, and Compositional Correctness for Real World Security

Mission Statement. *Language-theoretic security* (LangSec) is a design and programming philosophy that focuses on formally correct and verifiable input handling throughout all phases of the software development lifecycle. In doing so, it offers a practical method of *assurance* of software free from broad and currently dominant classes of bugs and vulnerabilities related to incorrect parsing and interpretation of messages between software components (packets, protocol messages, file formats, function parameters, etc.).

This design and programming paradigm begins with a description of valid inputs to a program as a formal language (such as a grammar or a DFA). The purpose of such a disciplined specification is to cleanly separate the input-handling code and processing code. A LangSec-compliant design properly transforms input-handling code into a *recognizer* for the input language; this recognizer rejects non-conforming inputs and transforms conforming inputs to structured data (such as an object or a tree structure, ready for type- or value-based pattern matching). The processing code can then access the structured data (but not the raw inputs or parsers' temporary data artifacts) under a set of assumptions regarding the accepted inputs that are enforced by the recognizer.

LangSec aims to (1) produce verifiable recognizers, free of typical classes of ad-hoc parsing bugs, (2) produce verifiable, composable implementations of distributed systems that ensure equivalent parsing of messages by all components and eliminate exploitable differences in message interpretation by the elements of a distributed system, and (3) mitigate the common risks of ungoverned development by explicitly exposing the processing dependencies on the parsed input.

As a design philosophy, LangSec focuses on a particular choice of verification trade-offs: namely, *correctness and computational equivalence of input processors*. It is informed by the collective experience of the exploit development community, since exploitation is practical exploration of the space of unanticipated state, inasmuch as defense is about its prevention or containment.

Root causes of insecurity through the LangSec lens. LangSec offers a unifying explanation for the existence of vulnerabilities and their continual perpetuation under current software design practices despite massive efforts at defining secure development practices. In short, the existence of exploitable bugs is a consequence of software designs that make verification and comprehensive testing infeasible and *undecidable in the formal sense.*

The propensity of software to admit/accommodate a much larger set of states than anticipated by the programmer is notorious. Exploits expose and use these unintended states, as they run on software-as-is, rather than the programmer's model of the software. Exploit computation can be modeled as a computation occurring on an automaton whose states and transitions are supersets of the programmer's intended model. When the program is meant to receive and process inputs (e.g., packets, messages, serialized data structures, contents of files in specific formats, etc.), these models closely resemble textbook examples of automata recognizing input languages of different Chomsky hierarchy classes – with the extra state and transitions being driven by crafted input off the safe recognition path.[1]

Bugs in input processing (wherever input is taken at a software module's communication boundary) clearly dominate other kinds of bugs. Hence the first order of business in securing software that does any communication is ensuring that no unanticipated state is entered and no unexpected computation occurs while consuming inputs. In practice, however, such code is often ad-hoc and lacks a clear, formal language-theoretic definition of valid payloads. What's worse, inputs are "checked" with recognizers that cannot possibly accept or reject them correctly, e.g., context-free formats with regular expressions.[2] In such cases, subsequent code assumes properties that couldn't possibly have been checked, and thus cannot be trusted to abide by their specification. Non-existence of unexpected computation is then highly unlikely, and unanticipated state conditions proliferate.

[1] Exploitation itself can be said to be the art of finding, leveraging, and manipulating such state in the target, see e.g., *Exploitation and state machines: Programming the "weird machine", revisited.* Thomas Dullien/Halvar Flake, Infiltrate 2011

[2] See http://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags.

Verification is a primary weapon against software reaching unanticipated state. A variety of safety and liveness properties can be verified; however, choosing the properties to verify is still a matter of software design and its engineering trade-offs and assumptions. We posit that handling of inputs must be the first target of verification; thus, input-handling code must be constructed in such a manner so as to be verifiable. The following obstacles, which we identify as design anti-patterns, must be removed for this to happen.

1. Ad-hoc notions of input validity. Verification of input handlers is impossible without formal language-theoretic specification of their inputs, whether these inputs are packets, messages, protocol units, or file formats. Therefore, design of an input-handling program must start with such a formal specification. Once specified, the input language should be reduced to the least complex class requiring the least computational power to recognize. Considering the tendency of hand-coded programs to admit extra state and computation paths, computational power susceptible to crafted inputs should be minimized whenever possible. Whenever the input language is allowed to achieve Turing-complete power, input validation becomes undecidable; such situations should be avoided. For example, checking "benignness" of arbitrary Javascript or even an HTML5+CSS page is a losing proposition.

2. Parser differentials: mutual misinterpretation between system components. Verifiable composition is impossible without means of establishing parsing equivalence between different components of a distributed system. Different interpretation of messages or data streams by components breaks any assumptions that components adhere to a shared specification and so introduces inconsistent state and unanticipated computation. In addition, it breaks any security schemes in which equivalent parsing of messages is a formal requirement, such as the contents of a certificate or of a signed message being interpreted identically (e.g., X.509 CSRs as seen by a CA vs. the signed certificates as seen by the clients[3] or signed app package contents as seen by the signature verifier versus the same content as seen by the installer (as in the recent Android Master Key bug).[4]) An input language specification stronger than *deterministic context-free* makes the problem of establishing parser equivalence undecidable. Such input languages and systems whose trustworthiness is predicated on the component parser equivalence should be avoided.

3. Mixing of input recognition and processing (a.k.a. "Shotgun parsers") Mixing of basic input validation ("sanity checks") and logically subsequent processing steps that belong only after the integrity of the entire message has been established makes validation hard or impossible. As a practical consequence, unanticipated reachable state exposed by such premature optimization explodes. This explosion makes principled analysis of the possible computation paths untenable. LangSec-style separation of the recognizer and processor code creates a natural partitioning that allows for simpler specification-based verification and management of code. In such designs, effective elimination of exploit-enabling implicit data flows can be achieved by simple systems memory isolation primitives.

4. Ungoverned development: Adding New Features / Language Specification Drift. A common practice encouraged by rapid software development is the unconstrained addition of new features to software components and their corresponding reflection in input language specifications. Expressing complex ideas in hastily–written code is a hallmark of such development practices. In essence, adding new input feature requirements to an already-underspecified input language compounds the explosion of state and computational paths.

Even with a LangSec-style approach to input parsing specification, the "new feature challenge" remains. In particular, developers require tools and intuition that can inform them of the consequences of a particular type of change to the input language specification, namely changes that entail an increase in required computational power. However, such intuition can only proceed from explicit enumeration of the dependencies between the new code and the accommodations for it in the input formats – and the corresponding additions to the recognizer. Such additions must remain coherent with the processing code assumptions, which the recognizer must remain capable of enforcing. For example, they should not change the class of the input language.

[3] *PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure*, Dan Kaminsky, Meredith L. Patterson, and Len Sassaman, Financial Cryptography 2010

[4] This family of bugs boils down to different views of the signed package by the verifier's Java-based implementation of unzip and its C++ implementation in the package installer; cf. `http://www.saurik.com/id/18`