The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them

Conference Paper · November 2016

DOI: 10.1109/SecDev.2016.019

4 authors, including:

Falcon Darkstar Momot
Leviathan Security Group
5 PUBLICATIONS **48** CITATIONS

SEE PROFILE

Sergey Bratus
Dartmouth College
86 PUBLICATIONS **1,319** CITATIONS

SEE PROFILE

The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them

Falcon Darkstar Momot
Leviathan Security Group
Seattle, WA
falcon@falconk.rocks

Sergey Bratus
Dartmouth College
Hanover, NH
sergey@cs.dartmouth.edu

Sven M. Hallberg
Hamburg University of Technology
Hamburg, Germany
sven.hallberg@tuhh.de

Meredith L. Patterson
Upstanding Hackers, Inc.
Brussels, Belgium
mlp@upstandinghackers.com

***Abstract*—Input-handling bugs share two common patterns: *insufficient recognition*, where input-checking logic is unfit to validate a program's assumptions about inputs, and *parser differentials*, wherein two or more components of a system fail to interpret input equivalently. We argue that these patterns are artifacts of avoidable weaknesses in the development process and explore these patterns both in general and via recent CVE instances. We break ground on defining the input-handling code weaknesses that *should* be actionable findings and propose a refactoring of existing CWEs to accommodate them. We propose a set of new CWEs to name such weaknesses that will help code auditors and penetration testers precisely express their findings of likely vulnerable code structures.**

I. Introduction

Many famous exploitable bugs of the past few years—such as Heartbleed, Android Master Key, Rosetta Flash, etc.—have been parser bugs. These parsers tended to give experienced code auditors the proverbial "bad feeling about this". However, a feeling is not a finding and is not actionable, no matter how unsafe the code might look. As such, telling programmers to be ever more careful [30, 25, 33] about sanitizing inputs [11, 37, 21] is not helpful.

An implicit but common assumption has been that properly following applicable specifications such as RFCs or ISO standards protects programmers against both unexpected effects of crafted inputs and incompatibilities. Security lapses in implementations of such standards are then often assumed to be due to the indefensible carelessness of individual programmers who failed to follow the standard.

Rather than blaming imperfect programmers, we should take a look at the root causes of their mistakes. This leads us to develop specific and easily verifiable requirements and procedures, which have the potential to end the blame and fix the problem.

A. Input-driven exploitation

There is a way in which all input-driven vulnerabilities are alike and exploited alike: *invalid input is processed instead of being rejected.* Its intended preconditions unsatisfied, the processing code drives the system through a sequence of states its designers did not foresee. An exploit can be thought of as a program encoded in crafted inputs, using fragments of the processing code outside of their intended preconditions as its primitive operations, and operating in the space of states that arise from precondition violations. [20, 14]

As Morris noted in 1973, the programmer "could begin each operation with a well-formedness check, but in many cases the cost would exceed that of the useful processing" [29]. To delineate between paranoia and prudently providing for the satisfaction of preconditions in application logic, certain questions must be answered: What are the properties of input that need to be checked and can be relied upon? What coherent sets of such properties can scale up to be implemented correctly by large groups of programmers? To what extent are the pitfalls properties of the input specifications themselves? The LangSec methodology seeks to answer these questions.

B. LangSec

In a nutshell, language-theoretic security (LangSec) is the idea that many security issues can be avoided by applying a standard process to input processing and protocol design: the acceptable input to a program should be *well-defined* (i.e., via a grammar), *as simple as possible* (on the Chomsky scale of syntactic complexity), and *fully validated before use* (by a dedicated parser of appropriate but not excessive power in the Chomsky hierarchy of automata).

In other words, LangSec methodology starts with an explicit grammar of expected inputs as a language. It postulates that a well-constructed parser must plainly follow this grammar, and must reject non-conforming inputs without operating on them any further. This requires a clear and obvious boundary between the input-validating code and the rest of the code, at which the validated properties of inputs are clearly documented (and match the input language specification).

Our fundamental measure of input language "safety" is the Chomsky complexity hierarchy. For a variety of reasons, LangSec supports limiting protocols and other input language specifications to grammars no more complex than *deterministic context-free*. This constraint is both formally testable and practical. For example, JSON, subsets of XML, and Protocol Buffers can be used consistently and intuitively to stay within these bounds if complex inner dependencies can be avoided.

Deviating from these principles opens a Pandora's box of bugs and exploits which, in general, cannot be algorithmically mitigated.

DOI 10.1109/SecDev.2016.21

C. Parsing weaknesses in existing CWEs

Before we present our taxonomy, we discuss how the existing CWEs cover parser weaknesses—and why a more comprehensive approach is needed.[1]

Existing CWE entries tend to characterize specific kinds of programming errors such as buffer overflows (CWE-805), but also represent broad attack types, e.g. cross-site scripting (CWE-79) and SQL injection (CWE-89). Many LangSec bugs could be filed under the abstract CWE-20, *Improper Input Validation*. This could hardly be more general, but also, again, hints that the weaknesses were mere programmer failure.

The CWE guides code analysis and application security testing in many cases. Code reviewers and auditors generally test that a program is secure against particular attacks and free from particular patterns of weakness, and the CWE enumerates a large corpus of items to check. In addition to the use of CWE as a categorization of CVEs, code reviewers and code review tools frequently frame individual findings as instances of a particular CWE. The CWE acts as a pre-packaged justification for the reporting of the finding, even where there is no actual evidence of exploitability. This is valuable because attackers usually have more time to develop creative exploits than security analysts.

Generally, a security analyst has to justify a finding in some way to report it. That justification might take the form of a reference to a standard, evidence of actual exploitability, or citation of a CWE or in another taxonomy. The analyst will then refer to the citation to suggest a remedy.

There are currently 11 potential mitigations listed in CWE-20, advising developers to attend to several necessary components in input validation, but the closest it comes to directing developers specifically to avoid writing shotgun parsers (see below) is to call for input canonicalization (transforming input into an application's internal format before validation). It is not explicit about the need to reject invalid input before processing.

When the MITRE and SANS top 25 was still maintained, a mitigations index was developed [4]. The index contained general mitigations for the vulnerabilities identified in the top 25, e.g., M1 *Establish and maintain control over all of your inputs* and M2 *Establish and maintain control over all of your outputs*. The guidelines are agnostic to the specific method and nature of the control for which they call. We aim our taxonomy to include such guidance and add rigor to existing vague guidelines.

II. Taxonomy

The existing CWE descriptions related to LangSec-type bugs are often inexact and concentrate on general effects rather than underpinnings of the bugs. We propose a new taxonomy that can be used to describe the relationship between the violation of LangSec requirements and resulting vulnerabilities:

- Shotgun parsing (ad-hoc validation during processing)
- Non-minimalist input-handling code
- Input language more complex than deterministic context-free
- Differing interpretations of input language
- Incomplete protocol specification
- Overloaded field in input format
- Permissive processing of invalid input

Where the above problems are allowed to exist, they are likely to cause considerable numbers of vulnerabilities. For example, we surveyed security bugs in OpenSSL from January 2015 to June 2016, by inspecting the reports of each vulnerability assigned a CVE number and evaluating whether the vulnerability could have been averted by avoiding one of the antipatterns itemized above.

We categorized an OpenSSL vulnerability as relating to the shotgun parsing antipattern when a discrete validation stage would have prevented the ingress of invalid input (frequently too-long input but occasionally other types) into application logic or complex input transformations. We found permissive processing of invalid input when a validator intentionally accepted invalid input and passed it through to application code, for instance to accommodate interoperability with a buggy implementation. In other cases, we found that a vulnerability was related to an incomplete protocol specification when it arose from a problematic interpretation of what constituted valid input (or what the meaning of that input was) and inspection of the relevant specification seemed to allow both the pre- and post-patch behaviors.

Not all software projects express all types of vulnerabilities from our taxonomy. For example, we did not categorize any of the surveyed OpenSSL vulnerabilities as resulting from an input language more complex than deterministic context-free.[2]

Of the 47 vulnerabilities enumerated, we estimate that 35 (74.5%) would have been averted if the design of that library had avoided the problems taxonomized here. Out of these, 13 seemed most attributable to shotgun parsing and 11 to attempts to process invalid input or corresponding failure to reject known-invalid input.

We estimate that only 12 reported vulnerabilities in that time period in OpenSSL were beyond the ken of LangSec. These were generally cryptographic in nature, or related to concurrency problems.

That a widely-used, critical cryptography library is mostly vulnerable not because of cryptography implementation flaws but because of the difficulty of processing even highly standardized input languages speaks to the need for a new way to consider such vulnerabilities beyond "be more careful."

III. Antipatterns

We now describe each of these antipatterns, and then discuss how each antipattern led to serious security bugs.

[1] We build on previous work [22], which classifies LangSec vulnerabilities according to the MITRE Common Weakness Enumeration database.

[2] This is not to say OpenSSL is structurally immune to such problems. On the contrary, if the recently-proposed anonymous authentication scheme of Delignat-Lavaud et al. [19] were implemented in OpenSSL, the path to such a vulnerability would be open.

a) Shotgun Parsing: Shotgun parsing is a programming antipattern whereby parsing and input-validating code is mixed with and spread across processing code—throwing a cloud of checks at the input, and hoping, without any systematic justification, that one or another would catch all the "bad" cases.

Shotgun parsing necessarily deprives the program of the ability to reject invalid input instead of processing it. Late-discovered errors in an input stream will result in some portion of invalid input having been processed, with the consequence that program state is difficult to accurately predict. This type of parsing can occasionally be detected by static means, since it is rooted in program structure [48].

Many injection vulnerabilities, such as SQL injection or XPath injection, fall into this category because they represent a failure to correctly validate user input before it is used. In fact, they usually represent a failure to validate it at all, so that later application code (a SQL parser or some other type of query engine) is the only validator the input goes through. This relates back to the shotgun parsing problem: the "validator" that accepts the input is both strewn throughout the program, and not deliberate but rather emergent.

For another extremely common example of this, consider a recent Rails bug, CVE-2016-0752 [8]. Here, directory traversal is possible because a developer failed to remember to filter dots. We can also model this problem as a failure to validate input before processing it, since what ultimately happened here and in the litany of similar cases is that the developer failed to specify valid input, and so validity checking devolved to the kernel or another component. The component (or composition of components) to which input checking falls in these cases tends to be unaware of the security requirements or high-level notions of acceptable input, and so a breach is inevitable.

b) Non-Minimalist Input-handling Code: Input-handling code should be minimalist in computing power. A regular language should be handled by a finite automaton implementation, not by a pushdown one, nor by a more powerful model. This precept is related in many ways to the need to avoid shotgun parsing and complex input languages, but is distinct from both: it is possible to have an overwrought parser that validates input before processing and accepts a language that is expressible by a simple grammar.

Initial input-handling code should do nothing more than consume input, validate it (correctly), and deserialize it. Bugs related to any computing power present in input-handling code that is over the bare minimum required by the language fall into this category. Computational power exposed at a validator is power and privilege given to the attacker, and must be minimized.

We note that the structure of the parser code is important for enabling meaningful security audits regardless of the parser's backend mechanism. For example, it took over 10 years to discover a chunk-length integer overflow bug, CVE-2012-2028, in Nginx's handling of the HTTP Chunked Encoding, even though a very similar bug in Apache, CVE-2002-3092, was thoroughly understood in 2002! This is despite Nginx's HTTP parser being explicitly structured as a hand-coded finite automaton—but the inputs and states of this automaton were all mixed together for all the grammar elements, and thus thoroughly unintelligible to auditors.[3]

One important class of problems falling into this category is the exposed reflection antipattern in the Java platform. For example, consider the Elasticsearch vulnerability CVE-2015-1427 [51]. The vulnerability arose from the use of the JVM language Groovy as a scripting language for user-originated query scripts. Developers attempted to sanitize the scripts. [4] Unfortunately, reflection was still allowed because the parser failed to constrain the input language to the minimum effective sublanguage[5], leading to unrestricted remote code execution. Certainly, one might view this as another instance of the dreaded missing check, but we posit the problem is more systemic than this. A minimalist domain-specific language used instead of Java or Groovy would almost certainly not have had this problem.

The litany of XML parser vulnerabilities also tends toward this category. The canonical mitigation for all such problems, systemized by Späth et al., is to limit the capabilities of the parser [46], making it more minimalist.

c) Input Language More Complex than Deterministic Context-Free: We recommend not letting language complexity go above deterministic context-free (DCF) first and foremost because of the issue of parser equivalence. Most systems these days contain not one, but several parser implementations for the same protocol; it is an implicit requirement for correctness and often security that these implementations be equivalent in how they interpret the protocol's messages. When testing equivalence, automation is desirable—but syntactic complexity beyond DCF sets a sharp theoretical limit to what can be achieved algorithmically.

As language complexity moves up the Chomsky hierarchy, it becomes harder to reason about a parser's behavior, such as whether it validates its inputs *correctly*. Rice's theorem [38] already tells us that non-trivial properties are undecidable over arbitrary programs, so we must naturally seek specialized ways to ensure that a parser accepts only valid inputs, such as constructing the parser from a formal grammar, as an automaton of the right class for that grammar on the Chomsky scale. Yet even when starting from formal grammars, an implementation's accepted language may be hard to reason about.

Notably, it is still generally undecidable whether two context-free grammars (i.e., parsers) correspond to the same

[3]The vulnerable parser, `ngx_http_parse.c`, contained 57 switch statements with 272 single-character clauses in 2.3K SLOC. Even though such code can be fast, it should not be hand-written!

[4]This, only after the developers of Elasticsearch discovered in CVE-2014-3120 [6] that allowing users to specify arbitrary code for execution in a query leads to a remote code execution vulnerability. Use of general purpose languages in this way is problematic for many reasons–they also tend to be more complex than deterministic context-free to parse, and tend to have many surprising features. Attempts to sandbox them lead to an endless fount of sandbox escapes.

[5]We posit that doing so is impractical in cases where it is not impossible.

language [28]. Thus it may be impossible to determine whether a given implementation is equivalent to another given implementation, or indeed even to the specification. We identify this theoretical result as a leading root cause for parser differentials such as those found in SSL implementations [32].

However, grammar equivalence *is* decidable for the *deterministic* context-free languages [43]. This class is generally well-studied and consists precisely of the LR-parsable languages [34]; standard algorithms such as LALR and others parse large subclasses of it very efficiently. Moreover, context-free grammars form an accessible language for specification, and parsers can be conveniently generated from them with existing software.

For a vulnerability that may be attributed to language complexity, consider CVE-2013-2729. An out-of-bounds memory access occurs in Adobe Reader when a BMP file using run-length encoding contains invalid *movement deltas*, operations that move the "cursor" in the output buffer. The implementation performed no bounds check, allowing pixel data to be written to arbitrary memory locations. Although it is tempting to assign blame to the missing bounds check, we ask why it was forgotten in the first place. Our answer is that these operations are highly context-sensitive, making it difficult to reason about their correct implementation. Complexity analysis would have highlighted them for scrutiny.

To see how movement deltas lift the language into context-sensitivity, note that, in a context-free setting, results would be generated sequentially, and the cursor could be considered an internal part of the abstract and individually verifiable parsing algorithm. Explicit movement operations turn this output pointer into mutable state at the application level; the input language is no longer free of this context. Control over this pointer given to user input requires more power to reason about its validity, leading to the missed check, and thus exploitability.

A related, yet distinct, issue is excessive computational power in the *semantics* of an input language. Obvious instances are Javascript and similar "scripting" languages embedded in input formats. Here we meet the halting problem and the full weight of Rice's theorem. Malicious code is impossible to detect with certainty.

An example of a vulnerability resulting from this is the recent Ethereum vulnerability allowing exploitation of "The DAO," the altcoin's version of a mutual fund or investment bank. Cryptocurrencies generally include a notion of a "transaction" or "contract," in the form of a set of instructions. In the case of Ethereum, this set of instructions had a unique property: deliberate Turing-completeness [7]!

Cryptocurrency contracts and their semantics should be viewed as protocols, since they are intended to describe instructions within specific bounds for other parties to read and perform. In the case of Ethereum, the Turing-completeness of its contracts, and in particular the allowance for recursion, allowed massive theft of funds [18, 15].

Ethereum's attempt to limit computational power through the "gas" mechanism [1]—which assigned a cost to individual operations but placed no limit on the complexity of program semantics—was ineffective. Unlimited complexity resulted in unanticipated program semantics; the currency collapsed, and was ultimately hard-forked in an attempt to annul the heist [16], while re-architecture of The DAO is ongoing [9].

For further evidence that Turing-complete input languages tend to lead to vulnerability, consider the catastrophic (that is, CVSSv2 base score 9.3) vulnerability presented in 2013 by Julia Wolf, CVE-2011-3402 [50]. TrueType fonts contain a Turing-complete bytecode language; following exploitation of a related memory corruption vulnerability, the "glyph program" used to describe scaling of glyphs could manipulate kernel-mode memory. It provides computational primitives needed for exploitation right there in the intended program functionality. It is worth noting that although in this case the font format exploited was Turing-complete, the functionality required by the exploit in Wolf's detailed walkthrough is no more than that of a linear-bounded automaton. This makes it a useful example of the need to absolutely minimize computational power of (weird) machines in program semantics.

d) Differing Interpretations of Input Language: Whether or not an input language is complex, different programs, different implementations of the same input language, and even different components of the *same program in the same runtime context* can interpret input differently, both from each other and from the specification. The result is that input produced by a trusted entity might become malicious on interpretation, or that validation methods such as application firewalls will fail to mask out invalid input. A correctly written parser is essentially equivalent to an application firewall [45].

An excellent example of this weakness is the series of bugs collectively known as the Android Master Key bugs [41, 40, 42]. In these, different components of the Android install chain—namely, the Java-based cryptographic signature verifier and the C++-based installer—disagreed in the interpretation of the ZIP-ed package contents, resulting in the attacker's ability to install entirely different contents than what was verified. The remedy eventually included handling package input data with the *same* parser.

An earlier but arguably higher-impact example was provided by Kaminsky, Sassaman, and Patterson [32], introducing the concept of *parser differentials* and demonstrating over 20 of these between the different libraries used by the X.509 SSL infrastructure libraries at the time. By manipulating the X.509 inputs, these attacks created different views of the apparently benign Common Name (CN) in Certificate Signing Requests (CSR) as seen and signed by a Certificate Authority (CA), and the same CN in the CA-signed certificates as seen by a browser using a different SSL library. Specifically, the browser would see a high-value CN instead of the benign obscure CN—and thus trust a malicious site that submitted the crafted CSR. For instance, CVE-2009-0408 [31] was a critical vulnerability of this type with respect to the Netscape PKI and TLS library `libnss`, leading to clients reading as signed certain properties not actually intended by the issuing CA. The vulnerability caused `libnss` to interpret certain certificates

as authenticating properties that the issuing CA did not intend, since the issuing CA and `libnss` interpreted the certificates differently.

e) Incomplete Protocol Specification: Attempting to write equivalent parsers is of course impossible if the language itself is ill-defined. For example, consider OpenSSL CVE-2016-0703, a high-severity OpenSSL bug involving an obsolete method of negotiating the client master key wherein part of it is sent in the clear. The protocol specification indicates how to handle "clear key bits", but says little about permitted scenarios and usages for them [27]. This specification-level incompleteness coupled with a faithful implementation of the protocol led directly to exploitability.

Another vulnerability, this time in `libnss`, allowed remote code execution on certificate validation. Cited as CVE-2009-2404 [36], this critical vulnerability resulted from a simple heap buffer overflow in `libnss`, related to a failure to allocate correct buffer size—or did it?

Our preferred interpretation addresses a different and more systemic aspect of this vulnerability. The flaw rests in the processing of a non-standard Netscape certificate syntax that uses regular expressions to define which hostnames a certificate is valid for. The specification is very difficult to locate (and may not exist at all), but it is clear to see how this extension ambiguates the X.509 subset used in this type of authentication. This ambiguity resulted in an interaction between null characters and regular expressions, which could have been prevented if a specification addressed whether null characters or regular expressions had their normal meanings in the CN component of the certificate subject DN.

f) Overloaded Field in Input Format: The reuse of data fields for different purposes can be a good indication of ad-hoc constructions or hasty additions—an obvious road to complexity and mistakes. On the other hand, consider a benign grammar such as the following:

$$\begin{aligned} S &\rightarrow \ '1'\ time \ \mid \ '2'\ count \\ time &\rightarrow \ 32bit \\ count &\rightarrow \ 32bit \end{aligned}$$

Here, the second field of the message is "reused" only in the sense that it occupies the same space in both forms.

Although it does not necessarily raise language complexity, overloading can serve as valuable circumstantial evidence. A classic method [26] for exploiting memory corruptions in the Windows low-fragmentation heap made use of a *chunk relocation offset* to elevate small buffer overflows to arbitrary writes. The feature was activated by placing the special value 5 in a field otherwise used as a byte counter.

Another high-impact vulnerability of this class relates to an NTP authentication bypass discovered in 2015 [24]. The field indicating which cryptographic key to use was overloaded to indicate "authentication not required"—which any attacker was allowed to assert at will, because the attacker is allowed to specify which key they are using.

g) Permissive Processing of Invalid Input: The traditional "robustness principle" dictates that one should "be liberal in what you accept" [2]. After leading developers to implement vulnerable programs for decades, this principle has attracted considerable discussion [10, 39]. We argue that one should not be liberal, but *definite*—or explicit—about what is accepted.

This class of vulnerability subsumes most cases of failure to validate program input, but in a more general way, since it encompasses both deliberate and accidental instances. Rather than concentrating on the need for programmers to account for and filter all malicious input, we recommend a strict whitelisting approach where the whitelist is generated by a grammar derived from a specification, which describes valid instances of types the program is prepared to accept.

When programs process invalid input instead of discarding it, the consequences can be very similar to shotgun parsing: application state is easily made inconsistent by an attacker who manufactures bad luck and selectively violates the specification. The consequences can also be dire. The well-known "Heartbleed" bug [5] was an instance of this class; heartbeat requests with a shorter payload than the asserted length are certainly not strings in the (extraordinarily complex) TLS protocol grammar, and yet OpenSSL attempted to process them anyway, with disaster the result.

IV. Remedies

There is an unfilled need to mitigate certain types of input processing failures *at the design level*. Where such vulnerabilities share a common root cause, the task of searching for and mitigating them individually is impossible to complete and is also a waste of the resources spent on it. We outline some design principles, mapped in the CWE style to specific vulnerabilities from our taxonomy, that promise to prevent parser bugs from continuing to be the menace that they are.

a) Completely separate input validation from application logic: Avoid writing shotgun parsers. Input should be fully validated by a machine expressible as a deterministic pushdown automaton. Only once validation succeeds should any application logic proceed. Programmers with formal experience can directly express the recognizer, but in practice, basic formats such as JSON, XML, Protocol Buffers, or ASN.1 are common. Only a small number of programs actually have input languages no more constrained than their data interchange format. In this case, input should be validated against a complete schema using bounded state and no more than one stack.

Parsing of a basic interchange format into an internal representation is sometimes referred to as *canonicalization*. Canonicalization and schema validation must exist as discrete steps. If validation logic is intermixed with other functionality or no validation beyond canonicalization is being done, security analysts should make an observation to this effect.

b) Minimize complexity of pre-validation code: This remedy is only possible for the simpler input languages, such as finite, regular, or context-free languages, which occupy the

bottom rungs of the Chomsky syntactic hierarchy. For each of these language classes, the validating parser that accepts only valid inputs and rejects invalid ones (a.k.a. a *recognizer* for the input language) has a well-known structure, ranging from the finite state to the pushdown automata class.

In general, the code responsible for input canonicalization and validation should be constrained to just those functions. This is not the place to introduce application logic. Canonicalization should only deserialize input, and validation should only verify that it matches the defined grammar.

The easiest way to implement this remedy is to make the structure of the parser code in charge of input validation follow the structure of the grammar. We look to parser construction toolkits like Hammer [49] (discussed in V) to allow developers to write code that literally embodies the grammar's productions. Such code will be highly amenable to auditing.

c) Avoid defining complex input languages: It is virtually never necessary to design an input language that cannot be reduced to a syntax validatable by a deterministic push-down automaton.

We recognize that this advice is meaningless to programmers and security analysts without a formal language background; therefore, we distill this recommendation into a simple rule stated in technical terms.

In essence, the most preferable language is one that can be fully validated by a regular expression. Programmers should try to use such languages, but of course this is not always possible, as these languages do not support recursive nesting of data structures.

When such nesting is needed, the best way to limit the corresponding parser complexity is to ensure that if one entity or statement depends on another, they be hierarchically related. To give a concrete example, it is acceptable to have XML tags with meaning and validity dependent on the structure in which they are contained—but not on structures elsewhere about the document, since that would likely promote the grammar to context-sensitive.

In other words, if the input of an application is too complex to be described by BNF (or EBNF or ABNF, without introducing complexity by way of prose values), it is too complex to be safe. In security, *the input data format is the code's destiny.*

ABNF exemplifies a particularly useful tool for this purpose, since it can easily represent length fields through its specific repetition rule [3]. Due to the use of ABNF in specifying Internet protocols, there are many parser generators available that use it, including the multi-language project APG [47]. We strongly recommend that developers use a parser combinator library or a parser generator; this will result in fewer parser differentials, and it centralizes the proof of correctness burden in code specifically designed to generate correct parsers.

Conditions such as allowed protocol state transitions can be excluded from this expression of the input language for simplicity, but should usually be a predicate to validation (that is, it should dictate what subset of the language is valid for any given state).

Checksums, message authenticity codes, and signatures represent another important set of idioms. We do not propose that applications necessarily take all data at face value once the parser accepts it; in many cases this would be self-defeating and would introduce truly excessive complexity into the parser. Attempting to write checksums and cryptographic primitives in terms of LR-grammar production rules is unlikely to produce security gains. However, it is important to represent the fields which store these values in a manner amenable to reliable parsing. Messages which are not accepted by the parser may be rejected without further processing; once the parser accepts the message, the program may have other checks to do.

It nearly goes without saying that general-purpose programming languages are inappropriate for interchange formats. The preceding sections detail several instances where accepting such a language, even in spite of alleged trust (as with fonts) or attempts to sandbox, results in remote code execution. However, the same is *not* true of binary interchange formats–if they follow the guidelines outlined in this paper, those might well be more amenable to secure computing than their human-readable counterparts.

In practice, most systems are not actually universal Turing machines, because they are capable of storing only finite amounts of state. In this, they are closer to linear-bounded automata, which can accept context-sensitive languages [35]. The bounding of state implies that the halting problem for such machines is indeed decidable, but the complexity of so deciding is bounded only by the amount of state to which the linear-bounded automation is limited [13]. Although it is not strictly impossible to reason about such languages, Blum shows that it is very difficult and often impractical.

The most complex languages for which there is a general solution to the equivalence problem are the deterministic context free languages, as mentioned above. A formal system for doing so is given by Sénizergues [44]; though it is hardly practical for software developers to verify implementations in this way, the solution shows that mechanical verification is possible and reasoning about at least some nontrivial properties of the languages is possible.

Our recommendation derives both from this result, and from the undecidability of the grammar equivalence problem for languages more complex than deterministic context free (discussed in *Input Language More Complex than Deterministic Context Free*, above). We further recommend that a parser combinator library or at least a code generator be used to derive the parser directly from the formal specification; this puts the verification-of-correctness problem on the shoulders of the developers of parser generators and parser combinators.

d) Be clear about specifications: The practice of making and following clear specifications will remedy both the differing interpretations of input language and incomplete protocol specification problems.

A clear, unambiguous specification makes it possible to write or generate validators. Where a specification is unclear, it is critical to document—preferably, in the grammar itself.

A complete specification leaves no bytes to chance. Character sets and alphabets must be defined, at least by reference to

a well-defined set, for each field. Each field must be defined in terms of its meaning, allowed content, allowed length, and expected place.

An excellent way to make a specification for a context-free input language is by writing it down in BNF. The exercise will ensure completeness and make the validator extremely simple to write.

e) Avoid overloading fields: Do not use special values in fields to have special meanings. A field's contents should be as straightforward as possible. If, for example, the presence of some field is optional but must be signalled, do not designate a special value of all zeroes or all ones for that field to indicate absence; to do so invites confusion. Instead, create an additional field that indicates whether the optional element is present or not, or where possible, simply infer the presence or absence of the field from the presence or absence of a representation of it (as in XML or JSON).

In general, the correct way to add additional functionality is to add another field. Do not repurpose existing fields, especially where existing implementations might use them in ways that must be deprecated.

f) Do not transparently correct for invalid input: If input does not validate correctly, either because it cannot be canonicalized, required entities are missing, or illegal entities are present, code should not make excuses for the input. For example, input handlers should reject input containing illegal entities, rather than discarding just the invalid portion of the input. They should also reject inputs that are missing terminating sentinels, rather than "helpfully" adding them. Code that accepts invalid input definitionally has functionality beyond the specification, and using input correction code to apply a transformation to input to "activate" a malicious payload is a favorite trick of attackers.

If it is necessary to support a buggy client that sends invalid input, it is preferable to amend the specification (whether for canonicalization or validation) to account for that input formally, rather than treating it as an exception. This more clearly states what the program actually accepts, and better supports design review processes in determining whether the compatibility accommodation presents a security threat.

The corresponding anti-pattern explains why the PDF and Flash formats are the top attack vectors—specifically due to the designed propensity of their parsers to "correct" faulty inputs. This anti-feature both allows attackers to co-opt the "correcting" rewriting mechanism as a part of their exploits, and to avoid detection of their maliciously crafted payloads—because it is never clear to a third-party checker which kinds of malformations are malicious, and which are "benign" due to being "fixable".

V. Hammer and related work

Hammer [49] is a parser construction kit designed to aid developers in applying the LangSec methodology under production constraints in the choice of language (Hammer targets C/C++ and has bindings for Java, Python, Ruby, Perl, Go, PHP, and .NET), and where code generation is not a supportable option. Hammer enables production programmers to write in parser-combinator style, making it obvious which properties of input are expected and checked.

For decades, a principled approach to parsing was presumed synonymous with Yacc and Bison. However, they are tooled for compiler construction to the exclusion of other applications, such as parsing of binary payloads, where the need for secure parsing is the strongest. Their modern successors such as Spicy[6] address binary parsing much better—where code generation is desired and possible. ANTLR comes closest to Hammer in its expressiveness, but is limited to the Java/C# ecosystem.

Hammer does not preclude code generation. Nail [12], a direct offshoot of Hammer, comes with a code-generation step. Still, Hammer supports fully inline programming for the industry environments where it is needed. In this it is similar to Nom [17], a Rust streaming parser combinators library.

Importantly, Hammer separates the parsing algorithm from its uniform API, designed to describe the input language, not the backend. Hammer currently offers five parsing backends for three different classes of languages, with the Packrat algorithm [23] as the most general backend—all through the same API.

VI. Conclusion

The existing orthodoxy of software security analysis that supposes that software could be bug-free if only programmers were more careful and stamped out all the bugs individually is untenable, as abundantly demonstrated by the ongoing software insecurity epidemic. Clearly, a more systemic approach is required.

Our taxonomy provides just such an approach. It offers clear mitigations for some of the most serious and abundant sources of vulnerability that exist today, at a design level, before software is released.

The classes of vulnerability we have taxonomized represent risky behavior that should be treated the same way as issues like many of those noted in CWE-398, *Indicator of Poor Code Quality*. Indeed, we suggest that CWE identifiers be assigned to each of the items in this taxonomy. They are each strong indicators of exploitability, and particularly strong indicators of subtle but severe bugs.

Unless and until we reconceptualize input processing errors as systemic failings of the types indicated in this paper—and clarify the linkage between these types of software issues and actual vulnerability in penetration test and code audit reports—the insecurity epidemic will continue.

References

[1] Gas Fees. Ethereum documentation. http://web.archive.org/web/20160319102705/http://ether.fund/tool/gas-fees.

[2] Requirements for Internet Hosts – Communication Layers. Internet Standard RFC1122, 1989.

[3] Augmented BNF for Syntax Specifications: ABNF. Internet Standard STD68, 2008.

[4] 2011 CWE/SANS Top 25: Monster Mitigations, 2011.

[5] CVE-2014-0160. MITRE CVE, 2014.

[6] http://www.icir.org/hilti/

[6] CVE-2014-3120. MITRE CVE, 2014.

[7] A Next-Generation Smart Contract and Decentralized Application Platform. Ethereum Wiki, 2016. http://web.archive.org/web/20160210235122/https://github.com/ethereum/wiki/wiki/White-Paper.

[8] CVE-2016-0752. MITRE CVE, 2016.

[9] slockit DAO Pull Request 274: Protect against recursive attack. Github pull request, July 2016. https://github.com/slockit/DAO/pull/274.

[10] Eric Allman. The robustness principle reconsidered: Seeking a middle ground. *ACM Queue*, 9(6), June 2011.

[11] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401, 2008.

[12] Julian Bangert and Nickolai Zeldovich. Nail: A Practical Tool for Parsing and Generating Data Formats. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 615–628, Broomfield, CO, October 2014. USENIX Association.

[13] Manuel Blum. Recursive function theory and speed of computation. *Canadian Mathematical Bulletin*, 9:745–749, 1966.

[14] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. Exploit Programming: from Buffer Overflows to Weird Machines and Theory of Computation. *;login:*, pages 13–21, December 2011.

[15] Vitalik Buterin. CRITICAL UPDATE Re: DAO Vulnerability, June 2016. https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/.

[16] Vitalik Buterin. Hard Fork Completed, July 2016. https://blog.ethereum.org/2016/07/20/hard-fork-completed/.

[17] Geoffoy Couprie. Nom, a byte-oriented, streaming, zero-copy, parser combinators library in Rust. In *2nd IEEE Language-theoretic Security & Privacy Workshop*. IEEE SPW, May 2015.

[18] Phil Daian. Analysis of the DAO Exploit. Blog, June 2016. http://web.archive.org/web/20160619113629/http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[19] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno. Cinderella: Turning shabby x.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 235–254, May 2016.

[20] Thomas Dullien. Exploitation and State Machines: Programming the "Weird Machine", revisited, April 2011. Infiltrate Conference.

[21] Ben Edmunds. *Never Trust Your Users. Sanitize ALL Input!* Apress, Berkeley, CA, 2016.

[22] N. W. Filardo. Mitigating Langsec Problems with Capabilities. In *IEEE Security and Privacy Workshops*, 2016.

[23] Bryan Ford. Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 36–47, New York, NY, USA, 2002. ACM.

[24] Matthew Van Gundy. NAK to the Future: NTP Symmetric Association Authentication Bypass Vulnerability, October 2015. http://www.talosintel.com/reports/TALOS-2015-0069/.

[25] William G Halfond, Jeremy Viegas, and Alessandro Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, volume 1, pages 13–15, 2006.

[26] B. Hawkes. Attacking the Vista Heap, November 2008. Ruxcon Conference, Sydney.

[27] Kipp E. B. Hickman. The SSL Protocol. Internet Draft, 1995.

[28] J. E. Hopcroft. On the equivalence and containment problems for context-free languages. *Mathematical systems theory*, 3(2):119–124, 1969.

[29] J. Morris Jr. Types Are Not Sets. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 120–124. ACM, 1973.

[30] Martin Johns, Christian Beyerlein, Rosemaria Giesecke, and Joachim Posegga. Secure Code Generation for Web Applications. In *Engineering Secure Software and Systems: Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings*, pages 96–113, 2010.

[31] Dan Kaminsky. Mozilla NSS NULL Character CA SSL Certificate Bypass, November 2009. https://packetstormsecurity.com/files/82678/Mozilla-NSS-NULL-Character-CA-SSL-Certificate-Bypass.html.

[32] Dan Kaminsky, Len Sassaman, and Meredith Patterson. PKI Layer Cake: New Collision Attacks Against The Global X.509 CA Infrastructure. Black Hat USA, August 2009. http://www.cosic.esat.kuleuven.be/publications/article-1432.pdf.

[33] Navdeep Kaur and Parminder Kaur. Input validation vulnerabilities in web applications. *Journal of Software Engineering*, 8(3):116–126, 2014.

[34] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.

[35] S.-Y. Kuroda. Classes of languages and linear-bounded automata. *Information and Control*, 7(2):207–223, 1964.

[36] Moxie Marlinspike. More Tricks for Defeating SSL in Practice. Blackhat USA 2009, 2009.

[37] Russ McRee. Pta: Practical threat analysis. *Information Systems Security Association*, pages 37–40, 2008.

[38] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[39] Len Sassaman, Meredith Patterson, and Sergey Bratus. A Patch for Postel's Robustness Principle. *IEEE Security & Privacy*, (10):87–91, 2012.

[40] Jay Freeman (saurik). Android Bug Superior to Master Key. http://www.saurik.com/id/18, 2013.

[41] Jay Freeman (saurik). Exploit (& Fix) Android "Master Key". http://www.saurik.com/id/17, 2013.

[42] Jay Freeman (saurik). Yet Another Android Master Key Bug. http://www.saurik.com/id/19, 2013.

[43] Géraud Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *Automata, Languages and Programming: 24th International Colloquium, ICALP '97 Bologna, Italy, July 7–11, 1997 Proceedings*, pages 671–681, 1997.

[44] Géraud Sénizergues. L(A)=L(B)? decidability results from complete formal systems. volume 251, pages 1–166, 2001.

[45] Z. Shaw. Ragel State Charts, 2009. https://zedshaw.com/archive/ragel-state-charts/.

[46] Christopher Späth, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. Sok: Xml parser vulnerabilities. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX, August 2016. USENIX Association.

[47] Lowell D. Thomas. APG, 2016. https://github.com/ldthomas?tab=repositories.

[48] K. Underwood and M. E. Locasto. In Search of Shotgun Parsers in Android Applications. In *IEEE Security and Privacy Workshops*, 2016.

[49] UpstandingHackers, Inc. UpstandingHackers/hammer, Parser combinators for binary formats, in C. https://github.com/UpstandingHackers/hammer.

[50] Julia Wolf. CVE-2011-3402: Windows Kernel TrueType Font Engine Vulnerability (MS11-087). CanSecWest 2013, 2013.

[51] Jordan Wright. Remote Code Execution in Elasticsearch - CVE-2015-1427, 2015. http://jordan-wright.com/blog/2015/03/08/elasticsearch-rce-vulnerability-cve-2015-1427/.