

Guns and Butter: Towards Formal Axioms of Input Validation

Robert J. Hansen *, Meredith L. Patterson[†]
{`rjhansen`, `mlpatter`}@`cs.uiowa.edu`
Department of Computer Science
The University of Iowa
Iowa City, Iowa

June 30, 2005

Abstract

Input validation has long been recognized as an essential part of a well-designed system, yet existing literature gives very little in the way of formal axioms for input validation or guidance on how to put in practice what few recommendations exist. We present basic formal axioms for input validation and apply them to SQL, where we demonstrate enhanced resistance to injection attacks.

blame for this sorry state of affairs on an infrastructure built by developers who are, in general, woefully lacking in knowledge of either the formal methods of computer science or the recommendations of the computer security community, a good portion of the blame must rest on the academic treatment of the subject. In our literature survey we found very little information about formal axioms of input validation, and of that, almost none featured examples to show implementors how these best practices could be applied to their work.

1 The Importance of Input Validation

A very large class of attacks against systems are really input validation attacks. Buffer overflows are exploited by inputs which are larger than the memory space allotted for them. SQL injection uses carefully-manipulated inputs to trick servers into running semi-arbitrary code using the server's own permissions. Many more exist; one need only read `comp.risks` [1] to find them.

While the academic world is tempted to pin the

*Mr. Hansen wishes to thank Professor Leon Tabak of Cornell College and Professor Doug Jones of the University of Iowa for assistance.

[†]Ms. Patterson wishes to thank David McNab, the author of PyBison.

2 Intended Audience

We hope that both theoreticians and implementors will benefit from this paper. In order to accommodate both audiences, we have elaborated some things which are self-evident to one group in the interests of making it more comprehensible to the other. Wherever possible, we use a uniform terminology: for instance, while context-free grammars and pushdown automata are equivalent and interchangeable, in this paper we will primarily refer to pushdown automata.

3 Existing Practices

3.1 Real-World Practice

Existing real-world validation processes are largely ad-hoc and/or taken from believed-authoritative references. Unfortunately, many of these references completely fail to address input validation. Those that do tend to do so in little detail, advocate the use of inferior mechanisms (largely regular expressions), and give a false sense of tradeoffs between security and convenience. This “guns or butter” mentality seems endemic to the literature, as shown in the following quote from Welling [2]:

Be careful when making things more restrictive, though, because a validation function that rejects one percent of valid data is far more annoying than one that allows through ten percent of invalid data.

In our random survey of six commonly-available texts, three said nothing of input validation; one referred to it but did not discuss it; and only two suggested the use of regular expressions to validate user input. Welling was one of the two, which should be further indicative of the poor state of security awareness: after all, bad advice is considerably worse than none.

3.2 Academic Practice

Our academic literature survey yielded results not much more impressive. Most of the existing literature is concerned with handling errors after bad input has occurred, not proactively ensuring bad input doesn't occur.

Furthermore, the problem of syntactically valid but semantically malicious input is broadly considered intractable. The literature which addresses this generally cites Beizer [3], a reading of which suggests the author did not feel the field possessed the tools necessary to address the problem:

Data validation is the first line of defense against a hostile world. Good designers will design their systems so that it doesn't just

accept garbage—good testers will subject their systems to the most creative garbage possible. Input-tolerance testing is usually done at the system level ... and consequently, the tester is usually distinct from the designer.

...

Every input to a system or routine can be considered as if it were a string of characters. The system accepts valid strings and rejects invalid ones. ... There's nothing we can do about syntactically valid strings, all of whose values are reasonable but just plain wrong—that kind of garbage we have to accept.

We courteously disagree with the preceding. Data validation is, indeed, the first line of defense; for that reason, it is imperative that data validation be incorporated into a design, not a tacked-on feature tested in an ad-hoc manner. Moreover, as we will demonstrate in this paper, data validation can be subjected to formal syntactic analysis which will trap many kinds of malicious inputs.

3.3 Conclusions

We conclude from our survey the following:

1. Existing literature does not sufficiently address input validation
2. Existing literature provides no useful axioms for input validation
3. The state of the art, as far as implementation goes, is regexp-based validation

In this paper we assume the first, address the second, and demonstrate the insufficiency of the third as we develop axioms of input validation and apply them to SQL.

4 Framework

4.1 Definitions and Observations

Our definitions and observations should be unsurprising to anyone with a background in theoretical computer science, and for that reason we will eschew most elaboration. Implementors may find Sipser [4] and/or Fleck [5] useful when reading this section.

Definition 1 (Validity) *A string S in a language L is valid if and only if S can be generated from the formal description of L .*

Definition 2 (P -languages) *A language L is a P -language if and only if all strings generated by L exhibit the property P .*

Observation 1 (Validity of Sublanguages) *All strings in the P -language L' derived from L are valid P -strings in L , regardless of whether L is a P -language.*

This observation, while quite general, has practical consequences. By constructing a properly constrained sublanguage of an insecure language, we may generate secure strings within that insecure language.

Definition 3 (M -set) *The M -set of languages is composed of all languages which require a mechanism of minimum strength M for their generation.*

Definition 4 (Syntactic Context) *A syntactic context for an input I in a string S is the maximum extent in S to which I can change the syntactic structure of S , up to an upper bound of S itself.*

We will denote the syntactic context of S for a given input I with the notation Υ_I^S . Variables are handled by observing that for any variable V defined in its own syntactic context Σ^T , any syntactic context Σ^S which references V will incorporate Σ^T .

Definition 5 (Semantic Context) *A semantic context for an input I in a string S is the maximum extent in which I can change the semantic meaning of S , up to an upper bound of S itself.*

We will denote the semantic context of S for a given input I with the notation Υ_I^S . Our discussion of Υ_I^S will be limited. Semantic analysis is beyond the scope of this paper except as it can be facilitated via syntactic analysis.

To the best of our knowledge, we are the first to formally specify the meaning of syntactic and semantic contexts, although many computer scientists have for years had an intuitive and/or ad-hoc sense of the same.

4.2 Input Validation Theorems

Our theorems pertaining to input validation should be as unsurprising as our preceding definitions and observations.

Theorem 1 (Minimum Validation Strength)

Given a language L which requires a minimally-strong computational mechanism M to generate strings $\in L$, a string $S \in L$ must be validated using a mechanism at least as strong as M .

Proof. Since validating a string against a language is done by applying the production rules of the language to create the string being validated, if it is possible to validate S using a mechanism weaker than M , then M is not the minimally-strong mechanism required to generate L 's productions. This contradicts our statement that M is minimally-strong.

Observation 2 (Maximal Validation Strength)

A language L requiring a minimally-strong computational mechanism M should not be validated using a mechanism stronger than M .

Hoglund and McGraw [6] note that “a complex computational system is an engine for executing malicious computer programs delivered in the form of crafted input.” Their concern—the inevitable compromise, given enough time, of systems—informs our observation. If we assume our systems will be compromised at some point, prudence requires that we expose to our attackers the weakest mechanism possible. An attacker given access to a strong mechanism has far more potential for mayhem than one using a weaker mechanism.

Theorem 2 (Range of Validation) *Given an input I and a string S , validation must occur over at least the range Σ_I^S .*

Proof. Since Σ_I^S is defined as the greatest extent to which I has the capability to influence the syntactic meaning of S , validating less than Σ_I^S may result in failure to validate all the syntactic changes I may introduce to S .

Validation of only I is sufficient in only those cases where it can be proven $\Sigma_I^S = I$.

Lemma 1 (Construction of Sublanguages)

Given pushdown automata G and H , where $H = \langle V, T, R, S \in V \rangle$, $L(G)$ is a sublanguage of $L(H)$ if $V(G) \subseteq V(H)$, $T(G) \subseteq T(H)$, $R(G) \subseteq R(H)$ and $S(G) = S(H)$.

Proof by Construction. For any string w in $L(G)$ \exists a production $P(w) \in G$. All variables, terminals and rules used in $P(w)$ are correspondingly in G . $S(P(w)) = S(G)$. $V(P(w))$ must necessarily exist in $V(G)$ which as a given exists in $V(H)$. Similar arguments exist for T, R, S . Thus, $P(w) \in H$ and $w \in L(H)$. Consequently, since any arbitrary string in $L(G)$ is also a string in $L(H)$, $L(G)$ is a sublanguage of $L(H)$.

Observation 3 (Mechanism Equivalence)

Any regular expression, finite-state automata or context-free grammar can be trivially converted into an equivalent pushdown automata, and thus fall under Lemma 1.

5 Axiomatic Input Validation

The conventional wisdom for validating user inputs is to use regular expressions to either filter out bad input or only allow in good inputs. A schism exists among developers over which is the correct (safe) way to use regexp validation: whether it is better to use regexps to only allow inputs known to be good or to use them to forbid inputs known to be bad. With respect to those engaged in such debate, we believe this misses the more important point: for a majority of input situations, regexps are computationally insufficient for input validation.

The consequences of using insufficient mechanisms are self-evident to computer scientists, but perhaps not to implementors. Attempts to validate Σ_I^S using a mechanism weaker than M will often fail to recognize invalid strings.

5.1 Guns or Butter

For instance, a regexp cannot be created which will reliably match $a^m b^n$; there will always be some string $a^m b^n$, $m \neq n$, which will pass this regular expression¹. Attempting to massage regexps into handling this language will inevitably lead to friction between what the regexp can do, what developers think it can do, and what users need it to do.

Those who say we must err on the side of safety will say we must only admit as inputs those which the regexp can verify as valid, and accept the false-positive rate (and its concordant inconvenience) as a necessary expense of security. Those who say we must err on the side of convenience will say we must only exclude those inputs which are known to be bad, and accept the false-negative rate (and its concordant insecurity) as a necessary expense of convenience.

Curiously, neither side appears to recognize what we consider to be self-evident: that whether an error is made on the side of safety or on the side of convenience, it remains an error.

5.2 Guns and Butter

Shifting to a more appropriate tool—in our $a^m b^n$ example, a pushdown automata—allows us to validate inputs as good or bad with perfect accuracy. No longer do we have to make the tradeoff between security and convenience; we can have both guns and butter. The mechanism involved is to create a subset of the command language in which it is only possible to generate secure strings. Lemma 1 gives us a mechanism for generating subsets of context-free languages and Observation 3 allows us to extend the lemma to regular languages and finite-state automata.

¹A note to implementors: regexps are not allowed to reference themselves. You can construct a Perl expression which will match $a^m b^n$ through self-reference, but the addition of self-reference keeps it from being a regexp.

The significance of these results should be sufficiently obvious that further elaboration is unnecessary.

Implementors are again referred to Fleck and Sipser for more complete discussions of computational mechanisms and their sufficiencies for given tasks.

6 Applying Axioms to SQL

The Structured Query Language, SQL, is an ANSI/ISO standard language for interacting with databases. Command strings are input to the database, wherein the string is parsed according to the rules of a pushdown automata or equivalent.

SQL is an extraordinarily complex language which offers tremendous flexibility, a flexibility which can be used against implementors as easily as implementors can use it against the problem domain. In this section we will show how formal axioms of input validation can give improved resistance to a certain class of attacks.

6.1 SQL Injection

SQL *injection* is a class of attacks against software connected to a database. An innocuous-looking statement, such as "select rights from table where username = I_1 and password = I_2 ", can be subverted with carefully-chosen I_n values.

For instance, if I_1 is "root" and I_2 is the string "' or '1' = '1'", the final statement passed to the database is "select rights from table where username = 'root' and password = "' or '1' = '1'". This malicious and malformed boolean expression will evaluate to true and root's rights will be granted.

The regexp-based solution to this is to restrict what letters are allowed as input. However, this is done against the wisdom of Theorems 1 and 2, which say we need to validate against Σ_I^S using a mechanism at least as strong as that used to generate valid command strings.

We do not know what the user will input, and as such, predicting Σ_I^S is problematic. However, using

the commonsense observation that $\Sigma^S \leq S$, we elect to validate over the entire command string.

So far, we have not validated anything. If we were to put the injected SQL string through an SQL parser, it would parse correctly. This means it is valid in a language sense; however, the injected SQL is not valid in a security sense. To validate it from a security perspective requires us to define a sublanguage of SQL which only generates safe strings.

Fortunately, defining a safe subset of SQL for this case is very simple. We define, using Lemma 1, our own sublanguage of SQL wherein the only production rules allowed are those which generate the sequence "select rights from table where username='TERMINAL-STRING' and password='TERMINAL-STRING'", where TERMINAL-STRING denotes a sequence of characters which derives no keyword or other syntactic structure.

We take whatever command string C we create from splicing together our command string and the user input, then see if C can be validated against our known-safe sublanguage, using a tool of sufficient computational strength to produce all valid strings in that sublanguage. In this case, since we know a pushdown automata is enough to generate SQL strings and our known-good sublanguage generates valid SQL strings, we know a pushdown automata is sufficient for our purpose. If C is a valid string in our sublanguage, the input is good. If it is not, the input is bad.

By so doing, we are secure against SQL injection attacks.

7 Cautions and Criticisms

We have not introduced a silver bullet. We mention this as a caution to those who are looking for a miraculous solution to the input-validation problem. This is not that; look elsewhere.

Instead of putting the burden on programmers to come up with regexps that validate input, we now put the burden on programmers to come up with restricted sublanguages that validate input. The programmer's life is not made easier by our results. However, we believe that working towards an achievable

goal beats Sisyphean labor.

Bad sublanguages can be created which do not meet the programmer's security expectations. While this can likely be ameliorated through expert systems and AI techniques, in the main this problem is undecidable.

The more complex the computational mechanisms involved become, the easier it is to run into the limits of computational decidability. We do not believe these axioms to be a practical way, at this time, for the validation of Turing-set inputs.

8 Source Code

We have implemented a (simple) Python webserver and database environment which implements our concept of sublanguage validation. It can be found at [INSERT URL HERE].

9 Further Research

Further research into the axiomatics of input validation seems appropriate, given the simplicity of our existing axioms.

The relationship between S , I , Σ_I^S and Υ_I^S is poorly-defined, leading to our recommendation of validating over the entirety of S . Interested parties should contact us for information on our null hypotheses in this matter.

Much work remains to be done in determining what constitutes a properly-constrained sublanguage. It is likely this will be extremely sensitive to context.

Research into AI techniques, expert systems and/or other computer-aided constraint set generation methods may be fruitful.

Finally, since any program may be viewed as an input given to a Universal Turing Machine, research into input-validation techniques may give insights into software verifiability. Obvious results, such as the Halting Problem, can be easily rederived from these input-validation axioms.

10 Concluding Remarks

It's said the best defense against a punch is not to be there when it lands. The same principle applies to computer security: the best defense against malicious inputs is to prevent them from being input.

We recommend the following practices to implementors who work with input validation:

1. Validate the entire context in which an input appears, not the input itself
2. Validate the context using a mechanism strong enough to parse the context
3. Stronger is not better; use the weakest mechanism which will do the job
4. Use problem-specific sublanguages to validate inputs
5. Eschew both "allow-everything" and "deny-everything" heuristics except as last resorts. In the event these heuristics become necessary, consider it unambiguous proof of the failure of the validation mechanism.

References

- [1] `comp.risks`, a USENET newsgroup.
- [2] Luke Welling and Laura Thomson, *PHP and MySQL Web Development*. Sams Publishing, 2005.
- [3] Boris Beizer, *Software Testing Techniques*. Van Nostrand Reinhold, 1983.
- [4] Michael Sipser, *Introduction to the Theory of Computation*. PWS Publishing, 1997.
- [5] Arthur Fleck, *Formal Models of Computation*. World Scientific, 2001.
- [6] Greg Hoglund and Gary McGraw, *Exploiting Software: How to Break Code*. Addison-Wesley, 2004.