

# A Patch for Postel's Robustness Principle

**Len Sassaman** | Katholieke Universiteit Leuven  
**Meredith L. Patterson** | Red Lambda  
**Sergey Bratus** | Dartmouth College

On Postel's Robustness Principle—"Be conservative in what you do, and liberal in what you accept from others"—played a fundamental role in how Internet protocols were designed and implemented. Its influence went far beyond direct application by Internet Engineering Task Force (IETF) designers, as generations of programmers learned from examples of the protocols and server implementations it had shaped.

However, we argue that its misinterpretations were also responsible for the proliferation of Internet insecurity. In particular, several mistakes in interpreting Postel's principle lead to the opposite of robustness—unmanageable insecurity. These misinterpretations, although frequent, are subtle, and recognizing them requires closely examining fundamental concepts of computation and exploitation (or equivalent intuitions). By discussing them, we intend neither an attack on the principle nor its deconstruction, any more than a patch on a useful program intends to slight the program. Our intention is to present a view of protocol

design that helps avoid these mistakes and to "patch" the principle's common formulation to remove the potential weakness that these mistakes represent.

## Robustness and Internet Freedom

Postel's principle acquired deep philosophical and political significance—discussed, for instance, in Dan Geer's groundbreaking essay "Vulnerable Compliance."<sup>1</sup> It created a world of programming thought, intuition, and attitude that made the Internet what it is: a ubiquitous, generally interoperable system that enables the use of communication technology to further political freedoms.

Yet this world of revolutionary forms of communication faces an insecurity crisis that erodes users' trust in its software and platforms. If users continue to see Internet communication platforms as weak and vulnerable to push-button attack tools that are easily acquired by a repressive authority, they will eventually become unwilling to use these platforms for important tasks.

The world of free, private

Internet communication must go on, and we must reexamine our design and engineering principles to protect it. Geer makes a convincing practical case for reexamining Postel's principle from the defender's position; Len Sassaman and Meredith L. Patterson arrived at a similar conclusion from a combination of formal-language theory and exploitation experience.<sup>2</sup>

## Robustness versus Malevolence

Postel's principle wasn't meant to be oblivious of security. For example, consider the context in which it appears in the IETF's Request for Comments (RFC) 1122, Section 1.2.2 "Robustness Principle":<sup>3</sup>

At every layer of the protocols, there is a general rule whose application can lead to enormous benefits in robustness and interoperability [IP:1]:

"Be liberal in what you accept, and conservative in what you send."

Software should be written to deal with every conceivable error, no matter how unlikely; sooner or later a packet will come in with that particular combination of errors and attributes, and unless the software is prepared, chaos can ensue. In general, it is best to assume that the network is filled with malevolent entities that will send in packets designed to have the worst possible effect. This assumption will lead to suitable protective

design, although the most serious problems in the Internet have been caused by unenvisioned mechanisms triggered by low-probability events; mere human malice would never have taken so devious a course!

This formulation of the principle shows awareness of security problems caused by lax input handling misunderstood as “liberal acceptance.” So, reading Postel’s principle as encouraging implementers to generally trust network inputs would be wrong.

Note also the RFC’s statement that the principle should apply at every network layer. Unfortunately, this crucial design insight is almost universally ignored. Instead, implementations of layered designs are dominated by implicit assumptions that layer boundaries serve as “filters” that pass only well-formed data conforming to expected abstractions. Such expectations can be so pervasive that cross-layer vulnerabilities might persist unnoticed for decades. These layers of abstraction become boundaries of competence.<sup>4</sup>

### Robustness and the Language Recognition Problem

Insecurity owing to input data handling appears ubiquitous and is commonly associated with message format complexity. Of course, complexity shouldn’t be decried lightly; progress in programming has produced ever-more-complex machine behaviors and thus more complex data structures. But when do these structures become too complex, and how does message complexity interact with Postel’s principle?

The formal language-theoretic approach we outline here lets us quantify the interplay of complexity with Postel’s principle and draw a bright line beyond which message complexity should be discouraged by a strict reading of the principle.

We then offer a “patch” that makes this discouragement more explicit.

### The Language-Theoretic Approach

At every layer of an Internet protocol stack, implementations face a *recognition* problem—they must recognize and accept *valid* or *expected* inputs and reject malicious ones in a manner that doesn’t expose their recognition or processing logic to exploitation. We speak of *valid* or *expected* inputs to stress that, in the name of robustness, some inputs can be accepted rather than rejected without being *valid* or *defined* for a given implementation. However, they must be *safe*—that is, not lead the current layer or higher layers to perform a malicious computation or exploitation.

In previous research, we showed that, starting at certain message complexity levels, recognizing the *formal language*—which is made up by the totality of *valid* or *expected* protocol messages or formats—becomes *undecidable*.<sup>5,6</sup> Such protocols can’t tell *valid* or *expected* inputs from *exploitative* ones, and exploitation by crafted input is only a matter of exploit programming techniques.<sup>7</sup> No 80/20 engineering solution for such problems exists, any more than you can solve the Halting Problem by throwing in enough programming or testing effort.

For complex message languages and formats that correspond to context-sensitive languages, full recognition, although decidable, requires implementing powerful automata, equivalent to a Turing machine with a finite tape. When input languages require this much computational power, handling them safely is difficult, because various input data elements’ validity can be established only by checking bits of context that might not be in the checking code’s scope. Security-minded programmers understand

that each function (or basic block) that works with input data must first check that the data is as expected; however, the context required to fully check the current data element is too rich to pass around. Programmers are intimately familiar with this frustration: even though they know they must validate the data, they can’t do so fully, wherever in the code they look. When operating with some data derived from the inputs, programmers are left to wonder how far back they should go to determine if using the data as is would lead to a memory corruption, overflow, or hijacked computation. The context necessary to make this determination is often scattered or too far down the stack. Similarly, during code review, code auditors often have difficulty ascertaining whether the data has been fully validated and is safe to use at a given code location.

Indeed, second-guessing developers’ data safety assumptions that are unlikely to be matched by actual ad hoc recognizer code (also called *input validation* or *sanity checking* code) has been a fruitful exploitation approach. This is because developers rarely implement full recognition of input messages but rather end up with an equivalent of an underpowered automaton, which fails to enforce their expectations. A familiar but important example of this failure is trying to match recursively nested structures with regular expressions.

“Liberal” parsing would seem to discourage a formal languages approach, which prescribes generating parsers from formal grammars and thus provides little leeway for liberalism. However, we argue that the entirety of Postel’s principle actually favors this approach. Although the principle doesn’t explicitly mention input rejection—and would seem to discourage it—proper, powerful rejection is crucial to safe

recognition. Our patch suggests a language in which the balance between acceptance and rejection can be productively discussed.

## Computational Power versus Robustness

It's easy to assume that Postel's principle compels acceptance of arbitrarily complex protocols requiring significant computational power to parse. This is a mistake. In fact, such protocols should be deemed incompatible with the RFC 1122 formulation.<sup>3</sup>

The devil here is in the details. Writing a protocol handler that can deal with "every conceivable error"<sup>3</sup> can be an insurmountable task for complex protocols, inviting further implementation error—or it might be impossible.

This becomes clear once we consider protocol messages as an input language to be recognized, and the protocol handler as the recognizer automaton. Whereas for regular and context-free languages as well as some classes of context-sensitive languages, recognition is decidable and can be performed by sub-Turing automata, for more powerful classes of formal languages, it's generally undecidable.

In the face of undecidability, dealing with every conceivable error is impossible. For context-sensitive protocols requiring full Turing-machine power for recognition, it might be theoretically possible but utterly thankless. These complex protocols, hungry for computational power, should be deemed incompatible with Postel's Robustness Principle.

Robust recognition—and therefore robust error handling—is possible only when the input messages are understood and treated as a formal language, with the recognizer preferably derived

from its explicit grammar (or at least checked against one). Conversely, no other form of implementing acceptance will provide a way to enumerate and contain the space of errors and error states into which crafted inputs can drive an ad hoc recognizer. Indeed, had this problem been amenable to an algorithmic solution, we would have solved the Halting Problem.

## Clarity versus Ambiguity in the Presence of Errors

It's also easy to assume that, no matter the protocol's syntax, Postel's principle compels acceptance of ambiguous messages and silent

A strict reading of the last sentence would forbid ambiguity (non-clarity) of "meaning." However, deciding a packet's meaning in the presence of any particular set of "technical errors" can be tricky, and some meanings might be confused for others, owing to errors. So, what makes a protocol message's meaning clear and unambiguous, and how can we judge this clarity in the presence of errors?

This property of nonambiguity can't belong to an individual message of a protocol. To know what a message can be confused with, we need to know what other kinds of messages are possible. So, clarity must be a property of the *protocol as a whole*.

We posit that this property correlates with the non-ambiguity of the protocol's grammar and, generally, with its ease of parsing. It's unlikely that the parser of a hard-to-parse protocol can be further burdened with fixing technical errors without introducing the potential for programmer error.

Thus, clarity can be a property of only an easy-to-parse protocol.

As before, consider the totality of a protocol's messages as an input language to be recognized by the protocol's handler (which serves as a de facto recognizing automaton). Easy-to-parse languages with no or controllable ambiguity are usually in regular or context-free classes.

Context-sensitive languages require more computational power to parse and more state to extract the message elements' meaning. So, they're more sensitive to errors that make such meaning ambiguous. *Length fields*, which control the parsing of subsequent variable-length protocol fields, are a fundamental example. Should such a field be damaged, the rest of the message bytes will likely be misinterpreted before the whole

**In the face of undecidability, dealing with every conceivable error is impossible. ... Complex protocols, hungry for computational power, should be deemed incompatible with Postel's Robustness Principle.**

"fixing" of errors. This is also a mistake. Prior formulations, such as IETF RFC 761, clarify the boundary between being accepting and rejecting ambiguity:<sup>8</sup>

The implementation of a protocol must be robust. Each implementation must expect to interoperate with others created by different individuals. While the goal of this specification is to be explicit about the protocol there is the possibility of differing interpretations. In general, an implementation must be conservative in its sending behavior, and liberal in its receiving behavior. That is, it must accept any datagram that it can interpret (e.g., not object to technical errors where the meaning is still clear).

message can be rejected thanks to a control sum, if any. If such a sum follows the erroneous length field, it might also be misidentified.<sup>4</sup>

Thus ambiguous input languages should be deemed dangerous and excluded from Postel's Robustness Principle requirements.

### Adaptability versus Ambiguity

Postel's principle postulates adaptability. As RFC 1122 states,<sup>3</sup>

Adaptability to change must be designed into all levels of Internet host software. As a simple example, consider a protocol specification that contains an enumeration of values for a particular header field—e.g., a type field, a port number, or an error code; this enumeration must be assumed to be

incomplete. Thus, if a protocol specification defines four possible error codes, the software must not break when a fifth code shows up. An undefined code might be logged ... but it must not cause a failure.

This example operates with an error code—a fixed-length field that can be unambiguously represented and parsed and doesn't affect the interpretation of the rest of the message. That is, this example of "liberal" acceptance is limited to a language construct with the best formal language properties. Indeed, fixed-length fields make context-free or regular languages; tolerating their undefined values wouldn't introduce context sensitivity or necessitate another computational power step-up for the recognizer.

So, by intuition or otherwise, this example of laudable tolerance stays on the safe side of recognition, from a formal language-theoretic perspective.

### Other Views

Postel's principle has come under recent scrutiny from several well-known authors. We already mentioned Dan Geer's insightful essay; Eric Allman recently called for balance and moderation in the principle's application.<sup>9</sup>

We agree, but posit that such balance can exist only for protocols that moderate their messages' language complexity—and thus the computational complexity and power demanded of their implementations. We further posit that moderating said complexity is the only way to create such balance. We believe that the culprit in the insecurity epidemic and the driver for patching Postel's principle isn't the modern Internet's "hostility" per se (noted as far back as RFC 1122<sup>3</sup>), but modern protocols' excessive computational power greed.

The issues that, according to Allman, make interoperability notoriously hard are precisely those we point out as challenges to the security of composed, complex system designs.<sup>6</sup> We agree with much in Allman's discussion. In particular, we see his "dark side" examples of "liberality taken too far"<sup>9</sup> as precisely the ad hoc recognizer practices that we call on implementers to eschew. His examples of misplaced trust in ostensibly internal (and therefore assumed safe) data sources help drive home one of the general lessons we argue for:<sup>5,6</sup>

Authentication is no substitution for recognition, and trust in data should only be based on recognition, not source authentication.

## IEEE SP 2012

33rd IEEE Symposium on Security and Privacy

20-23 May 2012

Westin St. Francis, San Francisco, CA, USA

Since 1980, the IEEE Symposium on Security and Privacy has been the premier forum for presenting developments in computer security and electronic privacy, and for bringing together researchers and practitioners in the field.

Register today

<http://www.ieee-security.org/TC/SP2012/>





We fully agree with the need for “checking everything, including results from local cooperating services and even function parameters,”<sup>9</sup> not just user inputs. However, we believe that a more definite line is needed for protocol designers and implementers to make such checking work. A good example is the missing checks for Web input data reasonableness that Allman names as the cause of SQL injection attacks. The downstream developer expectations of such reasonableness in combination with data format complexity might place undecidable burdens on the implementer and prevent any reasonable balance from being struck.

### The Postel’s Principle Patch

Here’s our proposed patch:

- Be *definite* about what you accept.
- Treat valid or expected inputs as formal languages, accept them with a matching computational power, and generate their recognizer from their grammar.
- Treat input-handling computational power as a privilege, and reduce it whenever possible.

Being definite about what you accept is crucial for the security and privacy of your users. Being liberal works best for simpler protocols and languages and is in fact limited to such languages. Keep your language regular or at most context free (without length fields). Being more liberal didn’t work well for early IPv4 stacks: they were initially vulnerable to weak packet parser attacks and ended up eliminating many options and features from normal use. Furthermore, presence of these options in traffic came to be regarded as a sign of suspicious or malicious activities to be mitigated by traffic normalization or outright rejection. At current protocol complexities, being liberal actually means exposing your

software’s users to intractable or malicious computations.

**R**eversing the ubiquitous insecurity of the Internet and keeping it free require that we rethink its protocol design from the first principles. We posit that insecurity comes from ambiguity and the computational complexity required for protocol recognition; minimizing protocol ambiguity and designing message formats so they can be parsed by simpler automata will vastly reduce insecurity. Our proposal isn’t incompatible with the intuitions behind Postel’s principle, but can be seen as its stricter reading that should guide its application to more secure protocol design. ■

### Acknowledgments

While preparing this article for publication, we received extensive feedback about both the Postel principle and our patch for it. We asked for permission to publish these letters in their entirety and are grateful for permissions to do so. Find these letters at <http://langsec.org/postel>.

### References

1. D. Geer, “Vulnerable Compliance,” *login*, vol. 35, no. 6, 2010, pp. 26–30; <http://db.usenix.org/publications/login/2010-12/pdfs/geer.pdf>.
2. L. Sassaman and M.L. Patterson, “Exploiting a Forest with Trees,” Black Hat USA, Aug. 2010; <http://langsec.org>.
3. R. Braden, ed., Requirements for Internet Hosts—Communication Layers, IETF RFC 1122, Oct. 1989; <http://tools.ietf.org/html/rfc1122>.
4. S. Bratus and T. Goodspeed, “How I Misunderstood Digital Radio,” submitted for publication to Phrack 68.
5. L. Sassaman et al., “The Halting Problems of Network Stack


Insecurity,” *login*, vol. 36, no. 6, 2011, pp. 22–32; [www.usenix.org/publications/login/2011-12/openpdfs/Sassaman.pdf](http://www.usenix.org/publications/login/2011-12/openpdfs/Sassaman.pdf).

6. L. Sassaman et al., *Security Applications of Formal Language Theory*, tech. report TR2011-709, Computer Science Dept., Dartmouth College, 25 Nov. 2011; <http://langsec.org/papers/langsec-tr.pdf>.
7. S. Bratus et al., “Exploit Programming: From Buffer Overflows to ‘Weird Machines’ and Theory of Computation,” *login*, vol. 36, no. 6, 2011, pp. 13–21.
8. J. Postel, ed., *DoD Standard Transmission Control Protocol*, IETF RFC 761, Jan. 1980; <http://tools.ietf.org/html/rfc761>.
9. E. Allman, “The Robustness Principle Reconsidered: Seeking a Middle Ground,” *ACM Queue*, 22 June 2011; <http://queue.acm.org/detail.cfm?id=1999945>.

**Len Sassaman** was a PhD student in Katholieke Universiteit Leuven’s COSIC research group. His work with the Cypherpunks on the Mixmaster anonymous remailer system and the Tor Project helped establish the field of anonymity research. In 2009, he and Meredith L. Patterson began formalizing the foundations of language-theoretic security. Sassaman passed away in July 2011. He was 31.

**Meredith L. Patterson** is a software engineer at Red Lambda. Contact her at [mlp@thesmartpolitenerd.com](mailto:mlp@thesmartpolitenerd.com).

**Sergey Bratus** is a research assistant professor in Dartmouth College’s Computer Science Department. Contact him at [sergey@cs.dartmouth.edu](mailto:sergey@cs.dartmouth.edu).

 Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.