A Patch for Postel's Robustness Principle

Len Sassaman, Meredith L. Patterson, Sergey Bratus

February 10, 2012

The famous Postel's Principle, a.k.a. Robustness Principle ("Be conservative in what you do, be liberal in what you accept from others"; we discuss this and other formulations later in this article) played a fundamental role in how Internet protocols were designed and implemented. Its influence went far beyond direct application by IETF designers, as generations of programmers learned from examples of protocols and server implementations it had shaped. We argue, however, that its misinterpretations were also responsible for the proliferation of Internet insecurity.

In this essay we discuss several subtle mistakes in interpreting Postel's Principle that are easy to make, and that lead to the very opposite of robustness – unmanageable insecurity as we have come to know it on the Internet. We discuss the view of protocol design that helps avoid these mistakes, and "patch" the Principle's common formulation to remove the potential weakness that these mistakes represent.

We would like to stress that the misinterpretations in question, even though frequent, are subtle, and require close examination of fundamental concepts of computation and exploitation (or equivalent intuitions) to clearly recognize them as such. Thus we intend neither an "attack" on the Principle nor its "deconstruction" – no more than a patch on a useful program intends to slight the program.

Robustness and Internet Freedom

The Postel Principle acquired deep philosophical and political significance — discussed, e.g., in Dan Geer's essay "Vulnerable Compliance" [1] — and created a world of programming thought, intuition, and attitude that made the Internet what it is: ubiquitous, generally interoperable, and enabling the use of communication technology to further political freedoms.

Yet this world of new, revolutionary forms of communication is now facing a crisis of insecurity, which erodes users' trust in its software and platforms. Seeing Internet communication platforms as weak and vulnerable to push-button attack tools easily acquired by a repressive authority will eventually destroy users' willingness to use these platforms for messages that matter.

The world of free and private Internet communication must go on – and we must re-examine our design and engineering principles to protect it – and this includes the applications of Postel's Principle. Geer makes a convincing practical case for re-examination of Postel's Principle, while Sassaman and Patterson arrived at it from a combination of formal language theory and exploitation experience.

Robustness vs malevolence

Postel's Principle as it is commonly known was not meant to be oblivious of security. For example, consider the context in which it appears in RFC 1122:

```
1.2.2  Robustness Principle

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
unenvisaged mechanisms triggered by low-probability events;
mere human malice would never have taken so devious a course!

--- http://tools.ietf.org/html/rfc1122
```

This formulation of the Principle clearly shows awareness of security problems caused by lax input handling misunderstood as "liberal acceptance". It would therefore be wrong to read Postel's Principle as encouraging implementors to generally trust network inputs. Note also the stated application of the principle is at every network layer.

Note also the RFCs 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 that conforms to expected abstractions. Such expectations can in fact be so

pervasive that cross-layer vulnerabilities due to them may persist unnoticed for decades[6]. In a word, layers of abstraction become boundaries of competence.

Robustness and the language recognition problem

Input data handling insecurity appears to be ubiquitous, and is commonly associated with complexity of message formats. Of course, "complexity" should not be decried lightly, as the progress of programming has been to produce ever more complex machine behaviors – and thus perhaps more complex data structures. But when does complex become too complex, and how does message complexity interact with Postel's Principle?

The formal language-theoretic approach we outline below and develop in [2, 3] allows us to quantify the interplay of complexity with Postel's Principle and to 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 safe manner that exposes neither the recognition logic nor further processing logic to exploitation. We speak of valid *or* expected inputs to stress the fact that, in the name of robustness, some inputs can be accepted rather than rejected without being valid or defined for a given implementation; these must, however, be *safe*, i.e., not lead the current layer or higher layers to perform a malicious computation (a.k.a. exploitation).

In [2, 3] we show that, starting at certain levels of message complexity, the recognition task viewed as the problem of recognizing a *formal language* made up by the *totality of valid or expected protocol messages or formats* becomes UNDECIDABLE. For such protocols, telling valid or expected inputs from exploitative ones is insurmountable, and exploitation by crafted input is therefore only a matter of exploit programming technique (cf. [4]). No "80/20" engineering solution for such problems exists, any more than it is possible to find an engineering solution to 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, while decidable, requires implementation of powerful automata, equivalent to a Turing machine with a finite tape.

When input languages require so much computational power, handling them safely is very hard to actually program, because validity of various input data elements can only be established by checking bits of context that may not be in scope for the checking code. The security-minded programmer understands 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. Security-minded programmers

are intimately familiar with this frustration: even though they know they must validate the data, they cannot 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 should 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 unlikely to be matched by actual ad-hoc recognizer code (a.k.a. "input validation" or "sanity checking" code) has been a fruitful exploitation approach. The reason for this is that full recognition of input messages is hardly ever implemented, and the equivalent of an underpowered automaton is used instead, which fails similarly to trying to match recursively nested structures with regular expressions.

We note that "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.

Even though rejection of inputs is not explicitly mentioned in the Postel's Principle, and would seem to be discouraged, properly powerful rejection is crucial to safe recognition. Our "patch" suggests a language in which the balance between acceptance can rejection can be productively discussed.

Mistake I: Computational power vs robustness

It is 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 above formulation.

The devil here is in the details. Writing a protocol handler that can deal with "every conceivable error" can be an insurmountable task for complex protocols, inviting further implementation error – or be simply 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, context-free, and 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 is, in general, UNDECIDABLE.

In the face of undecidability, dealing with "every conceivable error" is impossible. For context-sensitive protocols that require full Turing machine power for their recognition it may be theoretically possible but utterly thankless. So complex protocols hungry for computational power to be recognized should be deemed incompatible with the Robustness Principle.

Robust recognition – and therefore robust error handling – is only possible when the input messages are understood and treated as a formal language,

with the recognizer preferably derived from its explicit grammar (or at least is 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 an ad-hoc recognizer can be driven by crafted inputs. Indeed, had this problem been amenable to algorithmic solution, we would have had solved the Halting Problem.

Mistake II: Clarity vs ambiguity in presence of errors

It is easy to assume that, no matter what the protocol's syntax, the Robustness Principle compels acceptance of ambiguous messages and silent "fixing" of errors. This is a mistake.

Prior formulations such as RFC 761 can be read as trying to clarify the boundary between being accepting ambiguity and rejecting it:

```
3.2 Discussion

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).

--- http://tools.ietf.org/html/rfc761
```

A strict reading of "... MUST accept any datagram that it can interpret (e.g., not object to technical errors where the meaning is still clear)" would draw the line at *ambiguity* (i.e., un-clarity) of "meaning", although 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 due to errors.

The question, then, is what makes the meaning of a protocol message clear and unambiguous, and how can this clarity be judged in the presence of errors.

This property of non-ambiguity can hardly belong to individual messages of the protocol: to know what a message can be confused with, one needs to know what other kinds of messages are possible. Thus 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 is not likely that the parser of a hard-to-parse protocol can be further burdened with fixing "technical

errors" with impunity, without introducing a potential for programmer's error. Thus "clarity" can only be a property of 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 fall mostly within the regular or context-free classes.

Context-sensitive languages require more computational power to parse and more state to extract the meaning of the message's elements. Consequently, they are more sensitive to errors making such meaning ambiguous. Length fields that 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 message could be rejected thanks to a control sum if any – note that if such a sum follows the erroneous length field, it may also be mis-identified).

Thus ambiguous input languages should be deemed dangerous and excluded from the Robustness Principle requirements.

Adaptability vs ambiguity

Robustness Principle postulates adaptability. Continuing the RFC 1122 quote:

> 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 (see below), but it must not cause a failure. [RFC 1122]

It should be noted that this example operates with an error code, that is, a *fixed length field* that can be unambiguously represented and parsed, and does not affect the interpretation of the rest of the message.

That is to say, 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, and tolerating their undefined values would not introduce context sensitivity or necessitate another computational power step-up for the recognizer.

Thus, by intuition or otherwise, this example of laudable tolerance stays on the safe side of recognition as seen from 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 [5] goes

further and calls for balance and moderation in the Principle's application.

We agree, but posit that such balance can only exist for protocols that moderate language complexity of their messages and thus the computational complexity and power demanded of their implementations, and that moderating said complexity is exactly the only way forward to creating such balance. We believe that the culprit in the insecurity epidemic and the driver for patching Postel's Principle is not "hostility" of the modern Internet per se (that hostility was noted as far back as RFC 1122), but the excessive computational power greed of modern protocols. We note that the issues that, according to Allman, make interoperability notoriously hard are precisely those we point out as challenges to security of composed, complex system designs in [3].

There is much in Allman's discussion that we agree with. In particular, we see his "dark side" examples of "liberality taken too far" as precisely the ad-hoc recognizer practices that we call on implementors to eschew. Allman's examples of misplaced trust in ostensibly "internal" (and therefore assumed safe) data sources help drive home one of the general lessons we argue for:

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

We fully agree with the need for "checking everything, including results from local cooperating services and even function parameters", not just user inputs.

However, we believe that a brighter line is needed for protocol designers and implementers to make such checking to work. A good example is the missing checks for web input data "reasonableness" that Allman names the cause of SQL injection attacks – in fact, the downstream developer expectations of such "reasonableness" in combination with data format complexity may place undecidable burdens on the implementor, and prevent any reasonable balance from being struck.

The Postel's Principle Patch

Our proposed "patch" is as follows.

```
- Be liberal about what you accept.

+ Be definite about what you accept.(*)
+
+ Treat inputs as a language, accept it with a matching
+ computational power, generate its recognizer from
+ its grammar.
+
+ Treat input-handling computational power as privilege,
+ and reduce it whenever possible.
```

(*) For the sake of your users, be definite about what you accept. Being liberal worked best for simpler protocols and languages, and is in fact limited

to such languages; be sure to keep your language regular or at most context free (no length fields). Being more liberal did not work so 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 the users of your software to intractable or malicious computations.

Conclusion

Reversing the ubiquitous insecurity of the Internet and keeping it free requires 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 the Postel's principle, but can be seen as its stricter reading that should guide its application to more secure protocol design.

References

[1] Dan Geer, "Vulnerable Compliance", *;login: The USENIX Magazine*, vol. 35, no. 6, December 2010, `http://db.usenix.org/publications/login/2010-12/pdfs/geer.pdf`.

[2] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Anna Shubina, "The Halting Problems of Network Stack Insecurity", *;login:*, December 2011.

[3] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Michael E. Locasto, Anna Shubina "Security Applications of Formal Language Theory", `http://langsec.org/papers/langsec-tr.pdf`.

[4] 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:*, December 2011.

[5] Eric Allman, "The Robustness Principle Reconsidered: Seeking a middle ground", *ACM Queue*, August 2011.

[6] Sergey Bratus (with Travis Goodspeed), "How I Misunderstood Digital Radio", submitted to Phrack 68