Curing the Vulnerable Parser

Design Patterns for Secure Input Handling

SERGEY BRATUS, LARS HERMERSCHMIDT, SVEN M. HALLBERG, MICHAEL E. LOCASTO, FALCON D. MOMOT, MEREDITH L. PATTERSON, AND ANNA SHUBINA

Sergey Bratus is a Research Associate Professor of Computer Science at Dartmouth College. He sees state-of-the-art hacking as a distinct research and engineering discipline that, although not yet recognized as such, harbors deep insights into the nature of computing. He has a PhD in mathematics from Northeastern University and worked at BBN Technologies on natural-language processing research before coming to Dartmouth. sergey@cs.dartmouth.edu

Lars Hermerschmidt is currently working as Information Security Officer at AXA Konzern AG, where he is leading software security activities. He is a PhD candidate in software engineering at RWTH Aachen University, where he started to work on correct unparsers to prevent injections and on automated security architecture analysis. hermerschmidt@se-rwth.de

Sven M. Hallberg is a programmer by passion, a mathematician by training, and calls himself an applied scientist of insecurity by profession. He contributed large parts to the Hammer parser library and wrote the DNP3 parser based on it. He is currently pursuing a doctoral degree at Hamburg University of Technology, Germany, where he tries to further apply LangSec principles to cybernetic systems. pesco@khjk.org

Programs are full of parsers. Any program statement that touches input may, in fact, do parsing. When inputs are hostile, ad hoc input handling code is notoriously vulnerable. This article is about why this is the case, how to make it less so, and how to make the hardened parser protect the rest of the program.

We set out to make a hardened parser for an industrial control protocol known for its complexity and vulnerability of previous implementations: DNP3 [1]. We started with identifying known design weaknesses and protocol gotchas that resulted in famous parser bugs; we soon saw common anti-patterns behind them. The lesson from our implementation was twofold: first, we had to nail down the protocol syntax with precision beyond that of the standard, and, second, we formulated and followed a design pattern to avoid the gotchas.

We've used this approach with other protocols. Our parser construction kit *Hammer* (https://github.com/UpstandingHackers/hammer) allows a programmer to express the input's syntactic specification natively in the same programming language as the rest of the application. Hammer offers bindings for C, C++, Python, Ruby, Java, .NET, and others, and is suitable for a wide variety of binary protocols.

Sadly, there is no silver bullet one could implement in a library and simply reuse in every program to fix unsafe input-handling once and for all. However, we found several design patterns for handling input correctly, and thus making programs resilient against input-based attacks. In the following sections we describe three of them: the *Recognizer*, the *Most Restrictive Input Definition,* and the *Unparser*.

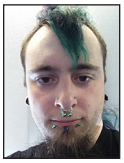
These patterns came from studying famous input-handling code flaws and what made them that way. Importantly, we found that the problems started with the choice of the input syntax and format that forced additional complexity on the code. The code flaws were made more likely by the choices of input structure; in a word, data format doomed the code.

"Don't trust your input" doesn't help to write good parsers. First, we need to deal with the standing advice of "Don't trust your input." This advice doesn't give the programmers any actionable solution: what to trust, and how to build trust? Without giving developers a recipe for establishing whether the input is trustworthy, we cannot expect correct software. This is a design issue, which no amount of penetration testing and patching can fix.

The problem of trust in the data is old. This is what types in programming languages arose to mitigate: the problem of *authenticating* the data, as James H. Morris Jr. called it in 1973 [2], before operating on it. We now call it *validating* the data, although our opponent is not Murphy—randomly corrupted data that leads to crashes—but Machiavelli: purposefully crafted data that leads to state corruption and compromise, aka unexpected computation.

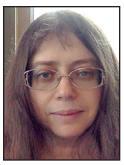
Trustworthy input is input with predictable effects. The goal of input-checking is *being able to predict the input's effects on the rest of your program*. Already, as we speak of checking the input, we assume that there is a checker separate and distinct from the rest of the code; we will make this distinction precise in the design patterns discussed below. The standing advice to validate input implicitly assumes that, if the input is valid, then its effects are *predictable* and do not include unexpected computation; it is safe to pass on to the rest of the program.

Michael E. Locasto is a Senior Computer Scientist at SRI International, where he works in the Infrastructure Security Group and leads several projects dealing with IoT security and secure energy systems research. He was previously an Associate Professor at the University of Calgary and an I3P Fellow at George Mason University. He is interested in why computer programs break and how we can get better at fixing them. michael.locasto@sri.com



Falcon Darkstar Momot is a Senior Security Consultant with Leviathan Security Group. He leads security reviews and penetration tests of software and networks at various large software companies, with an eye to process improvements. He received a BSc in computer science from the University of Lethbridge and is an MS student at Athabasca University. In his spare time he teaches people how to use amateur radios and works on a team to maintain an operational Bell System No. 1 Crossbar. falcon@iridiumlinux.org

Meredith L. Patterson is the founder of Upstanding Hackers. She developed the first language-theoretic defense against SQL injection in 2005 as a PhD student at the University of Iowa and has continued expanding the technique ever since. She lives in Brussels, Belgium. mlp@upstandinghackers.com



Anna Shubina is a Research Associate at the Dartmouth Institute for Security, Technology, and Society. She was the operator of Dartmouth's Tor node when the Tor network had about 30 nodes total. ashubina@cs.dartmouth.edu

Safety—that is, predictability of execution—comes from the combination of both the input format and the code checking it being simple and well-structured.

How do we know that reading a file that contains a hundred records is safe? How can we be sure that the execution is predictable? We'll have to start with the idea that a single record can be predictably read, and that the actions required for the reading are repeatable. One way to do so is to make sure the validity of each record can be judged apart from the contents of others, and that any objects constructed from it depend only on that record. This means that the records encode independent objects that follow each other (rather than nesting in each other). A pattern is allowed to repeat without limit, or up to a certain number of times, but its structure must be rigid; a pattern cannot contain itself recursively, directly or indirectly. Then, if it's safe to call the code that parses a record once, it's safe to call it repeatedly.

Some nesting of objects in a record is allowed but only in a pre-defined pattern: if we draw the objects containing each other as a tree, the shape of that tree is rigid except for possible repetition of a node where that kind of node is allowed. Supposing that each object is parsed by a separate function, the shape of the call graph is similar to the shape of the tree. This roughly corresponds to so-called regular syntax (as in regular expressions).

In short, when parsing such regular formats, the answer to "What should I do next?" or "Is the next part of the input valid?" doesn't depend on reexamining any previous parts. Thus the code that works predictably once is sure to work again.

However, not all formats can be so restricted. In HTML or XML, for example, elements can be nested in elements like themselves to an arbitrary depth. The same is true for file systems that have directories and for formats that emulate such file systems such as Microsoft's OLE2. Other formats, like PDF, have other kinds of container objects that can nest to any depth.

For such formats, whether it is safe to invoke the code that parses an object again and again may not be predictable, because it could be called under a potentially infinite set of circumstances. Should the result depend on the path to the top of the tree of objects or, worse, on the sibling nodes in that tree, such dependencies may now pile up infinitely. Unlike the regular case above, the shape of the tree is no longer rigid; much variation in its form can occur. Now the code needs to foresee a potentially unlimited number of possible paths and histories after which it gets called; the more its behavior is supposed to depend on reexamining other objects, the harder it is to get it right (and the harder it is for a programmer to have a succinct mental model of its behavior that has any predictive power whatsoever).

Thus the simpler the better; and only with the simplest formats can some assurance be obtained. The simplest syntax patterns are *regular* and *context-free*. Context-sensitive patterns are much harder to parse, and the code is much harder to reason about. In fact, such reasoning poses undecidable or intractable problems for formats that seem fairly intuitive and straightforward. We refer the reader to [3] and http://langsec.org/ for the theory; here, we'll look at the common scenarios of how things go wrong instead.

How Input Handling Goes Wrong

From a certain perspective, input data is "just" a sequence of symbols or bytes. But this sequence drives the program logic involved in construction and manipulation of some objects. These objects drive the rest of the program and must do so predictably.

The program should make no assumptions about these objects beyond those that the parser constructing them validates. If it does, the likely effect of its code working on data it does not expect will be exploitation. This relationship between the parser's results and assumptions made by the rest of the program is crucial, but the absolute majority of programming

languages do not provide any means of expressing it. Yet it has multiple ways of going wrong. Either the data's design is so complex that it invites bugs, or the programmer misunderstands the kind of validation that the data needs. Let us look at some of these examples.

Input too complex for its effects to be predictable. Safety is predictability. When it's impossible to predict what the effects of the input will be (however valid), there is no safety.

Consider the case of Ethereum, a smart contract-based system that sought to improve on Bitcoin. Ethereum operators like the decentralized autonomous organization (DAO) accepted contracts—that is, programs—to run in a virtual environment on their system; the code was the contract. The program that emptied the DAO's bank was a valid Ethereum program; it passed input validation. Yet it clearly performed unintended computation (creative theft of funds) and should not have been allowed to run.

Could the DAO have made this determination beforehand, algorithmically? Certainly not; Rice's theorem says that no general algorithm for deciding non-trivial properties of general-purpose programs may exist, and predicting the effects of a program on a bank such as DAO's is beyond even "non-trivial"—even the definition of malice in this context may not be amenable to complete computational expression. We will not dig into this theory here but will instead appeal to intuition: how easy would it be to automatically judge what obfuscated program code does before executing it? A Faustian Ethereum smart contract is hardly any easier. From the viewpoint of language-theoretic security, a catastrophic exploit in Ethereum was only a matter of time: one can only find out what such programs do by running them. By then it is too late.

Arbitrary depth of nesting vs. regexp-based checking. The arrangement (ordering and relative location) of objects in input requires a matching code structure to validate. Famously, regular expressions do not work for syntactic constructs that allow arbitrary nesting, such as elements of an HTML or XML documents or JSON dictionaries. These constructs may contain each other in any order and to any depth; their basic well-formedness and conformance to additional format constraints must be validated at any depth.

Regular expressions (regexps), which many Web applications erroneously use to check such structures, cannot do it. Regexps were originally invented to represent finite state machines, and those are incompatible with arbitrary-depth nesting. Thus regexps are best suited to checking sequences of objects that contain and follow each other in a particular order, repeat one or more times (or zero or more times), but do not infinitely nest; in other words, a finite state machine has no way of representing trees that can go arbitrarily deep. One can write a pattern that nests to some given depth N, but what about an input byte sequence where objects nest to depth $N + 1$? The attacker can craft just such an input and bypass the check.

Although regexp extensions found in modern scripting languages such as Perl, Python, and Ruby extend the power of their regexps beyond finite state machines, it is quite hard to write such patterns and get them right. Put differently, finite state machines cannot handle recursion well; a stack is needed there, and stack machines make a different, more powerful class of automata. Try writing a regexp without back references to match a string where several kinds of parentheses must nest in a balanced way. It cannot be done; the same problem arises with matching nesting XML elements of several kinds.

Perhaps the best known example of this mistake was the buggy anti-XSS system of Internet Explorer 8. Using regexps to "fix" supposed XSS led to non-vulnerable HTML pages being rewritten into vulnerable ones, the fix adding the actual vulnerabilities [5]. Web app examples of vulnerable checks of (X)HTML snippets are many and varied.

Context sensitivity. The lesson of the previous pitfall—still not learned by many Web apps—is that judging input must be done with appropriate algorithmic means, or else the program won't be able to tell if the data is even well-formed. But this is not the only trouble there can be.

Besides being well-formed, objects should only appear where it is legal for them to appear in the message. Judging this legality can be troublesome when the rules that determine validity depend not just on the containing object or message (i.e., the "parent" of the object we are judging), but on other objects as well, such as "sibling" objects in that parent or even some others across protocol layers.

For example, imagine that an object contains a relative time offset, in a shorter integer field, which is relative to another object that has the longer absolute value. For the relative value to appear legally, there has to be an absolute value somewhere preceding it, and the checker must keep track of this. This situation actually occurs in DNP3.

A closer-to-home example is nested objects that each include a length field. Since these lengths specify where each (sub)object ends (and another begins), all these lengths must agree with each other, and with the overall length of the message; that may be a *quadratic* number of checks on these fields alone!

The infamous Heartbleed bug arose from just such a construct: the agreement between the length fields of the containing `SSL3_RECORD` and the `HeartbeatMessage` contained in it was not checked, and the inner length was used to grab the bytes to echo back. Set that inner length to 65535, and that's how many bytes

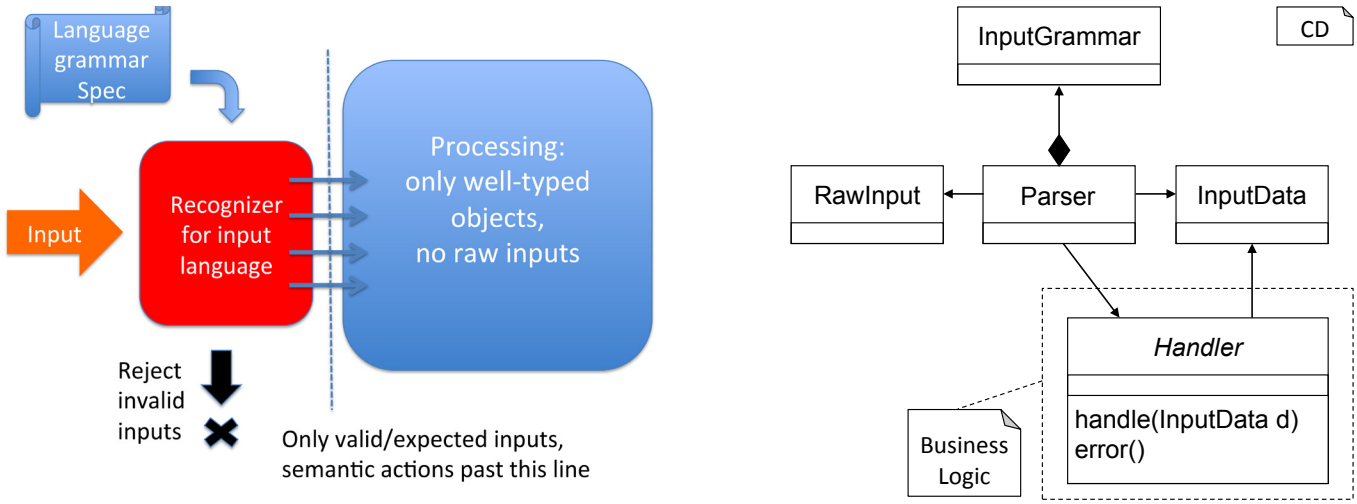


Figure 1: The Recognizer Pattern for validating raw input and providing it to the business logic: (a) input data flow through the Recognizer Pattern; (b) the Recognizer Pattern as a UML class diagram

OpenSSL included, even though the overall message length was set at a modest four bytes. The GNU TLS Hello message CVE-2014-3466 similarly took advantage of three nested lengths that were expected to agree but didn't get checked.

An older but equally famous example was the 2002 pre-authentication bug in OpenSSH 3.3 exploited by GOBBLES; there, the lengths of all SSH options would need to sum up to the length of the packet, and instead overflowed an integer allocation size variable before the crafted packet could be discarded.

Such formats where validity rules require checking properties across object boundaries are called *context-sensitive*. They require more complex checkers—and are more error-prone. Indeed, it is violating these relationships that's first tried by exploiters: a forgotten check is more likely there and thus an action on unchecked data that's not what the code expects.

It's best to be able to judge an object's legality based either just on its own content or on what type its parent object is; that is, *context-free* syntax is preferable to the more complex *context-sensitive*.

Transformation before validation. Another lesson from vulnerable ways of handling such a seemingly straightforward format as XML is that input messages should be checked as they arrive, *without additional transformations*, least of all those driven by the elements of these messages themselves. This has been a source of famous vulnerabilities with XML entities.

An XML document may include *entities*, syntactic elements that will be resolved and replaced, by string substitution, throughout the body of the document. Substitutions may occur in many rounds if entities include other entities, which, in turn, will be parsed and substituted, all before the document can be finally validated. The simplest consequence of this is that a short document can expand to gigabytes in size by using several levels of entities and repetition, the so-called "billion laughs" attack.

XML entities may also include references to external documents that need to be fetched and inserted before the input data object can be constructed and validated. Fetching an XML external entity (XXE) may already be an undesirable action in and of itself, and can trigger execution of other code; at the very least it creates network connections and can exfiltrate files, or even lead to remote code execution.

It becomes instantly clear that XXEs are trouble when you consider that an *action is taken based on input before that input has been fully validated*. XXEs bring actions into the recognition process, thus breaking the separation between recognition and processing. By comparison, JSON has no such feature, and JSON objects are judged as they are received. This may account for an order of magnitude difference in CVEs related to XML (850 at the time of this writing, of which 216 are XXE-related) vs. JSON (96).

The shotgun parser. We say we have a *shotgun parser* where validation is spread across an implementation, and program logic grabs data from the input-handling code before the full data's correctness is assured. This makes it very hard to follow the dependencies and assumptions made by the code, which, in turn, leads to vulnerabilities and unexpected behavior. The antidote for this is separation of concerns: validation first, then a clear boundary at which the data has been validated to a clear specification—and not used before.

But what comes out at that boundary? It is data as objects: constructed and fully conforming to the definitions of the data structures to be extracted from input. Reaching in to use them

before they are ready is an anti-pattern that resulted in Heartbleed (a remote memory leak) and many remote code executions like the 2002 OpenSSH bug or the GNU TLS Hello bug.

A million-dollar misnomer. Another key misconception about input data is that it is generally benign but can contain unsafe elements that should (and can) be "sanitized" or "neutralized." The choice of words suggested that having these elements removed or altered makes the data safe overall.

As a typical result of this (mis)understanding, the input is transformed by filtering it through regexp-based substitutions, where the regexps match the "bad" syntactic elements and replace them with some "safe" ones or suppress them.

The problem with this intuition is immediately clear: validity as predictability of execution is the property of the entire input, not of a few characters!

Deserialization is parsing, too! It should be clear by now that deserialization is not a trivial concern to be handled by some auxiliary code; it is a security boundary. This boundary exists between every pair of components that communicate outside a strong typing system or that use different structures to represent data.

It is the deserialization code's responsibility to create the conditions that the rest of the program can trust; otherwise any assurance of good program behavior is lost. That's why the properties of the serialized payload should be as simple as possible to check and, once checked, reliable enough to ensure predictable behavior.

Simply put, what a deserializer cannot check, the rest of the code should not assume. If serialized objects aren't self-contained and validatable on their own, the game is already lost; so many Java deserialization bugs, Python unpickling bugs, Remote Procedure Call bugs, and so on have turned into exploits.

The Recognizer Design Pattern for Input Validation

Input validation needs design patterns. Ensuring that input data is safe to process is a distinct, specialized role for code. As a matter of program architecture, any specialized code should be isolated in a dedicated component. Design patterns are a natural way to express the relationships of this component with others.

The main input-handling pattern we discuss is the Recognizer Pattern. As a whole, a recognizer has the sole task of accepting or rejecting input: it enforces the rule of full recognition before processing. This pattern concentrates the logic responsible for strictly matching the input's syntax with the specification and discarding any inputs that don't match.

The Recognizer Pattern in Figure 1 describes the relationships between five main elements: the InputGrammar, the Parser, the RawInput, the Handler, and the data type representing the input data within the program (called InputData in Figure 1 (b)). The locus of the Recognizer Pattern is the Parser. The Parser uses the InputGrammar as a definition of the valid input syntax. For input sequences read from the RawInput that comply with that syntax, the Parser produces a correctly instantiated InputData object representing the input in the programming language's type system. Importantly, the Parser only invokes the `handle()` method of the Handler interface after creating InputData objects. The Handler interface must be implemented by the "business logic" of the application. This arrangement cleanly separates the parsing logic from subsequent processing within the business logic, as the Handler can only access InputData validated by the Parser. This provides a crucial guarantee to the remainder of the business logic that the data has been validated and that such validation is structurally sound (i.e., it cleanly handles InputData objects nested within each other).

Most Restrictive Input Definition

In order to fully take advantage of this pattern, the input syntax specification expressed as the Grammar component should have a minimum of complexity needed to represent input objects. This point is very important because it openly acknowledges the price of adopting the Recognizer Pattern. Part of the value of adopting this approach is that you have a clear idea of what data you accept, but you give up attempting to accept arbitrarily complex data. Practically speaking, this means purposeful, thoughtful subsetting of many protocols, formats, encodings, and command languages, including eliminating unneeded variability and introducing determinism and static values. The design principle for creating predictable programs is to *choose the most restrictive input definition for the purpose of the program;* we acknowledge that it may be challenging to completely articulate the purpose of the program well enough, and that errors may still exist deeper in the program logic.

Parser Combinators: Don't Fear the Grammar!

At the heart of the Recognizer Pattern is keeping the admitted inputs to a strict definition of valid syntax. Being definite about the input gives the pattern its power; but how to do so without undue burden?

Historically, computer scientists wrote such definitions in special languages such as Augmented Backus-Naur Form (ABNF). Unfortunately, that's one more language—and another set of tools—for developers to learn; too much investment for handling what might seem a simple binary format! Moreover, after having written the input data definitions as a grammar (say, for yacc or Bison), one would need to write them *again*, in code, to construct the actual objects.

To add to developer confusion, yacc and Bison focus primarily on compiler construction, not binary parsing. The code they generate is quite unreadable: it's a large state machine with none of its internals named in a way to make sense to humans. Interfacing processing code with it is hard and has led to many mistakes.

Finally, another concern about grammars is that they have subtle gotchas to confuse their developers, such as left recursion's incompatibility with classic *LL(k)*-parsing algorithms.

Fortunately, the *parser combinator* style of writing input handling code provides a graceful way around these obstacles. The parser combinator style of programming defines the grammar of the input language and implements the recognizer for it at the same time. Thus it repackages strict grammar constraints on input in a form much more accessible to developers than do bare grammars, while retaining all of the rigor and power.

We took the parser combinator approach, and implemented the Hammer parser construction kit to specifically target parsing of binary payloads (e.g., describing bit flags and fields that cross byte boundaries is simple in Hammer, unlike in character-oriented parsing tools). Hammer targets C/C++, where the need for secure parsing is the strongest, yet modern tools for it (such as ANTLR) are not available.

Hammer supports hand-writing code that looks like the grammar and captures the definition of the recognized language in an eminently readable form. However, it does not preclude code generation. For example, Nail [4], a direct offshoot of Hammer, comes with a code-generation step.

But didn't ASN.1 solve this problem? The formidable ASN.1 standard was expected to solve the problem of unambiguously representing protocol syntax. Separating the syntax from encoding and specifying the encoding rules separately was supposed to open the way for automatically validating data against specification. The security gain from this would be obvious.

In reality, ASN.1 encoding rules and code generation tools created enough complexity and confusion to result in a series of high-profile bugs. The more permissive BER seems to be doing worse than DER: 45 vs. 26 related entries out of a total 95 ASN.1-related CVEs (based on a simple keyword search). Overall, the security record of ASN.1 does not suggest an equivalent security win for code generation.

Specifying a format with combinators. Here is an excerpt showing what our parser combinator code looks like. Remember, under this style everything gets its own parser, even a bit flag. This may seem excessive, but it truly defines the format from the ground up, and makes it clear, at every point, what structure is expected from inputs, and which properties have been checked and are being checked. Since Hammer targets binary protocols, it provides primitives for a field containing a given number of bits, `h_bits`, and a way to limit such a bit field to a range of possible integer values, `h_int_range`.

These individual parsers are connected up to parsers for each sub-unit of the message with *combinators*, such as sequencing (`h_sequence`, arguments are a NULL-terminated sequence of constructs that must follow each other), repetition (`h_many`, `h_many1`, `h_repeat_n` for the same respective meanings as *, + and {n} in regexps), or alternatives (`h_choice`).

Let's build up the parser for a DNP3 application header, which starts with a four-bit sequence number followed by four single-bit flags, then a one-byte function code (FC), and is optionally followed by a 16-bit field called "internal indications" (IIN), of which two bits are reserved. Whether a payload is a response or a request is determined by the flag combination. Not all combinations of flags are legal, and IIN is only legal in payloads that represent protocol responses, not requests. All these dependencies must be checked before the payload can be acted upon—or else memory corruption awaits.

We start with building up the bits for flags and their allowed combinations:

```
bit =  h_bits (1, false );
one = h_int_range(bit, 1, 1); // bit constant 1
zro =  h_int_range(bit, 0, 0); // bit constant 0

conflags = h_sequence(bit, zro, one, one, NULL); // confirm
reqflags = h_sequence(zro, zro, one, one, NULL); // fin, fir
unsflags = h_sequence(one, one, ign, ign, NULL); // unsolicited
rspflags =  h_sequence(zro, bit, bit, bit, NULL); // response
```

Then comes the start of the header, with its several valid alternatives. The rest are illegal and will be discarded.

```
seqno = h_bits(4, false /* unsigned */ );
conac = h_sequence(seqno, conflags, NULL );
reqac = h_sequence(seqno, reqflags, NULL );
unsac = h_sequence(seqno, unsflags, NULL );
rspac = h_sequence(seqno, rspflags, NULL );
iin  =  h_sequence(h_repeat_n(bit, 14), reserved (2) , NULL );
...

req_header =
  h_choice(h_sequence(conac, confc, NULL),
       h_sequence(reqac, reqfc, NULL), NULL);

rsp_header =
  h_choice(h_sequence(unsac, unsfc, iin, NULL) ,
       h_sequence(rspac, rspfc, iin, NULL), NULL);
```

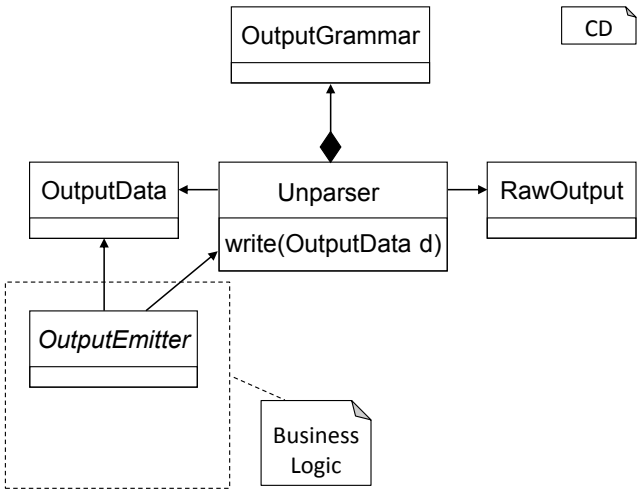


Figure 2: The Unparser Pattern for creating valid output illustrated as a UML class diagram.

Not shown here are the parsers for the one-byte function code field (`confc`, `reqfc`, `unsfc`, and `rspfc`), which enforce the appropriate value ranges. For example,

```
fc = h_uint8();
reqfc = h_int_range(fc, 0x01, 0x21);
```

and so on.

This example shows how the parser combinator-style code defines the expectations regarding the input precisely and implements a recognizer for them at the same time. But there's more—this recognizer doubles as the constructor of the parsed objects! For more detail, see [1].

Handling Output: The Unparser Pattern

So far we've only considered the case of an adversary that can directly provide input to a program. However, in interconnected systems, e.g., a Web server and a database, there are back-end systems like the database that only process input provided by the front-end Web server. Nevertheless, unexpected input to the front end may manipulate its output so that the back end interprets this in a way not intended by the developer. Therefore we need to discuss how to make back-end systems safe from indirect input attacks, where hostile inputs are passed by another program. Examples include SQL injection (SQLi) and cross-site scripting (XSS) and are most common in, but not limited to, text-based languages like SQL and HTML.

This injection into the output of the front end cannot, generally speaking, be prevented by the Recognizer at the front end. The reason is simple: the Recognizer enforced the specification of the *input* language; the language expected to be *output* by a program is different, and the Recognizer has no information about it. Hence it cannot reject those inputs that cause problems in output.

Commonly, textual output is created by concatenating fixed strings like SQL query parts with program input. Since textual languages like SQL use special tokens such as quotation marks to separate data from code, those tokens must be encoded when used within the program's output. Otherwise, input might change the meaning of the created output by using these tokens. Using templates where variables are replaced by input data, e.g., to create HTML, suffers from the same core problem: naïve creation of output with string concatenation that is not aware of the string being a language parsed by another program.

A defensive design pattern must encapsulate this awareness. For creating output and ensuring it is well formed, we developed the Unparser Pattern shown in Figure 2. Its operation is essentially reverse to that of the Recognizer: it uses a language specification (an OutputGrammar) to serialize existing valid objects to that specification.

Just as the Parser is the only class meant to read from RawInput, only the Unparser writes output to the RawOutput. Therefore, creating output from the perspective of the business logic works by instantiating OutputData objects and filling them with data without caring whether this data might contain special tokens of the output language. The Unparser takes these objects and creates a serialized output. It uses the definition of the OutputGrammar to ensure tokens possibly contained in the OutputData are encoded properly.

SQL's prepared statements interface is a special case of this pattern that had not been generalized to other output languages; we correct that. Our OutputData class provides an interface similar in function but more general and strongly typed. More about unparsers can be found in [6]; *McHammerCoder* (https://github.com/McHammerCoder) is our binary unparser kit for Java.

Finally, connecting the Recognizer, Most Restrictive Input Definition, and Unparser patterns using a business logic that translates InputData to OutputData results in a *Transducer*. The special case when InputGrammar and OutputGrammar are the same can be employed as a transparent filter at the trust boundary of a system. It acts like a syntactic firewall, improving the system's predictability by enforcing a strict input specification. We implemented this approach in our DNP3 exhaustive syntactic validation proxy and recommend it for other protocols.

Conclusion

After decades of repeated embarrassing failure, the larger programmer community accepted that "rolling your own crypto" was simply the wrong approach; effective cryptography required using professional tools.

This realization came none too soon, but a bigger realization awaits: *rolling your own parser is just as bad or worse*. Faulty input-handling is a bigger threat to security than faulty crypto, simply because, as a target, it comes *before* crypto and leads to full compromise. Solid design and professional tools are needed, just as with crypto; otherwise, the insecurity epidemic will continue.

References

[1] S. Bratus, A. J. Crain, S. M. Hallberg, D. P. Hirsch, M. L. Patterson, M. Koo, and S. W. Smith, "Implementing a Vertically Hardened DNP3 Control Stack for Power Applications," Annual Computer Security Applications Conference (ACSAC), Industrial Control System Security Workshop (ICSS), December 2016, Los Angeles, CA.

[2] J. H. Morris, Jr., "Types Are Not Sets," in *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73)*, October 1973, pp. 120–124.

[3] L. Sassaman, M. L. Patterson, S. Bratus, M. E. Locasto, and A. Shubina, "Security Applications of Formal Language Theory," *IEEE Systems Journal*, vol. 7, no. 3, September 2013; Dartmouth Computer Science Technical Report TR2011-709.

[4] J. Bangert and N. Zeldovich, "Nail: A Practical Tool for Parsing and Generating Data Formats," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-bangert.pdf.

[5] E. V. Nava and D. Lindsay, "Abusing IE8's XSS Filters," 2010: http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf.

[6] L. Hermerschmidt, S. Kugelmann, and B. Rumpe, "Towards More Security in Data Exchange: Defining Unparsers with Context-Aware Encoders for Context-Free Grammars," in *Proceedings of 2015 IEEE Security and Privacy Workshop*, pp. 134–141: http://spw15.langsec.org/.