

Structured Strings

Dr. Colin Hirsch

October 17, 2007 (preview)

“The differences between theory and practice are greater in practice than in theory.”

Abstract

Security problems like sql code injection and cross-site scripting vulnerabilities can be traced to the, common, use of unstructured strings to represent structured data and code. This paper gives an explanation of the issue, and develops and discusses an alternative generic encoding for structured string data that, by adding an appropriate, yet minimal layer of abstraction for meta-information, is very simple, and has the potential of immediately eliminating an entire attack vector.

1 Introduction

The world wide web is built on standards that handle code and data in the form of plain strings: The final content of a web page that consists of HTML, JavaScript and CSS is usually generated from an HTML or XML template, uses SQL to access a data base, script languages like PHP and Python to process dynamic content, and finally HTTP to be transferred to the client.

Unfortunately, all of these standards and *languages* were *not* designed with security in mind, the effects of which are seen and felt by the entire computer industry and its users.

For example, scripting languages could use type information like an additional flags for strings to signal whether a string might be safe for a given purpose, rather than silently forwarding input from untrusted sources. In this paper the focus is however on an earlier stage, on the saferrepresentation of code and data.

All mentioned standards and *languages* have in common that the input consists of a plain string; plain in the sense of being initially unstructured, i.e. a sequence of characters of which, at best, the length is known. The syntax of every language, implicitly or explicitly, defines how to find structure in these plain strings, e.g. S-expressions for Scheme, or the abstract syntax tree for C code.

The process of recovering the structured data from the unstructured input consists of the following conceptual stages.

First, according to the lexical rules of the language, the *scanner* decides where the input string is broken into *tokens*, i.e. substrings of the input string. Second, according to the grammar of the language, the *parser* transforms the sequence of tokens into a data structure, usually some kind of tree, for further processing.

Sometimes the tokens are augmented with some auxiliary information before reaching the parser stage, however in the following we assume that tokens are plain strings.

The scanner sequentially reads the input string, accumulating all characters into a token until a lexical rule is invoked. That is, lexical rules contain special characters or character sequences whose occurrence in the input signal the end of the current token. Which lexical rules apply is subject to which rule was invoked last.

Example 1.1. Consider the following, informally defined, lexical rules on strings of letters, quotes, and spaces. Assume that there are two kinds of tokens that we call *identifiers* and *literals*.

1. Identifiers start with a letter and reach to the next non-letter, excluding the non-letter.
2. Literals start with a quote and reach to the next quote, including the quote.

Let us call *scanner2* the scanner that works with these two rules. With these rules, spaces outside of literals separate consecutive identifiers, but are not themselves ever part of a token. For example *scanner2* tokenises **abc xyz'hi ho'** into the three tokens **abc**, **xyz** and **'hi ho'**, or two identifiers followed by a literal.

In practice, all¹ languages have much more elaborate lexical rules. There are several reasons for this, one of which can be put into a simple question regarding Example 1.1, “how can a literal contain a quote?” The answer is that it currently can’t, and that we need to add another rule that can override rule 2 in a special case:

3. Within a literal, two consecutive quotes do not end the literal; instead, they get replaced by a single quote.

Let us call *scanner3* the scanner that uses all three rules. Rule 3 answers our question, as *scanner3* tokenises the input **'quote"test'** into the single literal **'quote'test'**, rather than the two literals **'quote'** and **'test'** as *scanner2* would have done.

If we forget to duplicate the quote that was intended to be part of the literal, *scanner3* tokenises **'quote'test'** into the literal **'quote'**, the identifier **test**, and then throws an error² because there is no “next quote”, as required by rule 2.

This small example shows exactly how SQL handles quoted string literals. . .

Fact 1.2. *Forgetting to duplicate quotes as shown is one cause of SQL injection vulnerabilities.*

Example 1.3. Consider the following (partial) SQL query to retrieve messages stored in a database.

```
select * from messages where user=
```

To complete the query string, the application must append an initial quote, the name of the user on behalf of which the query is performed, and a closing quote.

```
select * from messages where user='alice'
```

¹Ignoring exceptions like the remarkable Unlambda programming language.

²It could also crash, or wait eternally for more input, but we choose an error.

Now suppose that the application does not correctly double all quotes in the username. A malicious username like `marvin' or 'a'='a` will then result in the query

```
select * from messages where user='marvin' or 'a'='a'
```

which changes the query to return *all* messages because the `'a'='a'` clause is always true.

Note 1.4. *The important thing to understand is the principle of the attack, that the concatenation of incorrectly prepared strings allows unforeseen structural changes, here an SQL query, that can be exploited. We call it a structural change because the parse tree of the malicious query has a different structure than the parse tree of the intended query.*

For SQL there are two common methods of protection against this kind of attack. One are the so-called *prepared statements* which transfers the query to from the application to the database in two independent steps. The first step fixes the structure of the query, the second step fills in the variables, in our example the username. Since the structure is fixed first, the variables can not change it.

This approach to prevent SQL injection attacks is fine. Unfortunately it is not generally used, otherwise the problem of SQL injection attacks would not exist. And unfortunately the approach does not easily generalise to all the other languages that are used on the internet.

Unlike, in a sense, the second method of protection for SQL generalises. Every language, including SQL, has rules on how to prepare strings so that they can be concatenated *without* unforeseen structural changes to the encoded data. Strict application of these rules, always and everywhere, would accomplish complete protection from these manipulations. This method is further complicated beyond “always and everywhere” because most languages have distinct, incompatible *quoting* and *escaping* rules for this preparation.

Claim 1.5. *The great absolute number of security incidents owing to incorrect or forgotten quoting and escaping [4, 6, 2, 8] shows that this approach is too brittle and error prone for practical application.*

Our aim is to reduce the complexity, and therefore increase the efficiency, maintainability and security of applications that handle structured data in the form of strings. This complexity is created by the *the conceptual weakness* weakness of plain string encodings not being an appropriate starting level of abstraction for this kind of data.

2 Our Suggestion

In this section we define an alternative that adds one simple layer to the encoding of string-based data that we call the *structured string (version X) encoding*, or StrucSE(X). Effectively we move all structural control from the syntax into a meta-layer that is not cannot be compromised by the actual data.

A sequence of bytes conforming to this definition is called a *structured string (version X) document*, or StrucSD(X). Bytes are the smallest addressed unit, and must consist of exactly 8 bits. The ordering of bits within a byte is not relevant, and not therefore defined.

A structured string document consists of at least one byte. The first byte determines the encoding version, and is followed by whatever that encoding version mandates.

Definition 2.1. Version 0 Encoding

1. A version 0 document consists of one byte with value 0x00.

Definition 2.2. Version 1 Encoding

1. A version 1 document consists of one byte with value 0x01, followed by one *object*.
2. An object consists of one byte that determines its kind, followed by whatever the kind mandates.
3. A string object consists of one byte with value 0x55, followed by a length ℓ , followed by a UTF-8 string of ℓ bytes.
4. A vector object consists of one byte with value 0x56, followed by a length ℓ , followed by ℓ objects of any kind.
5. A length consists of the ASCII representation of an unsigned integer ℓ with $0 \leq \ell < 10^{10}$, formatted as per "%09u".

This is the whole definition, orthogonal and straightforward. It is left as an exercise to the reader to find out by which factor the HTML standards document could be reduced when layering HTML on the structured string encoding, and thereby removing everything on quoting, escaping, character sets etc.

The following examples show how encodings that are currently based on plain strings could be based on the structured string encoding instead, and discusses why this might be a good idea.

Example 2.3. An **SQL statement** can be represented as structured string document consisting of one vector object with one string sub-object for every token that results from tokenising the original statement according to the lexical rules of SQL, including the appropriate reversal of all escaping and quoting.

Changing an SQL database to accept queries in this StrucSE brings two immediate benefits. First, a very small and generic StrucSE library can replace the SQL-specific scanner in the database. And Second, this interface to the database is immune to SQL code injection attacks.

It is immune because the length that prefixes every SQL token in the StrucSD is a jail from which the token's contents are unable to escape.

An application that prepares an SQL statement knows the lengths of all potentially malicious strings that are destined to become a token in the statement. It encodes these strings as string objects in the statement's StrucSD: It prepends the string object header according to StrucSE1 to the concatenation of a single quote, the string itself, and another single quote.

The prepended length in the StrucSD is what effectively establishes the jail.

As mentioned earlier, SQL statements can also be secured with prepared statements, but there are several reasons why to use the structured string encoding for SQL. Note first that this is no either-or situation as the structured string encoding can be applied to any or all of directly executed SQL statements, statements that are prepared, and the parameters when executing a previously prepared statement. Using the StrucSE for gives the developer more choice and flexibility, since she is not forced to resort to prepared statements in order to prevent injection attacks.

An alternative encoding of SQL statements as StructSDs could be defined based on the abstract syntax tree of the statement according to the SQL grammar. In order to keep the first example simple, we have chosen to encode tree-structured data only in the following example.

Example 2.4. An **XML or HTML document** can be represented a structured string document as follows. An entity **name** with attributes **attr1** through **attrX** and respective attribute values **value1** through **valueX** is encoded as structured string vector object of length three where

The first object in the entity’s vector is a string object with content **name**.

The second object in the entity’s vector is another vector object consisting of $2 * X$ string objects with respective values **attr1**, **value1**, . . . , **attrX**, **valueX**.

The third object in the entity’s vector encodes the content enclosed by the entity, if any. Conceptually this content consists of text and markup, i.e. a sequence of text and other entities. This is encoded into a vector object by consecutively creating (a) a string object for every maximal text, i.e. for every text that is delimited by entities, and (b) a vector object of length three for every entity, obtained by recursively applying the entity encoding rule.

(For “empty” entities like `
` (HTML), `
` (XML) or consecutive open/close tag sequences like `<p></p>` (either), this third vector of the entity is therefore empty too.)

In line with the idea of structured strings, this encoding of XML and HTML is defined to *not* support any of the XML and HTML escaping mechanisms. When transcoding an XML or HTML document into the the corresponding structured string document, all escaping must be reversed.

For simplicity this example only encodes the basic content of an XML or HTML document. For practical application it needs to be completed to handle the full XML and/or HTML specification

Example 2.5. The following string³ is a structured string version 1 document.

```
1V000000003U000000004nameV000000002U000000004
attrU000000005valueV00000000100000U00047<42
```

It also happens to conform to the structure of the structured string encoding of XML documents; the representation of the same data in its usual plain string XML form is as follows.

```
<name attr="value">7&lt;42</name>
```

Note how the structured string version is exempted from the application of escaping and quoting to obtain the intended document, and remember that this is always true because there are no special characters in structured string version 1’s string objects.

The following example illustrates the “context scenario” for the other examples in this paper.

Example 2.6. Suppose that Bob User wants to read his messages on `www.example.org`. He points his browser at `http://www.example.org/login.html` and enters his name as “bob user”, which is then submitted to a dynamic page by getting the following URI.

³The line break is not part of the string.

`http://www.example.org/messages.lang?username=bob%20user`

The space in the username was replaced by the browser with an escaped representation of its ASCII value as required by the HTTP specification. The HTTP server at `example.org` analyses the URI string and calls the server-side script in `messages.lang`. The script language makes available a variable `username` to the script that is set to the unescaped version of the corresponding substring of the URI.

At some point the script takes the (first) query string from 1.3 and appends a single quote, an escaped version of the value of `username`, and another single quote. This time the escaping is performed according to the SQL standard.

The resulting query is sent to the database where it is parsed and unescaped, again according to the SQL standard.

In case of more complicated SQL queries, it can happen that parts of the query string need to be first escaped according to SQL, and the result escaped according to the scripting language. This double escaping is even necessary when both languages have common lexical rules. For example assume a scripting language that shares the property that single quotes in literal strings must be duplicated. Then the literal string `'quote'test'` must be entered as `'quote""test'` in the script source. The first duplication is removed when the script language parses the script, the second when the database parses the query.

In any case, the example shows how different quoting and escaping rules need to be applied to every piece of data that flows back and forth between a web client, and a web server.

For an example below we need an StrucSE version for parts of the HTTP protocol.

Example 2.7. In an StrucSE version of HTTP, the POST data `attr1=value1&...&attrX=valueX` from the URI can be encoded as a vector object consisting of $2 * X$ string objects with respective values `attr1`, `value1`, ..., `attrX`, `valueX`.

The same encoding can also be used for the key/value pairs that make up most of an HTTP header, and it is the same encoding that was used above for the attributes of an XML or HTML entity. Like in 2.4, all string values are intended for literal interpretation, without any quoting or escaping. The request URI from 2.6 results in the following StrucSD:

```
1V000000004U000000004httpU000000015www.example.orgU000000013
messages.langV000000002U0000000008usernameU000000008bob user
```

Claim 2.8. *General adoption of the structured string encoding favours simplicity and security.*

Towards simplicity, we have said that the StrucSE allows the removal of all quoting and escaping from languages that use it. Towards security, we have shown how StrucSE can render a database interface immune to SQL injection attacks, and continue here with an extended example.

For the remainder of this section, assume that StrucSE1 is used for SQL, HTTP and HTML, as shown in 2.3, 2.7 and 2.4 above.

Suppose now that the script `messages.lang` from 2.6 contains an StrucSD version of the SQL query from 1.3. That is, the top-level vector of this StrucSD contains one entry for every token of the statement template, excluding the final parameter with the username. Once the script is invoked with the POST data encoded as per 2.7, it suffices to search for the string object that

contains the username and move or copy the object from the POST data's vector object to the SQL queries vector object. There is no need for (un)escaping and (un)quoting, and no danger of SQL code being injected into the query.

The web server might however be vulnerable to a different class of problems, the so-called **cross site scripting** attacks. Compared to the SQL code injection vulnerabilities discussed in this paper, the class of cross site scripting vulnerabilities is very much broader. The common problem is that somewhere an input string is not properly quoted and escaped, resulting in the possibility of injecting malicious content into an HTML page.

While SQL code injection attacks the server, cross site scripting attacks the client. Depending on the exact nature of the vulnerability, different private data stored on the end user's machine can be compromised.

One common example is a web application that does not properly prepare, i.e. quote, escape and sanitise, the messages posted into its forum or blog. This can lead to a vulnerability that can be exploited by posting specially crafted messages that attack every user that opens the message.

Unfortunately the nature and diversity of cross site scripting vulnerabilities prevent a single effective countermeasure. The following examples show some of the contributions that the StrucSE can make at least towards partial solutions.

Example 2.9. An immediate and direct benefit of using the StrucSE for XML or HTML is that no piece of data can escape from its jail, thereby preventing changes to the structure of the document.

The explanation is basically the same as for SQL 2.3. Suppose for example that an application concatenates the tag ``, a message read from the database, and the tag ``, in order to print the message in boldface.

The message `bla bla bla dangerous ` subverts this intention by moving the string **dangerous** outside of the **b**-entity. The structural change created by this message can be easily seen in the DOM [3] tree of the resulting document.

With HTML based on StrucSE, the change is prevented because the document structure is fixed by the hierarchy of the StrucSE objects which cannot be changed from within the payload of the objects. For simple messages, or any other content that is not supposed to contain markup, 2.9 is sufficient to prevent all cross site scripting problems.

Otherwise it is arguably a big step in the right direction since any other (security-related) processing can (a) assume that certain changes to the DOM tree are impossible, and (b) need not deal with the complexity resulting from the quoting and escaping rules of XML and HTML, which even allows a limited form of *obfuscating malicious code*.

Example 2.10. Imagine that a new HTML is based on StrucSE as in 2.9, and that it contains a new attribute **atags** for the **div**-entity. The value of the **atags** attribute is a comma separated list⁴ of entity names. Within the sub-tree of the DOM rooted at a **div**-entity that contains an **atags** attribute, the browser must ignore all entities that are not contained in the list of the **div**'s **atags** attribute.

Note that this example does not take into account attacks based solely on attributes of allowed

⁴A vector object would be more appropriate, but impossible without more fundamental changes to HTML.

values. For practical application, the mechanism would need to be extended, for example by a list of allowed attributes for every allowed entity.

For our example message forum, the message could be surrounded by **div** tags with an **atags** list of harmless entities like **b,i** and **ol,ul,li**, but excluding potentially dangerous entities like **img** and, in particular, **script**.

The **atags** attribute is easier to implement, and more reliable and secure, than current security mechanisms on current HTML that operate by hunting for potentially malicious sequences in the message body. Easier (and more reliable) because a recursive traversal of part of the DOM tree with a couple of string comparisons at every node is sufficient. More reliable (and easier) because it uses a white list of allowed entities, rather than a black list of potentially malicious sequences that need to be recognised through all quoting and escaping.

3 Randomised Discussion

The structured string encoding is not in competition with XML, or any other language. We suggest the redefinition of XML, and all other languages, to be based on structured strings rather than plain strings. The real competitor to the idea of structured strings is the idea of giving white-space, or other characters, a special role in plain strings.

Note 3.1. *To reiterate, layering XML and other languages on structured strings would simplify their specification, and*

...free resources to concentrate on more interesting questions than which character must be escaped how and when and where,

...eliminate SQL injection, and other, vulnerabilities,

by adding a very thin layer of abstraction that provides the languages with a basic encoding at a more appropriate layer of abstraction than plain strings.

Note 3.2. *A fixed “length of the length” of objects in encoding version 1 was chosen to simplify the decoding process (otherwise the “length of the length” would need a dedicated terminator, which is against the spirit of the StrucSE: know thy lengths in advance, or it would need to be prefixed by the “length of the length of the length”, which recursively defers the issue). The actually chosen “length of the length” of 9 is rather arbitrary and could be extended — for the final definition or for a future version, if and when the need arises.*

Note 3.3. *Not every StrucSD has the structure discussed in the HTML section, and therefore not every StrucSD can be used as input to an HTML processing application. This is a side-effect of the restriction of the StrucSE to only two kinds of objects. Alternatively one could add further kinds of objects whose structure is a step closer to what is needed, for example a dictionary object. However a very simple method, a small deterministic finite-state top-down tree-automaton, suffices to check whether structured string can be seen as transcoded HTML document.*

Note 3.4. *The version 1 encoding does not limit the nesting depth of objects, therefore version 1 documents that exceed any given length exist, even though the length of individual strings is limited.*

Note 3.5. *The following extensions to the StrucSE could be considered for future inclusion, either into the final version 1 or a future extended version.*

- A raw object consists of one byte with value `0x52`, followed by a length ℓ , followed by ℓ bytes of raw data.
- A list object consists of one byte with value `0x4c`, followed by the nesting depth of the list in the document's object hierarchy encoded like a length, followed by zero or more objects of any kind, followed by one byte with value `0x4c`, followed by the same nesting depth repeated.
- A document object consists of a StrucSD version n , where n is less or equal to the version of the containing document.
- A map object consists of one byte with value `0x4d`, followed by a length ℓ , followed by 2ℓ objects, a string, an arbitrary object, etc. This would simplify application of the structured string encoding to things like HTTP headers or JSON.

These rules are completely generic and therefore more in the spirit of the structured document encoding than the possible extensions that were mentioned in the context of HTML.

Note 3.6. Many binary data formats and protocols, for example IFF, Mach-O and Diameter, that have some kind of encoding for lists of objects, put the total length of the list (payload) in bytes at the beginning of the list. The StrucSE diverges by putting the length of the list in objects. This local structural measure allows for more efficient encoding and verification of StrucSDs because it eliminates redundancy — that would otherwise need to be checked.

Note 3.7. We consider conceptual and implementational **simplicity**, elegance and orthogonality the most important traits in order to achieve correctness, and therefore security, and efficiency: a simple solution is easy to write, debug, review, verify, tune... In order to achieve simplicity, we turn to a modular division of responsibilities, often in the form of **layering**, that allows every component of a solution to focus on one clearly defined task.

The proposed solution aims for simplicity and a cleanly layered design, the consequences of which can be seen for example in 3.9 or 3.10 below.

Note 3.8. The handling of string data is often **line based**, in particular in the Unix-world where there is a large tool box for line-based data processing: `diff/patch`, `awk`, `grep`, `sort`, ... Should the StrucSE, or something similar, find sufficient adoption, a similar set of generic applications and editors would be created that follow the implied **hierarchical structure** of the documents.

Regarding HTML and XML, it should be rather easy to modify applications to read and write the StrucSE of such documents in addition to the currently used plain strings.

Note 3.9. The **slight overhead in size and structure** that is possibly created by mapping other string encodings to structured strings is not considered a problem; whenever space is an issue, a data compressor can be used to eliminate the redundancy; in particular when several documents are handled together, the compression can be potentially greater than any clever and complicated alternative to the StrucSE.

Note 3.10. The possibility of structured strings to **contain bytes of arbitrary values** is not considered a problem; whenever a document needs to pass a legacy medium that is not 8-bit clean, a common data encoding like `base64` or `hexdumping` can be applied.

Note 3.11. The **inability to circumscribe Unicode code points**, a consequence of the complete missing of escape sequences in structured strings, is not considered a problem; the tools used to create and handle StrucSDs should provide any means for entering arbitrary code points that are appropriate to the application and its users at hand.

Note 3.12. *The restriction to support only UTF-8 strings was made to simplify all future applications. Tools and editors converting from and to structured strings can handle necessary conversions for as long as they are required. This pushes the complexity to the fringes, thereby keeping the core clean and simple, rather than forcing every application to handle every legacy character set with our current legacy character-set mayhem which lead to high-profile security vulnerabilities [5] even today.*

Note 3.13. *The choice of UTF-8 is expected to be future-proof and, unlike other encodings like UTF-16 or UCS-4, avoids endian issues, and is therefore more in the spirit of machine-independent string encodings. Documents that would have been smaller with a UTF-16 encoding can be compressed as mentioned in 3.9.*

4 A C++ Implementation

In order to demonstrate the StrucSE in practice, a prototype C++ library to manipulate, encode and decode StrucSDs will be made available for download at [1]. The source archive includes all documentation and build instructions; this section contains more general notes on the implementation that are not immediate to using the library.

Note 4.1. *The implementation is intended to showcase the simplicity of the approach and is therefore not optimised for performance. The error checking and input validation should be complete, however the error messages need to be extended for production use.*

Note 4.2. *The included HTML generator does not correctly quote and escape according to the HTML standard. Adding this is left as an exercise to the reader that serves to illustrate the complexity of the issue, complexity that is eliminated when using the StrucSE.*

Note 4.3. *In order to simplify installation, this prototype does not have requirements beyond the standard C++ library. For real-world code we highly recommend the use of additional libraries wherever possible, e.g. Boost's [7] `shared_ptr` facility rather than the included `util::pointer`.*

5 Conclusion

This paper showed how a small, simple and generic additional layer between plain strings and languages like SQL and XML can simplify the implementation of applications and libraries dealing with these languages, close some security vulnerabilities that are based on manipulating the data structure, and potentially have positive effects on performance. This paper is only the beginning, with some of the most obvious next steps being:

- Finalise the specification of the structured string encoding to make it fit for real-world use.
- Look for further security related use cases and benefits, and general use cases like JSON.
- Implement basic tools like editors and diff/patch for structured strings.
- Implement structured strings in at least one chain of applications: web browser, web server, server-side scripting language, relational database.
- Analyse the full impact of basing XML and HTML on structured strings and formulate a legacy-free version of the corresponding standards.

6 Acknowledgments

The author would like to thank Thomas Lewandowski for his review of an early version of this paper, and Tobias Haustein for the interesting discussions that led to its inception.

References

- [1] “Prototype Implementation of Structured Strings”
<http://www.umbrialogic.com/research.html>
- [2] “Common Weakness Enumeration – Vulnerability Type Distribution”
<http://cwe.mitre.org/documents/vuln-trends.html>
- [3] “Document Object Model (DOM)”
<http://www.w3.org/DOM/>
- [4] “SecurityFocus’ BugTraq Mailing List”
<http://www.securityfocus.com/>
- [5] “Multiple Products Full/Half Width Unicode Detection Evasion Vulnerability”
<http://www.securityfocus.com/bid/23980>
- [6] “US-CERT’s National Cyber Alert System”
<http://www.us-cert.gov/>
- [7] “Boost C++ Libraries”
<http://www.boost.org/>
- [8] “Do We Really Need a Security Industry?”
http://www.wired.com/politics/security/commentary/securitymatters/2007/05/securitymatters_0503