Language-agnostic Injection Detection

Lars Hermerschmidt
AXA Konzern AG, Germany
Email: lars.hermerschmidt@axa.de

Andreas Straub
RWTH Aachen University, Germany
Email: andreas.straub@rwth-aachen.de

Goran Piskachev
Fraunhofer IEM, Germany
Email: goran.piskachev@iem.fraunhofer.de

***Abstract*—Formal languages are ubiquitous wherever software systems need to exchange or store data. Unparsing into and parsing from such languages is an error-prone process that has spawned an entire class of security vulnerabilities. There has been ample research into finding vulnerabilities on the parser side, but outside of language specific approaches, few techniques targeting unparser vulnerabilities exist.**

This work presents a language-agnostic approach for spotting injection vulnerabilities in unparsers. It achieves this by mining *unparse trees* using dynamic taint analysis to extract language keywords, which are leveraged for guided fuzzing. Vulnerabilities can thus be found without requiring prior knowledge about the formal language, and in fact, the approach is even applicable where no specification thereof exists at all. This empowers security researchers and developers alike to gain deeper understanding of unparser implementations through examination of the unparse trees generated by the approach, as well as enabling them to find new vulnerabilities in poorly-understood software.

I. Introduction

Injections are the most widespread and critical vulnerabilities according to the OWASP Top 10 [1]. They come in multiple names, hence Cross Site Scripting (XSS) (Top 7 on the list) is just one variant of the general injection problem.

For prominent languages, like HTML and JavaScript, injection vulnerabilities can be mitigated by using existing frameworks that correctly generate documents of such languages. These frameworks prevent injections by utilizing unparsers that correctly encode all data which developers pass to a HTML or SQL document.

However, injections may occur in every output of a program, not only in these well known languages. Figure 1 provides an overview of this generalized situation. More specifically, whenever an unparser takes a typed representation of data and serializes it to an untyped one, i.e. a String, the syntax of the serialized representation has to preserve the type information. If the unparser fails to enforce that syntax for some input, this is called an injection vulnerability. Note, that this untyped representation or document does not have to be the output of a program. Commonly it is passed to another component within a system, where a parser is used to reconstruct the typed representation, i.e. a parse tree (PT).

Following the LangSec mantra "treat all valid inputs as a formal language" [2], [3], the language an unparser produces as output has to be defined in a context-free grammar as well. Since producing output is seen as an easy task, programming languages offer only the ability to write Strings without defining their syntax. Therefore, developers implicitly define the output language of their program and have to write encoder code by hand, which is known to be error prone [4]. To prevent injections, developers should define the output's grammar, use an unparser generator like McHammerCoder [5], and use the generated unparser to produce valid encoded documents of the defined language. Sadly, usage of unparser generators or equivalent constructs in programming languages that automatically generate an encoding and apply it is rare. And developers unintentionally create more undefined languages every day. For example, programs write records to a file or stream one by one, separated by a new line and rely on this separation when reading the records. This implicitly creates a language with new line as keyword. However, during unparsing into that language this keyword is commonly not encoded, which results in an injection vulnerability.

In this paper, we investigate a dynamic analysis method to identify injections in existing source code. Of course, there is a whole tool industry for static and dynamic application security testing that tries to spot injections for well-defined, previously known languages, like HTML and SQL. However, existing tools are not made for detecting injections in languages their inventor has never seen before. This is for good reason, because their task at hand is already undecidable since they face the halting problem which results in false positives and thus poor user acceptance, even for known languages.

In order to reach out to developers, inform them about injection vulnerabilities in their code, and change their approach to writing unparsers, we are fine with entering the land of false positives, since the goal is not to solve the problem once and for all, but instead to raise awareness. Compared with existing approaches from industry and academia for injection detection [6]–[8], we do not limit the search to know languages but include those carelessly created.

To accomplish that, we apply grammar mining techniques – which `AUTOGRAM` [9] uses for parsers – to unparsers and call this approach `MARGOTUA` as this is the reverse problem. By doing so we extract keywords of the language produced by unparser code. These are used in a fuzzing step to create inputs that cause injections. To determine if an input successfully exploited an injection vulnerability, `MARGOTUA` compares the parse tree of a known good input with those created by the fuzzer. Clearly, our proof-of-concept implementation relies upon some assumptions to successfully detect injection vulnerabilities in existing Java unparser code, which we also discuss in this work. Most notably, it requires the user to identify the unparser's entry point.

© 2020, Lars Hermerschmidt. Under license to IEEE.
DOI 10.1109/SPW50608.2020.00060

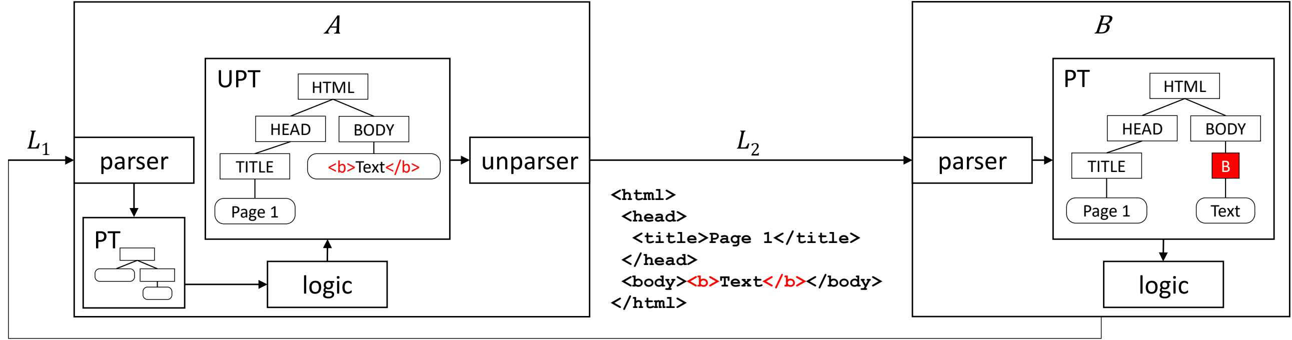


Fig. 1: Overview of the components that are involved in an injection vulnerability. HTML is used as an example for L_2

The remainder of this paper is structured as follows: We first review related work. In Section III, we lay out the general problem of injection vulnerabilities and provide a more formal definition. MARGOTUA is presented in Section IV, and we evaluate this approach and its limitations by assessing multiple unparser implementation in Section V. By highlight future work and drawing a conclusion in section VI, we close the paper.

II. Related Work

One of the few works targeting the security of unparsers was proposed by Hermerschmidt et al. [10]. The authors focused on unparsing and encoding malicious user data into a document whose language is defined by a grammar containing the context sensitive encoding rules. They generated encoder and decoder for HTML and JavaScript and evaluated them using penetration testing techniques. Their follow up work [5] showed how encoders can be derived. Although they provide a mitigation by construction to the injection problem they do not, however, search for specific injection vulnerabilities.

While there is very little work targeting unparsers, interesting results have been achieved in the analysis of parsing code. Notably, Höschele et al. [9] have pursued extraction of the grammar recognized by a given parser using AUTOGRAM. They leverage dynamic taint analysis to match up the typed outputs of the parser with their corresponding input fragments. They make use of call graphs from traced fuzzing runs in order to infer the hierarchical structure in which the input is processed. From this, they are able to synthesize very good approximations of the grammar covered by the fuzzed inputs. Their approach is applicable to data understanding, program understanding, test generation, and program decomposition.

A static analysis approach is used by Doh et al. [11] to analyze PHP programs that dynamically generate HTML documents as strings. It compares data-flow information extracted from the programs with a context-free reference grammar. This reference grammar has to be known ahead of time. Another drawback to this approach is that the programs need to be annotated at all points where critical strings are generated.

III. Formal Problem Definition

To better illustrate our approach, we first describe the general sequence of actions necessary for an injection vulnerability to occur which is illustrated in Figure 1. An injection attack is launched by providing a program A some input document written in a language L_1. From this document, A's parser creates a program internal representation of this document, called parse tree (PT), where individual values are typed. When processing the document, A's logic operates on the input via the PT. L_1 has to be defined by a grammar [2], so that documents can be formally recognized, i.e. accepted as a member of the language or rejected.

At some point, the program A reaches out to another component B, so it has to create a document written in a language L_2 that is sent to B. To accomplish this, in an ideal world, A creates an unparse tree (UPT) that is the in-program, typed representation of the document it is about to send. Next A calls the unparser for L_2 to transform the typed information from the UPT into a document that contains the untyped representation conforming to the formal definition of L_2. This setting also describes the prominent example of XSS where B, i.e. a browser, originally sends a request written in L_1, i.e. HTTP, to A and receives a document wrapped in HTTP that is written in L_2, i.e. HTML.

The receiving component B, being the target of the injection attack, uses a parser for L_2 to create a parse tree (PT) containing a typed representation of the document. The parser of B relies on the syntax of L_2 to correctly process the document. It has no means to detect or prevent an injection.

At the core of an injection attack is to change the context in which B's parser of L_2 interprets input that has been handed over to A in a document written in L_1. Note that a change in context results in a *modified* PT created by B's parser. Regarding textual languages, the contexts during parsing are separated by keywords of the language. Therefore, to perform an injection attack, keywords of L_2 have to be smuggled through the parser of L_1, A's program logic, and the unparser of L_2 to change the PT of L_2 within the receiving component B. Counterintuitively for developers, this changes the intended semantics of the document noted in the program's unparser code.

Since legitimate documents written in L_1 might contain elements of the syntax of L_2, it is not A's parser, but A's unparser which is responsible to encode these entries, such that they do not interfere with the syntax of the created L_2 document. If an unparser fails to do so we call it an injection vulnerability. More formally, an unparser that does not ensure a correct unparser-parser round-trip $parse(unparse(t)) = t$ for any given UPT t has an injection vulnerability [10]. Note that this definition for injection vulnerabilities and the solution also holds for binary languages that do not rely on keywords, but on length fields to separate contexts within the parser. To perform an injection into a binary language the unparser of L_2 has to fail in validating the length of some content and be tricked into overwriting the length field.

However, in real world software, the aforementioned clean structure mostly does not exist. Therefore, we face additional challenges to determine if an injection is present in a given unparser implementation. First of all, there is no single UPT t, because, after parsing, t is a PT. This means the UPT and the PT of a language commonly are constructed by two different implementations, resulting in different object structures not necessarily holding the same data. Therefore, the formal definition for a correct unparser-parser round-trip can not be used directly to check unparser correctness. As presented in Section IV, we rather only assess the PT for deviations from a known good sample to detect injections.

In addition, there exists unparser code which implicitly defines a language and skips the creation of a UPT. Since, from the code itself, it is often not obvious which keywords exist in this language, the unparser code and hence the language is commonly missing proper encoding for the language's keywords. One could call this anti-pattern a shotgun unparser. If the language does not have syntax for encoding, fixing the injection vulnerability requires fixing the language first. Afterwards, all (un)parsers for the language have to be updated to process the encoding correctly.

Although there is a solution and general guideline on how to eliminate the formal problem of injection vulnerabilities [5], [12], in practice it is rather difficult to enforce such a solution on a codebase without the ability to automatically spot violations. Therefore, we strive to demonstrate the scale of the problem and its impact by providing such an analysis in order to raise developers' awareness for using correct unparsers.

An ideal analysis should automatically detect unparsers in source code, injection vulnerabilities within these unparsers, and at best create sample exploits to prove the impact. One could also consider to integrate the analysis into Integrated Development Environments (IDEs) to notify developers immediately whenever they write ad-hoc unparser code. Ideally, a sustainable solution is to eliminate the possibility of creating insecure unparsers in the first place. However, this remains a challenge for future programming languages.

In this work we focus on the problem of identifying injection vulnerabilities within a given unparser for a textual language to highlight the existence of such injections within existing implementations.

IV. Automatic Analysis of Unparsers

To approach our ultimate goal of identifying injection vulnerabilities in unparser code according to our previously given definition, we created `MARGOTUA`, the concept of which we lay out in the following. `MARGOTUA` targets Java applications, since it is one of the most popular programming languages with rich open source communities and relevance in enterprise applications that process data in custom languages.

We use techniques inspired by `AUTOGRAM`'s approach, through which they obtain a context-free grammar for given program inputs [9]. However, in our current approach, we do not extract the full grammar, but only the set of keywords of the language, and the contexts in which these are valid. This, by itself, is already very useful information, as these keywords control the contexts, which influence parsers when creating the parse tree. Therefore, knowledge about them can be leveraged in guided fuzzing of the unparser in order to find injections. In the following, we describe the steps `MARGOTUA` performs in a pipeline in more detail.

A. Deriving Unparse Trees

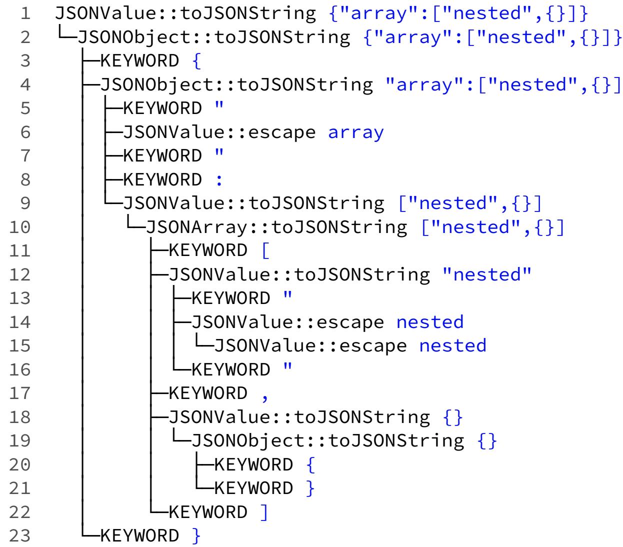
Höschele et al. [9] have shown that associating parts of a program with the input they process is a viable approach to deduce the grammar this input adheres to. From bare string input they obtain a parse tree, representing the syntactic structure of that input. By repeating this for multiple inputs, collecting all the corresponding parse trees, they are able to synthesize a context-free grammar describing the inputs.

We adapt this idea to the inverse scenario of unparsing. In such software, given structured data as an input, a bare string output is generated. Our goal is to correlate each part of this structured input data with a corresponding slice of the bare output string that is derived from that part of the input. We can then also identify those parts of the output that have no counterpart in the input, meaning they are created during the unparsing process. These are the keywords of the language. Together, this gives us a UPT that contains structural information on how the output can be decomposed into its constituent parts of input fragments and keywords.

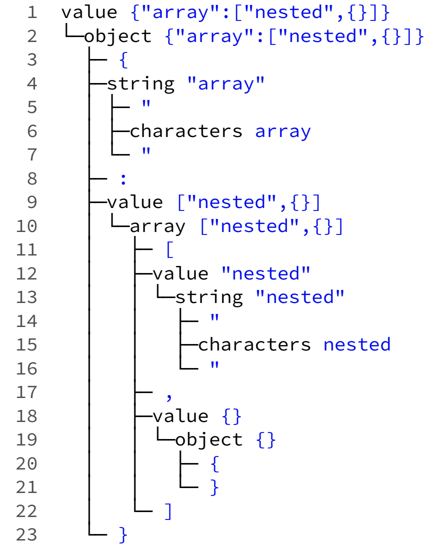
To extract UPTs from given unparser code we observe the running program via tracing through the Java Debug Interface (JDI) while it transforms the Java object structure representing the UPT into untyped output. `MARGOTUA` achieves this by assigning a unique taint to each object of the UPT using the dynamic taint analysis system Phosphor [13]. This enables us to track value propagation for them simultaneously.

Through JDI, the locally visible program state is constantly snapshotted during execution. This way, a snapshot for each method call, return, as well as source code line is produced. Notably, the program state contains the variable assignments including their associated taint tags. The collection of all these snapshots make up an execution trace.

From such a trace, we extract the call tree by matching up corresponding call and return events. As is evident in the example in Figure 2a, the call graph structure yielded by this step is already very close to the final UPT. Any method calls



(a) An example UPT for the JSON.simple library derived by MARGOTUA. The output slices associated with each node are shaded blue.



(b) A ground truth derivation of a UPT according to the JSON specification.

Fig. 2: UPTs for the same JSON example.

appear in this tree as nodes named after the containing class and method, separated by two colons. Additional nodes, like those containing keywords, are added later on in the pipeline, during the actual UPT derivation. First, however, the call tree is enriched with return value information, which is shown in the UPT shaded in blue.

Generally, the values returned for each call can be copied directly from the corresponding return event within the trace. This does not suffice for our analysis however, because, as an artifact of optimization, many internal unparser methods return `void`. In Java, for performance reasons, String objects, like the unparser's ultimate output, should generally not be constructed by iterative concatenation. Instead, classes like `StringBuilder` and `StringBuffer` or `Writer` are used to accumulate additions with less overhead. Many unparsers use and modify these objects across method boundaries, passing them along as a return parameter to their subcalls, which then return `void`. As a result, the structure of the output is often hidden behind side effects on such effectively global state.

To handle this pattern and to ensure that return values are present for each method call, we apply a simple heuristic. For methods with no return value, we look for a parameter of one of the aforementioned classes. Using the snapshots at call and return time, we extract the difference in content of this object. The resulting string is added into the call tree, taking the place of the corresponding missing return value.

Finally, a UPT is inferred from the call tree. This is achieved by recursively examining calls, in order to characterize the call's return value. The analysis starts at the unparser's entry point, which must be a method call returning a string. This entry point needs to be indentified by the user. The return value is decomposed into parts of the original input to the program, keywords created in the call, and values returned from subcalls. Keywords are identified by their lack of taint. In some unparsers, like the example from Figure 2a, these intermediate return values can always be cleanly decomposed into subcall return values and keywords. However, input data fragments may also appear in variables.

One example where analyzing these variables is necessary is shown in Figure 3. This is taken from an unparser for URLs which is implemented in the `toString` method of the class. As this method has direct access to the individual parts stored in fields, which it essentially just needs to concatenate correctly, it makes little use of subcalls. By incorporating variables into the decomposition analysis, these scenarios can be handled gracefully.

The keywords that were identified as previously described are inserted into the UPT as `KEYWORD` nodes. For variables, nodes named according to the variable's identifier are introduced. At each level of the tree, the nodes are positioned according to their order in the decomposition of the parent node. This ensures that an in-order concatenation of the associated values of sibling nodes always yields the value associated with the parent.

At this point, we are able to deduce the UPT for an individual input. Given a corpus of inputs, one could now proceed in similar fashion to AUTOGRAM and generalize the resulting set of UPTs into a grammar. Thus, we instead opt to directly use the UPT to guide a targeted fuzzer with keywords and their location.

```
1  URL::toString http://user:pass@example.com/
2  │              path?query#fragment
3  ├─scheme http
4  ├─KEYWORD :
5  ├─KEYWORD /
6  ├─KEYWORD /
7  ├─URL::userInfo user:pass
8  │ ├─username user
9  │ ├─KEYWORD :
10 │ └─password pass
11 ├─KEYWORD @
12 ├─Host::toHostString example.com
13 │ └─Domain::toString example.com
14 ├─path /path
15 ├─KEYWORD ?
16 ├─query query
17 ├─KEYWORD #
18 └─fragment fragment
```

Fig. 3: An example UPT for the galimatias library derived by MARGOTUA. The output slices associated with each node are shaded blue.

B. Information extraction

To identify injections we are mainly interested in determining the keywords of the language, since they have special meaning to a parser of the language. For example, in JSON, string literals are enclosed in double quotation marks (`"enclosed string"`), items in collections are separated by comma characters (`[1, 2, 3]`), etc.

We leverage the syntactic information contained in UPTs about the context of keywords within documents of the language produced by the unparser. Especially those keywords detected next to an input data fragment in the UPT can be used in fuzzing to likely produce injections within that part of the input.

With this information at hand, we can very easily assemble injection candidate in the next step.

C. Fuzzing

The last step of the MARGOTUA pipeline is a guided fuzzer, depicted in Figure 4. It uses the keyword information to generate potential injection attacks and evaluates them by subjecting both the injection and the original input data to an unparser-parser round-trip and comparing the results.

The first step of this process is the generation of the injection attempts. This is done by generating injection candidates, which are fragments that can then be inserted in a specific place into the otherwise unmodified input. We call this place the *site* of the injection. Candidates are made up of two parts: a prefix and a payload.

The prefix is constructed from keywords that were found next to the output slice derived from the input at the *site*. Their purpose is to ”escape” the syntactical construct immediately enclosing the *site* when it is unparsed.

The payload is constructed from all keywords that were extracted from the UPT. This way, valid fragments with syntactic significance can be generated.

Through combination of such prefixes and payloads, we obtain candidates that have a high likelihood of changing the syntactic structure of the output generated by the unparser, i.e. a high likelihood of a successful injection.

Let us use the UPT from Figure 2a as an example, where we are attempting to inject into the target `nested` in line 14. The candidate prefix in this case would be the double quote character `"`, as it occurs directly after the target in line 16. Payloads could then be randomly generated from any selection of the other keywords.

These payloads are passed to the unparser to test the implementation. At the moment MARGOTUA does not recognize the interface of the unparser code automatically. Hence, in order to run MARGOTUA on a test subject, a small piece of code has to be written to adapt the unparser to a pre-defined interface. We call this piece of code the *stub*. It has to provide the following prerequisites:

1) The *stub* has to mark the subject unparser’s entry point in order to make targeted analysis of just the unparsing code possible.
2) The *stub* must prepare the input objects for the unparser. It is not yet possible to generate these inputs automatically.
3) It must also provide the ability of inserting an injection candidate supplied by MARGOTUA into such input objects.
4) For some subjects that handle unordererd collections, simply comparing the *benign* and *malicious* parser outputs directly is error-prone, because there might be undefined behavior in how precisely the collection’s component items are ordered at parse-time. In order to significantly reduce false positives in theses cases, the *stub* also has to provide a comparison metric for parser outputs.

In order to identify whether an injection attempt is successful, we utilize an unparser-parser round-trip as described in section III, wherein UPTs are unparsed by the given unparser code and the results are parsed again. As described earlier however, object structures can differ between unparser and parser, so we can not simply compare the unparser input with the parser output.

To resolve this issue, we first run the unparser-parser round-trip on a benign input. This yields an object structure as emitted by the parser, the *benign output*. Next, we apply our injection candidate to the same input, preserving its structure. We can then run the unparser-parser round-trip a second time, now using this input modified with the injection candidate. This again yields an object structure from the parser, the *malicious output*. Since we did not change the structure of the input when applying the modification, the two outputs we have collected should also have the same structure if encoding and decoding is implemented correctly. If we find a difference in the structures between *benign* and *malicious* output, however, this means we have identified an injection.

Using this method, we can evaluate whether a given candidate actually produces a successful injection. We can thus use the previously described mechanisms for constructing candidates and evaluating them to perform very targeted fuzzing of

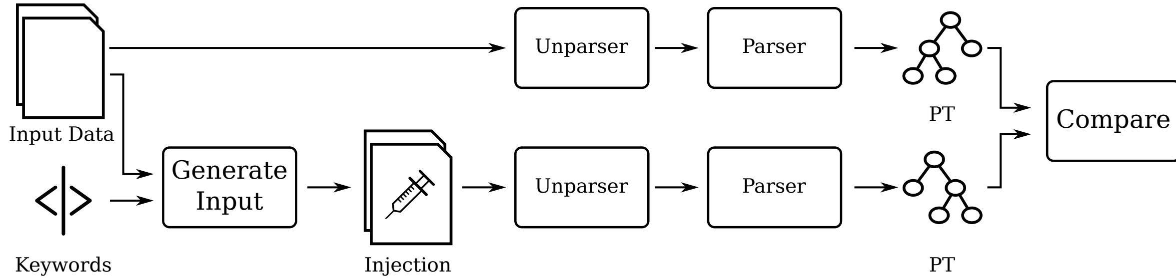


Fig. 4: The fuzzing process

the unparser. Even in the presence of parser differentials [14] among the unparser and corresponding parser the approach is applicable, since even if unparser and parser disagree on the language the resulting change of the parse tree can be detected.

V. Evaluation

During the implementation of `MARGOTUA`, we identified some limitations of the approach which we find worth mentioning. We first present these limitations and subsequently discuss them in the context of a case study of three unparser implementations.

Most notably, unparser code that only generates outputs of a very small subset of a language can only be used to extract that subset. One prominent example of this are applications that build up ad-hoc SQL queries using string concatenation. Commonly, large parts of such SQL statements are predetermined by string constants containing the SQL statement with just one variable containing input. Since only this variable crates structural variability in the output, `MARGOTUA` can not identify individual keywords of the SQL language but rather recognizes the string constants used. Hence, `MARGOTUA`'s fuzzer does not detect such injections.

Due to the proof-of-concept state of `MARGOTUA`'s implementation the taint tracking is not implemented for primitive types (not including arrays with primitive types) in the tracing step. As a result, parts of inputs that are of a primitive type can not be identified correctly in the output, degrading the quality of the UPT. However, since we are targeting injections of textual languages, which are constructed from strings, the impact of this limitation is low for the purposes of our evaluation.

At the moment `MARGOTUA` does not recognize the interface of the unparser code automatically. Hence, in order to run `MARGOTUA` on a test subject, a small piece of code has to be written to adapt the unparser to a pre-defined interface. We call this piece of code the *stub*. It has to provide the following prerequisites:

1) The *stub* must prepare the input objects for the unparser. It is not yet possible to generate these inputs automatically.
2) It must also provide the ability of inserting an injection candidate supplied by `MARGOTUA` into such input objects.
3) The *stub* has to mark the subject unparser's entry point in order to make targeted analysis of just the unparsing code possible.
4) For some subjects that handle unordererd collections, simply comparing the *benign* and *malicious* parser outputs directly is error-prone, because there might be undefined behavior in how precisely the collection's component items are ordered at parse-time. In order to significantly reduce false positives in theses cases, the *stub* also has to provide a comparison metric for parser outputs.

Since `MARGOTUA` requires a stub for each individual target it assesses, we evaluate our approach through the lens of case studies on open source software. As we have already hinted at in Section II, there are no approaches targeting the same problem we are addressing. Therefore, no direct comparisons to other works are possible. Instead, our first goal was to assess the general viability of `MARGOTUA` by observing the UPTs it is able to extract. By comparing such intermediate results of our processing pipeline with known ground truths, we aim to identify strengths and limitations of this technique. We will show some examples to reason about which software patterns our approach is able to correctly analyze, and where it falls short, giving rise to future work.

Ultimately though, we are most interested in actually finding injection vulnerabilities in unparsers. To test for such capabilities, we modified the subjects to insert deliberately placed vulnerabilities. Specifically, we disabled encoding steps present in the software so as to simulate missing encoding. We then ran `MARGOTUA` to examine whether the vulnerabilities we implanted could be identified.

A. Subjects

We selected evaluation subjects handling languages that cover a variety of different grammar structures, like repetition and recursion. We target libraries for one specific language as opposed to general utility libraries or large applications, as such more minimal and specific pieces of software lend themselves more easily to targeted examination. As our approach identifies injections through unparser-parser round trips, the evaluation targets need to include both an unparser as well as a parser for the same language.

1) JSON.simple: JSON.simple [15] is a very popular library for the JSON [16] format, with over 47000 Github repositories depending on it. As can be observed in Figure 2a, MARGOTUA is able to derive very accurate UPTs for this library. In comparison with a ground truth derivation for the same example, shown in Figure 2b, it is apparent that the overall structure is very similar. One notable difference, however, is that the derivations for the `string` nonterminal is inlined in the UPT extracted on this example by MARGOTUA. This is the case because the unparsing of string values is similarly inlined in the implementation of the library itself, i.e. it is not handled by a separate method, but instead performed directly whereever such a method would be called from. Because structural distinction is lost in this kind of optimization, the accuracy of the extracted result suffers by necessity.

Whereever a `string` appears as a value in an `object` or as part of an `array`, the impact of this loss of information is low, because in these cases the extracted UPT simply lacks the additional indirection that is present in the ground truth derivation. This can be observed comparing line 12ff. in Figure 2a and Figure 2b. Conversely, as in JSON, `object` keys can only ever be strings, they do not appear as descendants of a `value` in the ground truth derivation (see line 4 of Figure 2b). For these, the inlining results in a more significant loss of structure, as is evident in line 5ff. in Figure 2a. Here, the string's contents and its delimiting keywords can not be distinguished from the other sibling nodes, including the keyword `:`. As a result, the quality of the context information for the fuzzer is lower.

This example illustrates that accurate UPTs can be extracted from execution traces. However, some kinds of implementation optimizations, like inlining, can degrade the accuracy of these results.

In the library's original state, we were not able to find any injection vulnerabilities. This is not surprising, as the library has seen widespread use and can thus be assumed to be well-tested. For our evaluation, we therefore disabled the encoding performed on string inputs in the `JSONValue::escape` method. On this patched variant of the library, MARGOTUA was consistently able to find injections within fewer than 100 tries in the fuzzing stage, with an average of 10 tries. When disabling the keyword context analysis, i.e. using only the knowledge of the overall set of keywords for fuzzing, these statistics already more than double. This speaks to the relevance of the extracted information.

2) galimatias (URL): galimatias [17] is a parsing and normalization library for URLs. We have already seen an excellent result for this subject in Figure 3. This UPT represents the correct syntactic structure of the example, and the node namings extracted from variables and fields accurately conveys their semantic meanings. One shortcoming that can be observed is that the two slashes (`//`) in lines 5-6 are not grouped in this UPT even though they represent one syntactic unit. This comes down to an implementation detail, however, and is not representative of a limitation of the approach itself.

URLs always follow the same basic structure, as they do not feature recursion or repetition. It follows that UPTs for them are accordingly similar to one another. This template-like format presents a problem for our approach when fuzzing. In order to assess injection success, as described in Section IV, we compare parser outputs. Because the structural variability is very limited for this subject, injections can not consistently be identified by our structural comparison heuristic.

```
1  CsvWriter::write 1,"two,",three
2  ├─KEYWORD 1
3  ├─KEYWORD ,
4  ├─KEYWORD "
5  ├─KEYWORD t
6  ├─KEYWORD w
7  ├─KEYWORD o
8  ├─KEYWORD "
9  ├─KEYWORD ,
  ...
```

Fig. 5: A truncated UPT extracted from a CSV library.

3) CSV: vsg-csv [18] is a library for the CSV file format [19]. This subject is one example our implementation currently can not handle. The reason for this is that this library does not make use of any subcalls. Instead, everything is inlined into one method, which handles the entire input in one big loop. As a result, our approach can not deduce any of the structure at all. This is demonstrated by the truncated UPT in Figure 5

This is not a true limitation of the underlying approach, however. Such cases could be handled by making more use of the line-by-line program state snapshots gathered in the tracing phase. Local variable state could be examined at each step, and all of these intermediate values used for the return value decomposition analysis as described in Section IV. Our current implementation simply lacks this feature.

VI. Conclusion and Future Work

In this work, we presented MARGOTUA, an approach to identify injection vulnerabilities in unparsers. Using grammar mining techniques, MARGOTUA extracts a UPT by observing the unparser execution. From that UPT, MARGOTUA infers the keywords of the language which the unparser produces. Guided fuzzing focusing on these keywords eventually reveals injection vulnerabilities in the unparser.

Fostering adoption of LangSec principles and a more widespread use of solutions that prevent injection vulnerabilities requires raising awareness for the problem. Therefore, we choose to identify vulnerabilities in existing code bases to bring the call to action right to developers of those unparses.

To enable developers to assess their own unparser implementation for injection vulnerabilities, we implemented MARGOTUA. Evaluating MARGOTUA on Java unparser libraries, we found that MARGOTUA is able to derive very accurate UPTs in *JSON.simple* and *galimatias (URL)*. However, inlining optimizations in the implementation can degrade the quality of these results. We further showed that MARGOTUA is a viable approach for identification of injection vulnerabilities for textual languages.

Also when faced with the question of whether to use an existing unparser library, MARGOTUA can help developers to identify vulnerabilities before they use the library. Notifying creators and users of existing unparsers hopefully creates awareness and thus improve adoption of known good solutions.

To this end, we encourage developers faced with injection vulnerabilities in their code to use a known solution or come up with an improved one that better fits their needs.

Although generating injection exploits is of interest to better illustrate the impact of injection vulnerability at scale, it still requires a human to judge on the system's context and the impact of the vulnerability. Therefore, it is questionable if this path of improvement of our approach really has value.

In contrast, extending MARGOTUA to extract complete grammars from unparsers is of value for developers. This would enable them to use unparser generators like McHammerCoder [5] to derive a correct unparser and encoding implementation from that grammar.

Since this work focuses on textual languages the detection of injection vulnerabilities in binary languages is an open problem. This includes the challenge to automatically identify length fields and their relation to other parts of the grammar.

Another avenue for improvement would be automatic detection of unparsers without having to write any stub code. When integrating MARGOTUA into the software development life-cycle, developers might be warned as soon as they create an unparser and whether they choose an appropriate solution.

However, to radically improve the security in input processing, (new) programming languages resp. their core libraries shall adopt LangSec concepts for parsing and unparsing. Rather then using pure string operations like split and concatenation, developers would then use parsers and unparsers per default. This way developers have the tools at hand to create a secure implementation.

REFERENCES

[1] A. van der Stock and OWASP Community, "Owasp Top 10," https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[2] L. Sassaman, M. L. Patterson, S. Bratus, and A. Shubina, "The Halting problems of network stack insecurity," *;login:, Usenix Magazine*, vol. 36, no. 6, pp. 22–32, 2011.

[3] S. Bratus, T. Darley, M. Locasto, M. L. Patterson, R. b. Shapiro, and A. Shubina, "Beyond Planted Bugs in "Trusting Trust": The Input-Processing Frontier," *IEEE Security & Privacy*, vol. 12, no. 1, pp. 83–87, January 2014.

[4] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A Systematic Analysis of XSS Sanitization in Web Application Frameworks," in *Computer Security – ESORICS 2011*, ser. LNCS. Springer, 2011, vol. 6879, pp. 150–171.

[5] T. Bieschke, L. Hermerschmidt, B. Rumpe, and P. Stanchev, "Eliminating Input-Based Attacks by Deriving Automated Encoders and Decoders from Context-Free Grammars," in *Security and Privacy Workshops*. IEEE, 2017.

[6] GitHub, "Semmle," https://semmle.com/.

[7] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: Effective taint analysis of web applications," *SIGPLAN Not.*, vol. 44, no. 6, p. 87–97, Jun. 2009.

[8] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: Accurate and scalable security analysis of web applications," in *Fundamental Approaches to Software Engineering*, V. Cortellessa and D. Varró, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 210–225.

[9] M. Höschele and A. Zeller, "Mining input grammars with AUTOGRAM," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 31–34.

[10] L. Hermerschmidt, S. Kugelmann, and B. Rumpe, "Towards More Security in Data Exchange: Defining Unparsers with Context-Sensitive Encoders for Context-Free Grammars," in *Security and Privacy Workshops (SPW)*. IEEE, 2015, pp. 134–141.

[11] K.-G. Doh, H. Kim, and D. A. Schmidt, "Abstract parsing: Static analysis of dynamically generated string output using lr-parsing technology," in *Proceedings of the 16th International Symposium on Static Analysis*, ser. SAS '09. Berlin, Heidelberg: Springer-Verlag, 2009, p. 256–272.

[12] S. Bratus, L. Hermerschmidt, S. M. Hallberg, M. E. Locasto, F. Momot, M. L. Patterson, and A. Shubina, "Curing the vulnerable parser: Design patterns for secure input handling," *;login:, Usenix Magazine*, vol. 42, 2017.

[13] J. Bell and G. Kaiser, "Phosphor: Illuminating dynamic data flow in commodity jvms," *Acm Sigplan Notices*, vol. 49, no. 10, pp. 83–101, 2014.

[14] S. Bratus, T. Darley, M. E. Locasto, and M. L. Patterson, "Langsec: Recognition, validation, and compositional correctness for real world security," *USENIX Security BoF Handout*, 2013. [Online]. Available: http://langsec.org/bof-handout.pdf

[15] F. Yidong, "JSON.simple," https://github.com/fangyidong/json-simple, 2014.

[16] ECMA, *ECMA-404: The JSON Data Interchange Format*. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), 2013. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-404.htm

[17] S. M. Mola, "galimatias," https://github.com/smola/galimatias, 2018.

[18] V. Garnashevich, "csv," https://github.com/vsg/csv, 2017.

[19] Y. Shafranovich, "Common Format and MIME Type for Comma-Separated Values (CSV) Files," RFC 4180, Oct. 2005. [Online]. Available: https://rfc-editor.org/rfc/rfc4180.txt