

Recognizing Dynamic Fields in Network Traffic with a Manually Assisted Solution^{*}

Jarko Papalitsas, Jani Tammi, Sampsa Rauti, and Ville Leppänen

University of Turku, 20014 Turku, Finland
{jastpa,jasata,sjprau,ville.leppanen}@utu.fi

Abstract. Payloads of packets transmitted over network contain dynamic fields that represent many kinds of real world objects. In many different applications, there is a need to recognize and sometimes replace these fields. In this paper, we present a manually assisted solution for searching and annotating dynamic fields in message payloads, specifically focusing on web environment. Our tool provides a simple and intuitive graphical user interface for annotating dynamic fields.

1 Introduction

Messages transmitted over network contain dynamic fields that often correspond to many kinds of real world objects, for example names and cities. In many cases, such as when we want to edit these fields or filter some of them out, we need to recognize and possibly replace parts of message payloads. An example of a real world application of dynamic field recognition would be deceiving a malicious adversary by providing him or her false information in a response message by altering the values of dynamic fields [1, 2].

The challenge here, especially in the diverse web environment, is that message payloads come in many different forms and it is impossible to create a solution that would automatically recognize all dynamic fields in all messages conforming to different kinds of application-specific protocols. Because such automatic protocol-independent recognition is infeasible [11], we present a manually assisted solution for recognizing and annotating dynamic fields in message payloads. Our solution works with protocols running on top of HTTP in the web environment.

In this paper, we present a solution that recognizes dynamic fields based on previously recorded transactions (request-response pairs). The user can then revise and confirm these fields using a graphical user interface. Once the dynamic fields have been recognized and revised, the recorded transactions are altered and replayed according to our needs, for example to feed fallacious information to a malicious adversary. Because this solution involves recording and replaying transactions, we call it the "record and play" -approach (see also [3]).

^{*} The authors gratefully acknowledge the support of The Scientific Advisory Board for Defence (MATINE).

The rest of the paper is structured as follows. In Section 2, the challenge of automatic field recognition is explained more closely. Section 3 discussed the infeasibility of fully automatic detection of dynamic fields in messages. Section 4 introduces our solution for dynamic field recognition, including the graphical user interface for manual field annotation. Finally, Section 5 concludes the paper.

2 Dynamic field detection

Our approach to entity recognition (see also [7, 10]) is based on the idea that the interesting parts of the data which include possible entities are the ones varying between similar traffic extracts. The data we are looking for is mostly named entities in custom formats. We want to preserve the non-entity parts of data valid for re-use without the need for interpreting the custom formats. The custom formats can be HTML, JSON or XML for example. As such, more traditional methods such as plain string matching the entities or matching with regular expressions are not suitable for our purposes [8]. This paper focuses on the problems and solutions that arise from recognizing these fields using data from multiple transactions. To explain the challenges and solutions following, some of the terminology used will be explained next with help of Figure 1. Plain recorded data in database is saved in request-response transaction pairs. The header and contents of the transactions are called **transactions**, **transaction samples** or simply **samples** in this paper. Depending on the context it may also only refer to the response header and contents.

As similar samples are grouped together by a suitable method – in our example case of HTTP protocol we use the string matched URL path – the static resources are pruned out by using a combination of length and binary comparison. After these operations, we are left only with **dynamic resources**. The dynamic resources are further processed with similarity analysis where SequenceMatcher of Python difflib [9] is used to determine the similarity of the sample in relation to others.

As stated above, a dynamic resource is a group of samples with specified amount of minor differences. These differences are called **dynamic fields** and they are property of a dynamic resource. Every sample within dynamic resource contains **dynamic field instances** that include the offset and size of the dynamic field within that specific sample. It could be said that the dynamic field is a manifestation of multiple dynamic field instances. Figure 1 as a whole represents a dynamic resource with one sample visible. The dynamic fields are represented by ordinals (#1, #2...). Other instances of the dynamic fields #1 and #5 are listed by their content.

Different dynamic fields may contain same information with each other. For this purpose an **entity** object is used to describe the connections between multiple dynamic fields. As in the previous example, the dynamic fields still contain multiple instances (values). The Figure 2 visualizes the connections between these objects. As an example, let there be an entity called “First Name”, the entity can appear multiple times in the

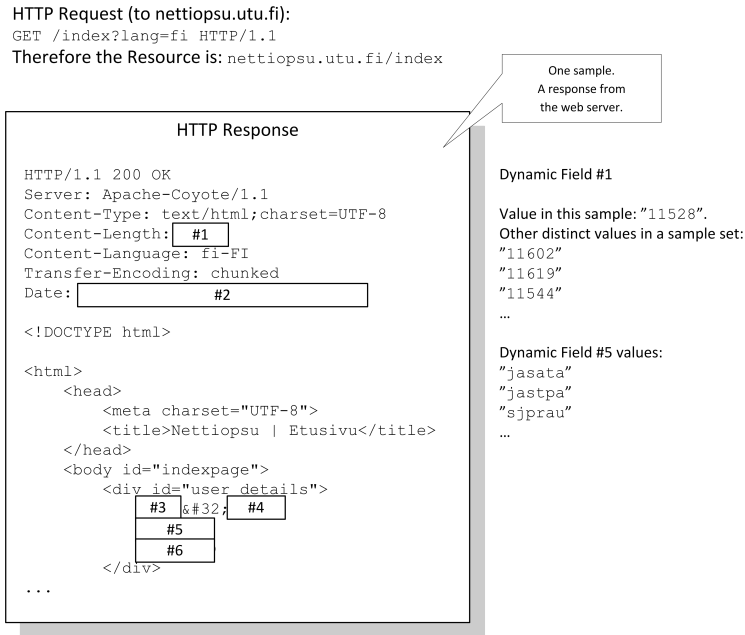


Fig. 1. Dynamic resource, sample and dynamic field

recorded content and as such has multiple dynamic fields assigned. The value of the entity “First Name” will still vary between samples (otherwise it would not be dynamic). For this reason the dynamic field has multiple values or as we called them, dynamic field instances.

3 Why automatic field detection is not feasible

In our approach, a solution functionality akin to a diff tool – displaying the differences between two files – is used to detect the dynamic fields, but with few twists. First, the comparison must span all samples simultaneously [5] so that a field that is without content in some samples will still be marked into all sources (start and end offsets set to equal). In other words, the problem is to find the longest common subsequence (LCS) of multiple strings, which is an NP-hard problem [12]. Second, multiple content specific LCS diff implementations are needed for improved dynamic field detection (see below for detailed description on detection problem). In what follows, we will discuss why such a solution also requires manual assistance when detecting dynamic fields in message payloads.

Given enough variance in samples, the success rate of field detection improves towards logically sound results. Let us consider the following example:

Jani Tammi

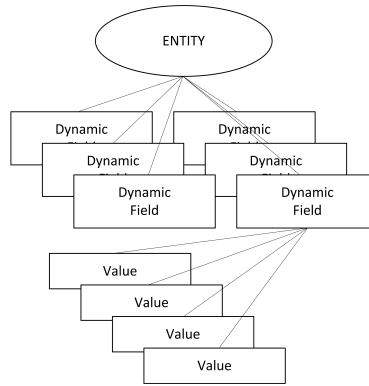


Fig. 2. Entities, dynamic fields and their values (instances)

Jarko Papalitsas

With only two samples to compare, what logically appears as either one field surrounded by tags or two fields separated by white space and surrounded by bold typeface tags, is identified by a field detection algorithm as four fields and five static delimiters (highlighted parts represent dynamic fields). The fourth dynamic field exists in the first name, just after the second i-character and before tag, but its content is empty. Having more variance improves the results as can be seen by adding one more name to our sample set:

Jani Tammi

Jarko Papalitsas

Ville Leppänen

The lack of common characters should now let the algorithm correctly identify two dynamic fields surrounded by tags. But what if names are treated as one field in the web application we are dealing with? It is a structural property of a full name to contain a space character between the first name and the family name. Data might contain a sample that deviates from this and yields us a result indicating just one dynamic field, but no strategy should be built trusting that an abnormal sample that guarantees the correct result will be present.

What about a system where names are treated separately as a first name and family name? While it would not be an issue if either value can be undefined (in which case, for that sample, that field simply appears as empty), we would have undesired results if the formatted output of the service chooses to omit separators between these fields.

This example has been greatly simplified to make understanding the issue easier, but it should still sufficiently explain why operator input

is necessary to verify the results of automatic detection and manually merge or divide fields in order to achieve more accurate field data.

Does this always matter and could some less than perfect field identification results just be accepted without laborious corrections? According to our experiences, in the situations where the range of input and output are finite and they are represented within the sample data, and when such fields do not represent entities that we are interested in, they can be left as they are. Let us consider a service that offers two languages. Sample data will produce a great number of dynamic fields for variance of two. For example, with Finnish and English:

```
<a href="/index >Etusivu</a>  
<a href="/index >Front page</a>
```

This is an example of static page content in two languages, making it appear as two dynamic fields. The input (a query string specifying the display language) and output (string shown above) range is finite and they do not represent real world entities that we would want to change. In this case, when the association is drawn between the input parameter and field results by a separate algorithm, it makes no difference to the results if the generated content is parsed in one or two pieces. In either case, the client receives exactly the same output.

There are borderline cases where the data actually represents entities, but the output variance is finite due to logical constraints (and not only within the scope of sample space). For example, a gender might be limited to two alternatives and bilingual UI would make that field dependant on (at last) two input parameters, producing four possible output values. What is not finite is the input range because while the language parameter may have just two values, the parameter(s) that identify the person are subject to entity faking and can have undefined variance from the perspective of the gender field.

We chose an approach where we deal with only two types of dynamic fields. Those that represent real world entities and those that do not. In the cases where we want to replace the existing field values, entity faking (value generation) is responsible for providing content even with unexpected input values, but this leaves open a question how to deal with unexpected input parameters to dynamic fields that are essentially reduced into look up tables of known input values and output values. Our current approach is to simply pick at random one of the known outputs and add this new input-output pair into the look up table.

4 A solution for manually assisted field detection

As the automatic dynamic field detection needs assistance and has relatively high complexity [12], manual field annotation is required. For possible further processing, being able to recognize which field instances are related to each other is essential. In practice, this means that in addition to finding the dynamic field instances from the transactions (in

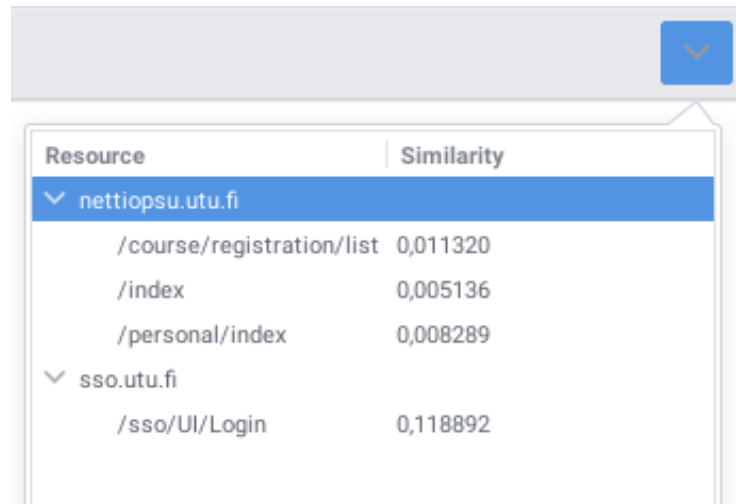
other words the non-static parts of a transaction), they need to be linked together via their dynamic fields.

Every dynamic field instance in every transaction needs to be marked (their offset and size specified) and they need to be linked together via the dynamic field common to each other to be comparable. The amount of manual work is of course dependent on the level of assistance given by the tool and previously performed automatic detection.

4.1 A GUI for manual annotation

Our current method for achieving manual dynamic field marking is a graphical user interface built using GTK 3 [4]. The graphical user interface should be compatible with the previously specified workflow, where multiple transactions within one dynamic resource is detected in order to find dynamic fields. While our implementation is currently very basic in its nature, it is also easy to use and intuitive. More advanced solutions to the problem will be also proposed in this section.

In our current implementation, the previously created database of transactions can be opened and preprocessed if not previously done. After detecting the dynamic records, they can be found on the dynamic resources menu as seen in Figure 3. The resources are ordered by the host and accessed by their path.



Resource	Similarity
▼ nettiopsu.utu.fi	
/course/registration/list	0,011320
/index	0,005136
/personal/index	0,008289
▼ sso.utu.fi	
/sso/UI/Login	0,118892

Fig. 3. List of the detected dynamic resources

After selecting the desired dynamic resource, dynamic fields belonging to the resource are shown on the sidebar alongside with the dynamic field instances associated to the currently selected transaction. The selected transaction is shown on the text buffer with the dynamic parts highlighted in yellow. The selected dynamic field instance, in turn, is shown

with dark blue background. This is shown in Figure 4 where Field 2 is selected and corresponding instance is highlighted in the text buffer.

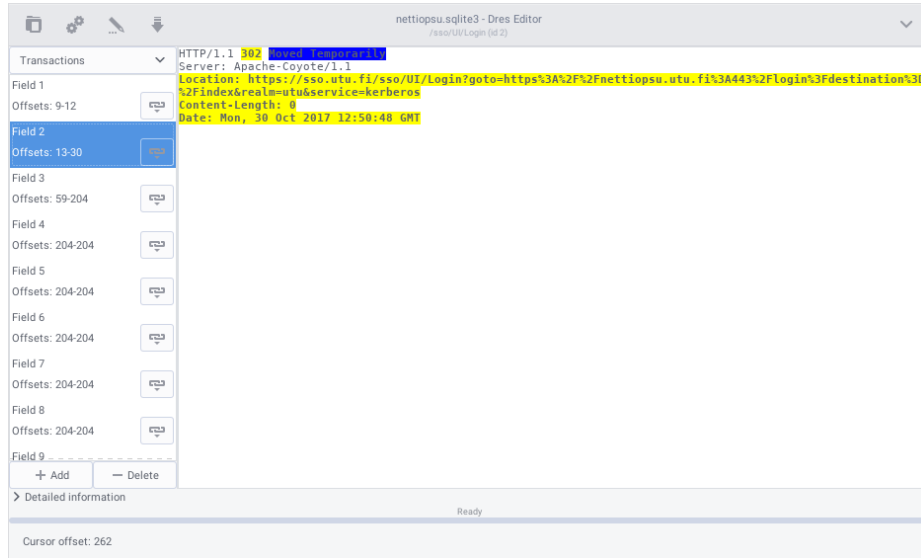


Fig. 4. Previously detected dynamic resources visualized

New dynamic fields can be added to the dynamic resource simply by clicking “Add” as in Figure 5. A new entry appears in the list of dynamic fields with no reference to any dynamic field instance. Selecting the dynamic part of characters from text buffer and clicking the “Link” button as in Figure 6 will add a reference between the dynamic field and the instance belonging to the shown transaction.

4.2 Challenges and potential improvements

The main challenge in the graphical user interface is to present multiple transactions of a dynamic resource in a sensible manner as the amount of possible transactions for one dynamic resource is relative to the amount of visits on that resource during recording phase. For example, the amount of transactions can be as low as two transaction or it might be fifty.

Our current solution is a simple solution to the problem but probably not the most efficient way to allow manual annotation. Currently our system lists all the available transactions for the selected dynamic resource in one menu. When selecting a transaction from the menu, the dynamic fields are kept on the sidebar but the links to the dynamic field instances are updated to reflect the newly selected transaction.

While being an intuitive and clear way of handling multiple transactions, our current solution does not always result in the most efficient workflow.

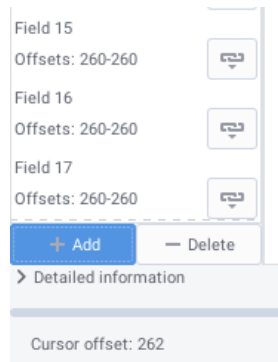


Fig. 5. Adding a new dynamic field to a resource

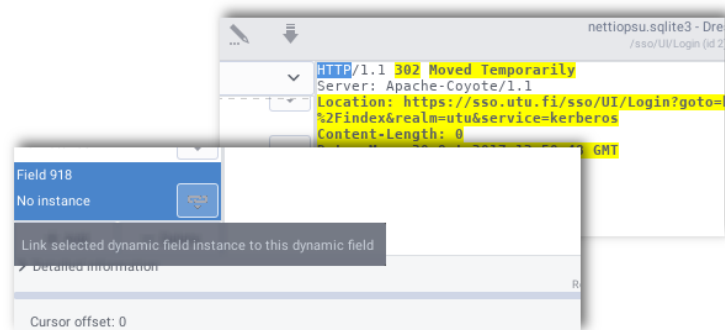


Fig. 6. Linking the selected field instance to a dynamic field

It drives the user to process the resources transaction by transaction rather than dynamic field instance by dynamic field instance first. The problem with this approach is that the dynamic field instance may grow in size depending on the other instances, meaning that the old instances need to be upgraded after the first pass.

One solution for avoiding the multi-pass problem is to present all the transaction samples simultaneously, in a similar fashion to a diff tool. While this approach will make it easier to annotate all field instances at once, the clarity will suffer when having a large number of samples. Figure 7 gives an idea what such user interface paradigm could look like. Here, multiple samples are presented in an open source multi-way diff tool Diffuse [6].

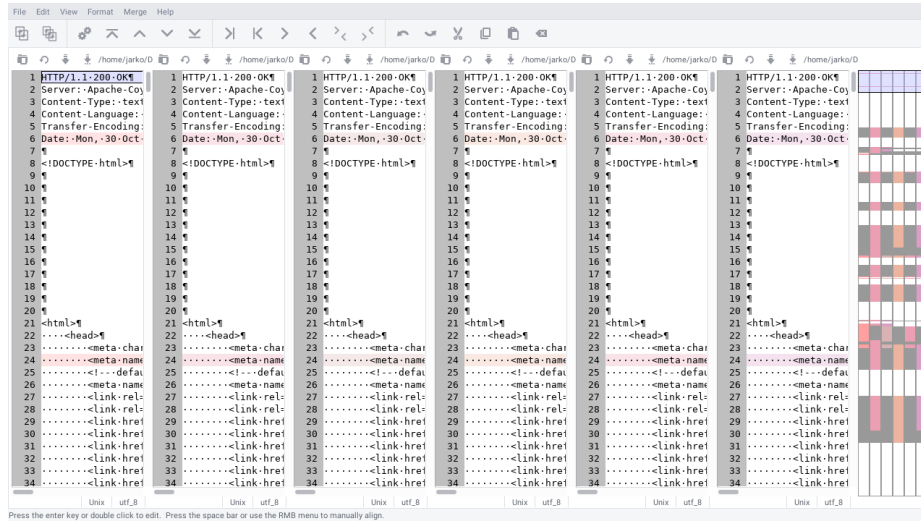


Fig. 7. Side-by-side comparison of six transaction samples

A third solution to the problem could be one where the software would prompt every field instance of the dynamic field after the first one was entered. This way the context would not constantly switch and the user could concentrate on one dynamic field at the time.

Some other assistive methods could be also implemented. For example, when selecting a dynamic field instance, the program could automatically check the other samples with the same characters before and after the field and suggest all the matches or alert if there are not at least one match in every field instance.

Currently, the manual workload is labour intensive. The improvements mentioned above, combined with possible future work on assistive methods for field detection, could positively affect the scalability of our system though. While the tool works only for saved offline data, the primary focus in our system is not to analyze the live traffic, but to detect the fields and simultaneously generate a template of the static content to be later on modified.

5 Conclusions

In this paper, we have described the challenge of detecting dynamic fields in the payloads of messages transmitted over the network. Dynamic field detection has many potential applications. It can be used for deception when values of dynamic fields are altered in order to deceive a malicious

adversary. Several network traffic analysis approaches might also benefit from recognizing dynamic fields. Finally, some message filtering solutions also require understanding of the message structure.

We have demonstrated that this problem cannot be satisfactorily solved with fully automatic field detection. We have therefore devised and implemented an easy-to-use graphical user interface to facilitate manual annotation of dynamic fields. While we have previously identified many ways to further improve our tool, we believe our approach provides a simple and intuitive way to recognize and annotate dynamic fields and provides a good foundation further work in this field.

References

1. M.H. Almeshekah and E.G. Spafford. Planning and Integrating Deception into Computer Security Defenses. In *Proceedings of the 2014 workshop on New Security Paradigms Workshop*, pages 127–138. ACM, 2014.
2. F. Cohen and D. Koike. Misleading attackers with deception. In *Proceedings from the Fifth Annual IEEE Information Assurance Workshop*, pages 30–37. IEEE, 2004.
3. W. Cui, V. Paxson, N. Weaver, and R.H. Katz. Protocol-independent adaptive replay of application dialog. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium*, 2006.
4. Gnome Developer. GTK+ 3 Reference Manual. <https://developer.gnome.org/gtk3/3.0/>.
5. R.W. Irving and C.B. Fraser. *Two algorithms for the longest common subsequence of three (or more) strings*, pages 214–229. Springer Berlin Heidelberg, 1992.
6. D. Moser. Diffuse homepage. <http://diffuse.sourceforge.net/>.
7. D. Nadeau, P.D. Turney, and S. Matwin. Unsupervised named-entity recognition: Generating gazetteers and resolving ambiguity. In *Proceedings of the 19th International Conference on Advances in Artificial Intelligence: Canadian Society for Computational Studies of Intelligence*, AI’06, pages 266–277. Springer-Verlag, 2006.
8. J. Papalitsas, S. Rauti, and V. Leppänen. A comparison of record and play honeypot designs. In *Proceedings of the 18th International Conference on Computer Systems and Technologies*, CompSysTech’17, pages 133–140, New York, NY, USA, 2017. ACM.
9. Python Software Foundation. difflib.
10. S. Sekine and C. Nobata. Definition, Dictionaries and Tagger for Extended Named Entity Hierarchy. In *LREC*, pages 1977–1980, 2004.
11. J. Tammi, S. Rauti, and V. Leppänen. Practical Challenges in Building Fake Services with the Record and Play Approach. Accepted for publication, 2017.
12. Q. Wang, D. Korkin, and Y. Shang. A fast multiple longest common subsequence (mlcs) algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 23(3):321–334, 2011.