Offset-Aware Mutation based Fuzzing for Buffer Overflow Vulnerabilities: Few Preliminary Results

Sanjay Rawat, Laurent Mounier
UJF-Grenoble 1, Grenoble INP, CNRS VERIMAG, Grenoble F-38041, France
{*Sanjay.Rawat, Laurent.Mounier*}*@imag.fr*

***Abstract*—This article presents few preliminary results and future ideas related to smart fuzzing to detect buffer overflow vulnerabilities. The approach is based on the combination of lightweight static analysis techniques and mutation-based evolutionary strategies. First, a static taint-analysis allows to identify the most dangerous execution paths, containing vulnerable statements those execution depend on user input streams. Then, concrete input are produced and executed on the vulnerable program following an *offset-aware* mutation strategy: at each step, the current input streams are mutated with specific values, *and at specific offsets*, depending on their ability to activate a target execution path. We provide few empirical results on a benchmarking dataset as a proof of concept and discuss future extension.**

***Keywords*-fuzzing, evolutionary algorithm, buffer overflow, taint analysis.**

I. Introduction

Smart fuzzing is an effective approach for discovering vulnerabilities in applications. The prefix *smart* implies that fuzzing is not performed purely randomly, but by taking advantage of some knowledge of the application. Thus, this technique could combine some *a priori* knowledge (e.g., the input formats), some results obtained from preliminary analysis of the software, or even some information gained during the fuzzing process. The goal is then to generate (or mutate) inputs such that they cause application to malfunction e.g. crash or hanged. The main challenge is therefore to generate inputs that allow to reach and activate some vulnerable part of the application. From this point of view, fuzzers based on Dynamic Symbolic Execution (DSE) have been proposed [1], [2]. The idea is to execute the application using some initial concrete inputs and to compute the resulting path-condition (the sequence of constraints that have been satisfied to execute this path). By negating one of these constraints we obtain a path condition for a new execution sequence. Thus, solving this new path condition will produce new inputs thereby increasing the coverage of the application. However, solving constraints turns out to be a costly operation, specially in the case of arbitrary data types.

Fuzzers based on evolutionary computing approaches (genetic algorithms, evolutionary strategies etc.) take a different approach by evolving the inputs based on its fitness which, in turn, is based on its success to reach the vulnerable point [3], [4], [5]. The basic idea is to instrument the application to observe its runtime behavior and, based on various metrics, change the inputs by doing crossover and mutation. Traditionally, crossover and mutation are done by changing random positions (based on some probability distribution) in the inputs. Based on our previous experience in this direction [4], we observe that doing a random mutation is not very efficient specially in order to solve string constraints which are combination of characters. In order to enhance the mutation operator, we propose an *offset-aware mutation* method which mutates string inputs at particular locations. The method improves the process of generating inputs that satisfy constraints. This article reports very initial results in this direction, but the final objective of this work-in-progress is twofold:

- to produce an automated smart fuzzer for buffer overflow vulnerabilities, which combines static analysis techniques (e.g, to find vulnerable execution paths) and evolutionary approaches (to execute these paths at runtime);
- to provide some metric on the exploitability of the discovered vulnerability (i.e., how easy is it to exploit them).

II. Overview

This section briefly introduces the problem and our intended approach to address that. Let us consider the following example to explain the problem and the solution.

The sample code in Listing 1 is taken from Verisec suite of vulnerable programs [6]. This program is vulnerable assuming that `checkD` and `c2` are user controlled buffer strings and `buf` is fixed size buffer. If `checkD` contains more than one "=\n" (combination of = and \n to pass checks at lines 6 and 11)), `buf` can be overflowed by `c2` (line 21). So, the problem is to generate a `checkD` string that contains "=\n" character sequence. The proposed methodology is composed of the following steps:

1. **Tainted Path Generation:** If there is a path starting from user provided input and reaching some vulnerable point in the program, knowledge of such a path helps in generating more useful (target-oriented) inputs. Our work relies on the availability of such a tainted path and we choose the

```
int copyData(char *checkD, char *c2, char *buf){
  char *outD=buf;
  char *c1;
  int nchar = 0, outD = 0; // index into outfile
  for (c1=checkD;*c1 != '\0';c1++){
    if (*c1 == '='){
      if (*c1++ == '\0')
        break;
      // =\n: continuation; signal to caller
      // it's ok to pass in more infile
      // BAD: forgot to reset out
      if (*c1++ == '\n'){
        nchar = 0;
        continue;
        }
      else {
        if (*c2 == '\0')
          break;
        nchar++;
        if (nchar > BASE_SZ)
          break;
        *outD = *c1; /* BAD */
        outD++;
        }
    }...
```

Listing 1. Excerpt of a vulnerable C code from Verisec suite of programs.

technique defined in our previous work [7]. We proposed a static analysis approach which associates a *taint environment* to each variable v, at each program location l. In addition to taint information, this environment also associates to each pair (l, v) a set of taint dependency sequences (TDS) explaining *why* the variable v is tainted at location l. More precisely, each TDS $t =< l_1, l_2, \ldots, l_n >$ is a sequence of program locations l_i a program execution path should traverse in order to reach l with an input-dependent value assigned to v. Thus, when l corresponds to a vulnerable statement, this TDS set exactly characterizes the set of "dangerous" execution paths. TDS example obtained from Listing 1 is the following: $< 5, 6, 11, 5, 6, 21 >$. It means that traversing these program locations (in this order) is a way to reach the vulnerable statement 21 with a tainted value of `c1` (which may correspond to a potential exploit).

2. Source-code Instrumentation: Code instrumentation plays a major role to record runtime behavior of the application. Once a TDS is obtained, we instrument the application at various locations (l_is) pertaining to this TDS to get its runtime behavior. We also instrument the application to trace the input data. The idea is to map the input `S` to TDS elements so that we know up to which offset i the input was used when last TDS element l_k was hit. The heuristic is to change the input *after* this offset i because the substring `S[1:i]` was satisfying the path condition to reach l_k, and, if the next TDS element l_{k+1} was not getting hit, it means the substring `S[i+1:]` does not satisfy some constraint. Therefore, it makes sense to mutate string after offset i. For example, if TDS element 6 was hit and the corresponding offset in the input is 50, we will mutate string from the offset 51. This will increase the chance of adding `\n` after this offset thereby producing the desired substring "=\n". we call this method as *offset-aware mutation*.

3. Mutation and Constraint Learning: As mentioned above, as a part of mutation operation, we append characters in the input string after a particular offset. These characters are not random but rather generated at runtime by observing fittest and worst inputs based on their fitness scores. Unlike any traditional evolutionary algorithm, we take into account *best and worst* inputs to improve the behavior of next generation of inputs. Our heuristic is based on the observation that it is the *presence* and/or *absence* of certain characters that makes an input *good* or *bad*. We make use of set theory to calculate the set of such characters that indirectly serves as *constraints learning and satisfaction* step. A more detailed description of this step is provided in [4]

4. Exploitability Metric: Once an abnormal behavior is obtained (e.g. crash), we want to classify this crash as exploitable or unexploitable. This will help the developers to prioritize the patching work. We propose to analyze process memory-snapshot after the crash together with the tainted data propagation to certain locations. As an initial metric, we resort to the heuristic that if any of the memory location and registers are pointing to tainted data, the so obtained crash points to an exploitable vulnerability. The idea is similar to the work presented in [8], but our objective is to make this automatic by considering other relevant properties.

III. Initial Results

Based on the ideas presented in the above section, we implement a prototype in Python to see the effectiveness of the approach on a a few sample programs from Verisec suite [6]. Currently the TDS are produced by performing (automatic) source-code analysis. This source code is then instrumented manually to trace the executions. In the future, we plan to perform these steps at the binary level using more sophisticated techniques (e.g., dynamic reaching definitions [9] in which reaching definitions are obtained from the execution trace of the application with a particular input; and binary-rewriting to enable input tracing). In table I, we compare the results obtained with those provided in [4]. In [4], we proposed an evolutionary algorithm that learns the simple constraints in the form of a set M of characters that make string based inputs to traverse a given tainted path. As mutation, the algorithm appends characters from M at *partially random* offsets. The figures in columns A1, A2 and A3 are the averages of number of iterations to generate desired input, taken over 20 runs of the experiment. A * denotes that corresponding algorithm could not generate any malicious input. It is, therefore, evident (column A1 vs. A2) that with the addition of offset-aware mutation, performance

of the algorithm mentioned in [4], improves. Note that the number of iterations in case of *edbrowse* is still high because the associated constraint on the input string is a combination of 3 characters starting *exactly* at offset 0 (which is hard to obtain by mutation).

Table I
EXPERIMENTAL RESULTS. LEGEND: A1 (RESP. A2)- EVOLUTIONARY STRATEGY WITH (RESP. WITHOUT) OFFSET-AWARE MUTATION, A3- RANDOM FUZZING

S.No.	Application	Name	Constraints	A1	A2	A3
1	sendmail	mime_fromqp	'=n'	27	154	370[1]
2	edbrowse	ftpls	'-- '	290	*	*

IV. CONCLUSIONS AND FUTURE WORK

This paper deals with an evolutionary-based approach for detecting effective buffer-overflow vulnerabilities. Starting from a set of *critical paths* (the so-called TDS, obtained during a static taint-analysis step), we apply mutation techniques at runtime to generate program inputs that allow the execution of these critical paths. With respect to our previous work ([4]), the main improvement we propose is to consider sequentially structured input (e.g., a sequence of characters). Our assumption is that such inputs are most of the time processed sequentially, from left to right, in the target program. Thus, knowing (by program instrumentation) that a given input starts to deviate from the critical path we target after having processed offset i, we know that this input should be mutated *after* this offset i. Experimental results obtained from the benchmarks we used in [4] show a significant improvement in the number of iterations required to craft a malicious input.

This works takes place in a larger perspective whose objective is to set up a fuzzing environment for software vulnerability detection. According to the general approach that we discussed here (identification of critical execution paths and input generation to activate the corresponding vulnerabilities), this environment will combine several program analysis techniques (both static and dynamic). In particular, we plan to improve the technique proposed in this paper in two main directions:

- The taint-analysis phase could be further refined in order to map some specific fields of the input data with each (branching) control location of the program. The taint information associated to these locations is then no longer a binary value (tainted/non tainted), but a field identification of some user input, as in [10]. This would allow to make the learning process more efficient and will weaken our (strict) assumption regarding how the input streams are processed (i.e., from left to right).
- The path condition associated to each critical path can be computed statically. Even if it cannot be solved in the general case, we may expect to partially solve some of its constraints (possibly using some concrete values). Again, this information should improve the mutation step.

Finally, our next objective is to experiment on larger real-world applications. In particular this approach can be generalized to other sequential data structures than character strings, like most file or network packet formats (both consisting in well-defined *field sequences*).

[1]It could not generate malicious inputs 5 times out of total 20 times. So the average is taken over 15 iterations

Acknowledgements: The authors thank the anonymous referees for their valuable comments.

REFERENCES

[1] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proc. of Network Distributed Security Symposium*. Internet Society, 2008.

[2] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proc. of the IEEE 31st Int. Conf. on Software Engineering*. IEEE Computer Society, 2009.

[3] C. D. Grosso, G. Antoniol, E. Merlo, and P. Galinier, "Detecting buffer overflow via automatic test input data generation," *Computers & Operations Research*, vol. 35, no. 10, pp. 3125–3143, 2008.

[4] S. Rawat and L. Mounier, "An evolutionary computing approach for hunting buffer overflow vulnerabilities: A case of aiming in dim light," in *Proc of sixth EC2ND 2010*. IEEE Computer Society, 2010.

[5] S. Sparks, S. Embleton, R. Cunningham, and C. Zou, "Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting," in *Proc. of the 23 Annual Computer Security Applications Conference*. ACSAC, 2007.

[6] [Online]. Available: http://se.cs.toronto.edu/index.php/Verisec_Suite

[7] D. Ceara, L. Mounier, and M.-L. Potet, "Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences," in *Proc. of the IEEE Int. workshop MDV'10*. IEEE Computer Society, 2010.

[8] C. Miller and N. Johnson, "Crash analysis using bitblaze," in *Proc. of the Black Hat USA 2010*, Las Vegas, US, July 2010.

[9] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Dynamic slicing in the presence of unconstrained pointers," in *Proceedings of the symposium TAV4*. ACM, 1991, pp. 60–73.

[10] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proceedings of the 2010 IEEE Symposium S&P 2010*. IEEE Computer Society, 2010.