0-knowledge fuzzing

Vincenzo Iozzo
vincenzo.iozzo@zynamics.com

February 9, 2010

Abstract

Nowadays fuzzing is a pretty common technique used both by attackers and software developers. Currently known techniques usually involve knowing the protocol/format that needs to be fuzzed and having a basic understanding of how the user input is processed inside the binary.
In the past since fuzzing was little-used obtaining good results with a small amount of effort was possible.
Today finding bugs requires digging a lot inside the code and the user-input as common vulnerabilies are already identified and fixed by developers. This paper will present an idea on how to effectively fuzz with no knowledge of the user-input and the binary.
Specifically the paper will demonstrate how techniques like code coverage, data tainting and in-memory fuzzing allow to build a smart fuzzer with no need to instrument it.

1 Introduction

Fuzzing, or fuzz testing, is a software testing methodology whose aim is to provide invalid, unexpected or random inputs to a program. Although the idea behind this technique is conceptually very simple it is a well known and widely established methodology employed in COTS software vulnerability discovery process.
The first appearance of fuzzing in software testing dates back to 1988 by Professor Barton Miller[1]; since then the technique has evolved a lot and it is not only used by attackers to discover vulnerabilities but also internally by many companies to find bugs in their software.
Over the course of time a lot of different implementations of fuzz testing have been researched, nonetheless it is commonly believed that there are two predominant approaches to fuzzing: Mutation-based and Generation-based.
The former is based on random mutations of known well-formed data, whereas the latter creates testing samples using templates describing the format of the software input.
Both approaches have their advantages and pitfalls. The former requires little effort to be implemented and it is reusable across different software. Nonetheless given the raising interest companies have shown in properly testing and developing products this approach will generally yield worse results than generation-based fuzzers.
The second approach has the advantage of obtaining better results in terms of bugs found, although it requires knowledge of the input format the binary expects and its reusability is bounded to binaries that deal with the same input format.

The difficulty of creating input models can range from low for public data formats to almost infeasible for proprietary formats.
In order to ease the process of creating input templates various approaches have been studied, most notably evolutionary fuzzers and in-memory fuzzers.
Both are derived from mutation-based fuzzers but for different purposes. The first type of fuzzers, in fact, by employing genetic algorithms attempts to generate sets of data which resemble as precisely as possible the input format. The latter, instead, first requires a human to manually identify specific functions inside the binary then mutates the input in-memory in order to prevent data validation which could lead to different code paths thus resulting in not fuzzing crucial pieces of an application.
Evolutionary based fuzzers suffer from the difficulty of identifying proper scoring and mutation functions and for this approach to be effective it usually requires more time than the generation-based one. In-memory fuzzing on the other hand has a high rate of false positives and negatives and it requires an expert reverse engineer in order to identify proper test cases.

In this paper the author presents an approach to fuzz testing based on in-memory fuzzing aiming at limiting human intervention and minimizing the number of false positives and negatives that currently affects this technique. The proposed methodology employs a range of known metrics from both static and dynamic program analysis together with a new technique for in-memory fuzzing. Specifically we will use data tainting for tracking user input, thus being able to identify locations in-memory suitable for testing; we will also employ static analysis metrics in order to identify functions in the binary that can be interesting from a security testing point of view.
To the best of the author‘s knowledge there are no public attempts at combining together these techniques for fuzz-testing purposes. A notable exception is Flayer which nonetheless only focuses on dynamic analysis and program paths manipulation in order to discover software defects.

The rest of this paper is organized as follows. In section 2 we provide basic background information on the metrics used. Section 3 discusses related work. Section 4 presents our approach and implementation. Finally we conclude and discuss future work directions in Section 5.

2 Background

In this section, we present background information on static analysis metrics, data tainting, and in-memory fuzzing.
In our implementation we use primary two static analysis techniques: cyclomatic complexity and loop detection.
Cyclomatic complexity is a software metric used to determine how complex in terms of code paths a function is. The computation is done on the number of edges and nodes a function contains. Intuitively the more the structure of the function is complicated the more complex the function is. In [2] the connection between function complexity and bugs presence has been discussed. Although there is not always a correlation between the two, it is reasonable to assume that more complex functions are prone to contain bugs given the amount of code they contain.
Another metric employed is loop detection. This algorithm takes advantage of some properties of a function flowgraph and its dominator tree in order to detect loops present in compiled code.

This technique is widely used in compilers for optimization purposes, and it has some interesting aspects from a security prospective as well. It is commonly known, in fact, that memory write often happens inside loops and that most compilers usually inline functions like *memcpy* so that the function will effectively result in a loop.

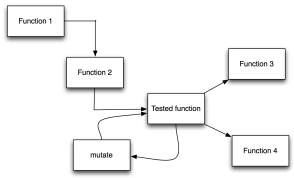
Another crucial piece of infrastructure for the proposed fuzzer is the data tainting engine. The goal of data tainting is to gather information on how user input is propagated through a binary. The concept of data tainting is intuitively very simple, one or more markings are associated with some data supposedly representing the user input and those markings are propagated following the program flow. Although it is possible to perform data tainting using static analysis the complexity of the task and the possible incomplete set of information led the author to choose a dynamic analysis approach to the problem by taking advantage of an existing dynamic data tainting framework called Dytan[6]. Using dynamic data tainting has the benefit of obtaining more precise and richer information on data propagation although it will not be able to explore program paths that are not executed at run-time. Given the nature of the fuzzer, obtaining information on non-executed code paths is of no interest as in-memory fuzzing relies on the ability to reach code paths by mutating a set of known good data.

Finally in order to monitor the effectiveness of our fuzzer, we employ a software testing measure known as code coverage. This technique verifies the degree to which the code of a program has been tested by tracing the execution of the binary. Although there are many different implementations of code coverage all using different criteria in terms of the kind of information to record, the author decided to implement the technique so that basic blocks execution is being traced. This implementation does not take into account code paths and therefore might be imprecise in some circumstances, nonetheless we consider this trade-off to be acceptable as it avoids to overly complicating the implementation and improves the fuzzer performance.

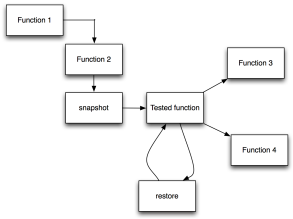
3 Related work

In this section we will briefly describe existing approaches to data tainting and in-memory fuzzing together with a brief description of Flayer[3] being it the closest work to the one described in this paper.

3.1 Existing in-memory fuzzing implementations



(a) Mutation loop insertion



(b) Snapshot restoration mutation

Figure 1: Known implementations of in-memory fuzzing

To the best of the author‘s knowledge in-

memory fuzzing was first introduced to the public by Greg Hoglund of HBGary in [4] and later further developed by Amini at al[5]. Currently there are two public methods: Mutation loop insertion and snapshot restoration mutation.
The first method works by inserting an unconditional jump from the function being tested to a function responsible for mutating the data residing in the process address space of the fuzzed binary. At the end of the mutation function another unconditional jump to the beginning of the currently tested function is inserted. The control flow graph of this approach is shown in 1(a).
This approach suffers of a number of drawbacks with a high rate of false negatives and stack consumption being the two major ones. Another disadvantage of this method is the general instability of the memory after a few fuzzing iterations.
The second approach works by inserting an unconditional jump from the beginning of the function being tested to a function responsible of taking a memory snapshot. This function will later call again the tested function. At the end of the analyzed function another unconditional jump is inserted. The jump points to a function responsible of restoring the memory, fuzzing data and executing again the fuzzed function. A control flow diagram employing this approach is shown in Figure 1(b). Although this method has some advantages in respect to the first one described, it still suffers from a high false positives rate and it is also slower given the need of continuously having to restore process memory.

3.2 Existing data tainting implementations

Dynamic data tainting has gained momentum in the last few years given the increase complexity of software. A lot of implementations of data tainting frameworks exist, for this reason the author decided to use a framework previously created by James Clause and Alessando Orso of Gatech called Dytan[6]. The decision was made based on a number of requirements.
First and foremost the ability to instrument binaries without any recompilation or access to the source code.
Another very important requirement was portability, most of the existing implementations are based on Valgrind[7] which does not support the Windows platform. The two most appealing candidates were Temu[9] and Dytan.
The first one is built on the top of a modified version of Qemu[8]. Although this would have respected both the initial requirements we think that a data tainting framework based on a virtual machine emulator is overkill for our goals. Besides the implementation in the author‘s opinion is not yet robust enough.
Dytan is implemented as a pintool[16]. It is a flexible framework and can run on both Linux and Windows.

3.3 Additional related work

As already mentioned in the previous section Flayer[3] is the most similar work to the approach discussed in this paper. The software combines data tainting and the ability to force code paths. Differently from many other data tainting tools Flayer has bit-precision markings. Although this grants a higher degree of precision in obtaining information on data propagation for the purpose of our work byte-precision markings are detailed enough.
Another limitation is the software the tool is based on; as already mentioned Valgrind does not support Windows which severly impairs the

usefulness of the tool.
Finally even if the main aim of the tool is not fuzzing it has the ability of forcing code paths and therefore it can be used to test various code paths. This method has three main drawbacks; the first one is a high number of false positives, the second one is the absence of a sample which can be later used by the attacker to reproduce the bug and finally a problem known as code-path explosion. This problem arises because the number of code paths to force increases exponentially with the complexity of the software.

4 Proposed approach and implementation

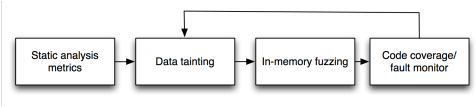


Figure 2: Fuzzer components

In this section we will present the idea and implementation of our work. As shown in Figure 2 our fuzzer can be divided into 4 parts.

4.1 Static analysis metrics

Static analysis algorithms are used to determine which functions could be potentially of interest for our fuzzer. We assign a higher score to functions that have a high cyclomatic complexity score and at least one loop in them; we then consider all the functions that have loops but a low cyclomatic complexity score and finally we take into account the remaining functions. Ideally we will add more metrics to the implementation, therefore this rather trivial scoring system should be replaced by a more sophisticated approach which takes into account scores coming from various metrics and weights them in respect to their relevance from a security prospective.

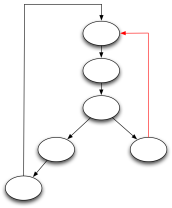


Figure 3: The edge in red is missed by the approximative cyclomatic complexity formula.

Cyclomatic complexity Cyclomatic complexity was first described by Robert McCabe in [10]. The purpose of this metric is to calculate the number of independent paths in a code section. Many formulation of this metric have been given, we briefly explain the ones that are relevant to our fuzzer.

Definition Let G be a flowgraph, E the number of edges in G, N the number of nodes in G and P the number of connected components in G.
Cyclomatic complexity is defined as:

$$M = E - N + 2P \tag{1}$$

A connected component is a subgraph in which any two vertices are connected to each other by paths. This formula originates from the cyclomatic number:

Definition Let G be a strongly connected graph, E the number of edges in G, N the number of nodes in G and P the number of connected components in G.

The *Cyclomatic number* is defined as:

$$V(G) = E - N + P \tag{2}$$

It should be notice that the cyclomatic number can be calculated only on strongly connected graphs, that is a graph in which from every pair of vertices there is a direct path connecting them in both directions. McCabe proved that the flowgraph of a function with a single entry point and a single exit point can be considered a strongly connected graph and therefore the cyclomatic number theorem applies and that $P = 1$, thus the resulting simplified formula is:

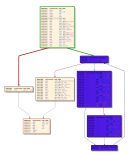
$$M = E - N + 2 \tag{3}$$

Intuitively when a flowgraph has multiple exit points the aforementioned formula doesnt hold true anymore. Another one should be therefore used:

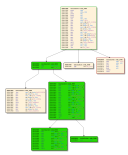
Definition Let G be a flowgraph, π the number of decision points in G and s the number of exit points in G. *Cyclomatic complexity* is defined as:

$$M = \pi - s + 2 \tag{4}$$

Applying (3) to functions with multiple exit points we will have, in fact, lower cyclomatic complexity values by a minimum factor of 2. Figure 3 shows typical edges and connected components missed by using (3).
Nonetheless the author believes that the less precise measurement can be used without impairing the results.
We implemented cyclomatic complexity calculation for each function in a module by using BinNavi API. A detailed explanation of the implementation can be found in[11].



(a) A function flowgraph, nodes in blue belong to a loop



(b) Dominator tree of the previous function, nodes in green correspond to the blue ones highlighted in picture (a)

(c) The nodes in green dominate the node in red in the dominator tree

Figure 4: Graphs used in loop detection algorithm

Loop detection algorithm As previously mentioned another metric, loop detection, is used to select functions. The first required step is to extract the dominator tree out of a function. Formally:

Definition A dominator tree is a tree where each node‘s children are the nodes it immediately dominates.
A node d is said to dominate node k if every path from the start node s to node k must go throught node d.

To give a visual example of a dominator tree of a function please refer to Figure 4. Nodes in blue in Figure 4(a) are highlighted in the dominator tree in green in Figure 4(b).
There are two known algorithms used to calculate the dominator tree of a flowgraph. It is out of the scope of this paper to discuss them. It should be noticed, though, that the

tool upon which we built our loop detection algorithm, BinNavi[12], implements Lengauer-Tarjan[13] dominator tree algorithm which is almost linear thus granting us a higher computational speed.
The second step is to calculate for each node its dominators. In Figure 4(c) the dominators of the node in red are the ones in green.
The last step is to search for edges from a node to one of its dominators. Recalling the definition of domination it is trivial to show that if there is an edge from a node to one of its dominators a loop is present.
Most complex assembly instruction sets have what are called implicit loops instruction, for instance *rep movs* in x86 ISA. Applying this algorithm to a flowgraph will therefore miss this type of loops.
In order to overcome this problem we will translate the function to an intermediate language called REIL[14] implemented in BinNavi. This intermediate language provides a very small set of instructions which helps in the process of unfolding implicit loops.
In [15] a detailed implementation of this algorithm can be found.

4.2 Data tainting

As stated before the author did not implement the data tainting framework employed by the fuzzer, nonetheless given the critical importance of data tainting for this project the author thinks it is important to briefly describe how dytan works and how we use this framework for our purposes.
We previously mentioned that data tainting is a technique to track user input inside a binary.
Tracking is usually performed by assigning markings to data while executing the binary.
Each data tainting implementation can choose the type of markings to use, more precisely it is possible to determine the granularity of those markings.
Dytan is able to either assign a single marking to each piece of input or have byte-level markings. We chose to use the second type of markings as it is more precise but at the same time does not cause an excessive overhead during the execution.
In order to make data tainting work it is important to define what data needs to be tracked. In Dytan it is possible to track user input coming from network operations, files access and command line arguments passed to the *main()* function. That is system calls and functions responsible for the aforementioned input sources are monitored and their output is tracked through the binary.
Another important factor to take into account while implementing a data tainting tool is a propagation policy.
A propagation policy is a set of rules followed while taint markings are assigned during program execution.
Dytan currently is able to perform control and data-flow or data-flow only analysis. The former tracks direct or transitive data assignments as well as indirect propagation due to control flow dependencies upon user input. The latter instead can only track direct and transitive data assignments. In our fuzzer we use the second approach as control flow analysis does not add any useful information on data locations to be tested.
Another problem to tackle while creating a propagation policy is how to deal with multiple markings assigned to the same input. Dytan currently assigns to the resulting taint marking the union of all the taint markings related to it. Although

for our fuzzer a different approach might grant better results we currently use the default dytan policy.
Finally we make dytan provide information on every instruction that assigns taint markings. That is for each of those instructions we obtain the state of taint markings on machine registers and on memory locations that are tainted at that specific program point.

4.3 In-memory fuzzing

We presented in section 3 the two known approaches to in-memory fuzzing. In this section we are going to present two slightly different approaches which we believe to gain better results given the amount of information we can gather from data tainting analysis.
We implemented our in-memory fuzzer on top of PIN[16]. PIN has the ability to add instrumentation functions before and after a binary is loaded in memory, functions and instructions.
Recalling that for each instruction that assigns taint markings we retrieve from data tainting analysis, we get the markings associated to machine registers and memory locations, we are able to precisely identify program points during binary execution that are suitable for fuzzing.
For both approaches we perform a number of steps:

1. Install an analysis function on image loading.

2. Install an analysis function before the function we are interesting in fuzzing is executed.

3. Install an analysis function before each instruction that assigns taint markings.

At point 1 we search for the address of the function we are interested in fuzzing and install the analysis function for that function. At point 2 we iterate through function instructions locating the ones that are of interest in order to install an analysis function as described in 3.
The first approach consists of mutating memory locations and registers in place. That is instead of allocating new memory and pointing instructions operands to it we modify the content of both memory locations and registers within their length boundaries.
We then continue the program execution until the program quits or new data is obtained from a tainted source.
This approach is more conservative than all the others as it does not change the memory layout thus the number of false positives is reduced but at the expenses of an increased number of false negatives.
The second approach works very similar to SRM 1(b). In addition to the first three steps we also add an instrumentation functions at the end of the tested function. This function will be responsible of restoring memory after fuzzing was performed. With the second approach the memory layout is changed as the fuzzer will allocate chunks of memory to be used during the fuzzing phase.
As for the first approach the program execution is continued until the application quits or new data is obtained from a tainted source.
Although our second approach is similar to SRM there are a few notable differences that have to be considered. First we do not take a full snapshot of the process memory but we only track modifications that occurred due to fuzzing during the execution of the tested function. The second difference is that memory is not totally restored after the function was fuzzed, this can

allow us to reduce the number of false negatives since possible bugs caused by a faulty execution of the function are not missed by restoring the full process memory.
It has to be noted that both approaches described here although more effective cannot be used without a proper amount of information gathered by the means of data tainting analysis or some similar techniques.

4.4 Code coverage

The combination of code coverage with fuzz testing has long been used in order to measure the effectiveness of fuzzing. We implemented code coverage on the top of BinNavi debugging API. The choice of using BinNavi debugger serves a double purpose, not only we are able to implement code coverage using lightwave breakpoints which highly reduce execution overhead but we are also able to monitor the execution for possible faults. We decided to implement code coverage at basic blocks level, that is a breakpoint is set at the beginning of each basic block in the tested binary. We perform code coverage first when the binary is executed with a known good sample, later it is calculated again every time the program is fuzzed. We require the fuzzing sample to perform at least as good as the known good sample, we also set a threshold defining the upper-bound after which the sample reaches the "halting point". The "halting point" is the point where the fuzzing process is re-initialized with a new known good sample as shown in Figure 2. Formally:

Definition Let C be the code coverage score of a known good sample, C_1 the code coverage score of a fuzzing sample, t a user supplied delta.

The following must hold true:

$$C_1 \leq C + t \tag{5}$$

The halting point is defined as:

$$C_1 = C + t \tag{6}$$

The code coverage score is calculated as follows:

Definition Let BB_t be the totality of basic blocks in a binary, BB_f the number of basic blocks touched in a single execution.
The *code coverage score* is defined as:

$$C = \frac{BB_f}{BB_t} \tag{7}$$

A detailed implementation of code coverage using BinNavi API can be found in [17].

5 Results and future work

In this paper we have described a new approach to fuzz testing which highly reduce instrumentation costs thus resulting very useful when dealing with large proprietary applications.
We have also shown how it is possible to combine static and dynamic analysis techniques to triage interesting functions from a security testing point of view.
Finally we have proposed a new approach to in-memory fuzzing which is more precise and less prone to false negatives than previous known techniques.
We do not have enough data to determine whether this approach has better results compared to other fuzzing techniques.
The author believes that compared to other mutation-based and evolutionary-based methodologies the one proposed in this paper will have

better results. In comparison to generation-based fuzzers our technique will have better results when dealing with complex software but worse results when the software input is simple. The main direction of future work will be focused on reducing false positives by employing constraint reasoners to determine whether a given bug is reproducible with valid but unexpected input.
Another important challenge is to implement more static analysis metrics to triage functions with a higher degree of precision.

Acknowledgments

The author would like to thank Thomas Dullien, Dino Dai Zovi and Shauvik Roy Choudhary for their suggestions and help while researching the topic.
The author would also like to thank James Clause and Alessandro Orso for having provided access to dytan source code and their help while testing and improving the original code base.
Finally we want to thank all the people who have reviewed the paper.

References

[1] B.P. Miller, L. Fredriksen, and B. So: "An Empirical Study of the Reliability of UNIX Utilities", Communications of the ACM 33, 12 (December 1990)

[2] Kan: Metrics and Models in Software Quality Engineering. Addison-Wesley. pp. 316317.

[3] W. Drewry, T. Ormandy: Flayer:exposing application internals, Proceedings of the first USENIX workshop on Offensive Technologies.

[4] G. Hoglund: Runtime Decompilation: The GreyBox process for Exploiting Software, Black Hat DC 2003

[5] M. Sutton, A. Greene, P. Amini: Fuzzing:brute force vulnerability discovery. Addison-Wesley.

[6] J.Clause, W. Li, A. Orso: Dytan:a generic dynamic taint analysis framework, Proceedings of the 2007 international symposium on Software testing and analysis

[7] Valgrind: http://www.valgrind.org

[8] Qemu: http://www.qemu.org

[9] Temu:http://bitblaze.cs.berkeley.edu/temu.html

[10] T. J. McCabe: A Complexity measure, IEEE transactions on software engineering, vol. se-2, no.4, december 1976

[11] V. Iozzo: Scripting with BinNavi - Cyclomatic Complexity

[12] BinNavi: http://www.zynamics.com/binnavi.html

[13] T. Lengauer and R. E. Tarjan: A fast algorithm for nding dominators in a owgraph, ACM Transactions on Programming Languages and Systems

[14] T. Dullien, S. Porst: REIL: A platform-independent intermediate representation of disassembled code for static code analysis, CanSecWest 2009

[15] V. Iozzo: Finding interesting loops using(Mono)REIL

[16] PIN: http://www.pintool.org

[17] V. Iozzo: Code coverage and BinNavi