PHPIL: Fuzzing the PHP Interpreter with Custom Bytecode

Vignesh S Rao*, Tarunkant Gupta†, Saastha Vasan‡, Deepthi L.R§
Department of Computer Science and Engineering, Amrita Vishwa Vidyapeetham
Amritapuri, India
Email: *vigneshsrao5@gmail.com, †tarunkant05@gmail.com, ‡saasthavasan@gmail.com, §lrdeepthi2002@gmail.com

***Abstract*—We aim to fuzz the PHP interpreter to search for bugs which may or may not be able to compromise the security of the interpreter and the system it is running on. In our research we propose to implement a fuzzing framework for the standard implementation of the PHP interpreter. What makes our fuzzer different from other PHP interpreter fuzzers is the ability to create syntactically and semantically correct code samples. We found in our research that most of the available PHP interpreter fuzzers, although able to create syntactically correct code samples, are unable to produce semantic correctness.**

We created our own intermediate language composed of custom opcodes, which is used by the code generator to generate the code samples. Code generator is governed by the rules which make sure that the resulting code follows the PHP syntax and symmetric conventions. The mutator is driven by the code generator and it performs the mutation on the generated intermediate language. We created a corpus which is used to store the desired code samples on which further mutations can be performed. Thus new inputs are generated by performing mutations to the code which increases the coverage, thus maximizing the chances of finding vulnerabilities. The lifter lifts the mutated intermediate language sample to the php code before feeding to the interpreter. The execution of the php code sample is monitored for any unexpected behaviour of the interpreter. A report is formed in case of any unexpected behaviour.

***Index Terms*—Fuzzing, Coverage, Corpus, Crashes, Mutation, PHPIL, Bytecode**

I. Introduction

PHP is a popular general-purpose scripting language that is especially suited to web development [1]. Originally developed in 1995, today php is used for the majority of websites and server side applications. Actually PHP is a server side scripting language which is used for connecting Web Page with a DataBase such as asp or jsp.

PHP interpreter like any other interpreter is a software that executes PHP code one line at a time. PHP interpreter plays an integral role in writing server side php script and today as PHP is used for the majority of website and server side applications there is a need for testing the security of the PHP code, there is a need to check the vulnerabilities present in the interpreter. Once found those vulnerabilities like memory corruption or access violations which can open doors to a pool of security issues such as memory leaks which might lead to remote code execution and other security attacks.

In this paper we are proposing a method to check for security issues in the PHP Interpreter using Fuzzing. Fuzzing or fuzz-testing unlike manual static and dynamic analysis is capable of detecting programming errors. For fuzzing the php interpreter, we create random automated unexpected inputs and pass it to the interpreter. The program execution is then monitored closely for any unexpected behaviour which might trigger system vulnerabilities. For this we will be developed a coverage guided fuzzer. A coverage guided fuzzer is aimed at maximizing the code code coverage which enables it to go to the deepest part of the program logic. New inputs are generated by performing mutations to the code which increases the coverage, thus maximizing the chances of finding vulnerabilities.

II. Literature Review

Before starting with the project we read some existing literature on the topic for background knowledge. In [2] fuzzing applications to find concurrency related vulnerabilities is discussed. The paper details various techniques for code coverage like edge based coverage and finding new paths that we included in our project as well. The paper also mentions usage of sanitizers which we are using to fuzz the PHP interpreter. Another paper that we came across was based on exploit techniques and mitigation [3]. The paper is based on Android devices, but we used the various exploit techniques that the paper details like return-oriented-programming to generate even better seed input for our fuzzer. Another approach that we thought of implementing was coming up with a novel technique to fuzz the interpreter by implementing machine learning models to generate code that can detect network vulnerabilities in an automated manner and feed that code to corpus when we are trying to create network crashes [4]. But after further research we found out that due to scalability factor and the manual interference that will be required to implement this model and this approach would not be feasible for our project. Some other work that we went through [5] [6], included the idea of fuzzing interpreters via abstract syntax tree representation of the code. This idea though appealing, had the downside of the generated code not being semantically correct. Thus we went with the idea of generating a custom bytecode representation for code generation.

Essentially, what we found missing in most of the programming language fuzzers that we came across was that all of them emphasized on syntactical correctness and not on semantically correct input. We attempt to fix this, by

Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on November 28,2024 at 10:52:49 UTC from IEEE Xplore. Restrictions apply.

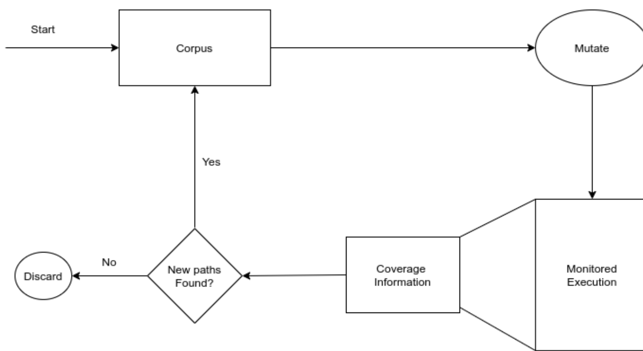

Fig. 1. High Level Fuzzing Architecture.

introducing a custom bytecode representation that will result in semantically correct input while maintaining the syntactical correctness of the the grammar. Another interesting approach that we use in our fuzzer is the use of templates instead of parsing grammar which is much easier to develop and tweak.

III. Proposed Approach

The aim of this project is to find crashing samples in PHP interpreter and for this we came with a novel approach that is fuzzing the PHP interpreter with the help of byte codes. Programs generated from the byte codes will be semantically and syntactically correct and that will help us to generate correct programs to feed into the PHP interpreter. As of our knowledge, there is no other PHP interpreter fuzzer that fuzzes on bytecode level, though this concept was used by S. Groß to fuzz javascript engines [7] and that gives us the opportunity to find more bugs in the PHP interpreter.

The following details a brief overview of how our fuzzer generates semantically and syntactically correct PHP code: Define Opcodes which you will need for your fuzzer in order to generate PHP code. PHP has its own opcodes but for our fuzzer we did not need that low level of opcodes, so we made our own opcodes like `BinaryOperation`that will contain all primary level of opcodes like `Add`, `Sub`, `Mul`etc. Define syntax of the Opcodes Create random variables and assign the values according to the syntax. Track all the variables/operations/instructions which would help us to debug/analyze the program. Added analyzers which helped us to find type/context/scope of a variable. Add constraints for bringing the semantic correctness, like having the `break`statement in the beginning of the loop doesn't mean anything. While calling a function the fuzzer can't choose the loop variable as argument variable for the functions. Now, we generate random programs and lift that to PHP syntax. Figure-1 showcases the high level fuzzing layout of our fuzzer.

IV. Intermediate Language

An intermediate language is an abstract programming language used by a compiler as an in-between step when translating a computer program into machine code. A programming language is really complex as it incorporates support to all the work that can be done using that particular programming language. One way of generating code is to add support to all the instructions in the native language itself in our case generating php source code itself but it will require us to give support to all the PHP instructions individually which will consume a lot of time and is not feasible considering how enormous the entire language is. So we came up with another method for dealing with this problem. We developed our own intermediate language bytecode which we are using to generate the PHP source code. As the central idea of this project is to develop a mutation based fuzzer for increasing the coverage, thus performing mutation on the bytecodes construct is much more effective and easier when compared to the source code or AST. The bytecodes can be used to create data flow graphs and gives us the actual control which is needed for achieving the maximum code coverage.

A. Byte Code

To achieve the maximum code coverage and providing support for different mutation strategies we came up with an idea of creating custom bytecodes which will be used to generate code and eventually be lifted to the actual PHP source code using a lifer. How this conversion is performed will be explained in detail later in the lifer's section. Using bytecodes gives us multiple advantages like the bytecodes can be used to data flow graph and gives us the actual control which is needed for achieving the maximum code coverage. It also enables us to come up with different mutation strategies which could not have been implemented easily as AST mutation.

The aim is to create code samples which are both syntactically and semantically correct before and after the mutation is performed and the resulting code must increase the code coverage.To guarantee the syntactical correctness of the generated code it is enough to convert the bytecode to the corresponding PHP code is possible. As explained earlier using bytecodes helps in implementing different mutation strategies which help in increasing the code coverage. As the mutation performed on the byte code is minimal, there is a very less possibility that the resulting code will be semantically incorrect. All the mutations are also governed by the rules which make sure that the resulting code follows the basic rules of the IL. Some of the rules which govern the semantic correctness are keeping a check for scope violations like if a variable is declared and defined in a particular scope then it is not available in other scope and making sure that a variable is initialized before operations are being performed on it.

V. Analyzers

In order to support some fuzzer mechanisms like mutations and some program requirements like syntactic and semantic correctness, we have implemented a number of program analyzers. This section goes over the description and implementation of each of the analyzers used in the fuzzer.

A. Scope Analyzer

The scope analyzer keeps track of the scope of each of the variables that is currently in use. A scope is represented as a

Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on November 28,2024 at 10:52:49 UTC from IEEE Xplore. Restrictions apply.

Python list. The scopes in active use are maintained via a stack. Whenever a new scope is encountered, a new list is pushed into this stack. Thus the top of the stack contains the current scope, and the bottom of the stack contains the global scope. The information about the scope of a variable is useful when the fuzzer has to randomly select a variable among all the defined ones. This gives the fuzzer more fine grained control so it can prefer the local variable as compared to global ones.

B. *Context Analyzer*

This is used to keep track of the program context. For example, we don't want to emit a 'return' instruction when we are not even within a function. The challenge here is that, like scopes, context's can be nested. For example, in a loop within a function, we are in the function context as well as the loop context. Thus keywords like continue, break and return are all valid. To keep track of the contexts we again use a stack. Currently we have divided the context to be of three types, namely global, loop and function. Whenever a new context is encountered, a new entry is pushed on to the stack. This new entry has the initial value of the last context. Thus if we are entering a loop inside of a function, the new context entry that is pushed already has the information that it is inside a function, so a return keyword can be used inside the loop.

C. *Type Analyzer*

This analyzer is used to infer the types of the variables used in the input program. Knowledge about types of variables is important in order to make the correct decision about which variable is to be used. For example, if a call is made to the inbuilt function "strlen", then the argument should ideally be a string rather than anything else. The type inference system is not yet complete and needs better algorithms in order to increase the efficiency. Currently the type of a variable is inferred based on the IL opcode in which it is being used in. Thus if there is a LoadInteger opcode the variable type is set to an integer. This is largely ineffective while inferring the types of function parameters. As PHP is a dynamically typed language, the function parameter types are not known before runtime. This remains an area of improvement for future research.

VI. Code Generator

Code generator is one of the core components of the project. It uses the byte codes to generate the intermediate code. As mentioned in preceding sections, we introduced a new intermediate language, which makes it easier to perform mutation while allowing the conversion of code from here to the PHP code. Our intermediate language consists of instructions, each consisting of an operation with a list of input and output variables.

The generation of intermediate language is governed by different rules to maintain the semantic correctness of the code. Variables are identified through integers and are required to be numbered consecutively starting from zero in every program. In our intermediate language control flow is implemented using special block instructions. Each block has a starting and an ending operation and has its own scope which means the variables declared inside this scope cannot be used outside the block. Furthermore, the input variables to the block instructions themselves, such as the condition variable in do-while loop, have to be defined in the outer scope. This reflects the behavior of common programming languages.

The following invariants must hold for every intermediate code generated for PHP -

- Variables are numbered consecutively.
- All input values to an instruction must be variables, there are no immediate values or nested expressions.
- All variables must be defined before they are used, either in the current block or an enclosing one.
- A block begin must eventually, either be followed by the corresponding closing instruction or by an intermediate block instruction, such as a `BeginElse` for which the same holds true.
- All inputs to block instruction must be defined in an outer block.
- The first input to a Copy instruction must be the output of a Phi instruction.

VII. Lifter

The final aim of the fuzzer is to feed a randomly generated PHP program to the interpreter. Since the core mutations and code generators work on the intermediate bytecode, there is a need to convert the bytecode to PHP code in order to feed it to the interpreter. The lifter is the component of the fuzzer that is tasked with this. It has handlers for each of the bytecode and the bytecodes corresponding syntax to convert to its PHP equivalent code. It also handles PHP specific variable scope syntax among the other things. In short, the lifter converts PHPIL to PHP code, taking care of the syntax.

VIII. Mutation

Mutators are used to introduce small changes in the existing valid code while still preserving its behaviour. Mutation process leverages an existing corpus of seed inputs during fuzzing. It generates inputs by modifying the provided seeds.

We propose to implement the following two mutator for our php fuzzer:

- Input Mutator
- Operation Mutator

One of the main scopes for future improvements would be to add more efficient mutators.

A. *Input mutator*

This method mutates the data-flow in a program by replacing one instruction with another. In our php fuzzer all the instructions are variables. So the input mutator actually replaces the instruction variables with any random variable. An overview of this is represented in Listing 1

Listing 1. Input mutator PHPIL code

Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on November 28,2024 at 10:52:49 UTC from IEEE Xplore. Restrictions apply.

```
#Before mutation
v3 = CallFunction v1 v9 v5 v6

#After Mutation
v3 = CallFunction v4 v9 v5 v7
```

B. Operation Mutator

This method mutates the parameter and operation of an instruction. This includes all types of operations like Unary, Binary, changing of the constant values, Comparators and also method calls. An overview of the fuctioning of this mutator is highlighted in Listing 2.

Listing 2. Operation mutator PHPIL code

```
#Before Mutation
v1 = LoadInteger 10
if(v1<20){
}
else{
}

#After Mutation

v1 = LoadFloat 30.0
if(v1>20){
}
else{
}
```

IX. COVERAGE

Randomly feeding arbitrary input to the PHP binary does not give us any feedback about the behavior of that input. If we happen to know how that input behaved, like what all code code paths were executed, then we can fuzz more efficiently. This section describes the approach that we used to gather information from the PHP application about the code paths that were executed and how we plan to use this information.

In order to gather the code coverage information from PHP, we will be using the popular /edge coverage/ metric. For this, we instrument PHP during compile time instrumentation. We used clang's `sanitizer-coverage` feature which causes all the branches in the control flow graph to be instrumented [8] with a call to the function `__sanitizer_cov_trace_pc_guard` (Listing 3). We are free to add the contents of this function will be discussed in the following paragraph.

Since we can write the body of the `__sanitizer_cov_trace_pc_guard` function, we can execute our code each time a new branch is hit. What remains here is how to share the coverage data with the fuzzer process. For this we will be using a shared memory region between the fuzzer and the PHP application. The shared memory region acts as a bitmap to hold the information whether an edge is hit or not. For a particular input if the nth edge is hit in the PHP executable, the nth bit in the shared memory region is set. Thus the fuzzer can know if a new path was found using the current input by comparing the bitmap before and after the execution of this input. Our implementation of the `__sanitizer_cov_trace_pc_guard` function, which handles the setting of the bits in the shared memory each time a new edge is hit, is described in Listing

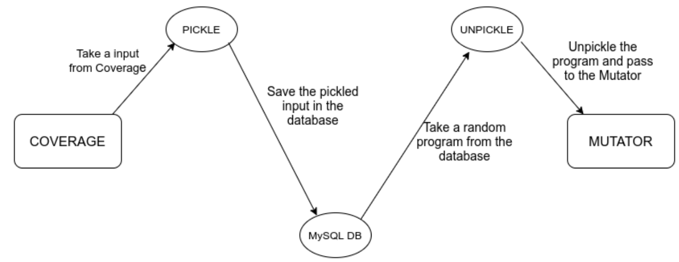

Fig. 2. Working of the Corpus.

Listing 3. Edge numbering code

```
void
__sanitizer_cov_trace_pc_guard(uint32_t *guard)
{
  // Duplicate the guard check.
  if (!*guard) return;

  // Calculate the index of bit to be set
  int idx = *guard/8;
  shmem[idx] |= 1 << (*guard%8);

  // Set the bit for this edge
  *guard = 0;

}
```

Since the core of our fuzzer is written in python, we wrote a Python module in C to interface with the Code Coverage data.

From the coverage data, the fuzzer gets the information about whether the current input triggered any new paths or not. If it did find new paths in the PHP binary, the particular input is sent for mutation and added to the corpus. Otherwise this input is marked as uninteresting and is discarded.

X. CORPUS

The corpus is like a database that contains a set of inputs for the fuzz target which leads to the discovery of newer paths in the existing code coverage. When starting a fuzzing process the fuzzer should have "Seed corpus" (the set of input that needed to be seeded for the mutator). The quality of the seed corpus has a huge impact on fuzzing efficiency as it allows the fuzzer to discover new code paths more easily. As PHP has a large codebase, so it would be having a lot of unique paths and if we would be using a file-based database then it could take a lot of space hence we used MySQL database in our fuzzer. A good reference for seed inputs would be regression tests included in the PHP codebase.

A. Working of Corpus

Figure 2 details how the corpus runs.

Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on November 28,2024 at 10:52:49 UTC from IEEE Xplore. Restrictions apply.

XI. Testing

The result was manually verified for syntactical and semantic correctness and we did not perform fuzzing on a scale. The whole project is still being extended and we plan to open source it soon so more people can run it at scale. For validation of the various components of the fuzzer, we planted intentional bugs in the PHP interpreter and checked that the fuzzer is able to reach those by mutation of another code sample. This also enabled us to verify our test harness that catches the crashes and logs them. Since the fuzzer is not fully completed yet (lots of mutation techniques to be added, a better looking frontend, network support for parallelly fuzzing multiple inputs for better performance etc), we have not yet used the fuzzer for actually finding bugs which will be done within the next couple of months.

XII. Conclusion

Our fuzzer uses the novel technique, fuzzing using byte code, for fuzzing the PHP interpreter. This is the first PHP fuzzer which fuzz on bytecode level and that makes this fuzzer different from other fuzzers. As this fuzzer uses a different technique, it intends to find new bugs on PHP interpreter.

We have implemented our own Intermediate Language (PhpIL) which will generate semantically and syntactically correct PHP programs, that will surely lower the errors when passing to the PHP interpreter. Mutating on the bytecode level will be more efficient than Abstract Syntax Tree (AST) because the fuzzing on bytecode level can prove to be more effective than AST.

An interesting area for further development would be to make the fuzzer more efficient by integrating both AST and ByteCode techniques together, which will provide us both grammar based coverage and system coverage. Also, currently the fuzzer is only a single threaded process. This a major limitation as it largely impacts the amount of samples we execute at a point. A future goal will be to provide multi thread/ remote procedure call support in order to use the underlying hardware judiciously.

Acknowledgment

We have taken efforts for this project but it would not have been possible without the support and guidance of our project mentor Dr Jayaraj Poroor and Mrs Deepthi LR. Our project coordinator Mrs. Lekshmi S. Nair, also played a vital role in assisting us with different parts of the project.

We are highly indebted to them for their guidance and constant supervision as well as for providing necessary information regarding the project and also for their support in completing the project.

Our thanks and appreciations also go to our colleagues in developing the project and people who have willingly helped us out with their abilities.

References

[1] PHP. The general-purpose scripting language, especially suited for web development. https://www.php.net

[2] Nischai Vinesh, Sanjay Rawat, Herbert BosCristiano, Giuffrida and M Sethumadhavan : "ConFuzz—A Concurrency Fuzzer", Advances in Intelligent Systems and Computing book series, vol. 1045.

[3] Vivek Parikh, Prabhaker Mateti, "ASLR and ROP Attack Mitigations for ARM-Based Android Devices", Communications in Computer and Information Science, vol. 746.

[4] R. Vinayakumar, K. P. Soman, Prabaharan Poornachandran, S. Akarsh: "Application of Deep Learning Architectures for Cyber Security", Advanced Sciences and Technologies for Security Applications.

[5] S. Groß, "Fuzzil: Coverage guided fuzzing for javascript engines". Master's thesis, TU Braunschweig, 2018.

[6] Marcin Dominiak, Wojciech Rauner: "Efficient approach to fuzzing interpreters." In BlackHat Asia, 2019.

[7] Junjie Wang, Bihuan Chen, Lei Wei, Yang Liu, "Superion: Grammar-Aware Greybox Fuzzing", arXiv 1812.01197

[8] Clang Sanitizer Coverage compile time instrumentation for runtime code coverage https://clang.llvm.org/docs/SanitizerCoverage.html

Authorized licensed use limited to: Univ of Calif Santa Barbara. Downloaded on November 28,2024 at 10:52:49 UTC from IEEE Xplore. Restrictions apply.