Fuzzing PHP for Fun and Profit

tripwire.com/state-of-security/vert/fuzzing-php-for-fun-and-profit

Craig Young

November 20, 2018

PHP is probably the single most prevalent server-side scripting language on the web. PHP has been the de facto choice for popular blog platforms like WordPress, Joomla and Drupal, which makes it a very attractive target for a wide range of attackers. It is also a very ideal system for demonstrating the power of American Fuzzy Lop (AFL) to identify memory corruption bugs within mature software.

From the AFL website:

> *American fuzzy lop* is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary.

In other words, the AFL compiler will add instructions to monitor the binary's execution flow, and the AFL fuzzer will use this instrumentation to recognize when a test case exercises a new state transition.

To begin, you'll need to have a Linux server with American Fuzzy Lop (AFL) and typical build tools (e.g. make, autoconf, etc) installed. Some distros offer AFL binary packages, and the latest AFL source package is always available from Michal Zalewski's web site: http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz. When building from source, be sure to build LLVM mode support to get the afl-clang-fast(++) compilers.

At a high-level, the steps for using AFL to fuzz a PHP function are:

1. Obtain PHP source
2. Configure and make PHP with the afl-clang-fast compiler and AddressSanitizer (ASAN)
3. Prepare a short PHP code sample to exercise the targeted function
4. Produce a few starting test cases to seed the fuzz
5. Run afl-fuzz and wait for crashes

Step 1: Obtaining the source

PHP source is available in the php-src GitHub repository, and source archives from each release or release candidate are listed in the releases section: https://github.com/php/php-src/releases

Download and extract the source on your system.

```
wget https://github.com/php/php-src/archive/php-7.2.11.tar.gz && tar xf php-
7.2.11.tar.gz
```

Step 2: Compiling

To build an instrumented PHP interpreter, we use *buildconf* to produce a *configure* script which we'll use to prepare a *Makefile* for compiling with *afl-clang-fast*, and then we make with ASAN enabled. For this step, you will need all the appropriate build dependencies for PHP.

For example, on Ubuntu, you will need build-essential, autoconf, bison, libxml2-dev and afl. You may also be able to use a package manager to install build dependencies like with apt build-dep. More information about obtaining PHP dependencies can be found in the PHP documentation.

(Note that if the function you wish to fuzz in PHP is actually a wrapper for some dependency like libgd, you'll want to first build that dependency with AFL and then link PHP against this instrumented lib.)

```
cd php-src-php-7.2.11
./buildconf --force
CC=afl-clang-fast CXX=afl-clang-fast++ ./configure
AFL_USE_ASAN=1 make
```

At this point, we could use the resulting binary (./sapi/cli/php) to start fuzzing, but the results are far from optimal. Even with the high-performance fork server in AFL, the overhead involved with initializing PHP severely limits the number of executions per second. In normal operation, *afl-fuzz* first starts a process but freezes it just after the OS has finished initializing the execution environment.

By creating an image of the process at this point, test cases can be run in AFL without the costs associated with linking and initializing libraries. This allows us to run PHP dozens or potentially hundreds of times per second (depending on the server), but there is still a lot of time spent initializing things within PHP for each test case.

Fortunately, the afl-clang-fast compilers support an advanced feature which allows us to minimize the overhead related to initializing the process for each test case. This feature, known as deferred instrumentation, effectively allows us to tell *afl-fuzz* that it can run further into the program before creating an image of the process. By taking this snapshot of the process after PHP has prepared itself for executing a script, each test case execution will take less time and processing.

To take advantage of this, we must add a single line to the C source code for the PHP command line tool. The goal is to add this line as close to where the user supplied code snippet will be processed as possible while still leaving the program in a "clean" state suitable for cloning the process. Most importantly, this means that it must be before threads or offset sensitive file descriptors are created.

For PHP, the main source file for the CLI tool is *sapi/cli/php_cli.c*, and the code for processing a PHP string to eval via *php -r* can be found by finding where *PHP_MODE_CLI_DIRECT* is used similar to the following:

```
1040         case PHP_MODE_CLI_DIRECT:
1041                 cli_register_file_handles();
1042                 if (zend_eval_string_ex(exec_direct, NULL, "Command line code", 1)
== FAILURE) {
1043                         exit_status=254;
1044                 }
1045                 break;
```

On this particular version, line 1042 is the start of the code we want to fuzz, so we can add *__AFL_INIT();* just before this line:

```
1040         case PHP_MODE_CLI_DIRECT:
1041                 cli_register_file_handles();
1042                 __AFL_INIT();
1043                 if (zend_eval_string_ex(exec_direct, NULL, "Command line code", 1)
== FAILURE) {
1044                         exit_status=254;
1045                 }
1046                 break;
```

Saving the file with this change, you can quickly rebuild the *php* binary:

```
AFL_USE_ASAN=1 make
```

On my server, this one line addition roughly triples the number of executions per second that I get when fuzzing the *unserialize()* function from PHP. (From around 30-40/sec to 100-110/sec with the same test cases.)

This definitely helped performance, but the real improvement comes when we allow *afl-fuzz* to loop over the same code rather than closing the process after a single test case. For this, we add a *while* loop around the *zend_eval_string_ex()* call using the *__AFL_LOOP()* macro as a break condition.

After this, my *php_cli.c* has the following:

```
1040         case PHP_MODE_CLI_DIRECT:
1041                 cli_register_file_handles();
1042                 __AFL_INIT();
1043                 while (__AFL_LOOP(100000))
1044                         zend_eval_string_ex(exec_direct, NULL, "Command line code",
1);
1045                 break;
```

We need to rebuild this again with:

```
AFL_USE_ASAN=1 make
```

Running the same test as before, my setup now yields 4000+ executions/second, meaning we have gained roughly a 100x improvement compared to the original source.

Step 3: Prepare a PHP code snippet to fuzz

For the purpose of this demo, we'll use the *unserialize()* function, which has produced good results for me in the past. For the sake of fuzzing, we want to write a line which will read some user-input from a file or from *stdin*. My preference is generally to use *stdin* so a basic sample is:

```
unserialize(file_get_contents("php://stdin"));
```

When supplied as an argument for *php -r*, this code will wait for data on *stdin* and supply it as input to *unserialize()*.

Step 4: Preparing sample input test cases

At this point, we need to give AFL some examples of what serialized PHP data looks like. We can do this by putting a few different data types into *serialize()* and saving the output in a new directory.

For example:

```
mkdir serialized_data && cd serialized_data
../sapi/cli/php -r 'echo serialize("a");' > string
../sapi/cli/php -r 'echo serialize(1);' > number
../sapi/cli/php -r 'echo serialize([1,2]);' > array_of_num
../sapi/cli/php -r 'echo serialize(["1","2"]);' > array_of_str
../sapi/cli/php -r 'echo serialize([["1","2"],["3","4"],[1,2]]);' > array_of_array
```

Step 5: Start the fuzz

Now that we have an instrumented binary and some test cases, we can begin fuzzing with *afl-fuzz*. In order to get useful results from address sanitization (ASAN), it is necessary to set an environmental variable so that PHP will disable its custom memory allocator. Without setting *USE_ZEND_ALLOC=0*, memory safety issues will be invisible to ASAN. My preference is to start a screen session with this variable and then fuzz within that.

```
USE_ZEND_ALLOC=0 screen
```

Within the first screen, the following command will start a fuzz:

```
cd ..
afl-fuzz -i serialized_data -o basic_fuzz -m none -- ./sapi/cli/php -r
'unserialize(file_get_contents("php://stdin"));'
```

For reference, this is saying that files in *./serialized_data/* will be used as initial inputs for the fuzzing queue in *./basic_fuzz/*. The *-m none* flag tells AFL not to artificially limit the memory available for the process, which is a necessary option when fuzzing an ASAN instrumented binary.

Over time, AFL will discover new paths, and the "total paths" field in the AFL UI should quickly increment. If it does not, there is likely something wrong with the fuzz. Eventually, you may encounter crashes, and these faulting test cases will be recorded to *./basic_fuzz/crashes/id** for analysis. Don’t get too excited about crashes when fuzzing *unserialize()* though because most will be benign crashes from ASAN not being able to allocate enough memory. This is because ASAN requires extra memory for tracking all allocations, and crafted serialized objects can trigger large memory allocations.

It’s easy to make a simple BASH loop to find any interesting crashes though. Starting from the *./basic_fuzz/crashes* path:

```
for FILE in $(ls id*); do cat $FILE | ../../sapi/cli/php -r
"unserialize(file_get_contents('php://stdin'));" 2>&1 | grep -E "SUMMARY|ERROR" | grep -
v "LargeMmap" && echo $FILE; done
```

For reference, the bugs I’ve found with this technique can be reviewed on the PHP bug tracker:

https://bugs.php.net/bug.php?id=74101

https://bugs.php.net/bug.php?id=74103

https://bugs.php.net/bug.php?id=74111

And from fuzzing exif functions: https://bugs.php.net/bug.php?id=76130

As an added bonus, each of these bugs was awarded a $500 bounty via the PHP (IBB) program on Hacker One.