Python Fuzzing 101

Python 3 to obecnie jeden z najpopularniejszych języków programowania. Stosunkowo prosta składnia oraz łatwość ekspresji powoduje, że jest on bardzo częstym wyborem. Popularność niesie za sobą masę bibliotek, które powstają nie tylko w Pythonie, ale także w innych językach (np. C) w celu poprawienia wydajności. Duża ilość kodu oraz ograniczone zasoby osobowe powodują, że oprócz wykonywanych regularnie procesów code-review dobrze jest rozejrzeć się za zautomatyzowanym sposobem poszukiwania błędów i włączyć go do procesu rozwoju oprogramowania w taki sposób, aby otrzymywać informacje o znalezionych błędach zaraz po kolejnym commicie. Taką możliwość daje fuzzing (lub property testing).

Metoda ta, opracowana na przełomie lat 80-90, stale dopracowywana i rozwijana, pozwala na zbadanie wielu stanów procesu poprzez dostarczanie odpowiednio przygotowanych danych i badanie zachowania uruchomionego kodu. Chociaż coraz częściej implementuje się fuzzing jako element pipeline CI/CD, to w gruncie rzeczy jest to jedna z odrębnych technik szukania błędów i podatności. W niniejszym artykule zaprezentuję przykłady użycia fuzzera Atheris do testowania kodu pythonowych bibliotek.

PRZYSPIESZONY WSTĘP DO FUZZINGU

Czym jest fuzzing? Zdefiniować go można jako automatyczną technikę testowania oprogramowania. Polega ona na losowym tworzeniu lub modyfikowaniu danych wejściowych i przekazywaniu ich do testowanego kodu (tzw. SUT – software under test). Celem tych działań jest odkrycie nieoczekiwanych zachowań programu (potencjalnie błędów bezpieczeństwa). Do przeprowadzenia takich testów wykorzystuje się całe środowiska fuzzingowe, w których najważniejszym elementem jest sam fuzzer. Oprócz tego środowisko takie oferuje narzędzia do np. deduplikacji przypadków, które doprowadziły do nieprzewidzianego zachowania programu. Fuzzery można różnie klasyfikować. Na potrzeby niniejszego artykułu wystarczy podział ze względu na „wiedzę" na temat SUT. Ogólnie rzecz biorąc, do najprostszych należą fuzzery **blackbox**, które nie posiadają informacji na temat struktury testowanego programu. Z kolei fuzzery typu **greybox** wykorzystują instrumentację kodu, śledząc ścieżki wykonania. Pozwala to zwiększyć efektywność w stosunku do pierwszego typu, natomiast stawiają pewne wymagania (jak np. możliwość kompilacji z odpowiednią instrumentacją). Natomiast fuzzery, które wykorzystują rozbudowane techniki analizy statycznej, dynamicznej, wykonanie symboliczne, należą do grupy **whiteboxowych** narzędzi i są najbardziej skuteczne z powyżej wymienionych. W tym artykule skupimy się głównie na dwóch pierwszych klasach.

Wykorzystanie losowo generowanych danych wejściowych „na ślepo" może nie być najskuteczniejszą metodą generowania przypadków testowych. Każdy, kto choć raz pisał testy np. jednostkowe w sposób intuicyjny, rozumie, jakie przypadki danych wejściowych powinny zostać przetestowane. W szczególności należy skupić się np. na wartościach brzegowych – tzw. edge case. Istnieje wiele trywialnych błędów, wynikających np. z nieintuicyjnych, ale bardzo prostych problemów matematycznych. Przykład, który często jest przytaczany, to liczenie odległości między dwoma punktami. Problem ten może zostać przetransformowany do codziennych dylematów – np. ile siatki potrzeba do ukończenia ogrodzenia składającego się np. z X słupków. W zależności od dalej poczynionych założeń odpowiedzi może być kilka (np. zależnych od tego, czy ogrodzenie ma stanowić łamaną otwartą czy też łamaną zwyczajną zamkniętą – zamykać dany obszar). Przekładając problem na bardziej programistyczny, często zdarzają się błędy off-by-one wynikające np. z przeoczenia faktu indeksowania od określonej wartości lub niewzięcia pod uwagę wartości krańcowej. Aby wygenerować sekwencję liczb od 0 do 17 (włącznie) z krokiem 1, programista Pythona powinien użyć konstrukcji `range(0, 18)`, zamiast intuicyjnego `range(0,17)`. Choć dokumentacja opisuje wywołanie jako `class range(stop)`, co może być mylące.

Testowany fragment kodu może przyjmować na wejściu pokaźną liczbę bajtów. Taki przypadek nie jest niczym niezwykłym, to z grubsza opis działania różnych parserów (np. plików graficznych), które są bardzo wdzięcznym celem do testowania przy pomocy fuzzerów. Sprawdzenie wielu wartości każdego bajtu obrazka jest procesem żmudnym i przy większych rozmiarach niemożliwym do zrealizowania w sensownym czasie. Z drugiej strony jest to działanie nie zawsze potrzebne. Przy takim podejściu hipotetyczny fuzzer przez większość czasu testowałby wartości, które nie mają większego wpływu na przepływ sterowania w programie – byłoby to po prostu marnotrawstwo zasobów. Takie działanie określa podstawowe fuzzery typu blackbox, które nie dysponują informacją na temat uruchamianego kodu. Ich główna zaleta to łatwość w równoległym testowaniu programu za pomocą wielu instancji, jednak brak dodatkowych informacji powoduje, że często nie są one w stanie odkryć błędów ponad te najprostsze. Ich działanie opiera się z grubsza na wielokrotnym uruchamianiu procesu ze zmienionymi danymi wejściowymi i zapisywaniu wejściowych bajtów, gdy dojdzie do nieoczekiwanego zatrzymania, programu. Powodem przerwania przetwarzania może być próba wykonania niedozwolonej operacji (np. dzielenia przez zero). W momencie gdy wzrasta skomplikowanie kodu, jest to działanie niewystarczające, aby dostatecznie dobrze przetestować program. Dlatego też nowocześniejsze fuzzery (punktem przełomowym był american fuzzy lop [1] – w skrócie AFL – napisany i opublikowany przez Michała *lcamtufa* Zalewskiego w 2013 roku) korzystają między innymi z mechanizmu

Python Fuzzing 101

Python 3 to obecnie jeden z najpopularniejszych języków programowania. Stosunkowo prosta składnia oraz łatwość ekspresji powoduje, że jest on bardzo częstym wyborem. Popularność niesie za sobą masę bibliotek, które powstają nie tylko w Pythonie, ale także w innych językach (np. C) w celu poprawienia wydajności. Duża ilość kodu oraz ograniczone zasoby osobowe powodują, że oprócz wykonywanych regularnie procesów code-review dobrze jest rozejrzeć się za zautomatyzowanym sposobem poszukiwania błędów i włączyć go do procesu rozwoju oprogramowania w taki sposób, aby otrzymywać informacje o znalezionych błędach zaraz po kolejnym commicie. Taką możliwość daje fuzzing (lub property testing).

Metoda ta, opracowana na przełomie lat 80-90, stale dopracowywana i rozwijana, pozwala na zbadanie wielu stanów procesu poprzez dostarczanie odpowiednio przygotowanych danych i badanie zachowania uruchomionego kodu. Chociaż coraz częściej implementuje się fuzzing jako element pipeline CI/CD, to w gruncie rzeczy jest to jedna z odrębnych technik szukania błędów i podatności. W niniejszym artykule zaprezentuję przykłady użycia fuzzera Atheris do testowania kodu pythonowych bibliotek.

PRZYSPIESZONY WSTĘP DO FUZZINGU

Czym jest fuzzing? Zdefiniować go można jako automatyczną technikę testowania oprogramowania. Polega ona na losowym tworzeniu lub modyfikowaniu danych wejściowych i przekazywaniu ich do testowanego kodu (tzw. SUT – software under test). Celem tych działań jest odkrycie nieoczekiwanych zachowań programu (potencjalnie błędów bezpieczeństwa). Do przeprowadzenia takich testów wykorzystuje się całe środowiska fuzzingowe, w których najważniejszym elementem jest sam fuzzer. Oprócz tego środowisko takie oferuje narzędzia do np. deduplikacji przypadków, które doprowadziły do nieprzewidzianego zachowania programu. Fuzzery można różnie klasyfikować. Na potrzeby niniejszego artykułu wystarczy podział ze względu na „wiedzę" na temat SUT. Ogólnie rzecz biorąc, do najprostszych należą fuzzery **blackbox**, które nie posiadają informacji na temat struktury testowanego programu. Z kolei fuzzery typu **greybox** wykorzystują instrumentację kodu, śledząc ścieżki wykonania. Pozwala to zwiększyć efektywność w stosunku do pierwszego typu, natomiast stawiają pewne wymagania (jak np. możliwość kompilacji z odpowiednią instrumentacją). Natomiast fuzzery, które wykorzystują rozbudowane techniki analizy statycznej, dynamicznej, wykonanie symboliczne, należą do grupy **whiteboxowych** narzędzi i są najbardziej skuteczne z powyżej wymienionych. W tym artykule skupimy się głównie na dwóch pierwszych klasach.

Wykorzystanie losowo generowanych danych wejściowych „na ślepo" może nie być najskuteczniejszą metodą generowania przypadków testowych. Każdy, kto choć raz pisał testy np. jednostkowe w sposób intuicyjny, rozumie, jakie przypadki danych wejściowych powinny zostać przetestowane. W szczególności należy skupić się np. na wartościach brzegowych – tzw. edge case. Istnieje wiele trywialnych błędów, wynikających np. z nieintuicyjnych, ale bardzo prostych problemów matematycznych. Przykład, który często jest przytaczany, to liczenie odległości między dwoma punktami. Problem ten może zostać przetransformowany do codziennych dylematów – np. ile siatki potrzeba do ukończenia ogrodzenia składającego się np. z X słupków. W zależności od dalej poczynionych założeń odpowiedzi może być kilka (np. zależnych od tego, czy ogrodzenie ma stanowić łamaną otwartą czy też łamaną zwyczajną zamkniętą – zamykać dany obszar). Przekładając problem na bardziej programistyczny, często zdarzają się błędy off-by-one wynikające np. z przeoczenia faktu indeksowania od określonej wartości lub niewzięcia pod uwagę wartości krańcowej. Aby wygenerować sekwencję liczb od 0 do 17 (włącznie) z krokiem 1, programista Pythona powinien użyć konstrukcji `range(0, 18)`, zamiast intuicyjnego `range(0,17)`. Choć dokumentacja opisuje wywołanie jako `class range(stop)`, co może być mylące.

Testowany fragment kodu może przyjmować na wejściu pokaźną liczbę bajtów. Taki przypadek nie jest niczym niezwykłym, to z grubsza opis działania różnych parserów (np. plików graficznych), które są bardzo wdzięcznym celem do testowania przy pomocy fuzzerów. Sprawdzenie wielu wartości każdego bajtu obrazka jest procesem żmudnym i przy większych rozmiarach niemożliwym do zrealizowania w sensownym czasie. Z drugiej strony jest to działanie nie zawsze potrzebne. Przy takim podejściu hipotetyczny fuzzer przez większość czasu testowałby wartości, które nie mają większego wpływu na przepływ sterowania w programie – byłoby to po prostu marnotrawstwo zasobów. Takie działanie określa podstawowe fuzzery typu blackbox, które nie dysponują informacją na temat uruchamianego kodu. Ich główna zaleta to łatwość w równoległym testowaniu programu za pomocą wielu instancji, jednak brak dodatkowych informacji powoduje, że często nie są one w stanie odkryć błędów ponad te najprostsze. Ich działanie opiera się z grubsza na wielokrotnym uruchamianiu procesu ze zmienionymi danymi wejściowymi i zapisywaniu wejściowych bajtów, gdy dojdzie do nieoczekiwanego zatrzymania, programu. Powodem przerwania przetwarzania może być próba wykonania niedozwolonej operacji (np. dzielenia przez zero). W momencie gdy wzrasta skomplikowanie kodu, jest to działanie niewystarczające, aby dostatecznie dobrze przetestować program. Dlatego też nowocześniejsze fuzzery (punktem przełomowym był american fuzzy lop [1] – w skrócie AFL – napisany i opublikowany przez Michała *lcamtufa* Zalewskiego w 2013 roku) korzystają między innymi z mechanizmu

JAK WYNALEŹĆ KOŁO NA NOWO, CZYLI O PRZEPISANIU WYSZUKIWARKI ARTYKUŁÓW „PROGRAMISTY”

INDEX: 285358

www.programistamag.pl

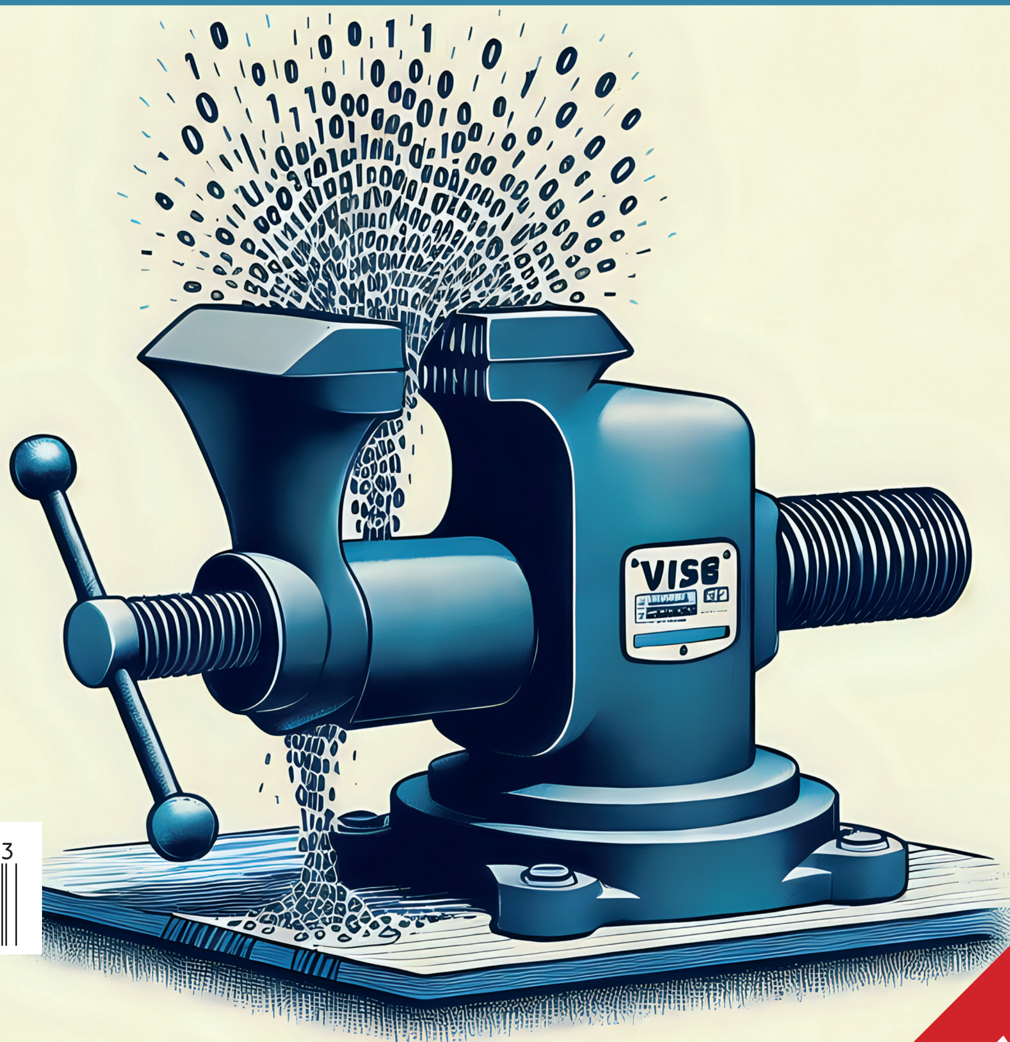
Magazyn programistów i liderów zespołów IT

programista

3/2024 (113)

Cena 28,90 zł (w tym VAT 8%)

FUZZOWANIE PYTHONA 101



- C, PHP, rozszerzenia i parametry nullable
- SMP we FreeRTOS
- Ile k... ...armowe” ... ...wanie?
- Emulacja, wirtualizacja i konteneryzacja – różnice i zastosowania
- O idempotencji, deduplikacji i ponawianiu...
- ...gmatyczne podejście do testów jednostkowych

NOWY NUMER JUŻ W EMPIKACH