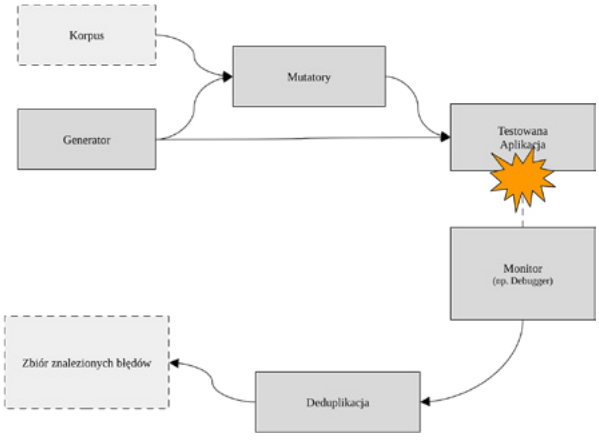
Fuzzing

Fuzzing jest z zasady prostą metodą testowania oprogramowania, polegającą na przekazywaniu programowi nieprawidłowych danych, a następnie sprawdzaniu, czy aplikacja poprawnie na nie zareagowała. Technika ta przeżywa obecnie drugą młodość jako niezwykle skuteczny sposób na odkrywanie błędów bezpieczeństwa, a my w tym artykule postaramy się omówić, co leży u jej podstaw.

WYSOKOPOZIOMOWY WSTĘP DO FUZZINGU

Załóżmy, że tworzymy od początku (ew. analizujemy od strony bezpieczeństwa) aplikację w C lub C++, której zadaniem jest wczytywanie plików w formacie PDF i renderowanie opisanych przez nie dokumentów do postaci bitmapy. O ile wiemy, że nasza aplikacja działa poprawnie dla testowych plików, które zebraliśmy lub stworzyliśmy, o tyle chcielibyśmy również, by działała stabilnie w przypadku nie do końca poprawnych PDFów lub dokumentów wykorzystujących elementy formatu w inny sposób, niż przewidzieliśmy. Jako programiści C/C++ zdajemy sobie również sprawę, że zakończenie naszej aplikacji z komunikatem „Segmentation fault" lub „Access Violation" może sygnalizować błąd związany z bezpieczeństwem, który w skrajnym wypadku może doprowadzić do wykonania kodu dostarczonego przez atakującego, a co za tym idzie – zainfekowania maszyny naszego użytkownika.

Fuzzing (lub *fuzz testing*) jest idealną praktyką, którą możemy zastosować w takim przypadku. Jest to automatyczna metoda testowania oprogramowania, której głównym założeniem jest wielokrotne podawanie aplikacji zmutowanych (przekształconych) danych wejściowych i obserwowanie, czy owe mutacje spowodują jej przedwczesne zakończenie z powodu błędu (lub bardziej formalnie: przejście w jeden z błędnych stanów).



Rysunek 1. Uproszczony schemat pojedynczego przebiegu fuzzera

Przebieg pojedynczego testu wygląda następująco (Rysunek 1):

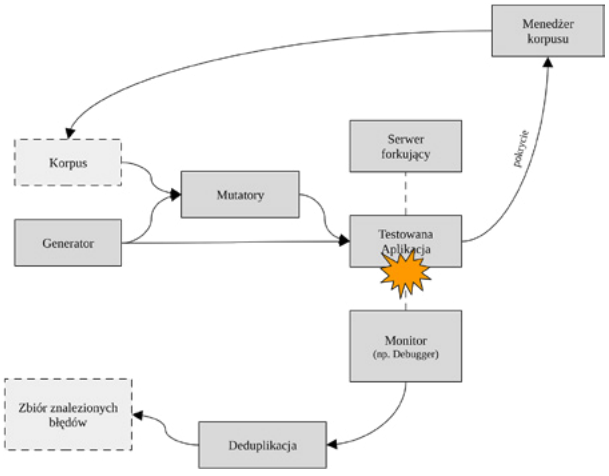
1. Z korpusu (zbioru danych wejściowych) wybierany jest, najczęściej losowo lub sekwencyjnie, pojedynczy plik. Alternatywnie uruchamiany jest program generujący zestaw danych wejściowych.
2. Wybrane dane są następnie poddawane mutacjom, które najczęściej mają charakter losowy (np. wybierany jest 1% bajtów w pliku, których wartość jest zmieniana na inną). W przypadku użycia generatorów mutacje mogą zostać wprowadzone już na etapie generowania danych – w tym wypadku oddzielna faza mutacji jest opcjonalna.
3. Zmutowane dane zostają podane testowanej aplikacji, której wykonanie jest monitorowane, np. przez odpowiednio stworzony lub oskryptowany debugger. W przypadku braku wykrycia nieprawidłowości w wykonaniu runda fuzzingu zostaje na tym etapie zakończona.
4. W przypadku wykrycia błędu zostają zebrane dostępne informacje o jego naturze – najczęściej jest to stos wywołań, informacje o stanie rejestrów czy kilka okolicznych instrukcji kodu maszynowego.
5. Tak powstały raport jest przekazywany do modułu deduplikacji, odpowiedzialnego za zapisywanie nowych błędów (oraz danych wejściowych, które doprowadziły do powstania danego błędu) oraz odrzucanie duplikatów.

Powyższy proces jest powtarzany wielokrotnie przez długi okres czasu (zwyczajowo od kilku godzin do kilku tygodni), co pozwala na przeprowadzenie milionów lub wręcz miliardów testów, a często również na znalezienie wielu błędów.

Należy w tym miejscu zaznaczyć, że każda z faz procesu (lub każdy moduł systemu fuzzującego) charakteryzuje się własnymi cechami i może być zarówno bardzo prosta, jak i bardzo złożona – stąd też każdej z nich poświęcimy osobną sekcję w niniejszym artykule.

WSPÓŁCZESNY FUZZING

Opisany w poprzednim paragrafie model systemu fuzzującego jest modelem klasycznym. W ciągu kilku ostatnich lat został on poszerzony o kilka istotnych elementów, o których można już powiedzieć, że weszły do kanonu (Rysunek 2).



Rysunek 2. Uproszczony schemat pojedynczego przebiegu współczesnego fuzzera

Przede wszystkim należy wspomnieć o użyciu pomiaru pokrycia testowanej aplikacji przez kolejne testy – pozwala to na wyodrębnienie mutacji docierających do nieprzetartych wcześniej fragmentów kodu, a co za tym idzie – rozbudowanie bazowego korpusu, co z kolei często powoduje dalszy wzrost pokrycia w kolejnych przebiegach [AFL-TECH]. Jest to swoisty algorytm genetyczny, który okazuje się być niezwykle skuteczny w praktyce [AFL-JPEGS] – więcej o tym podejściu piszemy w sekcji „Fuzzing a pokrycie kodu".

Drugim elementem wartym wspomnienia jest serwer forkujący (ang. *Fork Server*) – jest to zoptymalizowany sposób uruchamiania testowanej aplikacji na systemach Unixowych (tj. takich, które używają modelu `fork+execve`) pozwalający na zwiększenie liczby uruchomień z kilku-kilkudziesięciu do kilku tysięcy na sekundę – więcej o serwerze forkującym piszemy w sekcji mu poświęconej.

Ewolucję przeszły również metody kompletowania wstępnego korpusu danych wejściowych, mutatory, algorytmy deduplikacji czy same metody wykrywania błędów w aplikacjach – o każdym z tych elementów wspominamy w dalszej części artykułu.

Warto również nadmienić, iż fuzzing, z uwagi na swoją naturę, jest procesem bardzo dobrze skalującym się, stąd też współcześnie podczas fuzz-testingu często wykorzystuje się wiele rdzeni procesora [AFL-USERS], a w skrajnych wypadkach wiele setek lub wręcz tysięcy serwerów [SCALE] [1000].

PRZYGOTOWANIE APLIKACJI DO TESTOWANIA

Choć fuzzing jest uniwersalną techniką testowania oprogramowania, nie ograniczoną do konkretnej technologii, języka programowania czy sposobu przetwarzania danych, najczęściej spotykany jest scenariusz, w którym mamy do czynienia z pojedynczym programem wykonywalnym, który wczytuje dane wejściowe z pliku, przetwarza je i kończy działanie. Opisana sytuacja pokrywa się ze sposobem działania prostych programów pomocniczych, w szczególności na systemach z rodziny Linux. W innych przypadkach możliwe jest takie przystosowanie testowanego kodu i środowiska, by założenia tego prostego modelu zostały spełnione. W szczególności:

- Kiedy mamy do czynienia z biblioteką, a nie konkretną aplikacją, możemy samemu stworzyć krótki program (tzw. *wrapper*), który wywoła odpowiednie funkcje owej biblioteki na danych załadowanych z pliku.
- Jeśli program odczytuje dane ze źródła innego niż system plików, możemy przekompilować go z odpowiednią zmianą (o ile posiadamy dostęp do kodu źródłowego), uruchamiać go w sposób wymuszający odczyt danych z pliku (np. poprzez interpreter Bash i operator przekierowania <, w przypadku programu korzystającego ze standardowego wejścia) lub wreszcie zmodyfikować sam fuzzer w taki sposób, by przekazywał dane wejściowe w sposób zrozumiały dla konkretnej aplikacji.
- Jeśli jeden zestaw danych wejściowych składa się z wielu plików, możemy każdy z owych zestawów spakować w pojedyncze archiwa (np. ZIP), a następnie rozpakowywać w fuzzerze bezpośrednio przed przekazaniem ich do testowanego programu.

I tak, choć zaproponowany model nie odpowiada w pełni wszystkim przypadkom spotykanym w rzeczywistości, większość z nich można do niego sprowadzić, i jest on wystarczająco prosty, by na jego przykładzie prowadzić dalsze rozważania.

TWORZENIE KORPUSU PLIKÓW WEJŚCIOWYCH

Jak wspomnieliśmy wcześniej, fuzzing opiera się na przekazywaniu niepoprawnych (najczęściej w losowy sposób) danych i obserwowaniu reakcji testowanego kodu. Co oczywiste, używanie *całkowicie* losowych danych wejściowych (np. odczytanych bezpośrednio z pseudourządzenia `/dev/urandom`) nie przyniesie oczekiwanych rezultatów – prawie każde oprogramowanie oczekuje na wejściu pewnej struktury, zazwyczaj praktycznie niemożliwej do odnalezienia w naiwnie wylosowanej sekwencji bajtów. Łatwo sprawdzić, że prawdopodobieństwo wylosowania prawidłowego pliku RAR, PNG lub DOC jest bliskie zeru. Używając danych wejściowych tego typu, „odbijalibyśmy" się nieustannie od początkowych linii kodu odpowiedzialnych za weryfikowanie poprawności nagłówków, nie zyskując nigdy możliwości przetestowania regionów odpowiedzialnych za samo przetwarzanie danych.

Z tego powodu w praktyce najlepiej sprawdza się fuzzing przy użyciu danych posiadających poprawne i jak najbardziej różnorodne struktury obsługiwane przez testowany program, z niewielką liczbą losowych zmian, które mogą ujawnić błędy w najbardziej skomplikowanych miejscach w kodzie. Dzięki takiemu podejściu z jednej strony możemy dotrzeć do jak największej liczby funkcjonalności oferowanej przez program, z drugiej zaś możemy prowokować ujawnianie się błędów poprzez drobne, potencjalnie nie oczekiwane zmiany na wejściu.

Cel ten można osiągnąć na kilka sposobów. Jednym z nich, który sprawdza się bardzo dobrze w przeważającej liczbie przypadków, jest przygotowanie początkowego, licznego korpusu całkowicie poprawnych danych wejściowych, a następnie wprowadzanie do nich drobnych, losowych mutacji w kolejnych iteracjach fuzzingu. W przypadku testowania implementacji obsługi plików PDF wiązałoby się to z zebraniem wielu różnych dokumentów zapisanych w tym formacie; podobnie z każdym innym formatem plików.

Można również nadmienić, że podejście to ma jeszcze więcej zalet. Nawet w przypadku fuzzingu z użyciem informacji o pokryciu kodu (o czym więcej piszemy w dalszej części artykułu), gdzie teoretycznie cała specyfikacja formatu mogłaby zostać losowo „odtworzona" przez fuzzer, posiadanie poprawnie sformatowanych plików na samym początku procesu testowania pozwala na zaoszczędzenie wielu godzin czasu zegarowego oraz procesora; nie wspominając o fakcie, że zazwyczaj odnalezienie wszystkich konstrukcji wspieranych przez konkretny format danych jest niemożliwe nawet przy wykorzystaniu danych o pokryciu kodu. Co również ważne, skompletowanie plików wejściowych w danym formacie pozwala na wielokrotne użycie takiego korpusu podczas testowania wielu implementacji owego formatu.

Istnieje kilka wypróbowanych sposobów na tworzenie tego typu zbiorów plików wejściowych, w przypadkach, gdy to nie my jesteśmy autorami danego formatu. Po pierwsze, wiele projektów z rodziny open-source oferuje gotowe zestawy danych testowych, zazwyczaj wykorzystywanych w testach regresyjnych. Za przykład może posłużyć tutaj biblioteka do obsługi multimediów o nazwie FFmpeg i związany z nią system FATE, który oferuje setki obrazów i plików audio/wideo zapisanych w najbardziej egzotycznych formatach danych. Gdyby nie istnienie owego systemu, odnalezienie plików zapisanych w wielu z tych formatów byłoby znacząco utrudnione. Czasami zdarza się również, że choć pliki testowe związane z danym projektem nie są publicznie dostępne do ściągnięcia, wystarczy poprosić o nie autora lub aktywnych developerów,