Programowanie Defensywne

Testy bezpieczeństwa

Arkadiusz Chrobot

Katedra Systemów Informatycznych

24 maja 2024

Plan

Wprowadzenie

Termin *testy bezpieczeństwa* ma wiele znaczeń. Najczęściej jest on rozumiany jako *testy penetracyjne*, czyli symulacje ataków przeprowadzane przez testerów bezpieczeństwa, których celem jest wykrycie podatności, zanim oprogramowanie zostanie zainstalowane na środowisku produkcyjnym. Testy te zaliczane są do etapu *testów systemowych* lub nawet *akceptacyjnych*. Jednakże bezpieczeństwo można sprawdzać już na znacznie wcześniejszych etapach rozwoju i testów. Kompleksowe podejście do tego zagadnienia można znaleźć w opracowaniu [1]. W ramach tego wykładu zostaną przedstawione tylko podstawowe wiadomości na ten temat, głównie te, które można zastosować na etapie programowania.

Testy jednostkowe

Testowanie wymagań bezpieczeństwa, czyli poufności, integralności i dostępności, należy rozpocząć już na etapie tworzenia najmniejszych komponentów. Zazwyczaj wtedy programiści tworzą *testy jednostkowe*, które sprawdzają zgodność działania tworzonego przez nich kodu ze specyfikacją. Aby zautomatyzować ten rodzaj testowania opracowano wiele narzędzi, między innymi dla większości języków programowania stworzono biblioteki do testów, których jedną z wielu zalet jest to, że pozwalają oddzielić kod testujący, od kodu produkcyjnego. Na tym etapie najczęściej tworzone są testy, które sprawdzają działanie tworzonego kodu dla typowych danych. Testowanie bezpieczeństwa wymaga jednak trochę innego podejścia. Nadal można stosować zarówno metodę *testowania funkcjonalnego* (inaczej: *metodę czarnej skrzynki*) oraz metodę *testowania strukturalnego* (inaczej: *metodę przezroczystej skrzynki*), ale dobór *przypadków testowych*[1] jest w tym wypadku kluczowy.

[1] Przypadek testowy to specyfikacja pojedynczego test („eksperymentu testowego").

Testy jednostkowe

Określając przypadki testowe można posłużyć się metodą *klas równoważności*, czyli podzielić dane wejściowe na zbiory, które mają podobną charakterystykę. Przykładowo, jeśli testowany komponent przetwarza pliki tekstowe w wielu formatach, to można podzielić je na pliki w formacie XML, YAML, JSON oraz pliki płaskie. Dalej, należy z każdej klasy wybrać przypadki, które są dla niej typowe (należą do „środka" klasy), jak i te, które są dla niej brzegowe, a nawet takie, które są tuż poza granicami tej klasy. Przykładowo, jeśli komponent akceptuje ciągi dużych liter i cyfr, które nie są krótsze niż jeden znak, ani dłuższe niż dziesięć znaków, to należy dobrać tak dane wejściowe dla testów, aby znalazł się tam ciąg składający się dokładnie z dwóch liter lub cyfr (lub obu znaków), z jednego znaku, z dziesięciu znaków, z jedenastu znaków, aby pojawiła się tam jedna litera itp. W przypadku testów sprawdzających walidację danych należy wybrać przypadki testowe z danymi, które powinny być zaakceptowane i z takimi, które powinny być odrzucone.

Testy jednostkowe

Slajd nr 7 zawiera kod funkcji `is_valid()` napisanej w języku Python, która korzystając z wyrażenia regularnego sprawdza, czy przekazany jej jako argument ciąg znaków jest prawidłowym adresem e-mail (odpowiedź `True` lub `False`). Taka funkcja mogłaby być użyta np. w aplikacji, która stosowałaby adresy e-mail użytkowników jako ich loginy.
Z kolei slajd nr 8 zawiera klasę napisaną w tym samym języku, z wykorzystaniem modułu `unittest`, która implementuje dwa testy dla wspomnianej funkcji. Pierwszy (metoda `test_valid_email()`), sprawdza, czy akceptuje ona prawidłowe adresy, a drugi (metoda `test_sql_injection()`), sprawdza, czy odrzuci ona adres, który jest próbą przeprowadzenia ataku typu SQL Injection.

Testy jednostkowe

```
1  #!/usr/bin/env python3
2  import regex
3
4  def is_valid(email):
5      regular_expression =
       ↪  r"^[a-zA-Z0-9_+&*-]+(?:\.[a-zA-Z0-9_+&*-
       ↪  ]+)*@(?:[a-zA-Z0-9-]+\.)+[a-zA-Z]{2,}$"
6      result = regex.match(regular_expression,email)
7      if result!= None:
8          return True
9      else:
10         return False
11
12 if __name__ == "__main__":
13     email = "john.smith@example.com"
14     print(is_valid(email))
```

Testy jednostkowe

```
1  #!/usr/bin/env python3
2  import unittest
3  from email_validator import is_valid
4
5  class TestEmaliValidator(unittest.TestCase):
6
7      def test_valid_email(self):
8          self.assertTrue(is_valid("a.chrobot@tu.kielce.pl"))
9
10     def test_sql_injection(self):
11         self.assertFalse(is_valid("a.chrobot@tu.kielce.pl'--
           ↪  "))
12
13 if __name__ == "__main__":
14     unittest.main()
```

Testy jednostkowe

Przedstawiony kod jest zaledwie minimalnym przykładem. Prawdziwe testy powinny być bardziej rozbudowane. Powstaje pytanie, ile przypadków testowych należy opracować dla testów bezpieczeństwa? W przypadku metody testowania funkcjonalnego, tę liczbę można oszacować na podstawie tego, ile jest klas równoważności i jakie przypadki testowe da się z nich utworzyć. W przypadku testowania strukturalnego, które najlepiej jest połączyć z przeglądami kodu, liczbę przypadków testowych można wyznaczyć obliczając złożoność cyklometryczną kodu.
Warto odnotować, że w przypadku modułów o bardziej skomplikowanych funkcjach, np. takich które przetwarzają znaczniki HTML, należy do konstrukcji testów jednostkowych posłużyć się dodatkowymi narzędziami, jak np. parsery HTML.

Testy jednostkowe

W doborze przypadków testowych dla testów bezpieczeństwa można kierować się następującymi regułami:

1. testowanie bezpieczeństwa ma większy priorytet dla kodu, który bezpośrednio jest związany z zabezpieczeniami,
2. najważniejsze testy bezpieczeństwa powinny sprawdzać takie zdarzenia jak: odmowa dostępu, odrzucenie danych wejściowych, niepowodzenie operacji,
3. przypadki testowe bezpieczeństwa powinny weryfikować, czy każde z kluczowych działań przebiega poprawnie.

Testy rozmyte

Ręczne generowanie dużej ilości danych wejściowych (i co za tym idzie — dużej liczby przypadków testowych) nie jest efektywne. Zamiast tego można użyć techniki testowania rozmytego (ang. *fuzzing* lub *fuzz testing*), która polega na automatycznym generowaniu takich danych. Przykładowo, jeśli testujemy walidację danych, to możemy użyć generatora wartości pseudolosowych, aby dodawał do niezaufanych danych wejściowych znaki, które są niedozwolone. W ten sposób można szybko stworzyć dużą liczbę testów, których manualne opracowanie byłoby trudne i być może nawet nie wzięlibyśmy niektórych z tych przypadków testowych pod uwagę. Dzięki takim testom zwiększamy prawdopodobieństwo wykrycia luk zabezpieczeń w testowanym kodzie.

Testy regresji

Testy regresji są szczególnym przypadkiem testów, ponieważ tworzone są po zidentyfikowaniu podatności i pozwalają zapewnić, że nie będzie ona ponownie wprowadzona do bazy kodu. Zaleca się, aby były one tworzone *przed* opracowaniem poprawki, a jeśli nie jest to możliwe, to chociaż równolegle z jej opracowywaniem. Takie testy pozwalają także lepiej poznać naturę podatności. Powinny być tak skonstruowane, aby nie tylko obejmowały te dane wejściowe, które pozwoliły odkryć daną podatność, ale także podobne zestawy danych. Np. jeśli wykryto, że aplikacja jest podatna na atak „Billion Laughs", to warto również zastosować testy z użyciem plików XML zawierających zewnętrzne encje, aby sprawdzić, czy to oprogramowanie nie jest także podatne na wyciek danych.

Testy integracyjne

Poziom testów integracyjnych, to etap testowania w ramach którego sprawdza się *współpracę* opracowanych komponentów. Można na tym poziomie zastosować te same metody, co w przypadku testów jednostkowych, ale stosowane techniki powinny być już inne. Przykładowo, niektóre komponenty mogą być nawet całymi podsystemami, takimi jak *frontend* aplikacji webowej. W takim przypadku można skorzystać np. z takich narzędzi, jak opracowany przez organizację OWASP, ▸ ZAP, stosowany głównie w testach penetracyjnych, ale oferujący również automatyzację testów, integrację ze środowiskami CI/CD oraz testy rozmyte. Można także opracować własne narzędzie dla takich testów, posługując się np. biblioteką ▸ Selenium.

Testy wydajności a bezpieczeństwo

Jednym z najtrudniejszych do obrony są ataki typu DoS i DDoS. Można jednak użyć testów wydajności (obciążeniowych, przeciążeniowych), aby oszacować jakie obciążenie danymi wejściowymi może być groźne dla aplikacji i które jej komponenty są najbardziej wrażliwe na ataki tego typu. Dzięki tym testom można np. ograniczyć akceptowany rozmiar danych do bezpiecznego. Zwróćmy uwagę, że biblioteki testów jednostkowych, np. JUnit, zawierają rozwiązania, które pozwalają ograniczyć czas ▸ wykonywania testów, co może pomóc w identyfikacji wrażliwych na rozmiar danych wejściowych komponentów, już na etapie testów jednostkowych. W przypadku testów na poziomie systemowym pomocne są narzędzia do generowania obciążenia i pomiaru czasu odpowiedzi aplikacji lub ogólnego zużycia przez nią zasobów. Do nich zaliczane są zarówno rozwiązania typu open source, np. ▸ Gatling lub ▸ JMeter, jak i komercyjne np. dostarczane przez firmę ▸ DynaTrace. Warto zauważyć, że tego rodzaju testy mogą także wymagać specjalnego środowiska, np. z ograniczonymi zasobami.

Monitorowanie

Niestety, testy wydajności i wynikające z nich poprawki nie pozwalają całkowicie zabezpieczyć aplikacji przed wspomnianymi wcześniej typami ataków. W związku z tym konieczne jest zidentyfikowanie najważniejszych wskaźników wydajności (ang. *Key Performance Indicators*) i ciągłe ich monitorowanie. W przypadku znacznego ich pogorszenia należy sprawdzić, czy przyczyną nie jest atak DoS lub DDoS. Do monitorowania można użyć darmowych narzędzi, takich jak Grafana.

Dobre praktyki

Tworząc testy bezpieczeństwa warto skorzystać z następujących dobrych praktyk:

- zastosować podejście TDD (ang. *Test-Driven Development*), w który testy są opracowywane *przed* napisaniem kodu, który je przechodzi, lub czasem są pisane równolegle z nim,
- wykorzystać testy integracyjne do sprawdzenia, czy nie dochodzi do wycieku danych,
- jeśli aplikacja nie ma testów bezpieczeństwa, to należy je dodać - ich opracowanie jest dobrym sposobem na poznanie jej kodu i działania.

Literatura


Elie Saad i Rick Mitchell. *OWASP Testing Guide, Version 4.2.* 2020. URL: https://github.com/OWASP/wstg/releases/tag/v4.2.

Pytania

?

Koniec

Dziękuję Państwu za uwagę!