Wrocław, 9.11.2016r.

Komunikator z szyfrowaniem

Bezpieczeństwo Usług Sieciowych - laboratorium

Grupa: wtorek nieparzysty, 17^{05} - $19:^{30}$
Wykonał: Rafał Szopa, 196211
Prowadzący: mgr. Inż. Przemysław Świercz

Cel zadania

Celem laboratorium było stworzenie kompletnego komunikatora sieciowego, objmującego zarówno stronę kliencką, jak i serwerową. Jednym z głównych założeń programu była implementacja protokołu **Diffiego-Hellmana**, odpowiedzialnego za bezpieczne uzgadnianienie kluczy między klientem a serwerem. Dodatkowo, cała komunikacja sieciowa miała odbywać się przy wykorzystaniu formatu wymiany danych **JSON**. Wiadomość przed wysłaniem należało natomiast zakodować przy pomocy kodowania transportowego **Base64**. Nie było restrykcij obejmujących wybór technologii realizacji zadania.

Wiadomości mogą być szyfrowane przy użyciu szyfru cezara, OTP XOR lub nie być szyfrowane w ogóle.

Szczegóły dotyczące uruchomienia aplikacji znajdują się w repozytorium w pliku READ.me.

Implementacja

Wybór technologii

Założenia projektu nie obejmowały wyboru technologii. Do realizacji zadania użyto dwóch zupełnie innych narzędzi. Serwer został bowiem napisany przy pomocy języka **JavaScript** oraz środowiska **Node.js**. Aplikacja kliencka została natomiast stworzona w środowisku **C# .NET** przy wykorzystaniu API **Windows Forms**.

Serwer

Wybór środowiska Node.js do realizacji serwera komunikatora był podyktowany kilkoma względami. Przede wszystkim rosnąca popularność Node.js, a co za tym idzie chęć poznania oraz przetestowania tej technologii. Środowisko jest bowiem ostanimi czasy bardzo chcętnie wykorzystywane do implementacji rozwiązań serwerowych. Opiera się na nim m.in. amerkańska platforma usługowa VOD – Netflix. Node.js charakteryzuje się wysoką skalowalnością. Dodatkowym atutem jest także menadżer paketów npm, umożliwiający wykorzystanie szerokiej gamy gotowych modułów.

Moduły wykorzystane do budowy serwera to m.in. „color" – umożliwiający wypisywanie komunikatów w konsoli w różnych kolorach oraz „net", który zawiera zestaw narzędzi potrzebnych do postawienia serwera (w przypadku projektu – TCP).

Serwer jest uruchamiany z linii poleceń przy pomocy komendy node server.js `[adres IP] [numer portu]`. Podczas pracy, serwer wypisuje na konsolę komunikaty wziązane z aktualnie wykonywaną pracą, np. o połączeniu/rozłączeniu klienta czy zmianie przez klienta sposobu szyfrowania.

Klient

Aplikacja kliencka została napisana przy pomocy języka C# w środowisku .NET. Wykorzystanie API Windows Forms pozwoliło stworzyć graficzny interfejs użytkownika. Głównymi zaletami technologii są popularność (a co za tym idzie wiele artykułó, materiałów szkoleniowych i innego rodzaju pomocy), a także IDE dostarczone przez firmę Microsoft – Visual Studio 2015, wraz z potężnym narzędziem, jakim jest wbudowany Debuger.

Do obługi tzw. wielkich liczb skorzystano z biblioteki BigInteger znajdującej się w przestrzeni nazw System.Numerics. Wielkim atutem tego rozwiązania jest stabilność rozwiązania oraz dostarczenie gotowej metody umożliwiającej wykonanie operacji matematycznej potęgowania modulo, wykorzystywanego przy generowaniu klucza.

Aplikacja kliecka uruchamia dodatkowy wątek odpowiedzialny za nasłuchiwanie nadchodzących do aplikacji wiadomości oraz ich obsługę. Takie rozwiązanie chroni przed „zamrożeniem" UI, prowadzącym do spadku User Expirience aplikcaji.

Sposób działania komunikatora

Klient nawiązuje połączenie z serwerem, a następnie wysyłą wiadomość {„request" : „keys"}. Serwer po odebraniu żądania, dodaje klienta do tablicy użytkowników, zaznaczając jednocześnie, że proces wymiany kluczy nie jest zakończony (co za tym idzie, klient nie bierze

jeszcze udziału w komunikacji). Następnie serwer generuje liczby zgodnie z wymaganiami stawianymi przez protokół Diffiego-Hellmana (tzw. *p* oraz *g*) i odsyła je klientowi. Klient z kolei generuje na ich podstawie *A* i odsyła do serwera. Serwer odsyła *B* – w tym momencie obie strony generują klucze symetryczne. Dodatkowo, serwer zezwala klientowi na udział w komunikacji.

Wartości *p* oraz *q* potrzebne w procesie generowania kluczy są wartościami stałymi i wynoszącymi odpowiednio 23 oraz 5.

Aplikacja została zaprojektowana w ten sposób, aby klient mógł „w locie” zmienić rodzaj szyfrowania.

Wnioski

Środowisko Node.js doskonale radzi sobie w roli serwera sieciowego. Mimo, że rozwiązanie to działa na jednym wątku, zastosowany model asynchroniczności działa bardzo dobrze, nie prowadząc do zatorów związanych z kolejkowaniem zadań. Ponadtko, architektura *event-driven* jest bardzo wygodnia przy tego rodzaju zastosowaniu.

Siła kluczy wykorzystywanych do szyfrowania komunikacji opiera się na wygenerowaniu dostatecznie dużych wartości liczbowych *p* oraz *q*, czego zabrakło w przypadku aplikacji serwerowej. Możliwe było skorzystanie z gotowych modułów wspierających generowanie dużych liczb pierwszych wraz z ich generatorami.

Największe trudności napotkałem podczas debugowania kodu napisanego w JavaScript. „Odrobaczanie" aplikacji wyskonywałem bowiem ręcznie. Często komunikaty o błędach, przenoszone przez obiekty wyjątków, były bardzo ogólnikowe, co wydłużało czas lokalizacji błędów.