FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Zranitelnosti komunikačních Web API protokolů
Student:	Bc. Michal Kovačič
Vedoucí:	Ing. Jiří Šmolík
Studijní program:	Informatika
Obor / specializace:	Počítačová bezpečnost
Katedra:	Katedra informační bezpečnosti
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Kromě obecných typických zranitelností Web API, jako je injection, data exposure apod., odhalují nové API technologie také nové zranitelnosti specifické pro dané komunikační protokoly. Zmapujte zranitelnosti specifické pro GraphQL a gRPC a implementujte na vybrané podporované platformě software pro ochranu proti běžným zranitelnostem daného protokolu.

- Analyzujte a popište známé zranitelnosti pro GraphQL a gRPC API a následně srovnejte s REST API z hlediska bezpečnosti.
- Diskutujte možnosti snížení rizika pomocí automatizované kontroly.
- Implementujte knihovnu pro zmíněné API technologie ve vybraném jazyce, které pomůže vývojářům snížit rizika diskutovaných útoků.
- Otestujte implementaci funkčními testy, zdokumentujte a demonstrujte použití v jednoduché aplikaci (volbu proveďte po dohodě s vedoucím práce).

Elektronicky schválil/a prof. Ing. Róbert Lórencz, CSc. dne 9. února 2023 v Praze.

Diplomová práce

ZRANITELNOSTI KOMUNIKAČNÍCH WEB API PROTOKOLŮ

Bc. Michal Kovačič

Fakulta informačních technologií
Katedra počítačové bezpečnosti
Vedoucí: Ing. Jiří Šmolík
4. května 2023

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2023 Bc. Michal Kovačič. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení, je nezbytný souhlas autora.

Odkaz na tuto práci: Kovačič Michal. *Zranitelnosti komunikačních Web API protokolů*. Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Obsah

Poděkování viii

Prohlášení ix

Abstrakt x

Seznam zkratek xi

Úvod 1

1 Cíl práce 3

2 GraphQL, gRPC, REST API 5
2.1 GraphQL . . . 5
2.1.1 GraphQL v praxi . . . 6
2.2 gRPC . . . 8
2.2.1 gRPC v praxi . . . 8
2.3 REST API . . . 9
2.3.1 REST API standard podle OpenAPI . . . 10

3 Analýza známých zranitelností GraphQL 13
3.1 Absence autentizačních a autorizačních mechanizmů . . . 13
3.1.1 Popis a ohodnocení zranitelnosti . . . 13
3.1.2 Srovnání zranitelnosti v GraphQL a REST API . . . 14
3.1.3 Analýza automatizovaného snížení rizika zranitelnosti . . . 14
3.2 GraphQL CSRF . . . 15
3.2.1 Popis zranitelnosti . . . 15
3.2.2 Srovnání zranitelnosti v GraphQL a REST API . . . 16
3.2.3 Analýza automatizovaného snížení rizika zranitelnosti . . . 16
3.3 GraphQL errors information Disclosure . . . 17
3.3.1 Popis zranitelnosti . . . 17
3.3.2 Srovnání zranitelnosti v GraphQL a REST API . . . 18
3.3.3 Analýza automatizovaného snížení rizika zranitelnosti . . . 18
3.3.4 Dodatek: GraphQL introspection . . . 19
3.4 Zranitelnost vůči útokům DoS . . . 19
3.4.1 Popis zranitelnosti . . . 19
3.4.2 Srovnání zranitelnosti v GraphQL a REST API . . . 20
3.4.3 Analýza automatizovaného snížení rizika zranitelnosti . . . 20
3.5 Dodatek: SQL injection . . . 21

4 Analýza známých zranitelností gRPC 23
4.1 Komunikace nešifrovaným kanálem . . . 23
4.1.1 Popis zranitelnosti . . . 23
4.1.2 Srovnání zranitelnosti v gRPC a REST API . . . 23

4.1.3 Analýza automatizovaného snížení rizika zranitelnosti 24
4.2 Nezabezpečená definice souboru protobuf 25
4.2.1 Popis zranitelnosti 25
4.2.2 Srovnání zranitelnosti v gRPC a REST API 25
4.2.3 Analýza automatizovaného snížení rizika zranitelnosti 25
4.3 gRPC SQL injection 26
4.3.1 Popis zranitelnosti 26
4.3.2 Srovnání zranitelnosti v gRPC a REST API 27
4.3.3 Analýza automatizovaného snížení rizika zranitelnosti 27

5 Tvorba prostředí pro demonstraci a testování automatizovaných snížení rizik zranitelností GraphQL 31
5.1 Tvorba virtuálního prostředí 31
5.2 Popis demonstrační aplikace 32
5.3 Instalace, konfigurace a spuštění GraphQL serveru 32
5.4 Tvorba výsledné zranitelné aplikace a demonstrace zranitelností 34
5.4.1 Metoda demonstrující absenci autentizačních a autorizačních mechanismů 34
5.4.2 GraphQL CSRF 35
5.4.3 Errors information disclosure 36
5.4.4 Zranitelnosti vůči útokům DoS 36

6 Tvorba prostředí pro demonstraci a testování automatizovaných snížení rizik zranitelností gRPC 39
6.1 Tvorba virtuálního prostředí 39
6.2 Popis demonstrační aplikace 39
6.3 Instalace, konfigurace a spuštění gRPC serveru 40
6.4 Tvorba výsledné zranitelné aplikace a demonstrace zranitelností 41
6.4.1 Metoda demonstrující komunikaci nezabezpečeným kanálem 41
6.4.2 Metoda demonstrující information disclosure pomocí souboru protobuf . . 42
6.4.3 Metoda demonstrující SQL injection 43

7 Tvorba knihovny pro automatizované opravy zranitelnosti GraphQL 45
7.1 Autentizace 46
7.1.1 Testování metody 47
7.2 Autorizace 48
7.2.1 Testování metody 48
7.3 Metoda pro automatizované snížení rizika GraphQL CSRF 49
7.3.1 Testování metody 49
7.4 Metoda pro automatizované snížení rizika errors information disclosure 50
7.4.1 Testování metody 50
7.5 Metoda pro automatizované snížení rizika DoS 51
7.5.1 Testování metody 52

8 Tvorba knihovny pro automatizované opravy zranitelností gRPC 53
8.1 Metoda pro snížení rizika komunikace nešifrovaným kanálem 53
8.1.1 Testování metody 54
8.2 Metoda pro snížení rizika protobuf information disclosure 56
8.2.1 Testování metody 56
8.3 Metoda pro snížení rizika SQL injection 57
8.3.1 Testování metody 58

9 Závěr 61

A Příloha A - skript tvorby lokální public key infrastructure **65**

Obsah přiloženého média **69**

Seznam obrázků

2.1 GraphQL vs REST API entry point 6

3.1 SQL injection v GraphQL 22

4.1 Příklad nešifrované komunikace pomocí gRPC[22] 24

5.1 GraphQL hello world 33
5.2 Počátek komunikace klienta s GraphQL serverem – paket prvního dotazu 34
5.3 Výsledek nasimulovaného CSRF 35
5.4 Chybové hlášení GraphQL 36
5.5 GraphQL server po dotazovém DoS 37
5.6 Graf závislosti rychlosti odpovědi na velikosti dotazu 38

6.1 gRPC Hello World 41
6.2 Zranitelnost komunikace nezašifrovaným kanálem 42
6.3 Zranitelnost protobuf file information disclosure 42
6.4 gRPC SQL injection 44

7.1 Autentizace v GraphQL od začátku 48
7.2 Autorizace funkcí s graphql-shield pro neautentizované a tedy neautorizované . . 49
7.3 Výsledek testování CSRF 50
7.4 Ukázka přetvářky chyb v GraphQL 51
7.5 Odmítnutí nezmapované persistent queries 52
7.6 Měření odezvy serveru na batching attack 52
7.7 Měření odezvy serveru na rekurzivní DoS 52

8.1 Šifrovaná komunikace se serverem gRPC 55
8.2 gRPC odmítnutí nešifrované komunikace 55
8.3 Metoda detekce nad zranitelným thesis.proto 57
8.4 Příklad využití funkcí pro předpřipravené dotazy do MySQL 59

Seznam výpisů kódu

3.1 Příklad deklarace chybových hlášení 18
3.2 GraphQL SQL injection example 22
4.1 Pseudokód vyhodnocování zpráv 25
4.2 Pseudokód zjednoduššení vyhodnocování zpráv 26
4.3 gRPC SQL injection example 26
4.4 Předpřipravené dotazy s parametry v gRPC 27

4.5 SQL query whitelisting 28
5.1 Simulace spuštění úspěšného phishing útoku za cílem CSRF 35
5.2 Simulace útoku rekurzivního DoS 36
5.3 Simulace dotazového DoS 38
6.1 Kód metody demonstrující komunikaci nezabezpečeným kanálem 41
6.2 Příklad automatizace naplňování zpráv v gRPC 43
6.3 gRPC SQL injection v praxi 43
7.1 Ukázka spojování schémat 45
7.2 Ukázka ručního rozšíření kontextové funkce v GraphQL 46
7.3 Facebooková metoda pro minimalizování zranitelnosti Graphql DoS 51
8.1 Ukázka využití TLS/SSL v gRPC 54

Chtěl bych poděkovat především svému vedoucímu, Ing. Jiřímu Šmolíkovi za jeho cenné rady a velkou dávku trpělivosti. Mé díky také patří všem, kteří mě v práci podporovali, jmenovitě mé rodině a přítelkyni. Rád bych také zmínil všechny, které jsem při tvorbě práce zanedbával, díky patří i vám.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 4. května 2023 .....

Abstrakt

Tato práce se zabývá analýzou vybraných zranitelností komunikačních web API protokolů GraphQL a gRPC. Začíná úvodem do obou zmíněných protokolů, následně pokračuje výčtem jejich bezpečnostních zranitelností společně s jejich analýzou a detailem. Stěžejní část této práce jsou metodiky pro snížení rizika diskutovaných zranitelností pomocí automatizované kontroly a implementace těchto metodik pro praktické použití. Tato část slouží jako výsledek práce.

Klíčová slova web API, zranitelnosti, GraphQL, gRPC, počítačová bezpečnost, bezpečnostní rozšíření

Abstract

This thesis is dedicated to analysis of chosen vulnerabilities contained in communication web API protocols GraphQL and gRPC. It begins with introduction to both mentioned protocols and continues with enumeration of their security vulnerabilities together with their analysis and detail. The crucial part of the thesis are methodologies for decreasing the threat of mentioned vulnerabilities by automatic supervision and implementation of the methodologies for practical use. This part is considered to be the outcome of the thesis.

Keywords web API, vulnerabilities, GraphQL, gRPC, cybersecurity, security extension

Seznam zkratek

gRPC	Google Remote Procedure Call
CSRF	Cross-Site Request Forgery
HTTP(S)	Hyper Text Transfer Protocol (Secure)
DOS	Denial of Service
API	Aplication Programming Interface
URL	Uniform Resource Locator
JSON	JavaScript Object Notation
TLS	Transport Layer Security
SSL	Secure Sockets Layer
ALTS	Application Layer Transport Security
GCP	Google Cloud Platform
PKI	Public Key Infrastructure
DAC	Discretionary Access Control
MAC	Mandatory Access Control
ACL	Access Control Lists
CVSS	Common Vulnerability Scoring System
NIST	National Institute of Standards and Technology
IDOR	Insecure Direct Object Reference
CORS	Cross Origin Resource Sharing
MiM	Man in the Middle
DDoS	Distributed Denial of Service

Úvod

Webová API představují možnosti rozšíření funkcionalit webových serverů, popřípadně webových prohlížečů. Tyto API vývojářům slouží k efektivnější a snažší tvorbě webových aplikací. Webových API existuje nespočetně mnoho druhů, např. API pro bluetooth, API pro animace...

Jedním z nejdůležitějších webových API jsou tzv. „komunikační webová API“, která slouží k tvorbě a správě metod komunikace mezi klientem a webovým serverem, popřípadně zjednoduššení stávajících komunikačních metod. Mezi největší výhody v používání komunikačních webových API pro vývojáře je zpracování uživatelského vstupu a rošíření možností pro uživatele vstup zadat bez nutnosti vývoje od úplného začátku. Uživatel zase ocení jednoznačné přijimání vstupu podobné s jinými aplikacemi používající stejné API.

Spolu s používáním jakýchkoliv komunikačních API přichází z pozice bezpečnosti dílčí problémy, které je potřeba řešit. Mezi tyto problémy se řadí obecné typy problémů, např. validace uživatelského vstupu, data exposure... a problémy specifické pro konkrétní API, které mohou mít stejně katastrofální následky. Jak katastrofální si mohla ověřit společnost USPS v roce 2018, kdy útočník využil autentizační slabinu v API sloužící pro přístup interní databáze společnosti. V této databázi bylo uloženo přes 60 milionů záznamů týkajících se zásilek ve světovém měřítku[1]. API pro jakoukoliv komunikaci, obzvlášt pro komunikaci přes Internet, musí být zvláště zabezpečena, jelikož představují největší vektor útoku na aplikaci.

Bezpečnost komunikačních API je často založena na prostředí, ve kterém je API nasazeno. Proto vývojáři různých API nechávájí některé bezpečnostní faktory na vývojářích aplikace používající dané API. Vývojář aplikace často musí např. zařídit dodržování principu nejmenších možných oprávnění. Touto filozofií se řídili i vývojáří webových komunikačních API GraphQL a gRPC, které jsou předmětem této práce. Pokud aplikace používající tato API nejsou správně nakonfigurované, mohou se daná API stát důvodem úspěchu útočníka. V rámci této práce navážu na vývojáře těchto 2 konkrétních API a vytvořím middleware, který zjednoduší zabezpečení aplikací využívající tyto API.

[1] https://nordicapis.com/5-major-modern-api-data-breaches-and-what-we-can-learn-from-them/

Kapitola 1
Cíl práce

Tato práce se zabývá zjednodušením zabezpečení aplikací využívající technologií GraphQL a gRPC. Dosahuje toho pomocí tvorby middleware, který poskytuje otestované možnosti pro minimalizování rizik známých zranitelností těchto API. Hlavními částmi této práce jsou:

- Analýza běžných zranitelností obou protokolů
- Analýza automatizovaného snížení rizika zranitelností obou protokolů z prvního bodu
- Demonstrace zranitelností vyšlých z analýzy na jednoduché aplikaci simulující reálné prostředí
- Tvorba middleware pro opravu zranitelností pomocí metod diskutovaných v druhém bodu
- Otestování tohoto middleware na aplikacích ze třetího bodu

Osobně bych rád posunul použitelnost moderních komunikačních API. Myslím si, že GraphQL i gRPC mají velký potenciál, který je omezený rizikem plynoucího z tvorby vlastního zabezpečení a rád bych pomohl tento problém odstranit.

Kapitola 2

GraphQL, gRPC, REST API

Tato kapitola se věnuje teoretickým základům technologií GraphQL a gRPC. Jejím cílem je shrnout teoretické znalosti nutné k pochopení a následnou analýzu nejznámějších zranitelností těchto technologií. V sekci popisující REST je také popsán standard OpenAPI, jenž je později v práci používán k bezpečnostnímu srovnání z hlediska daných zranitelností vůči GraphQL a gRPC.

2.1 GraphQL

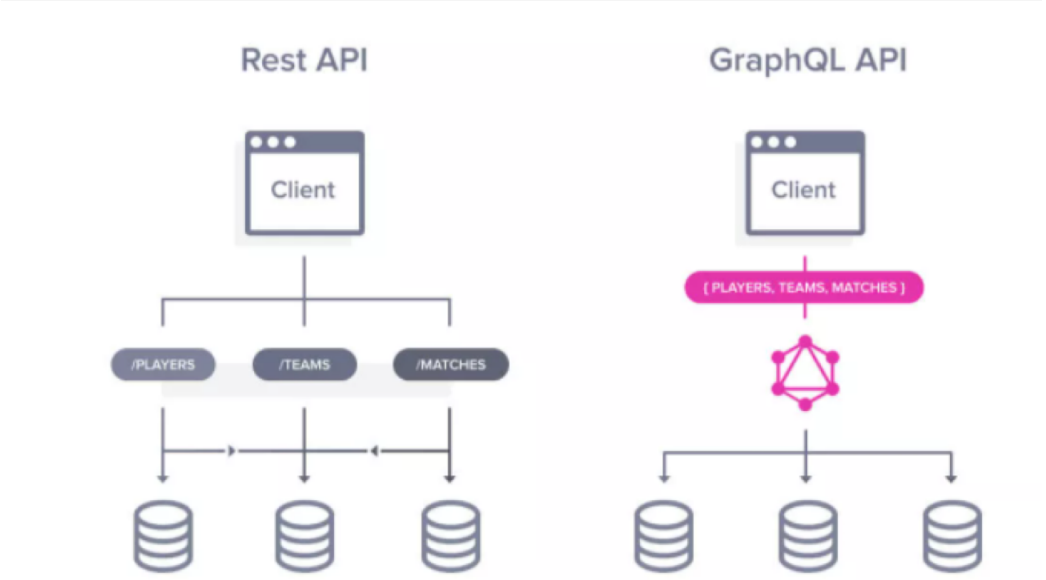
GraphQL je technologie sloužící ke vzdálené komunikaci zveřejněná roku 2015[1]). Skládá se ze dvou primárních částí, které jsou nezbytné pro pochopení její funkčnosti.

První z těchto částí je jazyk uzpůsobený k pokládání dotazů a jejich řešení (tzv. „query language“, česky „dotazovací jazyk“) nazývaný „GraphQL query language“. Pomocí tohoto jazyka si administrátor aplikace využívající GraphQL může velice přesně nadefinovat, jaké požadavky mohou uživatelé dané aplikace pokládat. Je pečlivě zdokumentován spolu s oficiálním know-how[2]. Mimo to také existují řady veřejně dostupných neoficiálních zdrojů, ze kterých lze čerpat vědomosti týkající se GraphQL query language (např. články, příspěvky na diskuzních fórech[1]...).

Druhá část GraphQL je služba na straně serveru, která se stará o přijímání a zpracovávání dotazů. Typicky je tato služba nasazena na serveru dostupného z internetu (např. skrz URL) a slouží jakožto vstupní bod do systému. Pomocí této služby GraphQL vytváří právě jeden vstupní bod do celé aplikace, díky čemuž uživatelé nemusí načítat data z několika URL (jako např.
u REST API), ale dostanou svá data najednou pomocí jednoho požadavku, viz. obrázek 2.1.

GraphQL slibuje vysokou propustnost (i přes využívání komunikace pomocí nekompaktního datového formátu JSON a to díky komprimaci dat), stabilitu (např. predikovatelné výsledky), ověřování splnitelnosti dotazů a mnoho dalších vlastností[1]. Jeho obrovskou výhodou je také univerzální portabilita, díky čemuž je využívaný v korporátní praxi. Díky této vlastnosti je žádané, aby nasazení tohoto protokolu bylo korektně zabezpečené. Pro nasazení GraphQL do reálného prostředí existují knihovny v mnoha jazycích (jejich konkrétní seznam je k dispozici online[2]). GraphQL je kompletně open-source, veškeré jeho úpravy jsou povoleny.

[1] Například na fóru Stack Overflow: `https://stackoverflow.com/questions/tagged/graphql`
[2] `https://graphql.org/code/`



Obrázek 2.1 GraphQL vs REST API entry point

2.1.1 GraphQL v praxi

V této sekci je popsáno, jak v praxi z pohledu kódu vypadá server a jakým způsobem mohou se serverem komunikovat klienti. Informace v této sekci jsou k dispozici v officiální dokumentaci GraphQL[2]. Vědomosti popsané v této sekci jsou stejné, jako v dané dokumentaci, ale popsány způsobem mně bližším.

Server se skládá ze dvou primárních částí - schématu a resolverů:

- Za schéma se považuje definice netriviálních typů (struktur), query a mutací. Query jsou klientské dotazy, díky kterým uživatelé získavají data z GraphQL. Jedná se o ekvivalent operace HTTP GET[3] pro aplikace využívající GraphQL. Mutace jsou klientské dotazy, které jakkoliv mění data na straně serveru. Jakýkoliv dotaz, jehož cílem je víc, než získání informací je považován za mutaci.

  Pro ilustraci zde uvádím příklad jednoduchého schématu s komentářem. Schéma je uloženo v textovém řetězci v alespoň jedné proměnné. Je možné složit více schémat dohromady, pokud např. neobsahují stejně pojmenovanou položku:

```
const typeDefs = gql‘

    #user data type
    #contains 3 non-nullable fields of type String
    #and one nullable field of type String
    type User{
        username: String!
        password: String!
        email: String!
        city: String
    }
```

[3] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET

```
    #list of possible queries
    #in form: name(params: paramsType): returnValue
    #can return primitive data type, or custom one
    #[ ] is an array sign
    type Query{
        getUsers(): [User]
        getOneUser(id: Int!): User
        sayHello(): String
    }

    #list of possible mutations
    #in the same form as queries are
    type Mutation{
        addUser(username: String!, password: String): String
    }
`;
```

- Resolver je obecné pojmenování funkce, jenž se zavolá v reakci na klientský dotaz. Pro výše uvedený příklad by definice resolverů mohla vypadat následovně:

```
const Resolvers = {
    type Query:{
        getUsers(parent, args, ctx, info){
            return Users;
        }
        getOneUser(parent, args, ctx, info){
            const ret = Users.find((u) => u.id === args.id);
            return ret;
        }
        sayHello(parent, args, ctx, info){
            return "Hello!";
        }
    }
    type Mutation:{
        addUser(_, {username, password}){
            const tmp = Users.find(
                        (u) => u.username === args.username);
            if(tmp) {return "User already exists";}

            const user = {
                username: username, password: password,
                email: random@gmail.com, city: "CTU"
            }
            Users.add(user);    //this is why its a mutation
            return "Success"
        }
    }

}
```

Tyto dva parametry (schémata a resolvery) jsou esenciální k zavolání konstruktoru GraphQL serveru. Mimo to je užitečné serveru přidat kontextovou funkci, což je funkce, která proběhně před vyřízením všech dotazů. Návratová hodnota kontextové funkce je po jejím skončení předána do všech dalších funkcí v kontextovém argumentu. Ve výše uvedených resolverech je tento argument vidět pod jménem „ctx“. Server také samozřejmě dovoluje používat i funkce ve standardní JavaScriptové formě.

Klient může se serverem komunikovat pomocí dat ve formě JSON. Jeho hlavní metodou jsou výše zmíněné query a mutace. Serveru odesílá dotazy ve tvaru:

```
#query:              ...yes, queries can have comments!
query{
    getOneUser(id: 1){  #name of the query(param_i: param_iValue)
        username
    }
}

#mutation
mutation {
    addUser(username: "MiKo", password: "1234", email: "rand@gmail.com"){
        #possibly return value fields - none in our example}}
```

Tyto dotazy může odeslat mnoha nástroji. Nejvýraznějším tímto nástrojem je GraphQL sandbox, webové UI poskytované GraphQL serverem určené k odesílání dotazů. Pro představu je sandbox ilustrovaný na obrázku TODO. Mezi další nástroje patří konzolové nástroje např. CURL, nebo klientský program napsaný v GraphQL query language. Každý z těchto nástrojů má své vlastní využití, jak je možno vidět v kapitole 5.

2.2 gRPC

gRPC (Google remote procedure call), druhá analyzovaná metoda vzdálené komunikace této práce dosahuje své funkcionality pomocí volání funkcí na vzdáleném počítači. Teoreticky je tato metoda podstatně jednodušší, než GraphQL – funguje na základě knihoven pro podporované jazyky[4], jenž se starají o to, aby jeden počítač mohl odeslat informaci o úmyslu spustit kód na druhém počítači.

gRPC slibuje podporu load-balancingu, autentizace, heart-beat protokolu a mnoho dalších vlastností[3]. Stejně jako GraphQL je universálně portabilní a využívaný v korporátní praxi, což z něho stejně tak činí objekt nutný zabezpečit. Je open-source, tedy jakékoliv úpravy kódu jsou povoleny. Veškeré zdrojové kódy společně s dokumentací jsou k dispozici online v oficiální dokumentaci[5].

2.2.1 gRPC v praxi

V této sekci je popsáno, jak v praxi z pohledu kódu vypadá server a jakým způsobem mohou se serverem komunikovat klienti. Informace v této sekci jsou k dispozici v oficiální dokumentaci gRPC[4]. Vědomosti popsané v této sekci jsou stejné, jako v dané dokumentaci, ale popsány způsobem mně bližším.

Základ gRPC je soubor protobuf (.proto), ve kterém jsou definované služby serveru a forma všech zpráv. Službou je v praxi míněna funkce nebo metoda na straně serveru, kterou uživatel může spustit. Soubor protobuf může vypadat například následovně:

[4]Konkrétní seznam podporovaných jazyků je k dispozici zde: `https://grpc.io/docs/languages/`
[5]`https://grpc.io/docs/`

```
syntax = 'proto3'   #syntax of everything below

package pkg;        #namespace equivalent

service srv {       #name of a server service

    #function foo, accepts a message, returns a reply
    rpc foo(msg) returns (reply) {}

    #function bar, accepts messages until stop
    #returns messages until stop
    rpc bar(stream msg) returns (stream reply) {}
}


message msg { #message named msg
    string text = 1;    #first part of the message is a string
    Int32 flag = 2;     #second part is an Int
}

message reply {   #message named reply
    string text = 1;    #only one part of a type String
}
```

Úkol serveru je mít definované všechny metody popsané v protobuf serveru. Tyto metody zapouzdří do třídy, která dědí z package pojmenované uvnitř protobuf serveru. Jednotlivé metody „redefinují“ metody ze souboru protobuf, je tedy nutné označit je klíčovým slovem *override*. Tyto funkce mají určený tvar:

```
#gRPC function in a cpp code
#return grpc::Status - struct with return codes, error values...
#accepts: const server context with background info
  const message* as a struct containing everything defined in protofile
  reply* as a struct containing everything defined in protofile

  Status foo(grpc::ServerContext ctx,
             const msg* message,
             reply* reply) {...}
```

Na straně klienta probíhá volání funkce deklarací a definicí zprávy, odpovědi a kontextu, které posléze naplní chtěnými údaji a poté zavolá funkci serveru téměř jako by byla lokální, s tím rozdílem, že jí volá pomocí ukazatele na spojení. Do dalšího detailu v této sekci nebudu zacházet, ale kompletní využívaní gRPC je k vidění v mnoha kapitolách této práce.

2.3 REST API

REST API je obecný název pro API vzdálené komunikace, které splňuje požadavky architektonického style REST. Jedná se o široce rozšířenou možnost komunikace
na bázi protokolu HTTP, využívaná např. společnostmi Twitter, Instagram, nebo spotify[5]. Využívání aplikací na bázi REST je ověřený postup používaný od roku 2003. Dobře vytvořené REST API umožňuje rozšiřitelnost a flexibilitu na straně serveru zároveň s nezávislostí na využitých technologiích.

Požadavky architektonického stylu REST jsou následující:[6]

- **Jednotný přenos informace**
  Mezi jednotlivými komponentami komunikace jsou informace přenášeny jednotným způsobem
- **Bezestavovost**
  Veškeré dotazy jsou na sobě navzájem nezávislé
- **Hierarchické uspořádání systému na straně serveru**
  Strana serveru je virtuálně rozdělena na hierarchický systém, jenž je pro klienta neviditelný. Jednotlivé klientské požadavky se tedy zpracovávají postupně podle hierarchie systému, s nímž klient komunikuje
- **Kód na vyžádání**
  REST API musí být schopné přenést část kódu na straně klienta. To může být užitečné např. k vyplňování formulářů
- Volitelné: **Možnost ukládání serverových informací na straně klienta**
  Tento mechanismus cílí na zefektivnění komunikace. Klient nemusí žádat server o data, která má uložená ve své paměti a tím ušetří požadavek. Příkladem tohoto mechanizmu mohou být např. známá cookies, které mají na starost právě tuto funkcionalitu

Vzhledem k rozšířenosti komunikace za pomocí REST API je nutné, aby jakýkoliv jiný způsob vzdálené komunikace byl schopný REST API konkurovat. Z tohoto důvodu je součástí této práce bezpečnostní srovnání GraphQL a gRPC s REST API. Analýza bezpečnostních rozdílů mezi těmito komunikačními prostředky odhalí, v čem je GraphQL, resp. gRPC lepší, než REST API a v čem se naopak potřebuje zlepšit. Vzhledem k tomu, že REST API je pouze obecný název pro API s výše zmíněnými omezeními, je nutné si určit nějaký standard, který bude k bezpečnostnímu srovnání použit.

2.3.1 REST API standard podle OpenAPI

REST sám o sobě není standard, nedefinuje přísná pravidla, nebo formát pro tvorbu architektury vzdálené komunikace. Jeho povaha tedy neumožňuje vytvořit žádný specifický standard. Nejblíže, co se standardu blíží jsou sady zdokumentovaných ověřených postupů z různých zdrojů. Mezi tyto dokumenty patří například směrnice od Microsoftu[6], nebo směrnice komunity Stack Overflow[7]. Ve všech těchto směrnicích (výše zmíněných i jiných) jsou popsány velice podobné principy (např. návratové kódy HTML, využívání formátu JSON...), které se defacto dají za „standard“ považovat. Abych ve tvorbě referenčního standardu byl co nejpřesnější, využiji k jeho tvorbě poučení z výše uvedených zdrojů obohacených o poznatky ze specifikace OpenAPI.

Specifikace OpenAPI, sada nástrojů a programovací jazyk, je určen k popisu API. Byla vytvořena společností Swagger (po původním jménu společnosti je dnes označována zmíněná sada nástrojů) a kompletně vydaná roku 2011. Díky schopnosti specifikace popsat REST API umožňuje např:[7]

- Klientům porozumět struktuře REST API bez nutnosti zkoumat zdrojový kód
- Generovat knihovny pro klienty
- Generovat dokumentaci API

[6] `https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design`
[7] `https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/`

První pohled, který nijak neodhalí užitek OpenAPI pro vytvoření standardu pro OpenAPI se mýlí. Na svých internetových stránkách[8] uvádí autor mnoho rad pro tvorbu REST API naznačuje, co by REST API mělo umět tím, že dokumentuje, co vše umí OpenAPI popsat. Dohromady s výše zmíněnými směrnicemi pro tvorbu REST API poskytuje specifikace OpenAPI dostatek informací k tvorbě standardu pro účely této práce. Při bezpečnostních srovnáních v této práci se budu ukazovat prioritně na specifikaci OpenAPI a poté, pokud v této specifikaci nebudou k problematice uvedeny žádné informace, se budu ukazovat na výše uvedené směrnice.

[8] `https://swagger.io/specification/`

Kapitola 3

Analýza známých zranitelností GraphQL

V této kapitole je obsažen seznam a popis nejznámějších zranitelností GraphQL. U každé zranitelnosti je uvedeno její ohodnocení pomocí systému CVSS v3.1 (industriální standard pro ohodnocení vážnosti zranitelností, hojně využívaný na FIT ČVUT), pro jehož výsledek jsem využil online kalkulátor zveřejněný agenturou NIST[a] (americká agentura zabývající se standardizací v oboru informačních technologií). Také se u každé zranitelnosti nachází její zasazení do bezpečnostního kontextu REST API a na konec analýza možností snížení jejího rizika.

[a] `https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator`

3.1 Absence autentizačních a autorizačních mechanizmů

3.1.1 Popis a ohodnocení zranitelnosti

GraphQL oficiálně spoléhá na programátory, aby své aplikaci poskytly své vlastní autentizační a autorizační mechanizmy[1]. Z tohoto faktu vyplývá, že kvalita implementace autentizace i autorizace v aplikaci přímo závisí na tvůrci, popřípadě správci aplikace. Pokud tento člověk nemá dostatečné bezpečnostní vzdělání, nebo jednoduše udělá chybu, otevře aplikaci útokům typu „insecure direct object reference“ (IDOR). útočník může pomocí těchto útoků snadno přistoupit k datům, jenž by měly být mimo hranice jeho působnosti.

Fakt, že jsem při své rešerši jsem nenašel žádné externí rozšíření, jenž by sloužilo k autentizaci nebo autorizaci, činí zranitelnost obzvlášť zákeřnou, jelikož veškeré autentizační i autorizační mechanizmy jsou potenciálně vyrobené amatéry. Na internetu existuje pouze řada návodů k tomuto problému, ale jsou bohužel často nekompletní, nebo obsahují suboptimální praktiky[2].

Při ohodnocení zranitelnosti jsem předpokládal nejhorší možnou alternativu: autentizace ani autorizace není implementována vůbec a útočník má právo použít metodu POST. V takovém případě je zranitelnost ohodnocena 10 body, jedná se tedy o katastrofální zranitelnost[3].

[1] Oficiálně proto, že tento fakt je zmíněný v oficiální dokumentaci: `https://graphql.org/learn/authorization/`

[2] Např. ve článku `https://daily.dev/blog/authentication-and-authorization-in-graphql` autor nejenže používá postupy z nekompatibilních verzí knihoven, navíc jeho návratové chybové kódy jsou příliš informující

[3] `https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator?vector=AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:N&`

3.1.2 Srovnání zranitelnosti v GraphQL a REST API

REST API stejně jako GraphQL funguje bez nutnosti autentizace. Narozdíl od GraphQL má ale standardní REST API2.3.1 podstatně větší podporu implementace autentizačních i autorizačních mechanismů. OpenAPI nabízí popis těchto mechanismů jako jednu ze svých základních funkcionalit[7].

Pro implementaci autentizace a autorizace v GraphQL existují pouze již zmiňované doporučené postupy včetně jednoho minimalistického zveřejněného v oficiální dokumentaci[4]. Nejedná se tedy o zcela ignorovaný problém, ale jak bývá u novějších technologií zvykem, není dostatečně prozkoumaný. Neexistuje pro něj mnoho řešení a ty, která existují, nejsou tak ověřená, jako řešení tohoto problému v REST API.

Jak GraphQL, tak REST API tuto zranitelnosti obsahují – obě technologie delegují problém autentizace i autorizace na správce systému. Avšak z praktického hlediska je nutné dát výhodu REST API, u něhož jsou autentizace i autorizace považovány za samozřejmost.

3.1.3 Analýza automatizovaného snížení rizika zranitelnosti

3.1.3.1 Analýza rozšíření GraphQL o autentizaci

Z hlediska autentizace je GraphQL webová aplikace, jako každá jiná. Pro autentizaci existují tři primární možnosti autentizace – pomocí „HTTP basic authentication“, cookies, nebo tokenu[8]. V této práci se budu soustředit primárně na autentizaci pomocí tokenu, protože zapadá do práce nejlépe. Narozdíl od cookies nepředpokládají stavový systém, popřípadě nějaké nasimulování systémových stavů a narozdíl od HTTP basic authentication nenutí uživatele zadávat heslo při každém požadavku. Později v práci, pkonkrétně v sekci 3.2.2, se ukáže další výhoda této metody.

Token slouží jako nezvratný důkaz o identitě uživatele. Nutně musí obsahovat uživatelské údaje autentizujícího a jejich digitální podpis pro prevenci falšování dat. V kostce je to vlastně všechno. Implementace takového tokenu není potřeba, jelikož existuje řada implementací (např. „Branca“[5], Microsoftí TGT token využívaný v jejich proprietárním protokolu Kerberos[6]...), které jsou navíc ověřeně funkční bez známých bezpečnostních chyb. Proto využiji některou z nich. Jedna z implementací vyčnívá převážně díky své portabilitě – JWT. [9]

JWT (JSON web token) je standardizovaný způsob formy dat v komunikaci bezpečnostního tokenu mezi dvěma entitami. Tento token je velmi populární pro autentizaci a další indikování identity pro následující zprávy. Podle OWASPu správné užívání tohoto tokenu *zabrání útočníkovi pozměnit identitu, nebo jiné charakteristiky tokenu*[10], což v kombinaci s jeho popularitou (tedy ověřením funkčnosti komunitou) činí ověřený a bezpečný způsob autentizace. Pro autentizaci v GraphQL proto využiji právě JWT.

Tento token obsahuje údaje ve formě kompaktního JSON, rozdělené na 3 části: [10]

1. **Hlavička**
   Obsahuje typ tokenu (JWT) a algoritmus využitý k podpisu
2. **Payload**
   Obsahuje data
3. **Podpis**
   Jak je patrné z názvu, obsahuje podpis

Poslední položka – podpis – brání modifikacím tokenu třetí stranou, čímž zajišťuje integritu indikování identity a činí jej vhodný pro účely autentizace.

Pro automatizované snížení rizika absence zranitelnosti tedy bude můj cíl vytvořit sadu funkcí pro bezpečnou autentizaci za pomocí JWT.

[4] `https://graphql.org/learn/authorization/`
[5] `https://branca.io/`
[6] `https://learn.microsoft.com/en-us/windows/win32/secauthn/ticket-granting-tickets`

3.1.3.2 Analýza rozšíření GraphQL o autorizaci

Autorizace v GraphQL se skládá ze dvou podproblémů. Uživatel by měl mít (či nemít) práva k provádění dotazů ve formátu query i mutace, (pojmy vysvětleny v sekci 2.1), i práva přístupu ke konkrétním prvkům.

K přidělování práv na dotazy řešení existuje – knihovna Graphql-shield[7]. Tato knihovna umožňuje vytvářet pravidla a namapovat tyto pravidla k funkcím. Pokaždé, když se funkce zavolá, před jejím provedením je ověřena platnost adekvátního pravidla (popř. sady pravidel). Pokud ověření selže, funkce se nezačne provádět.

Vzhledem k dostupnosti a funkčnosti graphql-shield je využití této knihovny dobré řešení problému autorizace provádění funkcí. Jediná chyba tohoto řešení je fakt, že při využívání pouze této knihovny je zajištění autorizace pro všechny funkce zdlouhavé, jelikož ke všem z nich je nutné napsat nějaké pravidlo. Tomuto problému částečně pomůžu v části autorizace přístupu k datům.

Pro přidělení práv položkám dat neexistuje žádné řešení. Vzhledem k počtu možností zdrojů dat[8] se není čemu divit.

Pro přidělení práv k prvkům existují následující dominantní přístupy[11]:

1. **DAC/MAC Access Control Lists** – přidělení seznamu práv jednotlivým objektům, podle něhož se pozná, kdo se vůči danému objektu může autorizovat
2. **RBAC** – oprávnění jsou přidělá každému na základě jeho role

Přístup využívající rolí je přizpůsobivější vůči výše zmíněnému graphql-shield, tedy proti si zvolím tento přístup. Ke snížení rizika absenci autorizace tedy implementuji rozšíření v podobě uživatelských rolí. Objekty rozšířím o položku oprávnění, které budou určovat, kdo má k objektu přístup. Také ukážu, jak lze pomocí rolí zjednodušit řízení přístupu k funkcím pomocí knihovny graphql-shield.

3.2 GraphQL CSRF

3.2.1 Popis zranitelnosti

Zranitelnost CSRF se netýká pouze GraphQL, jedná se o obecnější zranitelnost všech aplikací na bázi protokolu HTTP. K ilustraci této zranitelnosti je nejlepší popsat útok CSRF a poté si definovat aplikaci s touto zranitelností jako aplikaci, na kterou je možno zaútočit útokem CSRF.

Útok CSRF je útok, při kterém útočník donutí uživatele, aby provedl nějaký (škodlivý) dotaz vůči aplikaci, ke které je autentizován[12]. To lze například pomocí techniky phishing. Uživatel si takto může nevědomky změnit heslo, popřípadě provést administrativní úpravu v aplikaci a poskytnout útočníkovi např. zadní vrátka.

GraphQL je speciální případ v tom, že ve svém základním nastavení neprovádí žádnou autentizaci. Jelikož popsaný útok autentizaci vyžaduje, tak z definice GraphQL ve svém základním nastavení zranitelné není. Je ale zranitelné např. v následujících speciálních příkladech:

- Útočník donutí položit dotaz uživatele, který se nachází v lokální síti, odkud je server přístupný
- Na serveru je logována IP adresa uživatelů a jejich dotazů. Útočník tak pomocí CSRF provede dotaz a svede jej na odlišného uživatele
- V dotazech jsou obsažena data z lokálního úložiště uživatele

[7] `https://the-guild.dev/graphql/shield`

[8] Např. Apollo GraphQL server podporuje všechny tyto datové kanály: `https://www.apollographql.com/docs/apollo-server/v2/data/data-sources/`

Ačkoliv se tedy nejedná o typickou zranitelnost vůči CSRF, GraphQL je zranitelné vůči podobným útokům, jako je CSRF.

Při ohodnocení zranitelnosti jsem taktéž předpokládal nejhorší možnou alternativu: naivního uživatele, kterého je jednoduché donutit požadavek provést a možnost provádět esenciálních administrátorské úkony (např. změnu hesla) jednoduchým požadavkem. V takovém případě je zranitelnost ohodnocena 10 body, jedná se tedy o katastrofální zranitelnost[9]

3.2.2 Srovnání zranitelnosti v GraphQL a REST API

REST API teoreticky může být zranitelné, např. pokud autentizační token je uložen v datech internetového prohlížeče, popřípadě uvnitř cookies. Cookies jsou soubory uloženy na obou stranách komunikace client-server obsahující některé komunikační detaily jako např. předměty v nákupním košíku na stránce e-shopu. Jejich cílem je, jak již bylo zmíněno2.3 zefektivnění komunikace. Tyto cookies rozšiřují povrch útoku o CSRF – bez nich by CSRF nebyl možný. Vzhledem k tomu, že cookies poskytují velké zefektivnění, kompletně tuto funkcionalitu vypnout není dobré řešení (ačkoliv z bezpečnostního hlediska se jedná o ideální možnost). Útoku naštěstí lze předejít i méně drastickými způsoby. Standard REST API[13] nejen pro tento účel definuje tzv. „Bearer authentication“, což je autentizace a autorizace pomocí tzv. „Bearer token“, tokenu, který se ukládá do hlavičky požadavku HTTP(s) a zaručuje autenticitu klienta. Díky tomuto způsobu ochrany není útok CSRF nijak běžný a tedy systémy na bázi REST API mají v tomto směru navrch. Avšak jakkoliv je tato ochrana jednoduše integrovatelná, je nutné si její funkčnost minimálně ověřit.

3.2.3 Analýza automatizovaného snížení rizika zranitelnosti

Při analýze opravy zranitelnosti CSRF se budu opět řídit směrnicemi OWASP[12]. Podle této směrnice existují 2 primární metody na ochranu proti CSRF:

1. **Token based mitigation**
   Ochrana proti CSRF na bázi autentizačního tokenu funguje tak, že ke každém dotazu komunikace je přiložen anti-CSRF token. Tento token se vytvoří náhodně na začátku každé session. Právě tato náhodnost znemožní útočníkovi donutit uživatele odeslat zfalšovaný dotaz, protože není možné tento dotaz vytvořit bez znalosti anti-CSRF tokenu.

2. **Double submit cookie**
   Double submit cookie je alternativa k využívání anti-CSRF tokenu, v případě, že je z nějakého důvodu problematické ukládání tokenu na straně serveru. V této technice se při každé session vygeneruje pseudonáhodná hodnota, která je uložena se na straně klienta. V komunikaci je pak při každém dotazu vkládána hodnota tohoto cookie do schované hlavičky.

Mimo těchto metod existuje řada „Defence in Depth“ principů. Tyto principy, taktéž zmiňované ve směrnici OWASP[12] fungují jakožto dodatečná obrana a bohužel nejsou nijak automatizovatelné. Jedná se o např. využívání atributu SameSite v souborech cookies atd. V práci se budu zabývat automatizovatelnými obrany proti CSRF.

Případ GraphQL s naimplementovanou autentizací, kdy autentizační údaje jsou přenášeny v hlavičce (nikoliv jako cookie soubor!) v sobě již zahrnuje implementaci Token-based CSRF mitigation. Díky této hlavičce je aplikace vůči CSRF již chráněná, jelikož ona sama slouží jakožto token. Odešle-li uživatel zfalšovaný dotaz, tato hlavička se do něho automaticky nevloží a útočník autentizační údaje nemá jak uhodnout. Důkaz tohoto tvrzení je vidět v sekci 5.4.2, kde se věnuji snížení rizika této zranitelnosti.

[9] `https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator?vector=AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:N&`

Pro automatizované snížení rizika CSRF je žádoucí vytvořit korektně fungující autentizaci na bázi tokenu. To mi dalo o důvod navíc tento typ autentizace využít, ale také zároveň o starost navíc, protože nekorektně vytvořená autentizace otevře výslednou aplikaci dalšímu útoku.

Alternativou automatizované obrany vůči GraphQL CSRF je CORS[10] –technika omezení možnosti prováděných dotazů na bázi tzv. originů.

3.3 GraphQL errors information Disclosure

3.3.1 Popis zranitelnosti

Vývojáři GraphQL si vzali ponaučení z chybových hlášení kompilátorů jazyka C++ a implementovali velice detailní a výřečná chybová hlášení. Bohužel neimplementovali žádnou hranici určující kdo dané chybové hlášení uvidí. Útočník může přehnaně výřečných hlášení zneužít pro zmapování funkcí modelu a schématu GraphQL. Tuto situaci uvedu na příkladu níže inspirovaný článkem týkajícím se této konkrétní zranitelnosti [14]:

```
query TrySearchActivities {
searchActivities(startDate: "2022-04-02", endDate: "2022-04-09"){
    name
    price
    description
}}
```

K tomuto příkladu je ve zdroji[14] uvedena i odpověď GraphQL serveru (upraveno pro estetické účely):

```
{"errors": [
    {
      "message": "Cannot query field \"searchActivities\"
                  on type \"Query\".
                  Did you mean \"searchHotels\",
                  \"searchLeisures\" or \"searchUsers\"?",

      "locations": [{
         "line": 144,
         "column": 3}]}]}
```

Ve výše zmíněném příkladě se uživatel dozvědel dvě informace:

1. Neexistuje funkce obsluhující dotaz „searchActivities“
2. Existují funkce „searchHotels“, „searchLeisures“ a „searchUsers“

Tímto způsobem si případný útočník může bez jakýchkoliv eskalací práv zmapovat celý systém zaštiťovaný GraphQL, čímž se značně zvyšuje riziko kompromitace systému.

[10] https://www.apollographql.com/docs/apollo-server/security/cors/

Při ohodnocení zranitelnosti je možné předpokládat právě jednu alternativu: útočník zmapuje celé prostředí. V takovém případě je zranitelnost ohodnocena 5,8 body, jedná se tedy o závažnou zranitelnost[11]. Tato zranitelnost sama o sobě nijak zničující není, ale umožňuje lehce odhalit jiné zranitelnosti a tím se stává stejně nebezpečnou, jako jsou ostatní zde zmíněné zranitelnosti.

3.3.2 Srovnání zranitelnosti v GraphQL a REST API

Standard REST API2.3.1 v případě této zranitelnosti jednoznačně mluví o využívání návratových kódu HTTP[15]. Tyto kódy jsou právě tak výřečné, aby uživateli poskytly co nejvíce informací ohledně chyby dotazu a zároveň dbaly na uchování důvěrnosti systému. Ve výše uvedém příkladě by standardní REST API vrátilo chybu *404: Not found*. V případě této zranitelnosti je REST API jednoznačně méně zranitelné.

3.3.3 Analýza automatizovaného snížení rizika zranitelnosti

Jakkoliv lítostivé, pro bezpečnost aplikace je nutné chybová hlášení GraphQL předělat. V praxi, kde je prostředí rozděleno na produkční a testovací část se v testovací části využijí chybová hlášení GraphQL a v produkčním prostředí využívat bezpečná chybová hlášení. GraphQL naštěstí umožňuje tvorbu nových chybových hlášení, viz. příklad 3.1, inspirován stejným článkem, kterým byl inspirován příklad výše[14]:

Výpis kódu 3.1 Příklad deklarace chybových hlášení

```
import {GraphQLError, GraphQLErrorExtensions} from 'graphql'

class InputValidationError extends GraphQLError {
          extensions: GraphQLErrorExtensions;
    constructor(message: string, field: string) {
        super(message);
        this.extensions = {
            statusCode: 500,
            code: "INPUT_VALIDATION_FAILED",
            field
        };
    }
}

#Error call:
post: (_parent:null, {id}: QueryPostArgs, context: Context) => {
    if(id >= context.db.posts.length) {
        throw new InputValidationError("Index out of range", "id")
    }
// Fetch and return the corresponding post
}
```

Daná chybová hlášení budu vytvářet, jak je zvykem, podle ověřeného systému. Využiji k tomu standartní HTTP chybová hlášení, čímž efektivně minimalizuji tuto zranitelnost a bezpečnostně přiblížím GraphQL k dobře implementovanému REST API.

[11] https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator?vector=AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:N&

3.3.4 Dodatek: GraphQL introspection

K podobným informacím, jako při zneužití zranitelnosti GraphQL errors information disclosure je možné dostat se podstatně jednodušeji, pomocí tzv. „GraphQL introspection“.

GraphQL introspection je funkcionalita GraphQL, která dovoluje uživatelům ptát se na informace týkající se podporovaných dotazů. Využití této funkcionality může vypadat následovně[16] (vlevo dotaz, vpravo potenciální odpověď):

```
{
    __schema {
        types {
            name
        }
    }
}
```

```
{
  "data": {
   "__schema": {
    "types": [
        {
          "name": "Query"
        }
        {
          "name": "String"
        }
        {
          "name": "Episode"
        }
        ...
```

V odpovědi na dotaz uvedený v příkladu výše se vyskytují datové typy. Pomocí dalších dotazů je za pomocí GraphQL introspection možné o těchto typech získat údaje a tím se získat seznam podporovaných dotazů a zmapovat si schéma aplikace. Příklad, jak něco takového lze udělat je popsáno v oficiální dokumentaci GraphQL v sekci introspekce [16] (ve zdroji je funkcionalita popsána jako vlastnost, ne hrozba).

Aby mělo nějaký smysl opravit zranitelnost GraphQL errors information disclosure, je nutné v produkčním prostředí funkcionalitu GraphQL introspection vypnout. Po odstranění obou těchto „zranitelností“ nebude mít útočník jak si zmapovat schéma GraphQL aplikace. Vypnutí introspekce je primitivní, stačí tento fakt zmínit v konfiguračních souborech, nebo konstruktoru serveru. V některých GraphQL serverech (např. Apollo serveru [17]) je v produkčním prostředí introspekce vypnutá automaticky.

3.4 Zranitelnost vůči útokům DoS

3.4.1 Popis zranitelnosti

Na GraphQL existují 2 útoky typu DoS. První z nich, „rekurzivní DoS“ (pojmenováno mnou pro účely této práce) využívá absenci rate-limitingu hloubky dotazů spolu se souvislostmi dat. Druhý z nich, „batching attack“ zneužívá možnosti dávkování dotazů.

Rekurzivní DoS, jak již bylo zmíněno využívá faktu, že GraphQL v sobě nemá zabudovaný žádný systém pro omezení hloubky dotazů. Příliš složitý dotaz pak může celý systém přetížit. GraphQL umožňuje vytvářet nekonečně hluboké dotazy, pokud existuje posloupnost odkazů mezi daty taková, že tvoří cyklus (např. autor odkazuje na knihy, které napsal a kniha odkazuje na jejího autora)[18]. Zlomyslný dotaz může vypadat např. následovně:

```
query { authors {
            books {
                authors {
                    books {
                        #repeat until server crashes }}}}}}
```

Batching attack způsobí DoS tím, že jeden uživatel pošle nadměrné množství dotazů jedním voláním (využije techniky „Batching“). Tím stejně jako v předchozím případě může přetížit systém. Tento útok může vypadat např. následovně (inspirováno směrnicí OWASP pro zabezpečení GraphQL[18]):

```
query {
    droid(id: "2000") {
        name
    }
    second:droid(id: "2001") {
        name
    }
    third:droid(id: "2002") {
        name
    }
    #call upon ridiculous amount of droids
    #to start a star war
}
```

Při ohodnocení zranitelnosti vůči výše popsaným útokům budu brát tyto útoky souhrně, jelikož oba vyžadují stejné podmínky a mají stejný výsledek. Zranitelnost jsem ohodnotil 8.6 body, podstatně závažněji, než by se na první pohled mohlo zdát[12].

3.4.2 Srovnání zranitelnosti v GraphQL a REST API

Zranitlenost DoS za pomocí útoků popsaných výše jsou specifické pro GraphQL. Co se REST API týče, úspěšnost útoků DoS závisí na architektuře systému (a samozřejmě výpočetní síle jednotlivých komponent, čehož se tato práce netýká). Jinak řečeno, úspěšný útok typu DoS proti REST API vyžaduje znalost vnitřní struktury systému, identifikaci slabin daného systému a jejich následné zneužití. Je zřejmé, že takový DoS proti systému na bázi REST je tedy obtížnější, než výše zmíněné specifické DoS na GraphQL a proto je REST API méně zranitelné vůči útokům typy DoS.

3.4.3 Analýza automatizovaného snížení rizika zranitelnosti

K prevenci DoS aplikace využívající GraphQL existuje samostatná kapitola ve směrnicích OWASP[19]. Tato kapitola radí následující kroky:

1. **Limitace dotazů**
   V GraphQL má každý dotaz hloubku a specifikovaný počet objektů, obě vlastnosti jsou v základním nastavení nekonečné. K limitaci DoS je nezbytně nutné tyto dotazy omezit na konstantu úměrnou aplikaci.

2. **Časový limit pro splnění dotazu**
   Jak je z názvu zřejmé, pomocí timeoutu efektivně zarazíme nepřátelský dotaz po předem stanoveném časovém limitu. To má jednu zásadní nevýhodu: aplikace bude po dobu časového limitu nedostupná a tím pádem toto opatření pouze redukuje DoS na útok „Create Lag on Service“ (pojem vymyšlený pro efekt)

[12] https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator?vector=AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:N&

3. **Omezení namáhavých dotazů**
   Tato technika v sobě zahrnuje analýzu náročnosti dotazu a poté zablokování těch dotazů, které byly vyhodnoceny příliš negativně. V ideálním světě je tato technika zázračná, ovšem v praxi nastává problém výpočtu ideální meze přijetí dotazu. Je-li příliš nízká, může zakázat legitimní dotaz. Je-li naopak příliš vysoká, přestane technika být účinná. K výpočtu náročnosti dotazu se také spotřebovává výkon.

4. **Rate Limiting**
   Limitováním počtu dotazů, které lze položit z jedné IP adresy znemožní DoS pomocí spamu dotazů. Jednoznačně se jedná o nutnou obranu proti DoS, obzvlášť pokud aplikace dovoluje pokládat některé dotazy bez autentizace (např. ukázková data na úvodní stránce)

Kromě následování těchto obecných rad existuje ještě jeden způsob obrany proti DoS využívající tzv. „persistent queries“.

Persistent queries je technika, kdy nastane očíslování dotazů podle dohody mezi všemi strany komunikace. Při komunikaci využívající persistent queries klient zasílá místo celých dotazů pouze jejich číslo, čímž šetří propustnost serveru[20]. Zavedení této funkcionality je sice ušetřena propustnost, ale sama o sobě nestačí:

1. **Nelze namapovat všechny dotazy**
   Extrémně velké dotazy, např. takové, ve kterých se uživatel ptá na jména 1001 droidů, namapovat nelze, jelikož takových dotazů je nekonečně mnoho.

2. **DoS hrubou silou je stále možné**
   Ve výsledku nezáleží, zda útočník pošle nekonečně mnoho dotazů týkajících se jmen droidů, nebo dotazů „16“.

Samo o sobě tato funkcionalita sice nestačí, ale existuje metoda obrany proti DoS založená na principu persistent queries, kterou úspěšně využívá Facebook[21]:

1. V testovacím (vývojovém) prostředí server povoluje všechny dotazy. Tyto dotazy se uloží
2. Při nasazení systému se všechny uložené dotazy uloží a namapují na jejich ID (podle metody persistent queries)
3. V produkčním prostředí jsou povoleny pouze dotazy dle jejich ID (namapované v předchozím bodu)

Tato obrana je podstatně striktnější než obecné rady OWASPU pro prevenci útoků typu DoS. Ve své podstatě jde o rozsáhlý whitelist, který sice útočníkům zcela jistě zakáže zpracovávání zlomyslných dotazů, ale může mít za následek odmítnutí legitimního dotazu.

Pro účely obrany proti DoS využiji v práci metodu Facebooku, jelikož v sobě obsahuje to nejlepší z rad OWASPU – limitaci i omezení dotazů. Přesto do implementace zahrnu měření hloubky dotazu, aby server nebyl položen v testovacím režimu. Časový limit ke splnění dotazu poté nebude potřeba dělat, jelikož je zaručené, že dotaz bude zvládnutelný. Zahrnout Rate Limiting je již triviální.

3.5 Dodatek: SQL injection

V obecném případě je SQL injection útok, kdy uživatel zadá takový vstup, který lze interpretovat jako kód pro interpreter databázových dotazů. Pomocí tohoto vstupu databáze vyhodnotí dotaz jinak, než jak bylo plánováno vývojářem aplikace. Příklad takového vstupu může být následující: *1 OR 1=1;*. Pokud by funkce tvořila dotaz např. jako „*query = ”SELECT * FROM _users WHERE id = ”+ args.id;*“, databáze by vyhodnotila dotaz jako „vyber všechny uživatele“. Tímto by vznikl únik dat, ačkoliv útok SQL injection umí udělat větší škody.

GraphQL, jako téměř jakákoliv aplikace přijímací uživatelský vstup, je zranitelné vůči SQL injection. Příklad uvedený výše by mohl uvnitř GraphQL vypadat zhruba jako v kódu 3.2.

Výpis kódu 3.2 GraphQL SQL injection example

```
    query User {                    export const loadUserByID = (id) => {
    user(id: "1 OR 1=1"){           const q = `SELECT * FROM _users where\
        id                                    _users.user.ID = ${id};`;
        name                        return db.query(q);
        #other fields               }
    }
}
```

Zranitelnost SQL injection bude kompletně řešena v rámci gRPC4.3. Analýza možnosti automatizovaného snížení rizika i jeho implementace proběhne v sekcích k nim určených. V rámci GraphQL nebudu tuto zranitelnost řešit, abych v rámci práce nedělal stejnou práci věc dvakrát, navíc práci, jenž se netýká tak úzce přímo GraphQL. Účel této sekce je upozornit na tuto zranitelnost a nutnost jejího ošetření (např. pomocí technik zmíněných v rámci gRPC). Zranitelnosti je ilustrována na obrázku 3.1.

Obrázek 3.1 SQL injection v GraphQL

Kapitola 4

Analýza známých zranitelností gRPC

V této kapitole je obsažen seznam a popis nejznámějších zranitelností gRPC. U každé zranitelnosti je uvedeno její ohodnocení pomocí systému CVSS v3.1 (industriální standard pro ohodnocení vážnosti zranitelností, hojně využívaný na FIT ČVUT), pro jehož výsledek jsem využil online kalkulátor zveřejněný agenturou NIST[a] (americká agentura zabývající se standardizací v oboru informačních technologií). Také se u každé zranitelnosti nachází její zasazení do bezpečnostního kontextu REST API a na konec analýza možností snížení jejího rizika.

[a] `https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator`

4.1 Komunikace nešifrovaným kanálem

4.1.1 Popis zranitelnosti

Servery gRPC nejenže podporují komunikaci pomocí nešifrovaného textu (tzv. „plaintext“), ale dokonce jej v základním nastavením využívají, jak je ilustrováno na obrázku 4.1. Navrch k tomu existuje funkce na straně klienta jménem *usePlaintext()*, která slouží jakožto indikátor navázání spojení pomocí nešifrovaného textu[22]. Tyto možnosti jsou velmi užitečné při testování, avšak do produkce se z očividných důvodů nesmějí dostat.

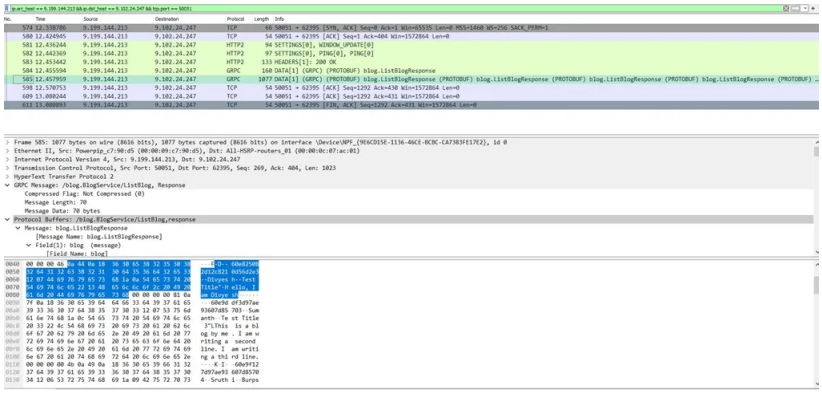
Tato zranitelnost je naprosto fatální. Při jejím ohodnocení jsem ani nemusel uvažovat nejhorší možnou alternativu, ohodnocena je 10 body[1].

4.1.2 Srovnání zranitelnosti v gRPC a REST API

Komunikace se systémem na bázi REST API je postavená na protokolu HTTP. HTTP je sice nešifrovaný protokol, avšak v moderní době již slouží pouze jako základ své bezpečnější nástavbě HTTPS, která šifruje spojení pomocí TLS. Dnes je již HTTP, starší verze protokolu, není využívána prakticky vůbec[2]. To platí do takové míry, že spousta moderních nástrojů (např. webové prohlížeče Mozzila Firefox, nebo Google Chrome) před spojením využívající HTTP

[1] `https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator?vector=AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:N&`

[2] Článek z roku 2018 obsahující amatérsky vytvořený seznam webových stránek využívající HTTP: `https://www.androidauthority.com/sites-still-on-http-889265/`. Jak je vidět, není jich mnoho



Obrázek 4.1 Příklad nešifrované komunikace pomocí gRPC[22]

důsledně varuje a dokonce zakazuje některé funkcionality[3]. Systémy na bázi REST API nejsou žádnou výjimkou - komunikaci s klientem šifrují alespoň pomocí TLS, jelikož komunikují skrze zabezpečený HTTPS (tento fakt je součástí bezpečnostních směrnic OWASP[23]). Tato zranitelnost se REST API tedy vůbec netýká.

4.1.3 Analýza automatizovaného snížení rizika zranitelnosti

gRPC při instalaci přichází s několika možnostmi autentizačních mechanizmů, pomocí kterých je možné učinit komunikaci s jiným systémy podstatně bezpečnější[24]. Mimo jiné tyto autentizační mechanizmy zcela řeší problém nešifrované komunikace, jelikož gRPC automaticky šifruje komunikaci s jakýmkoliv autentizovaným protějškem. Vzhledem k tomu, že tato funkcionalita je již součástí gRPC a přímo řeší probíraný problém, přímo se nabízí ji využít.

Dle oficiální dokumentace gRPC autentizace[24] podporuje gRPC 3 autentizační metody:

1. **Autentizace využívající TLS/SSL**
   gRPC plně integruje autentizační mechanismus na základě TLS/SSL. Vzhledem k rozšířenosti tohoto řešení na celém internetu pravděpodobně půjde o preferovanou metodu.

2. **Autentizace využívající ALTS**
   ALTS je metoda autentizace zařízení vyvinutá společnostní Google[4] používaná v aplikacích běžící na implementaci cloudu společnosti Google jménem GCP. Vzhledem k této restrikci toto řešení zamítám.

3. **Autentizace využívající Google tokeny**
   Tato metoda funguje na základě OAuth2tokenů a je využívaná pro přístup ke Google API. Opět se jedná o nepřijatelnou restrikci.

Je tedy zřejmé, že pro metodu snížení rizika této zranitelnosti využiji autentizaci na bázi TLS/SSL

Automatizovaná oprava této zranitelnosti se bude tedy skládat z právě jednoho kroku - donucení serveru, aby komunikoval pouze s klienty autentizovanými pomocí SSL/TLS. Pro vývojáře vytvořím za tímto účelem funkci s jednoduchým rozhraním, čímž se snažím dosáhnout zvýšení motivace k jejímu využívání.

[3] Např. stahování, viz. `https://blog.mozilla.org/security/2021/10/05/firefox-93-protects-against-insecure-downloads/`
[4] Detaily k dispozici zde: `https://grpc.io/docs/languages/go/alts/`

4.2 Nezabezpečená definice souboru protobuf

4.2.1 Popis zranitelnosti

Soubor protobuf je konfigurační soubor na straně serveru, který obsahuje definice služeb a zpráv[4]. Tento soubor je jeden z klíčových souborů celého gRPC. Detailněji je popsán v kapitole 2.2.1. Jelikož definuje, jaká data bude server odesílat, musí být vývojář nesmírně opatrný při jeho tvorbě. Je potřeba, aby ve zprávách bylo obsaženo jen nejnutnější, aby klient dostal právě nezbytné množství informací[22] (viz. „princip nejmenších možných oprávnění“, anglicky „principle of least privilege“)[5].

Zranitelnost v podobě odhalení nepotřebných dat („data exposure“) je velmi obecná. Týká se téměř všech aplikací, které nějak s daty pracují. V případě gRPC je možné této zranitelnosti dosáhnout jedním způsobem navíc a to právě díky uvedenému protobuf souboru. Při ohodnocení zranitelnosti jsem uvažoval nejhorší možnou alternativu - zpráva obsahuje navíc položku, která povede k eskalaci privilegií na úroveň administrátora systému. Zranitelnost jsem ohodnotil opět 10 body.[6].

4.2.2 Srovnání zranitelnosti v gRPC a REST API

Popisovaná zranitelnost je velice konkrétně zaměřená na protokol gRPC. Jak již bylo zmíněno, data exposure jako taková se týká i většiny aplikací na bázi REST API, ale při porovnání zranitelnosti mě zajímá převážně, jestli existuje nějaký mechanismus umožňující data exposure na úrovni tvorby zpráv.

V REST API nejsou zprávy nijak předem definované. Standard REST API2.3.1 je takový, že zpráva se seskládá do formátu JSON (popřípadně jiných formátů) ze všech dat, které jsou k dispozici z databázové vrstvy aplikace[15]. Tedy námi konkrétní popisovaná zranitelnost se REST API nijak netýká, ačkoliv obecně data exposure je samozřejmě problém i systémů na bázi REST API.

4.2.3 Analýza automatizovaného snížení rizika zranitelnosti

Celá tato zranitelnost je založená na tom, že vývojář aplikace jakýmkoliv nedopatřením rozšíří definici zprávy o nebezpečné položky. Ideální automatizovaná oprava by vyžadovala automatickou formu vyhodnocování formátu zprávy a ve své primitivní formě by vypadala podobně jako na příkladu 4.1.

Výpis kódu 4.1 Pseudokód vyhodnocování zpráv

```
function evaluateSafety:
    messageThreatLevel <- 0

    foreach(message in file.protobuf):
        messageThreatLevel <- 0
        foreach(field in message):
            messageThreatLevel += getThread(field)
        if(MAX_ACCEPTABLE_THREAT_LEVEL < messageThreatLevel):
            return "DANGEROUS ON MESSAGE message"

    return "SAFE"
```

[5] https://owasp.org/www-community/Access_Control

[6] https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator?vector=AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:N&

Jakkoliv by taková forma automatizované kontroly byla skvělá, má jeden problém – získat přesnou míru rizika obsažené v jednom poli zprávy je nelehký úkol. Jednotlivá pole mohou být buďto naprosto nepřijatelná, jako např. pole „server_administrator_password“ a zamítnuta, nebo mohou být nebezpečná jenom v nějakých případech, např. „server_OS“. Tyto položky, jenž jsou nebezpečné jen v nějakých případech by musely před odesláním projít dynamickou kontrolou, která by kontrolovala zprávu v kontextu celé komunikace.

Dynamická kontrola zpráv diskutovaná v odstavci výše bohužel není možná z důvodu absence podpory samotného gRPC. Uvnitř gRPC neexistuje žádné podmíněné odesílání zpráv, takže celá kontrola by musela proběhnout v každé funkci napsané vývojářem aplikace, což nelze automaticky kontrolovat. Navíc není možné předpovědět kontext jakéhokoliv serveru využívající gRPC. Pro snížení rizika zranitelnosti si tedy vytvořím seznam zakázaných položek a funkci výše zjednoduším na podobu znázorněnou psuedokódem níže 4.2.

Výpis kódu 4.2 Pseudokód zjednoduššení vyhodnocování zpráv

```
function evaluateSafetySimplified:
    foreach(message in file.protobuf):
        foreach(field in message):
            if(field exists in dangeousList):
                return "DANGEROUS ON MESSAGE message"

    return "SAFE"
```

Výsledný kód bude poté vracet hodnotu udávající míru závažnosti položek uvnitř zpráv (což není ilustrováno v kódu 4.2, který slouží pro účel vytvoření představy) a volitelně logovat hlášení týkající se nalezených nedostatků. Míra závažnosti bude zjednodušena tak, že je vráceno číslo udávající počet potenciálně rizikových položek. Zpracování návratové hodnoty již nechám na administrátory jednotlivých aplikací, jelikož mohou existovat situace, kdy potenciálně nebezpečnou položku je nutné ve zprávě nechat.

4.3 gRPC SQL injection

4.3.1 Popis zranitelnosti

SQL injection jako obecná zranitelnost je v této práci již popsána v kapitole týkající se analýzy známých zranitelností GraphQL v sekci 3.5, kde je popsán základ zranitelnosti.

gRPC, přijímá uživatelský vstup v podobě parametrů funkce a tím pádem se otevírá zranitelnosti SQL injection snadno. Způsob extrahování jeho dat ze zpráv činí otevření vůči SQL injection ještě snadnější. Příklad, kdy uživatel zadá identifikátor jako *1 OR 1=1;* uvedený v sekci 3.5 by uvnitř gRPC mohl vypadat následovně jako v kódu 4.3 uvedeným níže.

Výpis kódu 4.3 gRPC SQL injection example

```
        std::string query = "SELECT account_balance FROM _users
                    WHERE username = " + msg->id();

        //naive function for querying database
        std::string res = querydb(query);

        reply->set_accountBalance(res);
        return Status::OK;
```

Při ohodnocení zranitelnosti jsem předpokládal alternativu, při které databáze svůj vstup nekontroluje vůbec a navíc obsahuje data dostačující pro ovládnutí systému, případně vydíraní uživatelů daného systému. V takovém případě je zranitelnost ohodnocena 10 body, jedná se tedy o katastrofální zranitelnost[7].

4.3.2 Srovnání zranitelnosti v gRPC a REST API

Jedna z podmínek REST API je hierarchické členění systému na straně serveru, zmíněno v sekci 2.3. Za předpokladu korektně pracující autorizace a autentizace je díky této podmínce výrazně snazší dbát na bezpečnost jednotlivých komponent systému, jako je např. databáze. Díky tomu je SQL injection v REST API podstatně menší problém, než v gRPC, jelikož standardní REST API (2.3.1) deleguje databázovou bezpečnost přímo na databázi. Je nutno mít na paměti, že REST API je pouze architektonický styl, tudíž tato zranitelnost se v systému používající REST API může vyskytovat.

4.3.3 Analýza automatizovaného snížení rizika zranitelnosti

OWASP SQL injection cheatsheet[25] zmiňuje následující možné opravy této zranitelnosti:

1. **Používání předpřipravených dotazů s parametry**
   Metoda předpřipravených parametrů je nejbezpečnější metodou psaní databázových dotazů do kódu. Pomocí této techniky lze vytvořit část dotazu s vynechanými prvky na místech, kam dodatečně vložíme uživatelský vstup. Dotaz v tomto rozpracovaném stavu je položen databázi, která vyhodnotí, jaký by měl být její výsledek a poté čeká na doplnění informací ke konkretizaci tohoto výsledku.

   V případě gRPC je používání předpřipravených dotazů oproti běžným dotazům relativně nebolestivé, jelikož gRPC pro práci se vstupem nutí vývojáře vstup postupně extrahovat. Tím pádem bezpečnější varianta kódu je pouze rozšíření „nebezpečné varianty“, ilustrováno v kódu 4.4.

Výpis kódu 4.4 Předpřipravené dotazy s parametry v gRPC

```
//? instead of '+ msg->username()'
std::string query = "SELECT account_balance
            FROM _users WHERE id = ? ";

//2 extra lines
stmt = dataBaseConnection->PrepareStatement(query)
stmt.setString(1, msg->id());

std::string res = stmt.query();
reply->set_accountBalance(res);
return Status::OK;
```

2. **Používání předpřipravených procedur**
   Jako druhý nejefektivnější způsob obrany proti SQL injection zmiňuje OWASP používání předpřipravených procedur. Jak je z názvu patrné, tato metoda je velice podobná metodě z prvního bodu, s tím rozdílem, že zde diskutované procedury jsou uloženy přímo v databázi.

   gRPC v nemá v používání této metody žádná specifika oproti jakýmkoliv jiným programům.

[7] https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator?vector=AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:N&

3. **Validace pomocí whitelistu**
   Jakožto třetí možnost je v OWASPU zmiňován whitelisting. Ten je založen na zakazování typů parametrů při zacházení s konkrétními tabulkami. Může se např. jednat o údaj jména města při dotazování se uživatelské tabulky, nebo indikátor řazení výsledků...

   gRPC opět nemá v tomto bodě žádná specifika, funkce validace může běžet na serveru využívající gRPC stejně, jako kdekoliv jinde, ilustrováno na příkladu 4.5 (kód je zjednodušen v tom, že switch přijímá řetězec. V reálném kódu by bylo nutné zamaskovat řetězec např. pomocí hashovací funkce).

Výpis kódu 4.5 SQL query whitelisting

```
std::set<unsigned int> legalOptions;
for(int i = 0; i < numOfParams; ++i){
    switch(params[i]){
        case value1: legalOptions.insert(TableX); break;
        case value2: legalOptions.insert(TableY); break;
                        ...
    }
}
return legalOptions;
```

   Tato metoda podle OWASPu není vhodná. Pokud je v aplikaci použita, svědčí o nevhodném designu dané aplikace. Je nepřehledná, obtížně rozšiřitelná a z dlouhodobého hlediska neprovozovatelná. Se všemi těmito uvedenými vlastnostmi souhlasím a této metodě se vyhnu jak v této práci, tak v profesním životě.

4. **Escapování uživatelského vstupu**
   Jako poslední linii obrany OWASP uvádí metodu escapování uživatelského vstupu, což je metoda, kdy server obalí uživatelský vstup uvozovkami v naději, že se poté bude interpretovat vždy jako řetězec. Slovo naděje bylo použito záměrně, jelikož jakákoliv forma escapování vstupu lze obejít tím, že útočník vstup přizpůsobí. Aplikace, která spoléhá na escapování znaků není z hlediska SQL injection bezpečná a proto se této metodě hodlám v této práci vyhnout. V profesním životě si ovšem dokážu představit případy, kdy pro rychlou záchranu tuto metodu využiji pro účely mírné prodloužení času, který útočník na útok spotřebuje. Tento čas lze využít pro tvorbu opravdové prevence útoku.

Z výše uvedených možností pro prevenci SQL injection je zřejmé, že automatizovaná snížení SQL injection není zcela možná. Není možné nijak donutit programátora, aby validoval vstup. V extrémním případě, otrlý programátor nenávidějící bezpečnost může napsat SQL dotaz tak, že využije předpřipravené dotazy s nula parametry a tím kompletně obejde jejich pointu.

Je ale možné jít snížení rizika naproti. Pro účely co možná největšího bezpečí vůči SQL injection v aplikaci gRPC udělám v práci kroky v co nejužším souladu se směrnicí prevence od společnosti OWASP zmiňovaného výše:

1. Vytvořím sadu předpřipravených procedur pro validaci jednotlivých parametrů. Tyto funkce budou rozdílné pro každý typ parametru a budou inspirované nejčastějšími případy využití.

2. Co nejvíce motivuji programátora, aby tyto funkce využíval podle možností vybraného jazyka např. jednoduchostí volání daných funkcí.

K implementaci tohoto automatizovaného snížení si pokládám 2 zjednodušení:

1. V řešení se budu držet jedné konkrétní databáze, a to MySQL databáze. Toto zjednodušení pokládám proto, že napsat dané řešení v obecné formě je rozsáhlé nad rámec této práce. Takové řešení by se skládalo z jedné abstraktní třídy, která by obsahovala základ této funkcionality a ze které by poté dědilo 343[8] podtříd obsahujících skutečnou implementaci. Některé tyto třídy by ani nemohly mít metody ve stejném tvaru, díky různým syntaxím SQL. Některé databáze jsou licencované, tedy není možné metody ani otestovat. Také mohou existovat databáze s API přístupné z disjunktními jazyků.

   Mnou implementované specifické řešení bude sloužit jako důkaz, že automatizovaně lze této zranitelnosti předejít a také jako šablona pro kohokoliv, kdo by chtěl automatizovaně dosáhnout snížení rizika SQL injection pro jinou implementaci databáze.

2. Implementované funkce budou zaměřené pouze na vybrané SQL metody - konkrétně ty, které jsou pro SQL injection očividně využitelné z podobných důvodů, jako bod jedna.

Stručně řečeno, vznikne nový query builder pro vybrané dotazy v jazyce C++ pro MySQL, nástavba nad officiální knihovnou lmysqlcppconn[9]. Tuto kombinaci jsem si vybral proto, že jsem při své rešerši nenašel žádnou adekvátní nástavu a proto, že MySQL, jakožto populární a open-source databáze je často využívaná pro projekty s nižším finančním rozpočtem, tedy ne nutně využívaná odborníky. Mým cílem je tímto krokem jít naproti bezpečnosti této cílové skupině. Samo o sobě to na kompletní zabezpečení nestačí, jediný způsob, jak se zranitelnosti vyhnout je vzdělání vývojáře k tomuto problému a jejich motivace problém řešit.

[8] počet různých databází dle magazínu „Towards Data Science“ ve článku dostupném zde: `https://towardsdatascience.com/top-10-databases-to-use-in-2021-d7e6a85402ba`

[9] ??

Kapitola 5

Tvorba prostředí pro demonstraci a testování automatizovaných snížení rizik zranitelností GraphQL

Tato kapitola obsahuje popis tvorby testovacího prostředí pro metodu vzdálené komunikace GraphQL. Konkrétně obsahuje tvorbu jednoduché aplikace, na níž budu demonstrovat jak jednotlivé zranitelnosti, tak efektivitu snížení rizika pomocí mnou vytvořeného middleware. Daná aplikace, mnou vytvořená, byla v souladu se zadáním práce schválena jejím vedoucím.

5.1 Tvorba virtuálního prostředí

Pro tvorbu testovacího prostředí se inspiruji nejčastějším řešením obdobných problémů na FIT ČVUT – využiji virtualizovaný počítač s UNIXovým operačním systémem, na který stáhnu jednotlivé frameworky a všechny jejich závislosti z externích zdrojů. Tímto způsobem zařídím nejen flexibilitu v tvorbě daného prostředí a jeho separaci od svého domácího operačního systému, ale i relativně věrnou simulaci reálné situace, kdy frameworky běží na UNIXových serverech. Jakožto základový operační systém jsem zvolil *LUbuntu* verze *22.04*[1], od něhož si slibuji kompatibilitu se stejnými programy, jako Ubuntu a malé nároky na paměť a výpočetní výkon. Pro samotnou virtualizaci využiji nástroj *Wmware Workstation 16 Player*[2], čistě z toho důvodu, že s ním mám zkušenosti a při práci s ním jsem narazil na minimum problémů.

Pro samotný GraphQL server existuje několik implementací, např. Express GraphQl[3], nebo Apollo GraphQL[4]. Pro účely této práce se nejlépe hodí Apollo GraphQL server, z důvodu implementace persistent queries (viz. sekce 3.4), které využiji k implementaci metodě automatizovaného snížení rizika GraphQL DoS útoků. Jiné servery obsahují persistent queries pouze ve formě pluginu a jejich funkcionalita není natolik ověřena. Apollo obsahuje i všechny ostatní nástroje zmíněné v kapitole 3, tedy pro tuto práci se ideální volbou.

[1] Ke stažení zde: `https://lubuntu.me/downloads/`
[2] Nejnovější možnost ke stažení zde: `https://www.vmware.com/products/workstation-player.html`
[3] `https://graphql.org/graphql-js/express-graphql/`
[4] `https://www.apollographql.com/docs/apollo-server/`

5.2 Popis demonstrační aplikace

Aplikace pro účely této práce je hodně inspirovaná aplikací Hello World z oficiální dokumentace Apollo GraphQL serveru[5]. Simuluje webovou aplikaci pro nadšené čtenáře zábavní literatury, na které si tito fanoušci mohou vyhledat informace o knihách, autorech... Konkrétně obsahuje následující:

- **Databázi knih a autorů**
  Databáze bude vytvořena tím nejprimitivnějším způsobem, který GraphQL považuje za nativní – pomocí JSON souborů. Propojovat program s SQL databází nemá pro účely této práce význam. Data uvnitř mé databáze budou navržena tak, aby bylo možné věrně simulovat útoky typu DoS – tedy bude možné na sebe data cyklicky navázat pro účel rekurzivního DoS a bude jich dostatečné množství pro batching attack. Oba útoky jsou vysvětlené v sekci 3.4.

- **Chybně nadefinovanou autentizaci a autorizaci**
  Touto funkcionalitou je simulován příklad, kdy aplikaci vytvářel amatér a tím otevřel bránu útoku GraphQL CSRF popsaný v sekci 3.2.2. Konkrétněji bude tato autentizace vytvořená tak, že po funkci login si server poznamená, který uživatel se přihlásil. Daný uživatel pak pro úspešné položení dotazu přiloží své uživatelské jméno jako parametr. Za žádných okolností nedoporučuji nikomu, aby tuto zjevně nevhodně navrženou autentizaci kdekoliv využíval. Navíc budu pro zjednodušení ukládat hesla v plaintextu. Zdůrazňuji, že to dělám jen proto, abych demonstroval zranitelnosti, popřípadě jejich absenci, nikoliv proto, že bych hesla ukládal v plaintextu rád. Důrazně doporučuji neukládat v plaintextu heslo ani na papíře, natož v aplikaci.

- **Metody pro registraci a dalších změn týkajících se dat**
  Za účelem testování je nutné zahrnout do aplikace i základní administrativní metody.

5.3 Instalace, konfigurace a spuštění GraphQL serveru

Jak již bylo v práci zmíněno, k aplikaci využiji GraphQL serveru Apollo GraphQL převážně kvůli funkcionalitě persistent queries.

V procesu se budu řídit příkladem, který má Apollo server uveden ve své oficiální dokumentaci[6]. Celý tento proces je automatizován v podobě skriptu uloženého na médiu, které je přiloženo k práci. Konkrétně bylo třeba následujících kroků[26][7]:

1. **Inicializace pracovního prostředí**
   mkdir graphql-server-example && cd graphql-server-example;
   mkdir server client; mkdir server/src server/src/data client/src; cd server/src[8]
   touch index.js resolvers.js schema.js
   touch data/users.js data/books.js data/authors.js

[5] `https://www.apollographql.com/docs/apollo-server/getting-started/`
[6] https://www.apollographql.com/tutorials/fullstack-quickstart/01-introduction
[7] Tento proces lze podstatně zjednodušit pomocí naklonování gitového repozitáře `https://github.com/apollographql/fullstack-tutorial`
[8] Od této chvíle bude potřeba pracovat pouze se serverem až do doby práce s persistent queries

2. **Instalace prerekvizit**
   sudo apt-get install nodejs npm

   - Node.js je open-source JavaScriptové runtime prostředí pro interpretaci JavaScriptového kódu mimo webové stránky[9]
   - npm je balíčkový systém zaměřený na balíčky související s JavaScriptem[10]

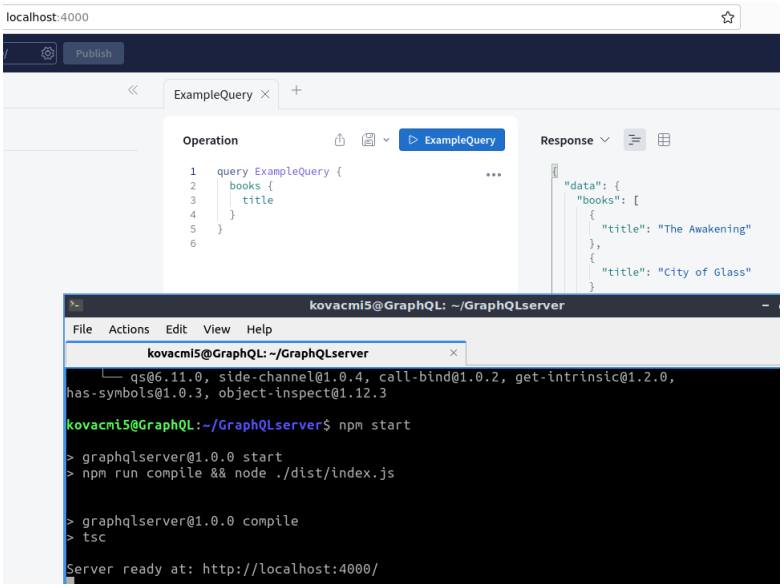
   npm install --save-dev n

   - n je zkratka pro „node“ a aktualizuje verzi nodejs

3. **Definice schématu, resolverů, serveru, datasety...**
   Schéma představuje definici datových typů. Resolverem se v GraphQL myslí komunikační funkce, která se provádí při odpovědi na dotaz. Samotný server je tvořen pomocí Apollo konstruktory. Všechny náležitosti GraphQL jsem uvedl do jim příslušných souborů vytvořených v bodě 1. Kompletní verze zranitelného serveru ve funkční formě je k dispozici na přenosném médiu přiloženém k práci.

4. **Nastartování serveru**
   npm start

Po těchto krocích již aplikaci využívající GraphQL běží, obsahujíce velice primitivní aplikaci s microdatabází knih. Funkčnost této aplikace je možné ověřit přes webové rozhraní serveru, v mém případě *http://localhost:4000*, viz. obrázek 5.1. Nyní stačí server rozšířit o funkcionality popsané v sekci 5.2.



Obrázek 5.1 GraphQL hello world

[9] https://nodejs.org/en
[10] https://www.npmjs.com/

5.4 Tvorba výsledné zranitelné aplikace a demonstrace zranitelností

Tato sekce obsahuje postup tvorby demonstrativní aplikaci popsanou v sekci 5.2. Také obsahuje praktickou ukázku zranitelností GraphQL na demonstrativní aplikaci. Kompletní verze zranitelného serveru je k dispozici na přenosném médiu přiloženém k práci.

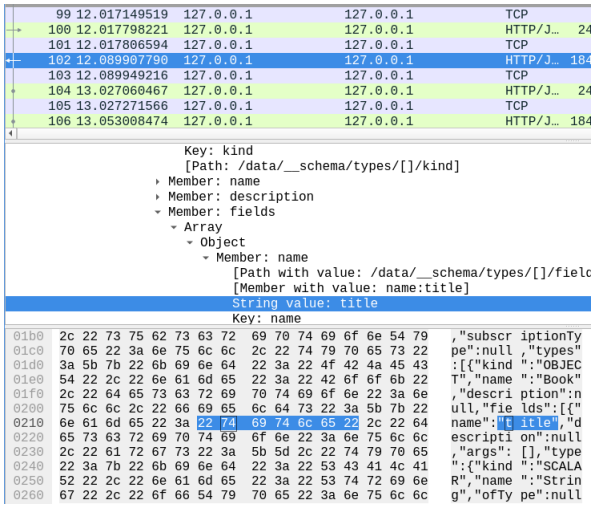
5.4.1 Metoda demonstrující absenci autentizačních a autorizačních mechanismů

Pro tuto zranitelnost není potřeba aplikaci nějak rozšiřovat. Tato zranitelnost již v práci demonstrována byla na programu „GraphQL Hello World“, ilustrovaného na obrázku 5.1. V aplikaci není obsažen žádný autentizační mechanismus a položený dotaz neobsahuje žádné identifikační údaje. K tomu, aby byl uživatel autentizován zbývá pouze jedna možnost, a to ta, že autentizace proběhla na pozadí automaticky a je před uživatelem skryta. K nezvratnému důkazu o tom, že tomu tak není je nutné prohlédnout jednotlivé pakety komunikace, které proběhnou mezi nastartováním služby na serveru a prvním dotazem klienta, tedy všechny pakety mezi body, kdy autentizace očividně musela proběhnout. Tyto pakety jsem prohlédl programem wireshark, jsou ilustrovány na obrázku 5.2[11] a byly následující:

1. **Paket 100 v uvedené komunikaci5.2**
   Z úplného konce paketu, ve kterém je uvedeno „operationName”: IntrospectionQuery“ je zřejmé, že se jedná o paket funkcionality GraphQL introspection. Tato funkcionalita již byla v práci zmíněna3.3.4, slouží ke zmapování GraphQL schématu.

2. **Paket 101 v uvedené komunikaci5.2**
   Zřejmě jde o TCP ACK paket.

Je tedy zřejmé, že autentizace ani autorizace v komunikaci opravdu neprobíhá, zranitelnost je v aplikaci přítomna.



Obrázek 5.2 Počátek komunikace klienta s GraphQL serverem – paket prvního dotazu

[11] Označený paket je paket prvního mnou zaslaného dotazu. První paket komunikace klienta se serverem je v komunikaci až číslo 100 protože odposlech obsahuje předešlé pakety klienta, jenž se snažil připojit k nespuštěnému serveru. Krásné číslo 100 vzniklo úplnou náhodou

5.4.2 GraphQL CSRF

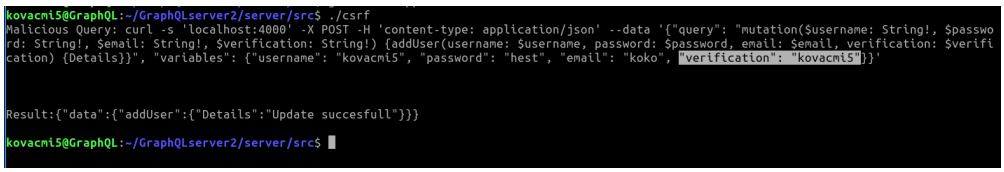
Pro nasimulování CSRF jsem vycházel ze slovníkové definice útoku – útočník zašle webový odkaz směřující na zranitelnou aplikaci, jenž donutí autentizovaného uživatele odeslat dotaz. Pro účely simulace jsem místo webového odkazu napsal miniprogram, s napevno zapsaným dotazem, jenž jakožto legitimní uživatel spustím. Tím simuluji webový link, který se ve slovníkovém CSRF dostane k uživateli. První část simulace, tedy způsob uživatele obdržení takového programu nechť úspěšně proběhla.

Tento program je ilustrován ve výpisu kódu 5.1. Ačkoliv je naprosto primitivní, rozhodl jsem se jej sem přiložit, protože naprosto totožným programem hodlám simulovat, zda CSRF funguje i přes korektně napsanou autentizaci. Pro úplnost je však k vidění již zde.

Výpis kódu 5.1 Simulace spuštění úspěšného phishing útoku za cílem CSRF

```
const char* maliciousQuery = "curl -s 'localhost:4000' -X POST -H
                    'content-type: application/json'
                    --data '{\"mutation\": \"{
                        addUser(username: "hacker",
                                password: "IamSoGood") {
                                details
                                }}\"}'";

std::array<char, 128> buffer;
std::string result("");
std::unique_ptr<FILE, decltype(&pclose)>
                pipe(popen(maliciousQuery, "r"), pclose);

while(nullptr != fgets(buffer.data(), buffer.size(), pipe.get()))
    result += buffer.data();
```

Výsledek je kromě návratové hodnoty možno ověřit tak, že se uživatel hacker zkusí přihlásit, jak je ilustrováno na obrázku 5.3. Je zjevné, že CSRF byl úspěšný, jelikož nový uživatel se přihlásil korektně, tedy aplikace je otevřená vůči CSRF.



Obrázek 5.3 Výsledek nasimulovaného CSRF

5.4.3 Errors information disclosure

K demonstraci tohoto útoku stačí zadat chybný dotaz vůči jakékoliv aplikaci a poté zhodnotit, jaké informace aplikace odhalí. Míra odhalených informací v mé aplikaci je k dispozici na obrázku 5.4. Aplikace opravdu odhaluje vše zmíněné v sekci 3.3 a navíc mnohem více, např. cestu k aplikaci uvnitř filesystému serveru.

Obrázek 5.4 Chybové hlášení GraphQL

5.4.4 Zranitelnosti vůči útokům DoS

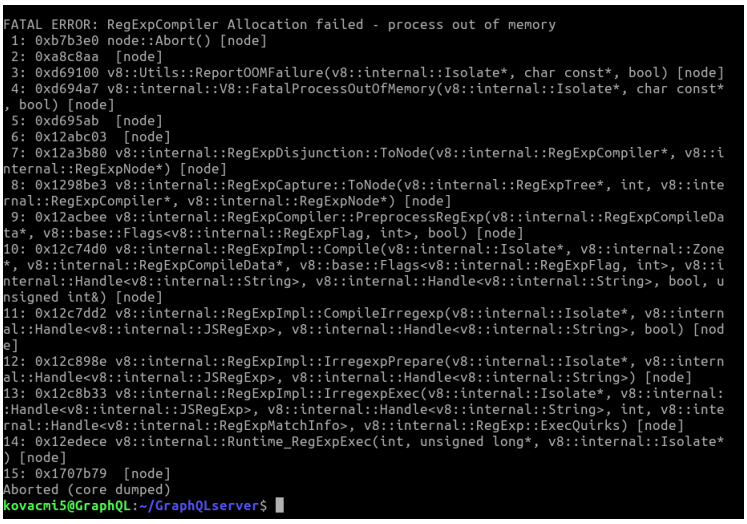
Na server GraphQL existují 2 specifické útoky typu DoS, viz. sekce 3.4 – batching attack a rekurzivní DoS. V této části budou popsány demonstrace obou těchto útoků.

Rekurzivní DoS využívá „rekurzivní“ povahy databázových záznamů GraphQL. V případě, kdy na sebe záznamy navzájem odkazují, je možné položit nekonečně dlouhý opakující se dotaz, jehož příklad je uvedený v sekci 3.4. K reprodukci této zranitelnosti bylo třeba vytvořit datový typ, který v sobě bude odkazovat v mém případě na knihu a zároveň, jenž bude odkazovaný knihou. Odpověď je zcela očividná – přidám do datasetu záznam autora, jemuž přiřadím jméno a jím napsané knihy. Poté již stačí najít hranici takového dotazu, který server přetíží. Na to jsem napsal program, jenž se postupně ptá na dotazy, kterým lineárně zvětšuje mohutnost a měří čas, za jaký server odpoví, viz. výpis kódu 5.2.

Výpis kódu 5.2 Simulace útoku rekurzivního DoS

```
std::string maliciousQuery = ("curl -s 'localhost:4000' -X POST -H
        'content-type: application/json' --data '{ \"query\": \"{");
    for(int i = 0; i < MAX_QUERY_DEPTH; ++i){
            maliciousQuery += "authors{posts";

            std::string toSend = maliciousQuery;
            toSend += "{id}";
            for(int j = 0; j < i + 1; ++j) {toSend += "}}";}
            std::cout << "sending: " << toSend + "\" }'" << std::endl;
            auto start = std::chrono::high_resolution_clock::now();
            SendQuery((toSend + "\" }'").c_str());
            auto end = std::chrono::high_resolution_clock::now();
            maliciousQuery += "{";
            std::cout << std::chrono::duration_cast\
    <std::chrono::microseconds>(end - start).count() << std::endl;
    }
```

Výsledné hodnoty jsem poté pro interpretaci plánoval znázornit grafem, ze kterého lze vyčíst, jak rychle vzrůstá doba odezvy serveru s narůstajícím dotazem. Tomu ovšem nebylo třeba, protože již v osmé iteraci cyklu programu 5.3 byl server položen pro nedostatek paměti, ilustrováno na obrázku 5.5. Doba odezvy se za tuto dobu nijak závažně nezvýšila. Nicméně DoS byl dosažen za pomocí přetečení kapacity paměti.



Obrázek 5.5 GraphQL server po dotazovém DoS

Batching attack je útok, kdy útočník odešle jeden nadměrně rozsáhlý dotaz, který server nedokáže zpracovat, viz. příklad s hvězdnými válkami popsaný v sekci 3.4. Abych tuto zranitelnost věrohodně reprodukoval, udělal jsem následující:

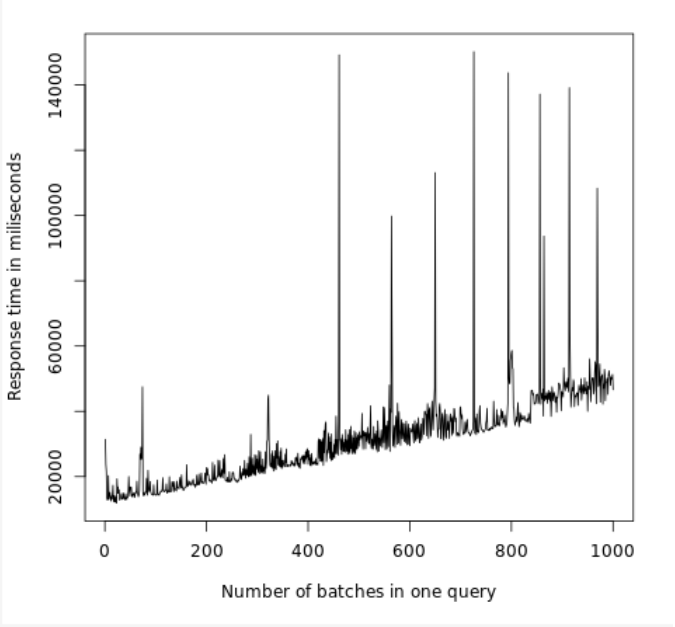
1. Přidal jsem datovému typu kniha identifikační údaj „ID“. Ačkoliv pro útok to není nutné (útok je plně možný provést pomocí jmen jednotlivých knih), napomáhám si tímto krokem k automatizaci a k demonstraci skutečné situace.

2. Přidal jsem do základního datasetu 1000 knih s originálními názvy „Title_x“ a autory „Author_x“, kde za x dosazuji ID jednotlivých knih.

Poté jsem odesílal dotazy takové, že první dotaz se tázal na první knihu a každý další dotaz se tázal na informace o jedné knize navíc oproti předchozímu dotazu. Pro ilustraci je automatizace tohoto postupu uvedena v kódu 5.3. V každém dotazu jsem poté měřil čas, jak dlouho trvalo serveru na dotaz odpovědět.

Hodnoty z programu jsem poté nechal vykreslit do grafu 5.6 pomocí jazyka R. Z grafu je jednoznačně vidět, že existuje nějaká forma přímé úměry mezi velikostí dotazu a odezvou serveru, jelikož má rostoucí tendenci. Také je možné vidět nějaké extrémy, které vysvětluji tím, že v tomto bodě jsem se začal dotazovat na nějaký blok dat, který server neměl ve skryté paměti a výchylka tedy značí overhead komunikace s hlavní pamětí.

K absolutnímu přetížení serveru jsem nedošel z několika důvodů. Největším důvodem je, že k demonstraci útoku není absolutní přetížení potřeba. Nyní je již dokázáno, že server je vytížen velkými dotazy více, než menšími, tudíž útok funguje. Druhým velkým důvodem je, že hranice, kdy dojde k úplnému DoS je velmi relativní vzhledem k cíli útočníka. Pokud by útočník chtěl způsobit DoS, musel by uvažovat na jak dlouho chce server zneškodnit (čím větší čas, tím větší dotaz), výkon serveru, nutnost využití více zdrojů dotazů (DDoS) atd. Z provedeného experimentu je zřejmé, že útok funguje a tím pádem považuji cíl za splněný.



Obrázek 5.6 Graf závislosti rychlosti odpovědi na velikosti dotazu

Výpis kódu 5.3 Simulace dotazového DoS

```
const char* maliciousQuery = "curl 'localhost:4000' -X POST -H
                    'content-type: application/json'
                    --data '{\"query\": \"{";


bool CreateComma = false;
for(int i = 0; i < NUM_OF_BOOKS; ++i){
    if(CreateComma)
        maliciousQuery += ","
    maliciousQuery += "books(id: \"" = to_String(i) + "\") {title}"

    auto start = high_resolution_clock::now();
    SendQuery(maliciousQuery + "}\"}'");
    auto stop = high_resolution_clock::now();
    std::cout << "duration_cast<seconds>(stop - start)" << std::endl;

    CreateComma = true;
}
```

Kapitola 6

Tvorba prostředí pro demonstraci a testování automatizovaných snížení rizik zranitelností gRPC

Tato kapitola obsahuje popis tvorby testovacího prostředí pro metodu vzdálené komunikace gRPC. Konkrétně obsahuje tvorbu jednoduché aplikace, na níž budu demonstrovat jak jednotlivé zranitelnosti, tak efektivitu snížení rizika pomocí mnou vytvořeného middleware. Daná aplikace, mnou vytvořená, byla v souladu se zadáním práce schválena jejím vedoucím.

6.1 Tvorba virtuálního prostředí

Tvorba virtuálního testovacího prostředí, kde vytvářím virtuální server je stejná, jako při tvorbě prostředí GraphQL v kapitole 5. Budou použity stejné nástroje, stejné OS se stejnými parametry a předpoklady.

6.2 Popis demonstrační aplikace

Aplikace vytvořená pro účely této práce bude představovat zjednodušenou formu vzdáleného úložiště. Jejím účelem bude dovolit uživatelům se připojit a poté provádět dotazy nad databází. Konkrétně bude obsahovat následující metody:

- **Test funkčnosti spojení**
  K účelům demonstrace komunikace skrz nešifrovaný kanál ze sekce 4.1 do aplikaci přidám jednoduchou metodu, která přijme od uživatele zprávu obsahující jeden textový řetězec, který následně vypíše na obrazovku a pošle jí zpět. Tato metoda simuluje metodu troubleshootingu, kterou by administrátor volal při testování aplikace. Pokud bude komunikace probíhat opravdu bez šifrování, bude textový řetězec jednoznačně viditelný v přenášených paketech

- **Metody operující nad databází**
  Pro účely testování obou zbývajících zranitelností (Protobuf Information Disclosure a SQL injection ze sekcí 4.2 a 4.3) se velmi hodí databázové operace, obzvlášť dotaz *SELECT*. Vzhledem k povaze této zranitelné aplikace, si tento bod zjednoduším. Zaprvé, pro důkaz zranitelnosti ani jednou ze zbývajících zranitelností není potřeba implementovat všechny databázové metody - stačí již zmíněný *SELECT*. Zranitelná aplikace tedy nic jiného umět nebude. Také

napíšu dvě metody, které obě budou *SELECT* volat, každá z nich bude demonstrovat jednu ze dvou zranitelností.

Aplikace tedy bude obsahovat tyto tři metody. V reálném světě by samozřejmě bylo i primitivní domácí úložiště podstatně složitější, ale pro demonstraci bude výsledná aplikace více než dostačující. Aplikace dále bude obsahovat primitivní formu autentizace a autorizace uživatele, jakožto důkaz, že metody jsou zranitelné i přesto, že jsou autentizované.

6.3 Instalace, konfigurace a spuštění gRPC serveru

K instalaci a zprovoznění frameworku gRPC jsem postupoval podle oficiální dokumentace[1]. Celý tento proces je automatizován v podobě skriptu uložené na médiu, které je přiloženo k práci. Konkrétně bylo třeba následujících kroků:

1. **Vytvoření pracovního prostředí**
   cd $HOME
   export MY_INSTALL_DIR=$HOME/.local
   mkdir -p $MY_INSTALL_DIR
   export PATH="$MY_INSTALL_DIR/bin:$PATH"

2. **Installace jednotlivých prerekvizit**
   sudo apt install -y build-essential autoconf libtool pkg-config cmake

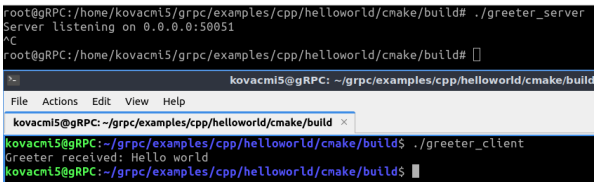
   - **build-essential** je souhrnný název balíčku základních nástrojů pro kompilaci (gcc, gdb...)[2]
   - **autoconf** je nástroj pro tvorbu konfiguračních skriptů pro některé tipy software [3]
   - **libtool** je knihovna, která maskuje komplexitu používání sdílených knihoven pod jedno rozhraní [4]
   - **pkg-config** je nástroj usnadňující kompilaci [5]
   - **cmake** je nástroj taktéž pro kompilaci [6]

3. **Naklonování repozitáře gRPC z gitlabu**
   git clone --recurse-submodules -b v1.53.0 --depth 1 --shallow-submodules https://github.com/grpc/grpc

4. **Lokální instalace gRPC**

```
cd $WORK_DIR
mkdir -p cmake/build
pushd cmake/build
cmake -DgRPC_INSTALL=ON \
    -DgRPC_BUILD_TESTS=OFF \
    -DCMAKE_INSTALL_PREFIX=$WORK_DIR \
    ../..
make -j 4
make install
popd
```

[1] K dispozici zde: `https://grpc.io/docs/languages/cpp/quickstart/`
[2] `https://linuxhint.com/install-build-essential-ubuntu/`
[3] `https://www.gnu.org/software/autoconf/`
[4] `https://www.gnu.org/software/libtool/`
[5] `https://www.freedesktop.org/wiki/Software/pkg-config/`
[6] `https://cmake.org/`



Obrázek 6.1 gRPC Hello World

Po výše uvedených krocích je již možné tvořit programy využívající gRPC, (např. Hello world, viz. obrázek 6.1[7]). Od tohoto bodu budu dále postupovat pro tvorbu aplikace popsanou v sekci 6.2. Celá aplikace, včetně zdrojových kódů klienta je vložena na přenosné médium přiložené k práci.

6.4 Tvorba výsledné zranitelné aplikace a demonstrace zranitelností

Tato sekce obsahuje praktický postup, pomocí kterého jsem vytvořil demonstrativní aplikaci popsanou v sekci 6.2. Také obsahuje postup demonstrace jednotlivých zranitelností.

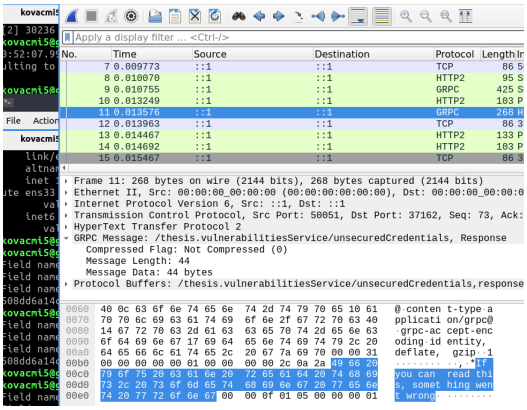
6.4.1 Metoda demonstrující komunikaci nezabezpečeným kanálem

Pro demonstraci této zranitelnosti by ani nebylo potřeba vytvářet zvláštní metodu, ale pro účely jednoznačnosti je to dobrý nápad. Tento primitivní kus kódu je ilustrován ve výpisu 6.1. Využití této metody pro odhalení, zda mezi sebou server a klient komunikuje pomocí nešifrovaného kanálu je jednoznačné. Klient odešle jemu známou zprávu a poté se podívá do packet snifferu, jestli tuto zprávu přečte. V mém případě jsem využil wireshark a úspěšně zachytil zprávu *If you can read this, something went wrong*, ilustrováno na obrázku 6.2.

Výpis kódu 6.1 Kód metody demonstrující komunikaci nezabezpečeným kanálem

```
        std::string text(msg->text());
        std::cout << "Recieved message: " << text << std::endl;
        rpl->set_text(text);
        return Status::OK;
```

[7] V programu klient (využívající spodní terminál) odešle zprávu obsahující libovolný text (editovatelný z kódu) a server mu nazpět odešle zprávu obsahující „Greeter recieved: + client_string“. Následné vypnutí serveru s programem nesouvisí.

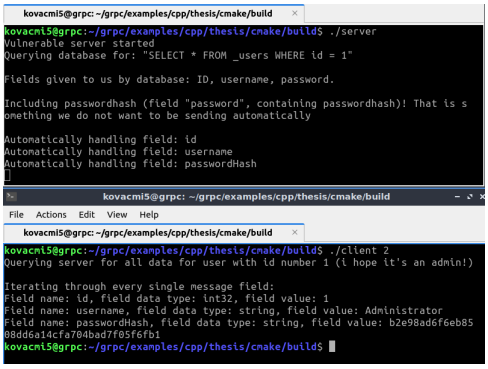


Obrázek 6.2 Zranitelnost komunikace nezašifrovaným kanálem

6.4.2 Metoda demonstrující information disclosure pomocí souboru protobuf

Aby byla demonstrovatelná tato zranitelnost, je nutné ukázat, jak jednoduché je vložit citlivý údaj z databáze do nevhodně definované zprávy v souboru protobuf. Pro tento účel byla vytvořena zpráva obsahující položky *ID*, *uživatelské jméno* a *heslo*. Stejné položky jsou zavedeny do tabulky *_users* v lokální MySQL databázi. Dané databázi je položen dotaz týkající se všech uživatelských položek, které jsou následně automatizovaně vloženy na příslušná místa do zprávy odpovědi. Tento postup simuluje jev, kdy programátor chtěl vytvořit co nejautomatizovanější řešení reakce na vzdálený *SELECT* bez toho, aby dbal na bezpečnost. Je snadno rozšiřitelný a relativně universální, používá-li se s rozmyslem. Výsledná metoda je ilustrována ve výpisu kódu 6.2.

K automatizaci naplnění zprávy využívám descriptory a obrazy zpráv, oficiální konstrukce gRPC určené mimo jiné právě pro tento účel. Jejich dokumentace je k dispozici v dokumentaci gRPC[8]. Uvedený kód 6.2 je pouze ilustrační, jeho plná verzi je k dispozici na přenosném médiu přiloženého k této práci. Jeho výsledek je ilustrován na obrázku 6.3, kdy se klientovi úspěšně podařilo zjistit SHA-1 hash hesla administrátora serveru.



Obrázek 6.3 Zranitelnost protobuf file information disclosure

[8] https://protobuf.dev/reference/cpp/api-docs/google.protobuf.descriptor/

Výpis kódu 6.2 Příklad automatizace naplňování zpráv v gRPC

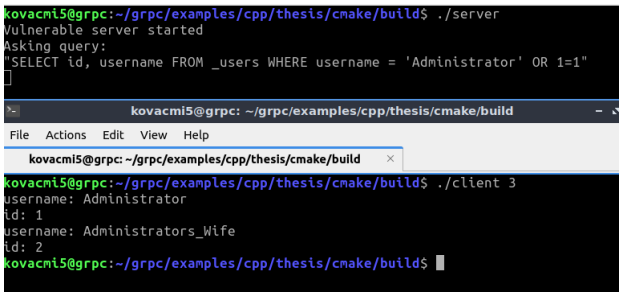
```
//grpc message descriptor and reflection
const Descriptor* desc = rpl->GetDescriptor();
const Reflection* refl = rpl->GetReflection();

std::string sPlaceholder("");
int iPlaceholder = -1;

//foreach field
for(int i = 0; i < desc->field_count(); ++i){
    const FieldDescriptor* field = desc->field(i);

    if(field->is_repeated())
        continue;

    result >> sPlaceholder;

    //get field type for proper parsing
    //and set[string/int] the field value
    switch(field->type()){
      case FieldDescriptor::TYPE_STRING:
        refl->SetString(rpl, field, std::move(sPlaceholder));
        break;
      case FieldDescriptor::TYPE_INT32:
        iPlaceholder = atoi(sPlaceholder.c_str());
        refl->SetInt32(rpl, field, std::move(iPlaceholder));
        break;
    }
}
```

6.4.3 Metoda demonstrující SQL injection

Pro demonstraci SQL injection jsem vytvořil co nejnaivnější možnou metodu. Tato metoda si připraví jednoduchý *SELECT*, který doplní vstupem od uživatele. Výsledek přeparsuje do zprávy pro klienta. Jelikož těchto výsledků může být více, tato metoda nebude posílat v odpovědi jednu zprávu, ale tzv. „message stream“, což je gRPC paralela s C++ streamy, pomocí které se dá odeslat libovolný počet zpráv. Kód této metody, ochuzen o detaily způsobu integrace programu s databází je ilustrovaný ve výpisu 6.3. K demonstraci této zranitelnosti je potřeba, aby zlomyslný klient zadal takový vstup, jenž databáze vyhodnotí jako kód místo dat. V mém případě jsem zvolil klasický „*OR 1=1*“, viz. obrázek 6.4.

Výpis kódu 6.3 gRPC SQL injection v praxi

```
std::string query = "SELECT id, username FROM _users
                     WHERE username = " + msg->username;

std::stringstream result(executeQuery(query.c_str()));
int id = -1; std::string user("");

while(result >> id){        //parse output, send all to client
    result >> user;
    sqlInjectionReply rpl;
    rpl.set_id(id); rpl.set_username(user);
    writer->Write(rpl); }
```



Obrázek 6.4 gRPC SQL injection

Existuje mnoho možností, jak otevřít svou zranitelnosti vůči útoku SQL injection. Jedna z těchto možností je uvedený *SELECT*, ale tento útok je možný se všemi dotazy, do kterých se dostává klientský vstup.

Kapitola 7

Tvorba knihovny pro automatizované opravy zranitelnosti GraphQL

V této kapitole je popsána autorem vytvořená knihovna k opravám diskutovaných zranitelností GraphQL. Tato knihovna slouží jako middleware mezi uživatelským kódem a GraphQL, ale její jednotlivé funkce lze zahrnout přímo do cílové aplikace. Kapitola také obsahuje demonstraci funkčnosti této knihovny pomocí testů diskutovaných s vedoucím práce.

Před popisem jednotlivých metod automatizovaných snížení rizik je úvodní část kapitoly věnována možnostem importu a využívaní výsledných knihoven, jelikož některé části jsou vyžadují speciální přístup využívající nástroje GraphQL. Knihovny obsahují mimo jiné rozšíření schématu, resolverů, kontext serveru (pojmy jsou popsány v sekci 2.1.1)..., které nestačí zakomponovat standardními JavaScriptovými metodami.

K zakomponování rozšíření schématu je nezbytné sjednotit 2 schémata, jelikož konstruktor některých GraphQL serverů (např. mnou využívaného Apollo Serveru) neobsahuje přetížení pro vícero schémat. Kromě ručního způsobu kopírování je vývojářům k dispozici nástroj *mergeTypeDefs* z knihovny *@graphql-tools*[1]. Tento nástroj poskytuje funkci přímo pro účel spojování schémat. Pro ilustraci uvádím praktický příklad ve formě výpisu 7.1.

Výpis kódu 7.1 Ukázka spojování schémat

```
const typeDefs1 = gql‘
    type Car = {Model: String!, HorsePower: Int!}
‘;

const typeDefs2 = gql‘
    type Motorbike = {Model: String!, HorsePower: Int!}
‘;

const typeDefs = mergeTypeDefs({typeDefs1, typeDefs2});

const server = new ApolloServer {
    schema: typeDefs, ...
}
```

[1] https://the-guild.dev/graphql/tools

Rozšíření resolverů staví vývojáře před stejný problém, jako je tomu se schématem. Tento problém má i stejné řešení – funkci *mergeResolvers* z knihovny *@graphql-tools*. Funguje naprosto stejně, jako v případě schématu.

Kontextová funkce je ve své podstatě zcela obyčejná funkce, jejíž návratová hodnota je využívána pro speciální účely. Není se tedy čemu divit, že při své rešerši jsem nenašel žádný nástroj pro spojování kontextových funkcí. Není ovšem nic jednoduššího, než udělat rozšíření ručně pomocí spojení návratových hodnot z většího množství funkcí, jak je ilustrováno ve výpisu kódu 7.2. Takto definovaný kontext poté stačí rozšířit o položky „tmp“ a pomocí jednoho řádku je problém vyřešen.

Výpis kódu 7.2 Ukázka ručního rozšíření kontextové funkce v GraphQL

```
const server = new ApolloServer {
    typeDefs, resolvers,
    context: async ({ req }) => ({
        const tmp = [];
        tmp[0] = foo();
        tmp[1] = bar();
        ...

        const ret = {};
        for(const element in tmp){
            ret = Object.assign(element);
        }
        return ret;
    })
}
```

7.1 Autentizace

Universální autentizaci pro aplikace využívající GraphQL jsem vytvořil za využití JWT, jak již bylo zmíněno v sekci 3.1.3.1. Primární částí funkcionality jsou tři mutace a kontextová funkce.

Mutace registrace obsahuje v parametrech uživatelské jméno, heslo, email a další méně podstatné údaje o uživateli. Tato funkce uživatele nezakládá, jelikož tím by otevřela vektor pro útok nekonečně registrací[2]. Místo toho si zanese informaci do konstrukce dočasných uživatelů a vrátí uživatelské údaje zašifrované svým soukromým klíčem („svým“ myšleno klíčem serveru), čímž je simulován verifikační email. Verifikační email se přímo netýká GraphQL a z časových důvodů jej musím pouze takto nasimulovat, v reálné aplikaci by na email uživatele odeslal odkaz ve tvaru *https://www.graphQLServer/queryverify(verifyString)Details*, kde verifyString je řetězec, jenž má funkce vrací. Dočasný uživatelé by v reálné aplikaci byly po určitém časovém intervalu nečinnosti smazáni.

Mutace ověření přijímá tento verifikační řetězec popsaný výše, dešifruje jej svým veřejným klíčem a poté v dočasných uživatelích hledá adekvátního uživatele. Pokud jej najde, přesune ho do skupiny regulérních uživatelů a bude smaže z dočasných uživatelů.

Mutace login přijímá uživatelské jméno a heslo v zahashované formě. Pokud uživatel existuje a heslo je v pořádku, vrátí adekvátní JWT vytvořený opět pomocí soukromého klíče serveru. Uživatel je poté povinen odeslat tento JWT v každém svém požadavku (v hlavičce v sekci autorizace), aby server daný požadavek zpracoval.

[2]Tento útok jsem sám pojmenoval pro efekt, oficiálně žádné jméno nemá. Funguje tak, že útočník za pomocí skriptu odesílajícího registrační formy způsobí nedostatek paměti na serveru pro legitimní uživatele

7.1.1 Testování metody

Pro testování autentizace uvádí OWASP doporučené postupy[3], podle kterých se budu řídit. Tyto postupy radí následující:

1. **Kontrola, zda uživatelské údaje neputují skrz nešifrovaný kanál**
   V části práce týkající se GraphQL se šifrovaným spojením nezabývám, jelikož se nejedná o častou, ani známou zranitelnost GraphQL a tedy je mimo rámec práce. Každopádně předpokládám, že celá komunikace bude v praxi šifrována skrz SSL/TLS, tedy že převážně heslo bude šifrované pomocí veřejného klíče serveru a pro externí entitu se stane nerozluštitelným.

   Ve většině případů je způsob přenášení hesel pomocí TLS/SSL považován za bezpečný[4], vyjma proti útoků MiM (Man in the Middle), kdy útočník již od začátku odchytne komunikaci, podstrčí uživateli nesprávný certifikát a tím efektivně odposlouchává celou komunikaci. Tomu se lze vyhnout pomocí dvojitého hashování - jak na straně klienta, tak na straně serveru, heslo tak bude uloženo zahashované dvakrát. Ve své implementaci předpokládám zahashashované heslo ve fázi pouze ve fázi přihlášení, jelikož při registraci musím vyžadovat jistou složitost hesla, což je důležitější, než předpokládat nepravděpodobný MiM útok právě ve fázi registrace. Důvěrnost JWT používaného po přihlášení je také zajištěna pomocí SSL/TLS. Navíc je do JWT přimíchaný náhodný řetězec, uložený na straně serveru pro účely znemožnění jeho reprodukce.

2. **Kontrola slabého zamykacího mechanismu**
   Zamykací mechanismus využívaný v aplikaci zajišťuje, že po pěti nesprávných pokusech zadání hesla je uživateli nastaven znak zablokování, který může odblokovat jenom admin. Tímto dle OWASPu je znemožněn útok hrubou silou. Pro normální aplikace je tento mechanismus kanón na vrabce a stačilo by po pátém neúspěšném pokusu o přihlášení odeslat uživateli email.

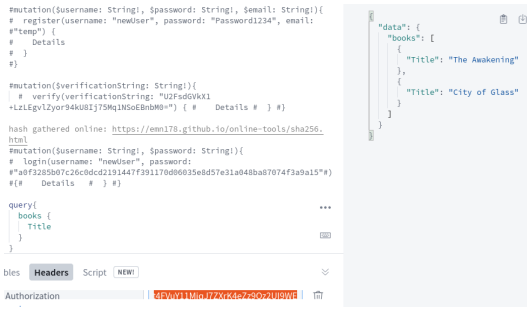
3. **Kontrola obcházení autentizačního schématu**
   Pro obejití autentizačního schématu v mé aplikaci je nezbytnou podmínkou aby vývojář zapomněl přiřadit funkcím ochranu pomocí knihovny graphql-shield (probíráno v sekci 7.2), nebo vytvořil mutaci, která nekontroluje oprávnění dat. Při korektním využívání mého rozšíření nenašli mnou provedené testy žádnou možnost pro obejití autentizačního schématu.

4. **Kontrola zranitelné funkcionality ukládání hesel**
   Ve svém schématu hesla ukládám ve formě 2* zahashovaná bezpečným hashovacím algoritmem.

5. **Testování zranitelnosti ukládání dat na straně klienta**
   Jediné, co se v mém případě může uložit do uživatelovy historie je heslo. Částečně zajišťuje bezpečnost fakt, že při přihlášení odesílám heslo již zahashované. Při registraci musí ovšem klient volat funkci *resetStore()*, pomocí které zaručeně smaže všechny stopy po hesle. Server má možnosti, jak donutit klienta po registraci tuto funkci zavolat pomocí přenesených JavaScriptových serverů.

[3] `https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/04-Authentication_Testing/README`

[4] Např. podle `https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html`



Obrázek 7.1 Autentizace v GraphQL od začátku

6. **Kontrola politiky hesel**
   Politika vytváření hesel je dělaná dle OWASPU, tudíž testem OWASPu očekávaně prošla. Konkrétně heslo musí být 8 znaků dlouhé, a obsahovat 3 ze 4 znakových tipů: malé písmeno, velké písmeno, číslo, speciální znak.

7. **Kontrola slabé funkcionality změny hesla**
   Jediná implementovaná funkcionalita změny hesla je změna hesla pro přihlášené uživatele. Tato funkcionalita se řídí doporučeními OWASPU: není možné specifikovat ID jiného uživatele, uživatel zadává své staré heslo, nové heslo se řídí politikou hesel, a funkcionalita není zranitelná vůči CSRF, jelikož kontroluje autorizační hlavičku (dokonce dvakrát, jednou pro splnění podmínek graphql-shield a podruhé pro ujištění, že uživatel mění svoje heslo, nikoliv cizí).

8. **Kontroly, jenž se na mou aplikaci nevztahují**
   Neimplementuji vícefaktorovou autentizaci, alternativní kanál, výchozí přihlašovací údaje, ani bezpečnostní otázky pro zapomenuté heslo.

Všechny výše uvedené body byly otestovány po vzoru předmětu BI-EHA vyučovaném na FIT ČVUT a nebyly nalezeny žádné nedostatky. Na obrázku **??** je ilustrován proces testování metod na funkčnost metod, politiku hesel a obcházení autorizace metod.

7.2 Autorizace

Autorizace popsaná v sekci 7.2, popisuje autorizaci k přístupu k datům a autorizaci k volání funkcí. Tato sekce shrnuje oba tyto typy.

Autorizace k datům probíhá za pomoci rolí. Uživatelům je přiřazen role v podobě textového řetězce. Tato část bohužel není možná univerzálně automatizovat, jelikož uživatelé mohou být uloženi v mnoha různých formách. Objektům je přiřazeno pole oprávnění v podobě rozšíření schématu o typ *Permissions*, které lze přiřadit jednotlivým objektům. Oprávnění obsahuje 2 položky - na koho se vztahuje (konkrétního uživatele, případně uživatelskou roli) a příznaky RW (read, write). Jednotlivé dotazy i mutace nyní lze rozšířit dvěma způsoby – přístup k nim lze řídit pomocí GraphQL shield, který zkontroluje roli uživatele a zda má k funkci přístup. Pomocí této role lze také zkontrolovat všechna oprávnění objektů a rozhodnout se, zda přístup k objektu povolit, či zamítnout.

7.2.1 Testování metody

Stejně jako pro autentizaci obsahuje OWASP doporučené postupy testování bezpečnost autorizace[5], jimiž se při testování budu řídit:

[5] `https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/05-Authorization_Testing/README`

Obrázek 7.2 Autorizace funkcí s graphql-shield pro neautentizované a tedy neautorizované

1. **Kontrola obcházení autorizačního schématu**
   Při korektnímu použití rolí a pravidel graphql-shield není možné přistoupit k datům neautentizovaný (díky kontrole JWT), ani bez role (na což je graphql-shield přímo vytvořený). Aby se uživatel mohl odhlásit, je potřeba vytvořit ještě jednu konstrukci, která zaznamenává platnost odesílaných JWT. Právě do této konstrukce je ukládán náhodný řetězec využívaný při podpisu JWT za účelem randomizace.

2. **Kontrola eskalace privilegií**
   Pouze role administrátora má přístup k funkcím ke změně privilegií.

3. **Kontrola IDOR útokům**
   Mnou vytvořená oprávnění se přiřazují jednotlivým objektům a tedy korektně vytvořená aplikace není zranitelná vůči útokům IDOR.

4. **Kontroly, jenž se na mou aplikaci nevztahují**
   Kontrola directory traversal se aplikací se vstupem GraphQL netýkají, jelikož GraphQL je přímo dělaný pro účel jednotného vstupu do aplikace, viz sekce 2.1. Aplikace také nevyužívá framework OAuth2.

Všechny výše uvedené body byly otestovány po vzoru předmětu BI-EHA vyučovaném na FIT ČVUT a nebyly nalezeny žádné nedostatky. Přikládám obrázek 7.2 primitivního využití GraphQL shield se základním chybovým hlášením.

7.3 Metoda pro automatizované snížení rizika GraphQL CSRF

Metoda pro automatizované snížení rizika GraphQL CSRF, zmíněná v sekci 3.2.2 je v této části již implementovaná. Spočívá v tom, že útočník nemůže vytvořit plnohodnotný dotaz pro jiného uživatele, bez znalosti detailu ohledně hodnoty autorizační hlavičky daného uživatele. Ta je utvořená zčásti pomocí soukromého klíče serveru, tudíž pokud lze reprodukovat, má aplikace větší problém.

7.3.1 Testování metody

V tomto případě OWASP přímo nespecifikuje, jak útok testovat, ale radí, jak se zranitelnosti vyhnout[6]. Na začátku tohoto návodu popisu nutné podmínky pro vznik této zranitelnosti. Druhý tento bod je znalost validních URL pro dotazy uživatele, které dle předchozí sekce nemá. Testování proběhlo za pomocí stejného skriptu, jako testování zranitelnosti za zranitelném serveru (viz výpis kódu 5.1). Výsledek je ilustrován na obrázku 7.3.

[6] https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/06-Session_Management_Testing/05-Testing_for_Cross_Site_Request_Forgery

```
kovacmi5@GraphQL:~/GraphQLserver2/server/src$ ./csrf
Malicious Query: curl -s 'localhost:4000' -X POST -H 'content-type: applicati
on/json' --data '{"query": "mutation($username: String!, $password: String!,
$email: String!, $verification: String!) {addUser(username: $username, passwo
rd: $password, email: $email, verification: $verification) {Details}}", "vari
ables": {"username": "kovacmi5", "password": "hest", "email": "koko", "verifi
cation": "kovacmi5"}}'



Result:{"errors":[{"message":"Not Authorised!","locations":[{"line":1,"column
":92}],"path":["addUser"],"extensions":{"code":"INTERNAL_SERVER_ERROR","except
tion":{"stacktrace":["Error: Not Authorised!","    at normalizeOptions (/home
/kovacmi5/GraphQLserver2/server/node_modules/graphql-shield/cjs/shield.js:27:
52)","    at shield (/home/kovacmi5/GraphQLserver2/server/node_modules/graphq
l-shield/cjs/shield.js:40:31)","    at Object.<anonymous> (/home/kovacmi5/Gra
phQLserver2/server/src/index.js:279:21)","    at Module._compile (node:intern
al/modules/cjs/loader:1196:14)","    at Object.Module._extensions..js (node:i
nternal/modules/cjs/loader:1250:10)","    at Module.load (node:internal/modul
es/cjs/loader:1074:32)","    at Function.Module._load (node:internal/modules/
cjs/loader:909:12)","    at Function.executeUserEntryPoint [as runMain] (node
:internal/modules/run_main:81:12)","    at node:internal/main/run_main_module
:22:47"]}}}],"data":null}
```

Obrázek 7.3 Výsledek testování CSRF

7.4 Metoda pro automatizované snížení rizika errors information disclosure

Metoda pro automatizované snížení rizika errors information disclosure, zmiňovaná v sekci 3.3 spočívá v uzpůsobení chybových hlášení.

Apollo GraphQL server k tomuto účelu poskytuje volitelný parametr v konstruktoru samotného serveru. Tento parametr se jmenuje *formatErrors* a jeho parametrem je funkce, jenž přijímá i vrací konstrukci s informacemi o chybě. V těle funkce se daná konstrukce dá pozměnit, v čemž také spočívá celá tato metoda. Pomocí sady podmínek jsem namapoval všechny chyby definované v officiální dokumentaci[7] na HTTP kódy následovně:

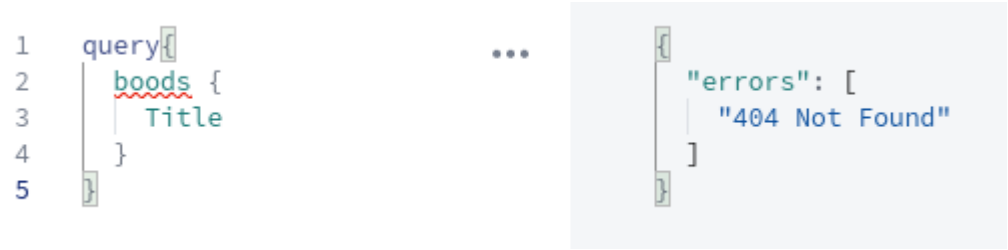
GRAPHQL_PARSE_FAILED :	400 Bad Request
GRAPHQL_VALIDATION_FAILED :	404 Not Found
BAD_USER_INPUT :	404 Not Found
UNAUTHENTICATED :	401 Unauthorized
FORBIDDEN :	403 Forbidden
PERSISTED_QUERY_NOT_FOUND :	404 Not Found
PERSISTED_QUERY_NOT_SUPPORTED :	503 Service Unavailable
INTERNAL_SERVER_ERROR :	500 Internal Server Error

Existence chyb *UNAUTHENTICATED* a *FORBIDDEN* naznačuje přítomnost nějaké autentizace. Není tomu bohužel tak – jde o chyby, jenž se uživateli vrátí v případě, že GraphQL slouží jako proxy jiné aplikaci, vůči které se uživatel neautentizoval. Jinou funkci než tuto dané chybové kódy nemají.

7.4.1 Testování metody

Tato metoda není příliš komplexní, co se testování týče. Parametr formatErrors zachytí chybu vždy a zbytek práce zařídí korektně vytvořené control flow. Fakt, že tento postup vrací chybu pozměněnou je ilustrováno na obrázku 7.4.

[7] https://www.apollographql.com/docs/apollo-server/v2/data/errors/



Obrázek 7.4 Ukázka přetvářky chyb v GraphQL

7.5 Metoda pro automatizované snížení rizika DoS

Metoda pro automatizované snížení rizika GraphQL, zmiňovaná v sekci 3.4 spočívá v zavedení Facebookové metody - přijímání pouze dotazů pomocí funkcionality persistentQueries.

Základem persistent queries je mapa mezi dotazy a jim přiřazenými ID. Tuto mapu lze vytvořit buďto ručně, nebo pomocí konsolového nástroje *persistentqraphql*[8]. Ten přetransformuje soubor s uloženými dotazy do tvaru požadované mapy. Tento nástroj se mi podařilo zprovoznit pouze pro soubory ve formátu .gql s obsahem v jejich adekvátním formátu. Ačkoliv podpora JavaScriptových souborů (i několik dalších tipů souborů) je implementována, pro demonstrační účely budu nástroj využívat následovně: *persistqueries file --gql --extension=gql*.

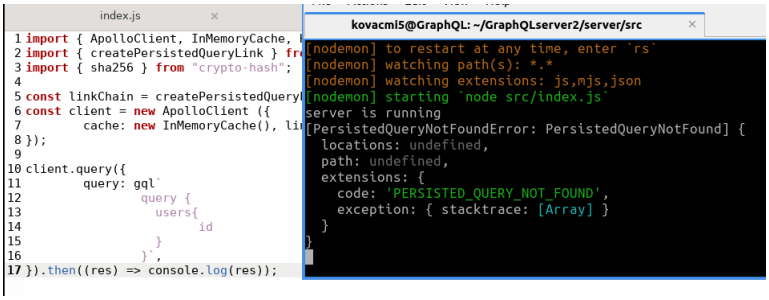
Po vytvoření této mapy zbývá přidat middleware, jenž při každém dotazu vyhledá v mapě daný otaz a následně jej provede, případně zamítne. Tento middleware má formu ilustrovanou ve výpisu 7.3.

V poslední části zbývá ukládat dotazy, pokud je server v režimu development. Jelikož middleware serveru vidí na jeho proměnné, není nic jednodušího, než k dosavadnímu řešení přidat podmínku, která rozhodne, jestli se do mapy kód uloží, nebo jestli se v mapě vyhledá. Abych nemusel využívat middleware podle express GraphQL serveru, vystačím se v implementaci s využitím kontextové funkce.

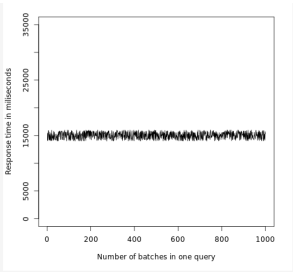
Výpis kódu 7.3 Facebooková metoda pro minimalizování zranitelnosti Graphql DoS

```
var app = express();
app.use (
'graphql',        //ui - in case of localhost, add full UI
(req) => {
    if(!req) {throw new Error("NO!")}
    if(req.body && req.body.id){ //check for persistentQuery ID
        const query = invertedQueryMap[req.body.id];
        if(query){  //success!
            req.body.query = query;
        }
        else {throw new Error("NO!");}
    }
    else {throw new Error("NO");}

    req.next();
})

...

server.addMiddleware(app);
```

[8] https://github.com/apollographql/persistgraphql



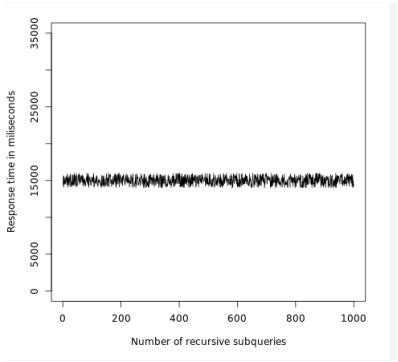
Obrázek 7.5 Odmítnutí nezmapované persistent queries



Obrázek 7.6 Měření odezvy serveru na batching attack

7.5.1 Testování metody

Pro otestování této metody je třeba ověřit si, zda GraphQL nezaznamenané dotazy skutečně odmítne (viz. obrázek 7.5, na kterém je ilustrován běh serveru a část kódu klienta s dotazem, který nebyl namapován) a zda je odmítne dostatečně rychle. Pro tyto účely jsem opět spustil své původní skripty, kterými jsem DoS testoval v sekci 5.4.4. Výsledky jsem zaznamenal do grafů pomocí jazyka R, ilustrovaných na obrázcích 7.6 a 7.7, z nihž je jasně vidět, že tímto způsobem již GraphQL přetížít server není možné. Grafy vypadají velice podobně – nejnižší naměřená hodnota odezvy v obou z nich je 14027 a nejvyšší 16403, což je oba činí velmi plochými. Měřítko jsem zvolil tak, aby bylo jasné, že oproti původních výsledkům na nezabezpečeném serveru (viz. obrázek 5.5) se jedná o velké zlepšení. Hodnoty odezev jsou jen o řád větší, než hodnoty „odezvy“ offline serveru.



Obrázek 7.7 Měření odezvy serveru na rekurzivní DoS

Kapitola 8

Tvorba knihovny pro automatizované opravy zranitelností gRPC

V této kapitole je popsána autorem vytvořená knihovna k opravám diskutovaných zranitelností gRPC. Tato knihovna slouží jako middleware mezi uživatelským kódem a gRPC, ale její jednotlivé funkce lze zahrnout přímo do cílové aplikace. Kapitola také obsahuje demonstraci funkčnosti této knihovny pomocí testů diskutovaných s vedoucím práce.

8.1 Metoda pro snížení rizika komunikace nešifrovaným kanálem

Jak již bylo v práci zmíněno v sekci 4.1.3, v implementaci nadepsané metody bude mým cílem zapouzdřit veškeré nezbytnosti pro donucení serveru k poslechu pouze na lince šifrované pomocí TLS/SSL. K tomuto účelu je potřeba funkční PKI.

PKI (public key infrastructure) je infrastruktura distribuce klíčů založená na asymetrické kryptografii. Při vzdálené komunikaci využívající PKI jsou stroje navzájem autentizované pomocí tzv. certifikátů. Certifikát je veřejný klíč, podepsaný důvěryhodnou třetí stranou – certifikační autoritou. Běžná internetová komunikace využívající protokol HTTPS využívá PKI např. k autentizaci webových serverů, což je vlastně součást problému, který řeším při tvorbě této metody.

Pro mé účely, lokální testování komunikace gRPC využívající TLS/SSL, si vytvořím vlastní miniverzi PKI přímo na serveru, kde probíhají testy. Pro třetí stranu bude nutné si certifikační autoritu buďto také vytvořit (což se může hodit např. pro tvorbu domácí síťové infrastruktury), nebo si vyžádat certifikáty od jakékoliv důvěryhodné certifikační autority[1] (což je pro nasazení aplikace na Internet tak či tak nezbytné). Postup tvorby PKI obsahuje ve větším detailu následující:

1. **Tvorba certifikační autority**
   K vytvoření certifikátů a klíčů v následujících krocích je třeba (důvěryhodné) certifikační autority. Pro své účely jsem si vytvořil novou, svou vlastní, nad kterou mám plnou kontrolu.

[1] např. od společnosti Digicert, `https://www.digicert.com/`

2. **Tvorba klíče a certifikátu serveru**
   Při tomto nejdůležitějším kroku je potřeba si vytvořit pár soukromého a veřejného klíče (podepsaného certifikační autoritou, tedy se jedná o certifikát) pro server. Jeho veřejný klíč budou klienti používat na šifrování zpráv, které si poté bude dešifrovat svým soukromým klíčem.

3. **Tvorba klíče a certifikátu klienta**
   Poslední krok je obdoba druhého kroku, ale tentokrát pro klienta. Klíč i certifikát slouží ke stejnému účelu, jako pro server.

Celý proces jsem udělal pomocí bashových příkazů využívajících knihovnu openssl[2]. Celý skript je k dispozici v příloze A a je k dispozici na přenosném médiu přiloženém k této práci.

Zbývá jen serveru i klientovi sdělit, že mají vytvořenou PKI využívat. K tomu využívá gRPC konstrukci „grpc::SslServerCredentialsOptions“[3], jenž se poté přikládá jako parametr do funkce sloužící k vytvoření spojení. Na straně serveru je tuto konstrukci nutné naplnit jeho certifikátem i soukromým klíčem a certifikátem certifikační autority. Na straně klienta je konstrukci potřeba naplnit jen certifikátem serveru. Poté stačí tuto konstrukci vložit do volání funkce pro zahájení spojení, ilustrováno ve výpisu kódu 8.1.

Výpis kódu 8.1 Ukázka využití TLS/SSL v gRPC

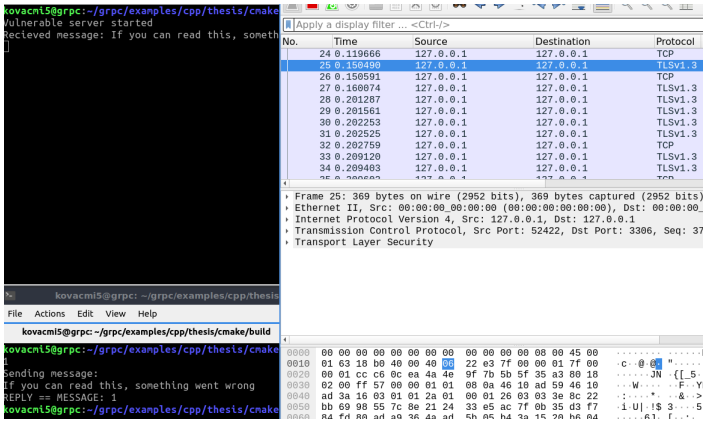
```
//server.cc:
        std::string serverCert = readFile("server.crt");
        std::string serverKey = readFile("server.key");

        grpc::SslServerCredentialsOptions sslOptions;
        sslOptions.pem_key_cert_pairs.push_back({serverKey, serverCert});
        sslOptions.pem_root_certs = readFile("ca.crt");

        serverBuilder.AddListeningPort(SERVER_ADDRESS,
                        grpc::SslServerCredentials(sslOptions));

//client.cc:
        grpc::SslCredentialsOptions sslOptions;
        sslOptions.pem_root_certs = readFile("server.crt")

        std::unique_ptr<vulnerabilitiesService::Stub> stub =
        vulnerabilitiesService::NewStub(grpc::CreateChannel(
        "SERVER_ADDRESS", grpc::SslCredentials(sslOptions)));
```

Výše uvedený kód je podstatně komplikovanější než kód připojení pomocí nešifrované komunikace. Aby programátor dbal na bezpečnost, nesmí být bezpečné řešení komplikovanější, než nebezpečné řešení. Celý proces otevření portu na kterém server začne bezpečně poslouchat zapouzdřuji do funkcí *bool AddSafeListeningPort(const char* serverAddress, const char* serverCrtFile, const char* serverKeyFile, const char* caCrtFile)*, resp. *bool AddSafeListeningPort(const char* serverAddress, const std::string& serverCrt, const std::string& serverKey, const std::string& caCrt)*, kde první funkce přijímá jako parametry cestu k souborům s nutnými informacemi a druhá funkce si tyto informace předává rovnou. Obdobně zapouzdřuji proces bezpečného připojení klienta.

8.1.1 Testování metody

Pro řádné otestování této opravy je nutné odpovědět následující otázky:

[2] https://www.openssl.org/
[3] https://grpc.github.io/grpc/cpp/structgrpc_1_1_ssl_server_credentials_options.html

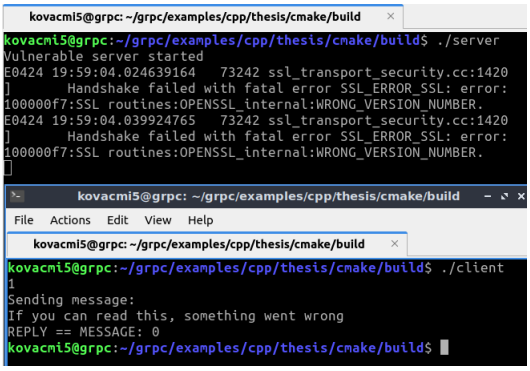


Obrázek 8.1 Šifrovaná komunikace se serverem gRPC

1. Je spojení skutečně šifrované?
2. Je spojení stále funkční? Rozumí si spolu klient i server i skrz TLS/SSL?
3. Je spojení šifrované vždy?
4. Odmítne server pokus o nešifrované spojení?

Pro odpovědi na první 2 otázky stačí využít jednu z funkcí ze zranitelného serveru (v mém případě funkci simulující testování připojení), a prohlédnout si komunikační pakety, ilustrováno na obrázku 8.1 (vrchní terminál představuje server, spodní terminál představuje klienta. Poslední řádek na terminálu klienta představuje true/false. Ve wiresharku jsou vidět pakety TLSv1.3, indikátor, že komunikace bude TLS využívat). Je jednoznačně vidět, že odpověď na první 2 otázky je ANO. Bezpečnost daného šifrování je zaručeno bezpečností protokolem TLS/SSL. Třetí otázku odpoví s jistotou pouze nekonečně mnoho pokusů. Jelikož na takový počet pokusů není čas, vystačil jsem si se stovkou pokusů, ze kterých byly šifrované všechny. Prohlašuji otázku za úspěšně zodpovězenou. V neposlední řadě by server měl odmítat připojení žádající o nešifrovanou komunikaci, což se také děje viz. obrázek 8.2.

Mnou vytvořená funkce tedy skutečně slouží jako jednoduchý interface pro bezpečné připojení, jenž je součást výsledku této práce.



Obrázek 8.2 gRPC odmítnutí nešifrované komunikace

8.2 Metoda pro snížení rizika protobuf information disclosure

Jak již bylo v práci zmíněno v sekci 4.2.3, řešením pro snížení tohoto rizika jsem zvolil funkci pro přeparsování souborů .proto s cílem najít nebezpečné položky zpráv. Abych si definoval, co je to nebezpečná položka, založil jsem konfigurační soubor, který obsahuje definice nebezpečných položek.

Tento konfigurační soubor je rozdělený na 2 části - část varující a část zakazující. Varující část obsahuje slova, jenž mohou potenciálně vést k information disclosure a zakazující část jsou slova, která k němu vedou vždy. Jednotlivá slova také mají předpony „contains“, nebo „match“. Contains naznačuje, že položky zprávy se nesmí jmenovat tak, aby jejich jméno bylo součástí slova po předponě. Match naznačuje, že že položky zprávy se nesmí jmenovat tak, aby jejich jméno bylo stejné jako slova po předponě. Mimo to také konfigurační server obsahuje část definující soubory .proto, které se budou kontrolovat a část definující, do jakého souboru se budou logovat události. Pro jednoduché pochopení přikládám příklad syntaxe tohoto konfiguračního souboru:

```
#set path to all your proto files here
proto:
    ./messages.proto

#set path to your log file
#!only 1 file per run allowed!
log:
    ./log.log

#catastrophic words
deny:
    match: password
    contains: pass

        ...

#potentially dangerous words
warn:
    contains: health

        ...
```

Obsah konfiguračního souboru byl inspirován službou sentry.io, konkrétně její schopností „čištění“ dat na mnoha různých serverech (tzv. „server side scrubbing“)[4].

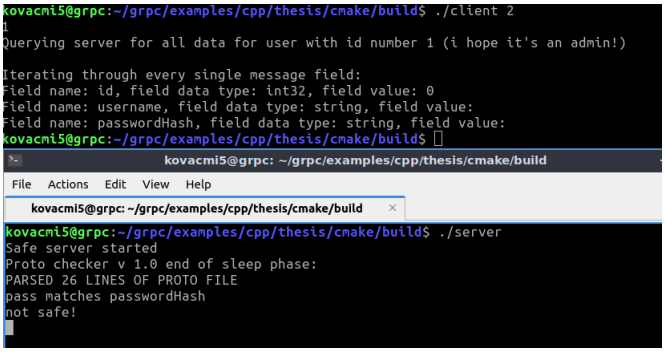
Zbytek funkce byl jednoduchý parser, který projde všechny .proto soubory definované v konfiguračním souboru a porovná je s hesly definovanými v konfiguračním souboru. Funkce poté vrátí int, jehož vrchních 16 bitů jsou definovány jako počet katastrofických hesel ve zprávách a spodních 16 bitů jsou definovány jako počet potenciálně nebezpečných slov ve zprávách.

8.2.1 Testování metody

Tato zranitelnost je v ohledu na její testování nejpřímočařejší. K otestování této opravy je nutné odpovědět defacto jen na jednu otázku: nalézá a zaznamenává funkce správně jednotlivé položky?

Po několika pokusech, kdy jsem funkci pouštěl nad různými soubory .proto a program vždy potenciálně nebezpečné heslo našel. Funkce obsahuje i implementaci normalizace dat, tedy např.

[4] https://docs.sentry.io/product/data-management-settings/scrubbing/server-side-scrubbing/



Obrázek 8.3 Metoda detekce nad zranitelným thesis.proto

množství whitespace neovlivní výsledek hledání. Ve všech příkladech tato funkce vrací korektní návratovou hodnotu, se kterou si poté programátor rozmyslí, jestli server pustit, nebo ne.

Nedostatek mnou provedených testů je ten, že mohu otestovat program pouze na datech ve různých tvarech, které vymyslím. Je možné, že existuje nějaká validní podoba zprávy, která mě nenapadne a tím pádem ji nemohu otestovat. Pokud ale soubor se zprávou není „záměrně obfuskovaný“, funkce pracuje korektně.

8.3 Metoda pro snížení rizika SQL injection

Jak již v práci bylo zmíněno 4.3.3, na obranu proti SQL injection byla zvolena metoda vytváření parametrických dotazů. Navíc, jak již také bylo zmíněno, bude oprava zaměřená jen na databázi mySQL, jelikož universálně je tato zranitelnost neopravitelná.

Aby parametrické dotazy byly uvnitř knihovny vůbec možné vytvořit, je nutné využít knihovny určené ke komunikací s databází. Pro mySQL k tomuto účelu slouží knihovna *mysqlcppconn*. Jelikož neexistuje žádný oficiální návod pro instalaci knihovny a jejímu linkování s programem na operačním systému Ubuntu, přikládám ho zde:

1. **Stáhnutí a instalace knihovny**
   sudo apt-get install lmysqlcppconn

2. **Využití korektních include maker**
   Include souboru mysql/jbdc, který by teoreticky měl obsahovat všechny soubory v balíčku lmysqlcppconn neexistuje. Do programu je potřeba naincludovat soubory mysql_driver.h, mysql_connection.h a soubory cppcon/*.

3. **Nalinkování k programu**
   Přepínače ke kompilaci jsou -L/usr/lib, -lmysqlcppconn. V grpc projektech se umístí do souboru cmake/build/CMakeFiles/server.dir/links.txt.

Pomocí funkce *prepareStatement(const char*)* uvnitř této knihovny je možné vytvářet parametrizované dotazy, pomocí kterých dá program vědět dopředu databázi, co se od ní bude chtít. Parametr funkce prepareStatement je dotaz, jehož některé vstupy jsou nahrazeny otazníky, které slouží jakožto znak indikující následné nahrazení hodnotou.

Využívání parametrizovaných dotazů je sice bezpečnější, než využívání „obyčejných“ dotazů, ale jak to tak bývá, programátorsky náročnější. Oproti obyčejným dotazům je potřeba přidat do kódu extra řádky, které se starají o nahrazování jednotlivých otazníků adekvátními hodnotami. K předejití zanesení programu zranitelností SQL injection jsou parametrizované dotazy obrovskou pomocí. Proto bude součástí mé práce funkcionalita, jenž zapouzdřuje parametrizované dotazy do jednoduché funkce, která se automaticky bude starat o bezpečnou formu dotazů.

Tato funkcionalita se skládá z více funkcí - jedna funkce na jeden tip dotazu (jedna funkce na *SELECT*, jiná funkce na *UPDATE*...). Jednotlivé funkce přijímají adekvátní parametry ke svým dotazům, tedy např. parametry dotazu *INSERT* jsou jméno tabulky, do které uživatel chce vložit, n-tice obsahující jména sloupců, které uživatel chce vyplnit a n-tice hodnot, jimiž chce naplnit vyplněné sloupce. Syntax této funkce následuje syntax dotazu *INSERT - INSERT INTO table_name (column_1, column_2 ...) VALUES (value_1, value_2 ...)*. Ostatní funkce přijímají parametry k nim adekvátní v pořadí syntaxe jejich dotazu.

Všechny tyto funkce mají podobný tvar. Nejdříve se z parametrů těchto funkcí postaví parametrizovaný dotaz, což jsem delegoval na třídu sloužící speciálně k tomuto účelu. Parametrizovaný dotaz vyrábím pomocí skládání řetězců a spoustu flow-control metod. Celou stavbu jsem naplánoval tak, aby všechny nepovinné parametry dotazů byly nepovinné i pro funkci (např. pokud cílový dotaz je *SELECT * FROM _users*, funkce lze zavolat jen se dvěma parametry bez starostí o klausuli *WHERE*). Pokud je cílem vynechat některé klausule z prostředka dotazu, je ovšem nutné odeslat prázdné parametry ((např. pokud cílový dotaz je *SELECT * FROM _users LIMIT 5*, funkci je nutné přidat dvě nultice imitující prázdnou klausuli *WHERE*).

Druhá část se skládá z předání parametrizovaného dotazu pomocí výrazu *prepareStatement(const char*)* a následné předávání v cyklu, což probíhá pomocí cyklů s vnořenými výrazy *statement-¿setString(i, value)*, kde i značí pořadí otazníku a hodnotu, kterou se daný otazník nahradí.

Zdůrazňuji, že mnou naprogramované funkce neslouží jako universální možnost administrativy celé databáze. Nejsou určeny ke správě databáze a nemají nahrazovat administrátorský přístup do databáze. Jejich využití se omezuje jako middleware mezi SQL databází a uživatelským vstupem přijatým z gRPC. Z tohoto důvodu jsem také vyrobil pouze 4 funkce pro uživatelem nejčastěji pokládané dotazy - *SELECT*, *INSERT*, *UPDATE* a *DELETE*.

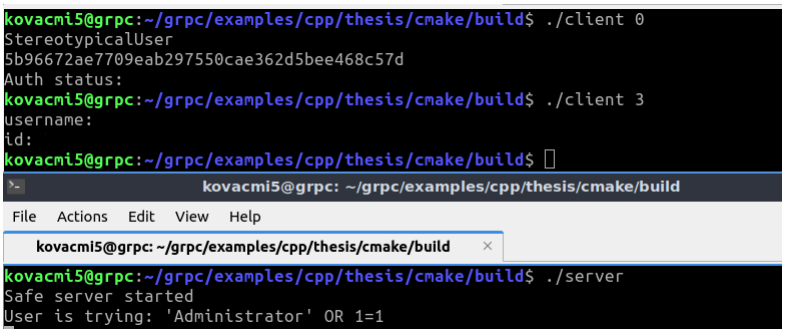
8.3.1 Testování metody

Pro otestování této metody je potřeba odpovědět na následující:

- Splňují funkce úděl dotazování databáze? Jsou jimi položené dotazy vždy funkční?
- Existuje nějaký způsob, jakým uživatel může do dotazu dostat kód?

Pro odpověď na první otázku je nutné otestovat všechny kombinace jednotlivých funkcí. Pokud všechny kombinace hodí správný výsledek, odpověď na první otázku je kladná. Samotné testování k práci přikládám pouze ve formě jedné ilustrace, jelikož kdybych do práce přidal všechny testy, následujících 5 stránek by obsahovalo pouze obrázky, které by v zásadě ani neměli přidanou hodnotu. Kombinace dotazů jsou následující:

- **SELECT** má 6 různých kombinací, díky nepovinným klauzulím *WHERE*, *ORDER BY* a *LIMIT*)
  Pro každou z těchto kombinací jsem položil několik dotazů, abych ověřil i funkčnost korektní stavby počátečního řetězce, např. umístění čárek. Funkce navíc odmítá nevalidní parametrizaci pomocí výjimky SQLException. Příklad funkčního SELECT je k vidění na obrázku 8.4.
- **INSERT** má jedinou kombinaci, tudíž pro testy jsem položil 3 dotazy, opět pro účel ověření stavby počátečního řetězce. Všechny fungovaly korektně.
- **UPDATE** má 2 kombinace. První, ne zcela očividná vynechává klausuli *WHERE*, tedy může se tázat např. na dotaz *UPDATE _users SET username='kovacmi5'*, jehož výsledkem je změna všech uživatelských jmen v tabulce na hodnotu kovacmi5. Jelikož toto je předpokládané chování MySQL, nejedná se o bug. Obě kombinace fungují v pořádku, celkově jsem testoval 6 dotazů.



Obrázek 8.4 Příklad využití funkcí pro předpřipravené dotazy do MySQL

- **DELETE** má jedinou kombinaci, kterou jsem zkusil ve 3 variantách. Všechny pracují korektně.

Z testů je zdá, že funkce se korektně tážou databáze. Pro zodpovězení druhé otázky je třeba do každého parametru daných funkcí vložit SQL kód, který by správně neměl proběhnout. Pro všechny parametry všech funkcí (tedy celkem 17 parametrů) jsem vkládal hodnoty „1=1“, „1 OR 1=1“, „1=1;“, „1 OR 1=1;“, „DROP TABLE _users“ a „DROP TABLE _users“. Žádná z těchto hodnot nezpůsobila SQL injection tedy se zdá, že metoda snížení rizika pro SQL injection pracuje korektně.

Kapitola 9
Závěr

Závěrečná kapitola této práce obsahuje zhodnocení výsledků z pohledu zadání a praktické využitelnosti. Je logicky uspořádaná podle bodů zadání, kde autor ke každému bodu vyjadřuje svůj názor.

Technologie GraphQL a gRPC

Začátek práce se týkal analýzy zranitelností GraphQL a gRPC API. K tomu bylo třeba se z danými technologiemi seznámit.

GraphQL je velice komplexní technologie s relativně vágní definicí. Tyto vlastnosti dělají GraphQL obtížně pochopitelné. Dokumentace GraphQL, která patří k nejnepřehlednějším, jakou jsem kdy viděl[1], srozumitelnosti nepomáhá, dokonce naopak. V sekci 2.1 jsem vynaložil své největší úsilí, abych tuto technologii shrnul do pochopitelné formy, bez toho, abych pokryl desítku stran a nezastínil účel této práce. Velice negativně také hodnotím možnosti, jenž poskytuje vývojáři. Např. z hlediska programovacích jazyků - ve sekci 2.1 jsem poskytl odkaz na seznam jazyků, ve kterých je možné s GraphQL pracovat. Realita je taková, že implementace serveru má na výběr ze všech těchto jazyků, pokud jsou JavaScript. Velice bych ocenil tuto technologii rozšířit na uživatelsky příjemnější jazyky (nebo alespoň vytvořit konstrukce GraphQL tak, aby JavaScriptové nováčky nemátly).

Z hlediska využitelnosti naopak musím GraphQL vyzvednout. Po uchopení této technologie je s její pomocí možné vytvářet velice přizpůsobivý endpoint jakékoliv aplikace. Pozitivně cením kompatibilitu technologie a její výkonnost[2]. Z těchto důvodů si myslím, že buďto GraphQL, nebo nějaký jeho budoucí nástupce bude mít velký vliv ve webové komunikaci (a jakékoliv jiné komunikaci tipu client-server) obrovský pozitivní vliv.

gRPC je naprostý opak GraphQL. Daná technologie je značně méně komplexní, zato naprosto přesně daná a pevně definovaná. Dokumentace gRPC je uspořádaná se čtivým formátem, ve které se dobře hledá. Je také značně jednoduché vyhledat rady k určitým problémům s touto technologií, jelikož tento problém bude mít stejné řešení ve všech jeho instancích. Také pozitivně oceňuji možnost vytvářet server ve všech jazycích, které jsem zmínil v sekci 2.2.1 a výkonnost gRPC[3].

[1] Porovnávám s například dokumentací C++ (dostupnou zde: `https://en.cppreference.com/w/`), laťka tedy není nijak nízko

[2] Přikládám pěkný článek o porovnání výkonnosti GraphQL a REST: `https://www.techtarget.com/searchapparchitecture/tip/When-GraphQL-wins-in-a-GraphQL-vs-REST-performance-comparison`

[3] Přikládám článek o porovnání výkonnosti gRPC a GraphQL: `https://hasura.io/learn/graphql/intro-graphql/graphql-vs-grpc/`

Všechny klady gRPC jsou způsobeny jeho omezeními, které jsou samozřejmě negativy. gRPC nepřepravuje data ve velice omezené formě a navíc se přepravovaná data nedají číst lidmi, jejich extrakce vyžaduje další akce. Odesílání zpráv klienta je také relativně náročná záležitost. Ze všech těchto důvodů nepředpokládám rozšíření gRPC, ale vidím jeho nezměrnou využitelnost v komunikaci tipu server-server, tedy např. v distribuovaných systémech.

Analýza známých zranitelností GraphQL a gRPC

Analýza zranitelností obou protokolů proběhla v kapitolách 3 a 4. Na první pohled je vidět, že GraphQL má těchto zranitelností více, než gRPC. Navíc, první zranitelnosti gRPC je vážná a rozšířená převážně z lenosti vývojářů, jelikož její funkční oprava je triviální[4]. Tento jev je způsobený převážně účelem, který dané protokoly plní.

Při komunikaci tipu client-server, jak je tomu v případě GraphQL, je nutné poskytnout klientovi míru svobody v jeho dotazování. Právě tato svoboda je defacto jediným zdrojem DoS útoků zaměřených speciálně na GraphQL a velice napomáhá absenci autorizace i autentizace, jelikož zcela universálně se tyto funkcionality tvoří velmi obtížně. Fakt, že komunikuje člověk také vede k vytvoření natolik specifických chyb a tím zranitelnosti error information disclosure.

gRPC naopak díky své přímočarosti a jednoduchosti vytváři malý prostor pro zanesení zranitelností. Za obě čistě protokolové zranitelnosti může lidský faktor - zranitelnost nešifrované komunikace je způsobena tím, že vývojáři si nedají práci navíc s konfigurací a protobuf information disclosure je způsobena neopatrností při vytváření definic zpráv. SQL injection, ačkoliv se týká obou protokolů analyzovaných v této práci, je více řešená v gRPC, jelikož formát přijatých dat svádí ke skládání primitivních dotazů bez validace vstupu.

Není ovšem jednoznačně určitelné, jestli jeden protokol je vysloveně bezpečnější než druhý. V základu by se tento závěr možná vyslovit dal, ale vzhledem k omezením gRPC není možné říct, jak by z hlediska bezpečnosti vypadala velká aplikace, jež by byla podstatně složitější naprogramovat s pomocí gRPC, než s GraphQL.

Analýza automatizované snížení rizika známých zranitelností

Analýza automatizovaného snížení rizika probíraných zranitelností proběhla také v kapitolách 3 a 4.

Zranitelnosti GraphQL se ukázaly být přesně podle mých předpokladů i předpokladů zadání práce. Všechny se týkaly nějakých funkcionalit této technologie, která se ukázala být destruktivní ve špatných rukou. Oproti tomu zranitelnosti gRPC měly zcela jiný charakter – týkaly se lidských chyb, které způsobují, že korektně fungující funkcionalita se pod špatných dozorem může obrátit proti serveru. Díky tomu nezapadají do rámce práce natolik, jako zranitelnosti GraphQL a bylo nutné si s tím poradit. Pozitivní je, že jejich automatizované snížení rizika bylo řešitelné pomocí C++, což mi poskytlo možnost improvizace do té míry, která byla potřeba.

Implementace automatizovaného řešení pro snížení rizika

Implementace automatizovaného snížení rizika probíraných zranitelností je ilustrována v kapitolách 7 a 8.

Zranitelnosti GraphQL bylo příjemnější řešit než zranitelnosti gRPC, jak jsem naznačil v předchozí sekci. V technologii GraphQL byly tímto řešením funkce obohacující, nebo modifikující,

[4] Jak moc triviální jsem se dozvěděl až při analýze snížení jejího rizika

u gRPC byla automatizovaná prevence lidských chyb. U obou technologií jsem teoreticky splnil účel, ale z řešení zranitelností GraphQL mám lepší pocit.

Otestování řešení automatizovaného snížení rizika

Otestování automatizovaného snížení rizika probíraných zranitelností je taktéž ilustrována v kapitolách 7 a 8.

Testování mých řešení byla pravděpodobněji má oblíbená část. U některých zranitelností, např. testování autentizace u GraphQL bylo vyžadováno komplexnější otestování, zato u jiných zranitelností nebylo co testovat – daná funkce buďto probíhá, jak má a tedy funguje, nebo neprobíhá a nefunguje. Tento případ byl nejvíce patrný u GraphQL errors information disclosure. Testy proběhly dle mého dobrého svědomí a přiložené médium obsahuje všechno potřebné k jejich napodobení.

Přínosy a nedostatky práce, možnosti navázání na práci

Tato sekce je věnována převážně výstupům práce. Také obsahuje její hodnocení a možnosti na její případné navázání.

Jakožto přínosy práce vidím následující:

- **Model hrozeb obou technologií z pohledu známých zranitelností**
  Jedním z výstupů práce je model hrozeb obou probíraných technologií. Známá byla pouze existence daných zranitelností, avšak ne ve formě modelu hrozeb. Ke zranitelnostem ve zdrojích chybělo např. jejich ohodnocení, možnosti automatizované nápravy (někdy i neautomatizované), možnosti zneužitelnosti a důkaz jejich existence v podobě jejich reprodukce (ne všechny uvedené body se týkají všech zranitelností). Obohacení seznamu zranitelností o dané body vidím jako jeden z přínosů práce.

- **Postup tvorby lokální aplikace obou technologií**
  Defacto vedlejším produktem této práce je relativně detailní postup k vytvoření funkčního lokálního prostředí využívající obě zmiňované technologie. Tento návod již existuje v adekvátních oficiálních dokumentacích, včetně možností jeho otestování, tedy tato část sama o sobě nepřináší velký přínos. Malý přínos vidím v tom, že mnou vytvořený návod je obohacený o seznam a popis prerekvizit a v pozdější části o instalaci MySQL knihoven pro C++ na operační systém Ubuntu a jeho integraci s gRPC. Pro samotnou instalaci knihovny návod sice existuje, ale chybný a pro nalinkování knihoven neexistuje vůbec (a už vůbec ne k gRPC).

- **Zranitelné servery a testování daných zranitelností**
  Přiložené médium obsahuje kompletní kód zranitelných serverů včetně skriptů či postupů na testování daných zranitelností. Tím jsem vytvořil prostředí, které zaprvé dokazuje existenci daných zranitelností a zadruhé umožňuje naživo vidět jejich podstatu. Vývojář může tyto poznatky využít k získání vědomostí potřebné pro nezanesení svých serverů těmito zranitelnostmi. Poznatky ohledně testování zranitelností spolu s jejich skripty se dají jednoduše přenést na jakýkoliv jiný server využívající technologie GraphQL, respektive gRPC.

- **Funkce pro automatizaci snížení rizika**
  Mezi výstupy práce samozřejmě patří kódy, kvůli byla tato práce psaná. V práci je také obsažen návod pro integraci daných knihoven do GraphQL serverů (u gRPC jde o primitivní include). Všechny funkce jsou otestovány a připraveny do provozu a jsou napsané tak, aby byly co nejpřenosnější a tím pokryly velkou množinu reálných serverů.

Nedostatky práce se z mého hlediska týkají především funkcí pro automatizované snížení rizika. Když jsem začal s tvorbou této práce, neměl jsem dostatečné vzdělání v technologiích GraphQL a gRPC (zpětně mi dochází, že ve webové komunikaci obecně) a moje představa byla, že dané zranitelnosti souvisí s nějakou chybou, buď to v kódu, nebo v nedomyšlení funkcionality. Díky této mylné představě jsem se také domníval, že automatizované snížení rizika bude komplexní a univerzální. Komplexní v některých případech řešení bylo, universální v některých také, ale zdaleka ne natolik, jak jsem si představoval.

Na nedostatky navazují má doporučení pro následovníky této práce. Některé funkcionality, které jsem jako součástí této práce naprogramoval, je definitivně možné vytvořit universálněji. Například útoky GraphQL Dos je možné zvládnout pomocí doimplementování funkcionality podmíněného odesílání zpráv přímo do gRPC, které by následně kontrolovalo zprávy před odesláním. Taková detailní oprava jedné, nebo 2 zranitelností ze stejné technologie vystačí na celou diplomovou práci.

Příloha A - skript tvorby lokální public key infrastructure

```
#!/bin/bash

Password=SecurePassword

# Generate valid CA
openssl genrsa -passout pass:$Password -aes256 -out ca.key 4096
openssl req -passin pass:$Password -new -x509 -days 365 -key ca.key \
    -out ca.crt -subj \
    "/C=CZ/ST=Czech/L=Prague/O=Test/OU=Test/CN=Root CA"

# Generate valid Server Key/Cert
openssl genrsa -passout pass:$Password -aes256 -out server.key 4096
openssl req -passin pass:$Password -new -key server.key -out server.csr\
    -subj "/C=CZ/ST=Czech/L=Prague/O=Test/OU=Server/CN=localhost"
openssl x509 -req -passin pass:$Password -days 365 -in server.csr \
    -CA ca.crt -CAkey ca.key -set_serial 01 -out server.crt

# Remove passphrase from the Server Key
openssl rsa -passin pass:$Password -in server.key -out server.key

# Generate valid Client Key/Cert
openssl genrsa -passout pass:$Password -aes256 -out client.key 4096
openssl req -passin pass:$Password -new -key client.key -out client.csr\
    -subj  "/C=CZ/ST=Czech/L=Prague/O=Test/OU=Client/CN=localhost"
openssl x509 -passin pass:$Password -req -days 365 -in client.csr \
    -CA ca.crt -CAkey ca.key -set_serial 01 -out client.crt

# Remove passphrase from Client Key
openssl rsa -passin pass:$Password -in client.key -out client.key
```

Bibliografie

1. *GraphQL, a query language for your API* [https://graphql.org/]. [B.r.]. Accessed: 2023-02-20.
2. *GraphQL dpcumentation* [https://graphql.org/learn/]. [B.r.]. Accessed: 2023-02-20.
3. *A high performance, open source universal RPC framework* [https://grpc.io/]. [B.r.]. Accessed: 2023-02-20.
4. *Introduction to gRPC* [https://grpc.io/docs/what-is-grpc/introduction/]. [B.r.]. Accessed: 2023-02-20.
5. JUVILER, Jamie. *REST APIs: How They Work and What You Need to Know* [https://blog.hubspot.com/website/what-is-rest-api]. [B.r.]. Accessed: 2023-04-03.
6. *What is restful API?* [https://aws.amazon.com/what-is/restful-api/]. [B.r.]. Accessed: 2023-03-07.
7. *About OpenAPI* [https://swagger.io/docs/specification/about/]. [B.r.]. Accessed: 2023-03-9.
8. SHAJI, Amal. *Web authentication methods compared* [https://testdriven.io/blog/web-authentication-methods/]. [B.r.]. Accessed: 2023-4-27.
9. *JSON Web Tokens Introduction* [https://jwt.io/introduction]. [B.r.]. Accessed: 2023-04-19.
10. *JSON Web Token for Java Cheatsheet* [https://cheatsheetseries.owasp.org/cheatsheets/JSON_Web_Token_for_Java_Cheat_Sheet.html#consideration-about-using-jwt]. [B.r.]. Accessed: 2023-04-03.
11. MANZOLILLO, Nick. *Access control models* [https://butterflymx.com/blog/access-control-models/]. [B.r.]. Accessed: 2023-4-27.
12. *Cross site request forgery prevention cheat sheet* [https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html]. [B.r.]. Accessed: 2023-03-28.
13. *Bearer Token* [https://swagger.io/docs/specification/authentication/bearer-authentication/]. [B.r.]. Accessed: 2023-03-24.
14. HAMMOU, Achraf Ait Sidi. GraphQL verbose error suggestions [https://escape.tech/blog/graphql-verbose-error-suggestions/]. 2022. Accessed: 2023-03-9.
15. *OpenAPI specification* [https://swagger.io/specification/]. [B.r.]. Accessed: 2023-03-09.
16. *GraphQL introspection* [https://graphql.org/learn/introspection/]. [B.r.]. Accessed: 2023-04-03.

17. STEMMLER, Khalil. *Why you should disable graphql introspection in production* [https://www.apollographql.com/blog/graphql/security/why-you-should-disable-graphql-introspection-in-production/]. [B.r.]. Accessed: 2023-04-03.
18. *OWASP GraphQL cheatsheet* [https://cheatsheetseries.owasp.org/cheatsheets/GraphQL_Cheat_Sheet.html#query-access-data-fetching]. [B.r.]. Accessed: 2023-03-09.
19. *OWASP GraphQL cheat sheet* [https://cheatsheetseries.owasp.org/cheatsheets/GraphQL_Cheat_Sheet.html]. [B.r.]. Accessed: 2023-03-28.
20. PANDYA, Dhaivat. *Persistent GraphQL queries* [https://www.apollographql.com/blog/apollo-client/persisted-graphql-queries/]. [B.r.]. Accessed: 2023-04-03.
21. STUBAILO, Sashko. *Benefits of Static Graphql Queries* [https://www.apollographql.com/blog/community/5-benefits-of-static-graphql-queries/#4-persisted-queries]. 2016. Accessed: 2023-04-03.
22. TEAM, IBM PTC Security. gRPC Security series: part 3. Security vulnerabilities in gRPC [https://medium.com/@ibm_ptc_security/grpc-security-series-part-3-c92f3b687dd9]. 2022.
23. *OWASP REST security cheatsheet* [https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html]. [B.r.]. Accessed: 2023-03-09.
24. *Authentication in gRPC documentation* [https://grpc.io/docs/guides/auth/]. [B.r.]. Accessed: 2023-02-20.
25. *OWASP SQL injection prevention cheat sheet* [https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html]. [B.r.]. Accessed: 2023-03-26.
26. *Get started with Apollo server - Apollo GraphQL docs* [https://www.apollographql.com/docs/apollo-server/getting-started/]. [B.r.]. Accessed: 2023-04-05.

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `src`
  - `GrahQL`
    - `VulnerableServer` ..... Zdrojové kódy zranitelného serveru
    - `Vulnerabilities` ..... Zdrojové kódy testování zranitelností
    - `ProtectedServer` ..... Zdrojové kódy jednotlivých částí oprav a testovací program
  - `gRPC`
    - `VulnerableServer` ..... Zdrojové kódy zranitelného serveru
    - `ProtectedServer` ..... Zdrojové kódy jednotlivých částí oprav a testovací program
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
- `thesis.zip` ..... zdrojová forma práce ve formátu LaTeX