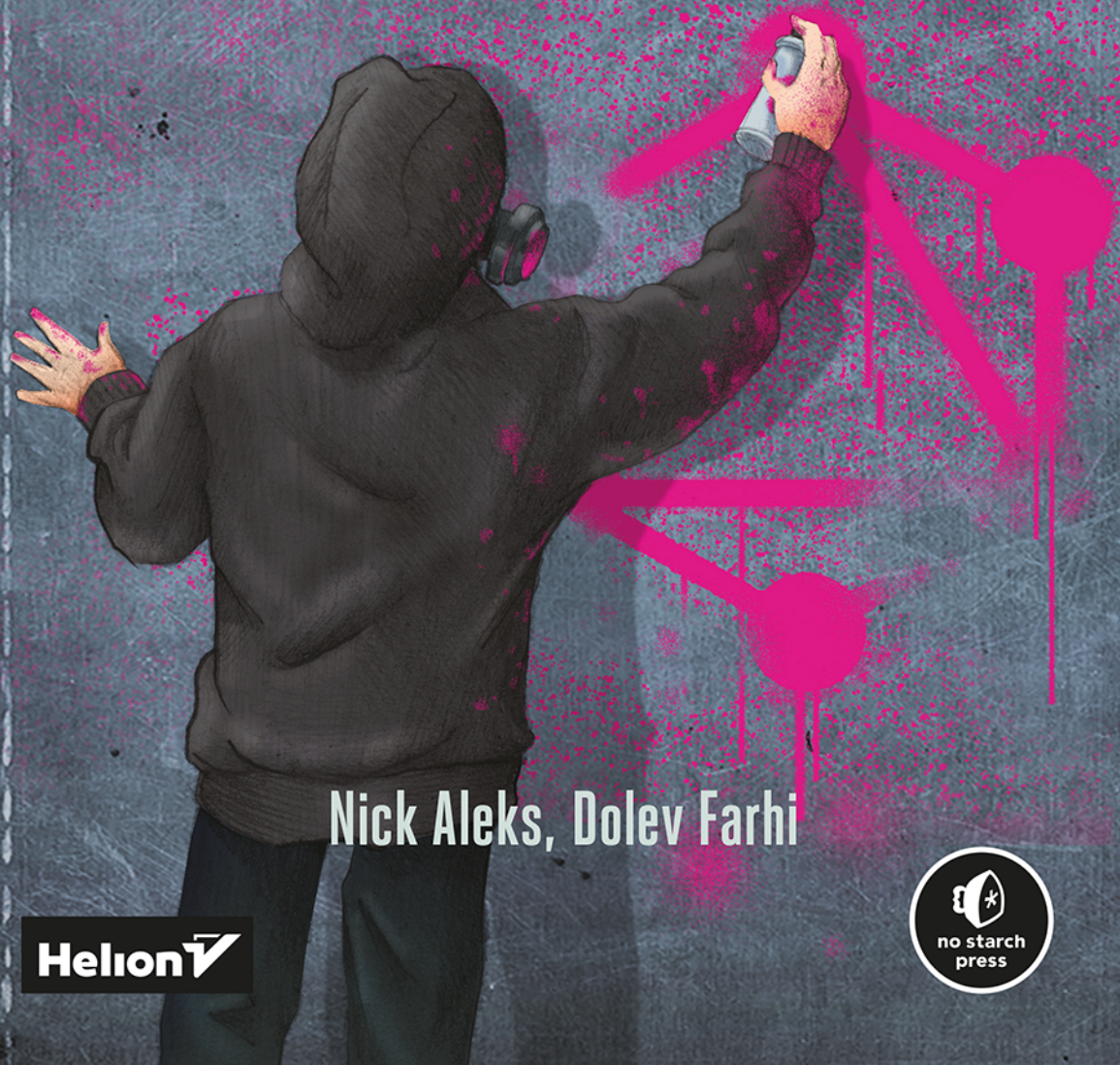
# Black Hat GraphQL

## *Bezpieczeństwo API dla hakerów i pentesterów*



Nick Aleks, Dolev Farhi

Helion

no starch press

Tytuł oryginału: Black Hat GraphQL: API Attacks for Hackers and Pentesters

Tłumaczenie: Robert Górczyński

ISBN: 978-83-289-1124-6



Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
*https://helion.pl/user/opinie/graphq*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

# Spis treści

**O AUTORACH** ..... **11**

**O REDAKTORZE TECHNICZNYM** ..... **11**

**PODZIĘKOWANIA** ..... **13**

**WSTĘP** ..... **15**

**WPROWADZENIE** ..... **19**
Dla kogo jest przeznaczona ta książka? ..... 20
Laboratorium dla książki i repozytorium kodu źródłowego ..... 20
Co znajduje się w książce? ..... 21

**1**
**WPROWADZENIE DO GRAPHQL** ..... **23**
Podstawy ..... 23
- Korzenie ..... 24
- Przykłady zastosowania ..... 24
- Specyfikacja ..... 25

W jaki sposób jest prowadzona komunikacja? ..... 25
- Schemat ..... 26
- Zapytania ..... 28
- Analizator składni zapytania i funkcje resolverów ..... 30

Jakie problemy rozwiązuje GraphQL? ..... 31
API GraphQL kontra API REST ..... 32
- Przykład oparty na API REST ..... 33
- Przykład oparty na GraphQL ..... 35
- Inne różnice ..... 38

Pierwsze zapytanie ..... 41
Podsumowanie ..... 44

**2**
**PRZYGOTOWANIE LABORATORIUM DO PRACY Z GRAPHQL ..... 45**
Środki bezpieczeństwa ..... 46
Instalowanie dystrybucji Kali ..... 47
Instalowanie klientów internetowych ..... 49
Wykonywanie zapytań z poziomu powłoki ..... 49
Wykonywanie zapytań za pomocą narzędzia graficznego ..... 50
Przygotowanie serwera GraphQL, w którym istnieją luki w zabezpieczeniach ..... 53
Instalowanie Dockera ..... 53
Wdrożenie aplikacji Damn Vulnerable GraphQL Application ..... 53
Testowanie DVGA ..... 56
Instalowanie narzędzi przeznaczonych do hakowania GraphQL ..... 56
Burp Suite ..... 57
Clairvoyance ..... 58
InQL ..... 60
Graphw00f ..... 61
BatchQL ..... 62
Nmap ..... 62
Commix ..... 63
graphql-path-enum ..... 64
EyeWitness ..... 65
GraphQL Cop ..... 65
CrackQL ..... 66
Podsumowanie ..... 66

**3**
**PŁASZCZYZNA ATAKU NA GRAPHQL ..... 68**
Czym jest płaszczyzna ataku? ..... 68
Język ..... 69
Zapytanie, mutacja i subskrypcja ..... 70
Nazwa operacji ..... 73
Pole ..... 75
Argument ..... 76
Alias ..... 78
Fragment ..... 80
Zmienna ..... 81
Dyrektywa ..... 82
Typy danych ..... 85
Obiekt ..... 86
Skalar ..... 86
Wyliczenie ..... 87
Unia ..... 88

Interfejs ... 90
Dane wejściowe ... 91
Introspekcja ... 92
Weryfikacja i wykonywanie zapytania ... 95
Najczęściej spotykane słabe strony ... 97
Reguła specyfikacji i słaba strona implementacji ... 97
Odmowa usług ... 98
Ujawnienie informacji ... 99
Błędy w mechanizmach uwierzytelnienia i autoryzacji ... 99
Wstrzykiwanie kodu ... 99
Podsumowanie ... 100

**4**
**REKONESANS ... 101**
Wykrywanie GraphQL ... 102
Najczęściej stosowane punkty końcowe ... 103
Najczęściej udzielane odpowiedzi na zapytania ... 104
Skanowanie za pomocą narzędzia Nmap ... 107
Pole __typename ... 108
Graphw00f ... 111
Wykrywanie narzędzi GraphiQL Explorer i GraphQL Playground ... 112
Używanie EyeWitness do skanowania pod kątem interfejsów graficznych ... 113
Próba wykonania zapytania za pomocą klienta graficznego ... 116
Sprawdzanie GraphQL za pomocą introspekcji ... 119
Wizualizacja introspekcji za pomocą GraphQL Voyager ... 124
Generowanie za pomocą SpectaQL dokumentacji introspekcji ... 126
Wyszukiwanie informacji w przypadku, gdy introspekcja jest wyłączona ... 126
Sprawdzanie implementacji GraphQL ... 127
Wykrywanie serwera za pomocą Graphw00f ... 130
Analiza wyników ... 131
Podsumowanie ... 133

**5**
**ATAK TYPU DOS ... 134**
Kierunki ataków typu DoS w GraphQL ... 134
Zapytania cykliczne ... 136
Relacje cykliczne w schemacie GraphQL ... 136
Jak wykrywać relacje cykliczne? ... 138
Luki w zabezpieczeniach związane z zapytaniami cyklicznymi ... 143
Luki w zabezpieczeniach związane z introspekcją cykliczną ... 144
Luki w zabezpieczeniach związane z fragmentami cyklicznymi ... 146

Powielanie pola ... 147
Sposób działania luki związanej z powielaniem pola ... 147
Testowanie pod kątem luk w zabezpieczeniach związanych z powielaniem pola ... 149
Przeciążenie aliasu ... 151
Nadużywanie aliasów na potrzeby ataków typu DoS ... 152
Łączenie aliasów i zapytań cyklicznych ... 153
Przeciążanie dyrektywy ... 154
Nadużywanie dyrektyw w atakach typu DoS ... 155
Testowanie pod kątem luk w zabezpieczeniach związanych z przeciążaniem dyrektywy ... 155
Przeciążanie limitu obiektu ... 156
Grupowanie zapytań za pomocą tablicy ... 158
Zrozumienie sposobu działania grupowania zapytań za pomocą tablicy ... 158
Testowanie pod kątem luk w zabezpieczeniach związanych z grupowaniem zapytań za pomocą tablicy ... 159
Łączenie zapytań cyklicznych i grupowania zapytań za pomocą tablicy ... 160
Używanie BatchQL do wykrycia dostępności grupowania zapytań za pomocą tablicy ... 162
Przeprowadzanie za pomocą narzędzia GraphQL Cop audytu podatności na ataki typu DoS ... 163
Zabezpieczenia GraphQL przed atakami typu DoS ... 164
Analiza kosztu zapytania ... 164
Ograniczenia głębokości zapytania ... 168
Ograniczenia oparte na aliasach i tablicach ... 169
Ograniczenia związane z powielaniem pól ... 170
Ograniczanie liczby zwracanych rekordów ... 171
Lista zapytań dozwolonych ... 171
Automatycznie trwale przechowywane zapytania ... 171
Przekroczenie czasu oczekiwania ... 173
Zapora sieciowa aplikacji internetowej ... 173
Brama działająca jako proxy ... 174
Podsumowanie ... 175

**6**
**UJAWNIANIE INFORMACJI ... 176**
Identyfikowanie sposobów ujawniania informacji w GraphQL ... 177
Zautomatyzowane wyodrębnianie schematu za pomocą narzędzia InQL ... 178
Jak sobie radzić z wyłączoną introspekcją ... 179
Wykrywanie wyłączonej introspekcji ... 180
Wykorzystanie środowiska innego niż produkcyjne ... 180
Wykorzystanie pola meta __type ... 181

Używanie funkcjonalności sugerowania nazwy pola ... 183
Poznajemy algorytm Edit-Distance ... 184
Optymalizacja używania funkcjonalności sugerowania nazwy pola ... 185
Uwzględnienie kwestii bezpieczeństwa ... 186
Wstawianie listy nazw pól ... 187
Wstawianie listy nazw typów w polu meta __type ... 189
Automatyzacja za pomocą narzędzia Clairvoyance funkcji sugerowania nazwy pola i wstawiania listy nazw ... 191
Nadużywanie komunikatów o błędzie ... 194
Analiza szczegółowego komunikatu o błędzie ... 195
Włączenie debugowania ... 197
Wyodrębnianie informacji ze stosu wywołań ... 198
Ujawnianie informacji za pomocą metody GET ... 200
Podsumowanie ... 200

**7**
**OMINIĘCIE UWIERZYTELNIENIA I AUTORYZACJI ... 202**
Stan uwierzytelnienia i autoryzacji w GraphQL ... 203
Modele wdrożenia in-band i out-of-band ... 203
Najczęściej stosowane podejścia ... 205
Testowanie uwierzytelnienia ... 212
Wykrywanie warstwy uwierzytelnienia ... 212
Ataki typu brute force przeprowadzane na hasła z użyciem funkcjonalności wstawiania listy nazw ... 214
Atak typu brute force na hasło z użyciem narzędzia CrackQL ... 216
Używanie listy nazw dozwolonych operacji ... 218
Podrabianie i wyciekanie danych uwierzytelniających JWT ... 219
Testowanie autoryzacji ... 222
Wykrywanie warstwy autoryzacji ... 222
Sprawdzanie ścieżek za pomocą graphql-path-enum ... 223
Przeprowadzanie za pomocą narzędzia CrackQL ataków typu brute force na argumenty i pola ... 224
Podsumowanie ... 226

**8**
**WSTRZYKIWANIE KODU ... 228**
Związane ze wstrzykiwaniem kodu luki w zabezpieczeniach GraphQL ... 229
Pole rażenia danych wejściowych o złośliwym działaniu ... 230
OWASP Top 10 ... 231
Płaszczyzna ataku polegającego na wstrzykiwaniu kodu ... 232
Argumenty zapytania ... 233
Argumenty pola ... 235

Argumenty dyrektywy zapytania ... 235
Nazwy operacji ... 236
Punkty dostarczania danych wejściowych ... 237
Wstrzykiwanie kodu SQL ... 238
Poznajemy typy ataków związanych z SQLi ... 239
Testowanie pod kątem luki SQLi ... 240
Testowanie aplikacji DVGA pod kątem luki związanej z SQLi z użyciem oprogramowania Burp Suite ... 240
Automatyzacja ataku polegającego na wstrzykiwaniu kodu SQL ... 247
Wstrzykiwanie polecenia systemu operacyjnego ... 249
Przykład ... 250
Ręczne testowanie aplikacji DVGA ... 251
Zautomatyzowane testowanie z użyciem frameworka Commix ... 253
Analiza kodu funkcji resolvera ... 255
Cross-Site Scripting ... 256
Luka XSS typu odbijana ... 256
Luka XSS typu przechowywana ... 258
Luka XSS oparta na modelu DOM ... 259
Testowanie aplikacji DVGA pod kątem luk XSS ... 260
Podsumowanie ... 265

**9**
**FAŁSZOWANIE I PRZECHWYTYWANIE ŻĄDAŃ ... 266**
Atak typu CSRF ... 267
Wyszukiwanie działań, które mogą prowadzić do zmiany stanu ... 269
Sprawdzanie pod kątem luk w zabezpieczeniach związanych z metodami POST ... 270
Automatyczne wysyłanie formularza w trakcie ataku typu CSRF ... 273
Sprawdzanie pod kątem luk w zabezpieczeniach związanych z metodami GET ... 274
Atak polegający na wstrzykiwaniu kodu HTML ... 277
Testowanie automatyczne za pomocą narzędzi BatchQL i GraphQL Cop ... 278
Unikanie ataków typu CSRF ... 279
Atak typu SSRF ... 281
Poznajemy typy ataków SSRF ... 282
Wyszukiwanie podatnych na ataki operacji, pól i argumentów ... 283
Testowanie pod kątem luki SSRF ... 284
Zapobieganie atakom typu SSRF ... 287
Przechwytywanie WebSocket ... 288
Wyszukiwanie operacji subskrypcji ... 289
Przechwytywanie zapytania subskrypcji ... 289
Ochrona przed atakami typu CSWSH ... 293
Podsumowanie ... 293

**10**
**EXPLOITY I UJAWNIONE LUKI W ZABEZPIECZENIACH ..... 294**
Odmowa usług ..... 295
Ogromne dane używane podczas ataku ..... 295
Wyrażenia regularne (CS Money) ..... 297
Zapytanie cykliczne introspekcji (GitLab) ..... 299
Aliasy dla powielania pola (Magento) ..... 300
Grupowanie zapytań za pomocą tablicy na potrzeby powielania pola (WPGraphQL) ..... 301
Fragmenty cykliczne (Agoo) ..... 303
Nieprawidłowa autoryzacja ..... 304
Umożliwienie dostępu do danych dezaktywowanym użytkownikom (GitLab) ..... 305
Pozwolenie nieuprzywilejowanemu pracownikowi firmy na modyfikowanie adresu e-mail użytkownika (Shopify) ..... 306
Ujawnienie liczby dozwolonych hakerów za pomocą obiektu team (HackerOne) ..... 307
Odczytywanie notatek prywatnych (GitLab) ..... 308
Ujawnienie informacji na temat transakcji płatności (HackerOne) ..... 309
Ujawnienie informacji ..... 310
Lista użytkowników GraphQL (GitLab) ..... 310
Uzyskanie dostępu do zapytania introspekcji za pomocą WebSocket (Nuri) ..... 311
Wstrzykiwanie kodu ..... 312
Wstrzykiwanie kodu SQL za pomocą parametru zapytania GET (HackerOne) ..... 312
Wstrzykiwanie kodu SQL w argumencie Object (Apache SkyWalking) ..... 314
Cross-Site Scripting (GraphQL Playground) ..... 315
Cross-Site Request Forgery (GitLab) ..... 317
Podsumowanie ..... 318

**A**
**LISTA RZECZY DO SPRAWDZENIA PODCZAS TESTOWANIA GRAPHQL ..... 319**
Rekonesans ..... 319
Odmowa usług ..... 320
Ujawnianie informacji ..... 320
Uwierzytelnienie i autoryzacja ..... 321
Wstrzykiwanie kodu ..... 321
Fałszowanie żądań ..... 322
Przechwytywanie żądań ..... 322

**B**
**ZASOBY DOTYCZĄCE BEZPIECZEŃSTWA GRAPHQL ..... 323**
Wskazówki i podpowiedzi dotyczące testów penetracyjnych ..... 323
Laboratoria hakerskie do samodzielnego wypróbowania ..... 324
Klipy wideo związane z zapewnieniem bezpieczeństwa ..... 324

# 3

# Płaszczyzna ataku na GraphQL

W tym rozdziale najpierw przedstawimy język GraphQL i jego system typów z perspektywy hakera. Następnie przejdziemy do zaprezentowania najczęściej ujawniających się słabych stron GraphQL. Mamy nadzieję, że masz pod ręką swój wyimaginowany czarny kapelusz, ponieważ dowiesz się, jak funkcjonalność można zmienić w słabość, jak błędną konfigurację można zmienić w wyciek informacji, a także jak błędy w implementacji mogą prowadzić do powstania możliwości przeprowadzenia ataku typu DoS.

## Czym jest płaszczyzna ataku?

**Płaszczyzna ataku** to suma wszystkich możliwych kierunków ataku, które można wykorzystać w celu naruszenia poufności, spójności i dostępności systemu. Na przykład wyobraź sobie fizyczny budynek składający się z drzwi wejściowych, bocznych drzwi

oraz wielu okien. Atakujący może postrzegać każde z tych okien i drzwi jako potencjalną możliwość uzyskania nieupoważnionego dostępu do budynku.

Z reguły ryzyko przeprowadzenia udanego ataku na system jest większe, gdy płaszczyzna ataku jest ogromna i system składa się z wielu aplikacji, baz danych, serwerów, punktów końcowych itd. Im więcej okien i drzwi ma budynek, tym większe niebezpieczeństwo, że jeden z tych punktów wejściowych będzie niezabezpieczony bądź otwarty.

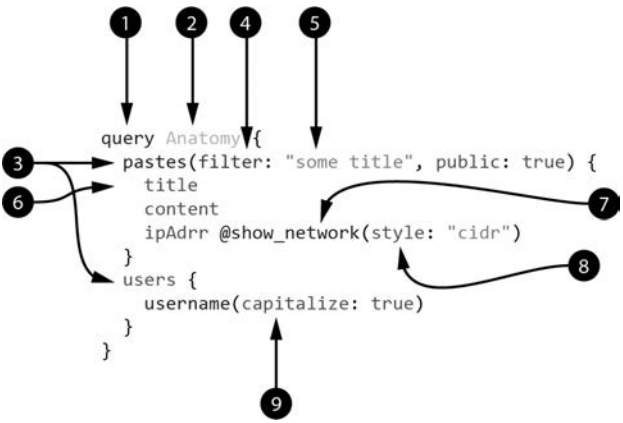
Płaszczyzna ataku zmienia się wraz z upływem czasu, zwłaszcza w sytuacji, gdy systemy i ich środowiska ewoluują. To stwierdzenie okazuje się szczególnie prawdziwe w środowisku chmury, w którym infrastruktura jest elastyczna. Na przykład serwer może istnieć tylko przez jakiś czas lub adres IP może się zmienić, niekiedy nawet wielokrotnie w ciągu dnia.

Warto przejrzeć wszystkie drzwi i okna w GraphQL oraz wymienić potencjalne kierunki ataku, aby można było je wykorzystać. Zrozumienie omówionych tutaj koncepcji pomoże w kilku kolejnych rozdziałach, gdy zagłębimy się w bezpieczeństwo ofensywne.

# Język

Na potrzeby omówienia płaszczyzny ataku GraphQL jego specyfikację podzielimy na dwie części: język i system typów. Rozpoczniemy od omówienia, z perspektywy klienta, języka używanego w celu wykonywania żądań do serwera API GraphQL. Następnie przejdziemy do przedstawienia systemu typów GraphQL z perspektywy serwera. Te koncepcje oraz inne komponenty wewnętrzne GraphQL poznasz na przykładzie specyfikacji GraphQL. W tym miejscu zamierzamy zająć się jedynie tymi aspektami, dzięki którym zdobędziesz wiedzę wystarczającą do sprawdzania kierunków ataku w kolejnych rozdziałach.

Język GraphQL składa się z wielu użytecznych komponentów, które mogą być wykorzystane przez klienty. Na pierwszy rzut oka sposób przedstawiania tych elementów w żądaniach może wydawać się dezorientujący. Na rysunku 3.1 pokazaliśmy przykładowe zapytanie GraphQL, którego komponenty są dokładniej wyjaśnione w tabeli 3.1 nieco dalej w rozdziale.




*Rysunek 3.1. Przykładowe zapytanie GraphQL*

Jak możesz zobaczyć, zapytanie GraphQL ma unikatową strukturę i to bardzo ważne, by zrozumieć jego różne elementy. Dokładniejsze przedstawienie poszczególnych komponentów zapytania zamieściliśmy w tabeli 3.1.

*Tabela 3.1. Komponenty tworzące zapytanie GraphQL*

| Lp. | Komponent | Opis |
|---|---|---|
| 1. | Typ operacji | Typ definiuje metodę interakcji z serwerem (zapytanie, mutacja bądź subskrypcja) |
| 2. | Nazwa operacji | Dowolnie utworzona przez klienta etykieta, używana w celu dostarczenia unikatowej nazwy dla operacji |
| 3. | Pole najwyższego poziomu | Funkcja zwracająca pojedynczą jednostkę informacji bądź obiekt żądany w ramach danej operacji (może zawierać pola zagnieżdżone) |
| 4. | Argument (pola najwyższego poziomu) | Nazwa parametru używanego do przekazania informacji do pola w celu dostosowania sposobu i wyniku jego działania |
| 5. | Wartość | Dane związane z argumentem przekazanym do pola |
| 6. | Pole | Funkcja zagnieżdżona, która zwraca pojedynczą jednostkę informacji bądź obiekt żądany w ramach operacji |
| 7. | Dyrektywa | Funkcjonalność używana do dekorowania pól, aby zmieniały sposób weryfikacji bądź wykonywania, modyfikując tym samym wartość zwracaną przez serwer GraphQL |
| 8. | Argument (dyrektywy) | Nazwa parametru używanego do przekazania informacji do pola lub obiektu w celu dostosowania sposobu i wyniku jego działania |
| 9. | Argument (pola) | Nazwa parametru używanego do przekazania informacji do pola w celu dostosowania sposobu i wyniku jego działania |

W kolejnych punktach znacznie dokładniej przedstawimy poszczególne komponenty zapytania, a także kilka dodatkowych funkcjonalności GraphQL, przy czym skoncentrujemy się na wpływie, jaki one mają na płaszczyznę ataku na GraphQL.

## Zapytanie, mutacja i subskrypcja

Podstawowe typy operacji — zapytanie, mutacja i subskrypcja — zostały już omówione w rozdziale 1., w którym pokazaliśmy także przykład użycia typu *zapytanie* (ang. *query*) w celu pobrania danych. (Z tego powodu w tym miejscu nie będziemy dokładnie omawiali typu zapytania). Dla hakera prawdziwą zabawą jest często modyfikowanie danych. Tworzenie, uaktualnianie i usuwanie danych na platformie docelowej umożliwia ujawnienie słabych stron logiki biznesowej.

**Uwaga**

*Zachęcamy, by podczas wykonywania żądań przedstawionych w przykładach zamieszczonych w tym rozdziale korzystać z laboratorium GraphQL, którego utworzenie omówiliśmy w poprzednim rozdziale. W celu wykonywania żądań do aplikacji DVGA skorzystaj z klienta Altair, w którym wpisz adres URL* http://localhost:5013/graphql.

## Mutacja

W języku GraphQL potężne możliwości związane z modyfikacją danych można odblokować dzięki wykorzystaniu tzw. **mutacji** (ang. *mutation*). Spójrz na przykład tego rodzaju zapytania.

```
mutation {
  editPaste(id: 1, content: "My first mutation!") {
    paste {
       id
       title
       content
    }
  }
}
```

Operacja mutacji została tutaj zdefiniowana za pomocą słowa kluczowego `mutation`. Następnie jest wywoływane pole najwyższego poziomu, `editPaste`, akceptujące argumenty `id` i `content`. (Dokładne omówienie argumentów znajdziesz w dalszej części rozdziału). Działanie tej mutacji polega na pobraniu fragmentu tekstu o identyfikatorze (`id`) wynoszącym `1` i uaktualnieniu tego fragmentu. Następnie pobierany jest uaktualniony fragment. To jest przykład mutacji, która jednocześnie zmienia i odczytuje dane.

**Uwaga**

*Podobnie jak w przypadku API REST — w którym żądania HTTP GET używa się do odczytywania informacji, choć można je wykorzystać również do modyfikacji danych — także w implementacji GraphQL można zignorować specyfikację i zaimplementować operacje zapytania w sposób pozwalający na zapisywanie danych. W rozdziale 9. wyjaśnimy, dlaczego zapisywanie danych za pomocą operacji zapytania GraphQL może być kiepskim pomysłem.*

## Subskrypcja

Operacja **subskrypcji** (ang. *subscription*) działa dwukierunkowo: pozwala klientowi na pobieranie w czasie rzeczywistym danych z serwera, a także pozwala serwerom na przekazywanie uaktualnień do klientów. Wprawdzie subskrypcje nie są tak często spotykane jak zapytania i mutacje, ale wiele serwerów ich używa i dlatego ważne jest, by poznać sposób ich działania.

Subskrypcja jest przekazywana poprzez protokół transportowy, najczęściej *WebSocket*, czyli protokół komunikacji w czasie rzeczywistym pozwalający klientom i serwerom na wymianę komunikatów w danej chwili za pomocą trwałego połączenia. Jednak specyfikacja GraphQL nie wskazuje, który protokół transportowy powinien być używany dla subskrypcji, więc można się natknąć na konsumenty używające różnych protokołów.

Gdy klient i serwer chcą prowadzić komunikację poprzez WebSocket, przeprowadzają operację wymiany informacji, w której wyniku następuje uaktualnienie istniejącego połączenia HTTP do połączenia używającego WebSocket. Omówienie wewnętrznych komponentów WebSocket wykracza poza zakres książki. Więcej informacji na temat tej

technologii znajdziesz w poście opublikowanym na blogu *https://portswigger.net/web-security/websockets/what-are-websockets*.

Skoro aplikacja DVGA obsługuje subskrypcje poprzez WebSocket, można zauważyć operację wymiany informacji między frontendem DVGA a serwerem GraphQL. Klient może wykorzystać subskrypcję do pobrania z serwera GraphQL pewnych danych, takich jak nowo utworzone fragmenty tekstu. Na przykład podczas przeglądania fragmentów tekstu dostępnych publicznie pod adresem *http://localhost:5013/* na karcie narzędzi programistycznych w przeglądarce WWW (Network) prawdopodobnie zobaczysz żądanie, które będzie podobne do tutaj przedstawionego.

```
GET /subscriptions HTTP/1.1
Host: 0.0.0.0:5013
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
Origin: http://localhost:5013
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: MV5U83GH1UG8AlEb18lHiA==
```

Pochodząca z serwera GraphQL odpowiedź na takie żądanie wymiany informacji może mieć następującą postać:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: aRnlpG8XwzRHPVxYmGVdqJv3D7U=
```

Jak możesz zobaczyć, to żądanie spowodowało, że klient i serwer przeszły z protokołu HTTP na WebSocket, na co wskazuje kod odpowiedzi `101 Switching Protocols`. Z kolei nagłówek odpowiedzi `Sec-WebSocket-Accept` informuje klienta o zaakceptowaniu przez serwer zmiany protokołu.

Po zakończeniu etapu wymiany informacji aplikacja DVGA wyśle żądanie subskrypcji poprzez nawiązane przed chwilą połączenie WebSocket.

```
subscription {
  paste {
     id
     title
     content
  }
}
```

Do zdefiniowania operacji subskrypcji używa się słowa kluczowego `subscription`. W omawianym przykładzie żądanie będzie dotyczyło pola najwyższego poziomu (`paste`) i następnie pobierze wartości pól `id`, `title` i `content`. Ta subskrypcja pozwala klientom na subskrybowanie pola `paste` — kiedykolwiek w aplikacji DVGA zostanie utworzony nowy

fragment tekstu, serwer GraphQL poinformuje subskrybentów o tym zdarzeniu. Dzięki temu klienty nie muszą nieustannie sprawdzać, czy na serwerze pojawiły się jakiekolwiek uaktualnienia. Takie rozwiązanie okazuje się szczególnie użyteczne, ponieważ w danej chwili serwer może nie mieć nic nowego do zwrócenia.

Jeżeli za pomocą klienta Altair spróbujesz przekazać tę operację subskrypcji do aplikacji DVGA, najpierw musisz przypisać adres URL subskrypcji. Można to zrobić w narzędziu klienta Altair przez kliknięcie przedstawiającej dwie strzałki ikony po lewej stronie paska bocznego i następnie wpisanie adresu URL w postaci ***ws://localhost:5013/subscriptions***. Aby teraz otrzymać dane pochodzące z subskrypcji DVGA, trzeba utworzyć fragment tekstu. W tym celu możesz wykorzystać interfejs aplikacji DVGA i utworzyć fragment tekstu za pomocą strony *Public Pastes*. Inną możliwością jest wykonanie mutacji, takiej jak przedstawiona w kolejnym fragmencie kodu, z poziomu innej karty narzędzia Altair.

```
mutation {
  createPaste(title: "New paste", content: "Test", public: false) {
    paste {
      id
      title
      content
    }
  }
}
```

Połączenie WebSocket jest podatne na lukę w zabezpieczeniach pozwalającą na przeprowadzanie ataków typu **CSWSH** (ang. *cross-site websocket hijacking*), która pojawia się, gdy w trakcie procesu wymiany informacji serwer nie przeprowadza weryfikacji pochodzenia klienta. Połączenie WebSocket może być również podatne na atak typu **MITM** (ang. *man-↪in-the-middle*), gdy transport komunikatów nie odbywa się poprzez zaszyfrowany kanał, taki jak TLS (ang. *transport layer security*). Istnienie wymienionych luk w zabezpieczeniach może mieć wpływ na działania przeprowadzane poprzez subskrypcje GraphQL. Ataki na WebSocket znacznie dokładniej omówimy w rozdziale 9.

## Nazwa operacji

**Nazwa operacji** GraphQL to etykieta unikatowo identyfikująca daną operację w określonym kontekście. Taka nazwa pojawia się w dokumentach wykonywalnych, które klient wysyła do usługi GraphQL. Wspomniany dokument może zawierać listę składającą się z jednej lub większej liczby operacji. Na przykład dokument przedstawiony na listingu 3.1 pokazuje pojedynczą operację zapytania, która żąda pola najwyższego poziomu `pastes` razem z polem zagnieżdżonym `title`.

*Listing 3.1. Dokument wykonywalny w postaci zapytania*

```
query {
  pastes {
    title
  }
}
```

Jeżeli dokument zawiera tylko pojedynczą operację i jest nią zapytanie, które nie definiuje żadnych zmiennych i nie ma żadnych dyrektyw, wówczas tę operację można przedstawić w postaci skróconej, bez słowa kluczowego `query`, jak pokazaliśmy na listingu 3.2.

*Listing 3.2. Skrócona wersja dokumentu zapytania*

```
{
  pastes {
    title
  }
}
```

Z drugiej strony dokument może zawierać wiele operacji. Jeżeli w dokumencie znajduje się więcej niż tylko jedna operacja tego samego typu, wówczas będzie trzeba używać nazw operacji.

Te nazwy są określane przez klienty, co oznacza, że mogą być całkowicie przypadkowe. Z tego powodu mogą wprowadzać w błąd analityków przeglądających dzienniki zdarzeń aplikacji GraphQL. Na przykład wyobraź sobie, że klient wysyła dokument, używając do tego nazwy operacji `getPastes`, przy czym zamiast zwrócić listę obiektów przedstawiających fragmenty tekstu, w rzeczywistości operacja powoduje usunięcie wszystkich fragmentów tekstu.

Na listingu 3.3 przedstawiliśmy przykład dokumentu zawierającego operacje zapytań o nazwach `getPasteTitles` i `getPasteContent`. Wprawdzie nazwy tych operacji są odpowiednie, biorąc pod uwagę żądaną treść, ale równie dobrze mogłyby być zupełnie niezwiązane z działaniami wykonywanymi przez zapytania. Tylko logika operacji i wskazane pola mają wpływ na dane wyjściowe operacji.

*Listing 3.3. Dokument zapytania zawierający wiele operacji, z których każda została oznaczona nazwą*

```
query getPasteTitles {
  pastes {
    title
  }
}

query getPasteContent {
  pastes {
    content
  }
}
```

Skoro nazwy operacji to dane wejściowe pochodzące od klienta, potencjalnie mogą być wykorzystane jako kierunki ataku polegającego na wstrzykiwaniu kodu. Niektóre implementacje GraphQL pozwalają na stosowanie znaków specjalnych w nazwach operacji. Aplikacja może przechowywać te nazwy w dziennikach zdarzeń audytów, aplikacjach stworzonych przez podmioty zewnętrzne bądź w innych systamach. Jeżeli te znaki specjalne nie zostaną poprawnie zakodowane, mogą doprowadzić do chaosu.

Kolejnym interesującym wnioskiem, do którego można dojść po dokładniejszej analizie kodu przedstawionego na listingach 3.1, 3.2 i 3.3, jest to, że za pomocą różnych dokumentów klient może żądać dokładnie tych samych danych. Taka elastyczność zapewnia klientowi ogromne możliwości, choć zarazem zwiększa liczbę możliwych żądań, co z kolei powiększa potencjalną płaszczyznę ataku na aplikację. Jeżeli analizator składni nie uwzględnia różnych sposobów, na jakie można przygotować zapytanie, staje się podatny na nieoczekiwane błędy.

## Pole

**Pole** to pojedynczy fragment informacji dostępnej w **zbiorze selekcji** (ang. *selection set*) operacji. Inaczej to lista ujęta w nawias klamrowy. W kolejnym fragmencie kodu polami są elementy `id`, `title` i `content`.

```
{
  id
  title
  content
}
```

Ponieważ te trzy pola znajdują się na najwyższym poziomie skróconej postaci zapytania, są znane również pod nazwą **pól najwyższego poziomu**. Pole także może mieć własny zbiór selekcji, co pozwala na przedstawianie skomplikowanych relacji danych. W kolejnym przykładzie najwyższego poziomu pole `owner` ma własny zbiór selekcji składający się z jednego zagnieżdżonego pola.

```
{
  id
  title
  content
  owner {
    name
  }
}
```

Tak więc zbiory selekcji tworzą pola, które z kolei mogą mieć własne zbiory selekcji, też zawierające pola. Czy przychodzą Ci na myśl jakiekolwiek związane z tym kwestie bezpieczeństwa? W rozdziale 5. wyjaśnimy, jak takie relacje cykliczne między polami mogą prowadzić do żądań rekurencyjnych i kosztownych, które mogą degradować wydajność działania i skutkować awarią serwera GraphQL.

Pola mają bardzo duże znaczenie podczas pracy z usługami GraphQL. Brak wiedzy na temat dostępnych pól może być niezwykle ograniczającym czynnikiem. Na szczęście w implementacjach wdrożono bardzo użyteczne narzędzie znane pod nazwą *sugestii nazwy pola* (ang. *field suggestion*). W przypadku gdy klient poda błędną nazwę pola, zwrócony przez serwer komunikat o błędzie będzie zawierał sugestię nazwy pola, do którego według serwera klient chciał uzyskać dostęp. Na przykład, jeśli zapytanie dotyczące pewnego

fragmentu tekstu w aplikacji DVGA będzie zawierało pole o nazwie `titl` (zwróć uwagę na błędny zapis), wówczas odpowiedź udzielona przez serwer będzie zawierała pewną sugestię dotyczącą nazwy pola.

```
"Cannot query field \"titl\" on type \"PasteObject\". Did you mean \"title\"?"
```

Dzięki funkcjonalności sugestii nazwy pola język GraphQL jest wygodnym, przyjaznym i prostym narzędziem nie tylko dla konsumentów API, ale także dla hakerów. Tę funkcjonalność możesz wykorzystać w celu znalezienia pól, o których istnieniu w przeciwnym razie nawet byś nie wiedział. Dokładne omówienie tej techniki ujawniania informacji znajdziesz w rozdziale 6.

## Argument

Podobnie jak w przypadku API REST, także klienty GraphQL mają możliwość wysyłania **argumentów** do różnych pól w zapytaniach, aby w ten sposób dostosować do własnych potrzeb wyniki zwracane przez te zapytania. Jeżeli raz jeszcze spojrzysz na rysunek 3.1 we wcześniejszej części rozdziału, zauważysz, jak argumenty mogą być implementowane na różnych poziomach — mamy tutaj na myśli pola i dyrektywy.

W kolejnym zapytaniu pole `users` ma argument `id` o wartości `1`. W razie braku wymienionego argumentu przedstawione zapytanie zwróciłoby pełną listę użytkowników w aplikacji DVGA. Argument pozwala na filtrowanie listy i umieszczenie na niej użytkowników o podanym identyfikatorze.

```
query {
  users(id: 1) {
    id
    username
  }
}
```

Zgodnie z oczekiwaniami odpowiedź udzielona na żądanie będzie zawierała pojedynczy obiekt użytkownika, składający się z identyfikatora i nazwy użytkownika.

```
{
  "data": {
    "users": [
      {
        "id": "1",
        "username": "admin"
      }
    ]
  }
}
```

Argumenty można przekazywać również do pól zagnieżdżonych. Spójrz na kolejne zapytanie.

```
query {
  users(id: 1)  {
    username(capitalize: true)
  }
}
```

W tym przypadku pole zagnieżdżone `username` ma argument o nazwie `capitalize`. Ten argument akceptuje wartość boolowską, stąd w omawianym przykładzie ma przypisaną wartość `true`. W aplikacji DVGA ten argument spowoduje, że GraphQL zmieni na wielką pierwszą literę zawartość pola `username` i zwróci ją w odpowiedzi udzielonej na żądanie, co w omawianym przykładzie spowoduje konwersję nazwy `admin` na `Admin`.

```
{
  "data": {
    "users": [
      {
        "username": "Admin"
      }
    ]
  }
}
```

Argumenty są *nieuporządkowane*, co oznacza, że zmiana ich kolejności nie będzie miała żadnego wpływu na logikę zapytania. W kolejnym przykładzie niezależnie od tego, który argument zostanie przekazany jako pierwszy, `limit` lub `public`, to nie będzie miało wpływu na sposób działania zapytania.

```
query {
  pastes(limit: 1, public: true){
    id
  }
}
```

Sposób przetwarzania i weryfikowania tych argumentów w całości zależy od aplikacji, różnice w implementacji mogą zaś prowadzić do powstania luk w zabezpieczeniach. Na przykład, skoro język GraphQL stosuje ściśle określone typy, przekazanie liczby całkowitej do argumentu oczekującego wartości tekstowej spowoduje wygenerowanie komunikatu o błędzie podczas weryfikacji. Jeżeli zamiast tego przekażesz ciąg tekstowy, wówczas weryfikacja na poziomie GraphQL zakończy się sukcesem. Mimo to aplikacja nadal powinna sprawdzić format takich danych wejściowych. Wartość w postaci np. adresu e-mail aplikacja może sprawdzić za pomocą wyrażenia regularnego dopasowującego adres e-mail bądź też szukać znaku `@` w przekazanej wartości.

Jeżeli aplikacja używa biblioteki dostarczającej niestandardowe typy skalarne dla adresu e-mail, wówczas taka biblioteka może samodzielnie przeprowadzać weryfikację, a tym samym twórcom aplikacji trudniej będzie popełnić błąd. Zewnętrzne biblioteki GraphQL, takie jak graphql-scalars (*https://github.com/Urigo/graphql-scalars*) dla JavaScriptu, dostarczają użyteczne i niestandardowe typy skalarne przeznaczone dla określonych zastosowań, np. jako znaczniki czasu, adresy IP, adresy URL witryn internetowych itd. Oczywiście w takich niestandardowych typach skalarnych również mogą istnieć luki w zabezpieczeniach. Na przykład znaleziona w bibliotece Pythona ipaddress luka w zabezpieczeniach (CVE-2021-29921) mogła umożliwić atakującemu pominięcie mechanizmu kontroli na podstawie adresu IP.

Jak możesz zobaczyć, argumenty zapewniają klientom ogromne możliwości w zakresie zmiany sposobu działania żądań i zarazem stanowią kolejny doskonały kierunek ataku. Skoro wartość argumentu pochodzi od klienta, potencjalnie może zawierać złośliwie działającą treść przeznaczoną do przeprowadzania ataków polegających na wstrzykiwaniu kodu. W rozdziale 8. dokładnie omówimy narzędzia i techniki używane do wykorzystywania luk w zabezpieczeniach argumentów, gdy ich wartości nie zostały poprawnie zabezpieczone przed atakami polegającymi na wstrzykiwaniu kodu.

## Alias

**Alias** pozwala klientowi na zmianę klucza odpowiedzi pola na inny. Na przykład w kolejnym fragmencie kodu dla pola `title` został użyty alias o nazwie `myalias`.

```
query {
   pastes {
      myalias:title
   }
}
```

Odpowiedź udzielona na żądanie będzie zawierała klucz `myalias` zamiast pierwotnej nazwy pola `title`.

```
{
  "data": {
    "pastes": [
      {
        "myalias": "My Title!"
      }
    ]
  }
}
```

Aliasy mogą okazać się użyteczne podczas pracy z identycznymi kluczami odpowiedzi. Zapoznaj się z zapytaniem przedstawionym na listingu 3.4.

*Listing 3.4. Powielone zapytania razem z odmiennymi wartościami argumentów*

```
query {
  pastes(public: false) {
    title
  }
  pastes(public: true) {
    title
  }
}
```

W tym zapytaniu dwukrotnie użyto pola `pastes`. W obu przypadkach został przekazany argument `public`, przy czym jego wartości są różne (`false` i `true`). Argument `public` pozwala na filtrowanie pod kątem określonego fragmentu tekstu na podstawie uprawnień: te publiczne są widoczne dla wszystkich klientów, podczas gdy te prywatne mogą być wyświetlane jedynie przez ich autorów. Zapytanie zamieszczone na listingu 3.4 skopiuj do klienta Altair, a następnie wykonaj je do aplikacji DVGA, co powinno spowodować wygenerowanie następujących danych wyjściowych:

```
{
  "errors": [
    {
      "message": "Fields \"pastes\" conflict because they have differing arguments.
      Use different aliases on the fields to fetch both if this was intentional.",
--cięcie--
}
```

Serwer GraphQL informuje o konflikcie powstałym podczas wykonywania tego zapytania. Ponieważ to samo zapytanie zostało wysłane z odmiennymi argumentami, serwer GraphQL nie jest w stanie przetworzyć ich razem. W takich przypadkach aliasy pokazują swoją użyteczność. Zapytanie może być zmodyfikowane w taki sposób, aby serwer uznał, że otrzymał dwa zapytania. Spójrz na listing 3.5, na którym pokazaliśmy, że użycie aliasów pozwala unikać konfliktów związanych z kluczami.

*Listing 3.5. Używanie aliasu w zapytaniu*

```
Query {
  queryOne:pastes(public: false) {
    title
  }
  queryTwo:pastes(public: true) {
    title
  }
}
```

W przedstawionej tutaj odpowiedzi udzielonej na żądanie zwróć uwagę na dwa klucze JSON, `queryOne` i `queryTwo`, dla poszczególnych aliasów wymienionych w zapytaniu zdefiniowanym na listingu 3.5. Poszczególne klucze JSON można traktować jako oddzielne odpowiedzi udzielone na różne zapytania.

```
{
  "data": {
    "queryOne": [
      {
        "title": "My Title!"
      }
    ],
    "queryTwo": [
      {
        "title": "Testing Testing"
      }
    ]
  }
}
```

**Uwaga**

*Wprawdzie alias zwykle może zawierać znaki alfanumeryczne, ale większość serwerów GraphQL w przypadku aliasów zawierających znaki specjalne będzie generowała błąd składni.*

Dotychczas aliasy wyglądały dość niewinnie. Zapewniamy jednak, że możesz je stosować jako broń. W rozdziale 5. wyjaśnimy, jak można używać aliasów podczas przeprowadzania różnego rodzaju ataków typu DoS, a w rozdziale 7. pokażemy, jak je wykorzystać do obejścia mechanizmu uwierzytelnienia.

## Fragment

**Fragment** pozwala klientowi wielokrotnie używać w zapytaniu GraphQL tego samego zbioru pól, aby w ten sposób zapewnić większą czytelność i uniknąć powtarzania pól. Zamiast powtarzać pola, można jednokrotnie zdefiniować fragment, a następnie wielokrotnie go używać w miejscach, w których jest potrzebny ten konkretny zbiór pól.

Do zdefiniowania fragmentu używa się słowa kluczowego `fragment`, po którym znajduje się dowolna nazwa, następnie słowo kluczowe `on`, na końcu zaś znajduje się nazwa obiektu typu.

```
fragment CommonFields on PasteObject {
  title
  content
}
```

W tym przypadku zdefiniowano fragment o nazwie `CommonFields`. Słowo kluczowe `on` powoduje, że ten fragment został zadeklarowany w odniesieniu do typu `PasteObject`, który zapewni dostęp do znanych już pól, takich jak `title` i `content`. Na listingu 3.6 pokazaliśmy przykład użycia tego fragmentu w zapytaniu.

*Listing 3.6. Zdefiniowanie fragmentu CommonFields i jego użycie w zapytaniu*

```
query {
  pastes {
    ...CommonFields
  }
}
fragment CommonFields on PasteObject {
   title
   content
}
```

Dzięki użyciu trzech kropek (...), nazywanych także **operatorem rozszczepienia**, do fragmentu `CommonFields` można odwoływać się w różnych miejscach zapytania, aby tym samym uzyskać dostęp do pól związanych z fragmentem tekstu, takich jak `title` i `content`. Nie ma żadnych ograniczeń w liczbie odwołań do fragmentu w danym zapytaniu.

**Uwaga**

*Na stronie GraphQL Working Group toczona jest nieustanna dyskusja dotycząca wprowadzenia argumentów dla fragmentu (*https://github.com/graphql/graphql-spec/issues/204*). W chwili gdy pisaliśmy te słowa, nie było jeszcze wiadomo, czy i kiedy taka możliwość zostanie dodana.*

Z perspektywy testów penetracyjnych istnieje możliwość utworzenia fragmentów, które będą odwoływały się do innych w taki sposób, aby odwołania cykliczne doprowadziły do stanu porównywalnego z atakiem typu DoS. Więcej informacji na ten temat znajdziesz w rozdziale 5.

## Zmienna

W operacji jako wartości argumentu można użyć **zmiennej**, co odbywa się przez zadeklarowanie zmiennej w dokumencie GraphQL. Zmienne okazują się użyteczne, ponieważ pozwalają uniknąć kosztownego tworzenia ciągów tekstowych w trakcie wykonywania zapytania.

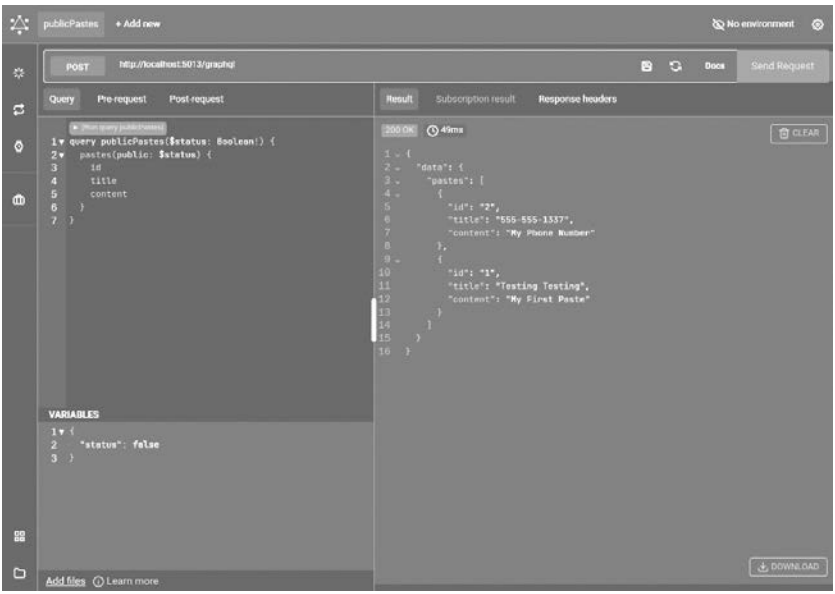
Zmienne definiuje się na początku operacji, tuż po jej nazwie. Na listingu 3.7 pokazaliśmy przykład zapytania, w którym użyto zmiennej.

*Listing 3.7. Zmienna status przekazana argumentowi public obiektu pastes*

```
query publicPastes($status: Boolean!){
  pastes(public: $status){
    id
    title
    content
  }
}
```

Przez użycie znaku dolara, `$`, zdefiniowano nazwę zmiennej, `status`, i jej typ, `Boolean`. Wykrzyknik umieszczony po typie zmiennej wskazuje, że jest ona wymagana do przeprowadzenia operacji.

Aby przypisać wartość zmiennej, można określić jej wartość domyślną w trakcie definiowania typu bądź też w dokumencie przekazać obiekt JSON zawierający nazwę zmiennej i jej wartość. W narzędziu klienta Altair zmienne definiuje się za pomocą panelu *Variables*, znajdującego się tuż pod panelem *Query* po lewej stronie okna, jak pokazaliśmy na rysunku 3.2.




*Rysunek 3.2. Panel Variables w narzędziu klienta Altair (lewy dolny róg okna)*

W omawianym przykładzie następuje przekazanie zmiennej o nazwie `status` razem z wartością boolowską `false`. Wymieniona wartość będzie użyta w miejscu każdego wystąpienia zmiennej w dokumencie. Zmienne ułatwiają wielokrotne używanie wartości przekazywanych argumentom w polach lub dyrektywach.

## Dyrektywa

**Dyrektywa** pozwala na **udekorowanie**, inaczej zmianę sposobu działania, pola w dokumencie. Zmiana sposobu działania może wpływać na to, jak dane pole będzie weryfikowane, przetwarzane bądź wykonywane przez aplikację. Dyrektywę można postrzegać jako „starszego brata" argumentu, ponieważ zapewnia mechanizm kontroli wyższego poziomu, np. warunkowe dołączanie lub pomijanie pól w zależności od pewnej logiki. Dyrektywy są dostępne na dwóch poziomach: zapytania i schematu. W obu przypadkach dyrektywę deklaruje się za pomocą znaku `@` i można w niej wykorzystać argumenty (podobnie jak pola).

Implementacje zwykle zapewniają wiele standardowych dyrektyw, a programiści API GraphQL mogą tworzyć własne według uznania. W przeciwieństwie do nazw operacji i aliasów, klienty mogą używać jedynie dyrektyw zdefiniowanych na serwerze. W tabeli 3.2 wymieniliśmy najczęściej stosowane dyrektywy domyślne, ich sposób użycia, a także miejsce ich zdefiniowania.

*Tabela 3.2. Najczęściej stosowane dyrektywy na poziomie schematu i zapytania*

| Lp. | Nazwa | Opis | Lokalizacja |
|---|---|---|---|
| 1. | `@skip` | Warunkowe pominięcie pola w udzielonej odpowiedzi | Zapytanie |
| 2. | `@include` | Warunkowe dołączenie pola do udzielonej odpowiedzi | Zapytanie |
| 3. | `@deprecated` | Sygnalizuje deprecjację komponentu schematu | Schemat |
| 4. | `@specifiedBy` | Określa niestandardowy typ skalarny (np. RFC) | Schemat |

Klienty mogą stosować dyrektywę `@skip` dla pola w celu jego dynamicznego pominięcia w odpowiedzi udzielonej na żądanie. Gdy konstrukcja warunkowa `if` w argumencie dyrektywy będzie przyjmowała wartość `true`, wówczas dane pole nie zostanie dołączone. Zapoznaj się z zapytaniem przedstawionym na listingu 3.8.

*Listing 3.8. Przykład użycia dyrektywy @skip do pominięcia pola owner w odpowiedzi udzielonej na zapytanie*

```
query pasteDetails($pasteOnly: Boolean!){
  pastes{
    id
    title
    content
    owner @skip(if: $pasteOnly) { ❶
      name
    }
  }
}

--cięcie--

{
  "pasteOnly": true
}
```

Kod zamieszczony na listingu 3.8 używa dyrektywy `@skip`, zawierającej konstrukcję warunkową `if`, do sprawdzenia wartości zmiennej boolowskiej `$pasteOnly` ❶. Jeżeli jej wartością jest `true`, całe pole `owner` (łącznie z polami zagnieżdżonymi) zostanie pominięte w udzielonej odpowiedzi.

Dyrektywa zapytania `@include` jest przeciwieństwem dyrektywy `@skip`. Spowoduje ona dołączenie pola (i jego pól zagnieżdżonych) tylko wtedy, gdy przekazany jej argument będzie miał wartość `true`.

Dyrektywa `@deprecated` jest inna niż przedstawione wcześniej `@skip` i `@include`, ponieważ klienty nie mogą jej używać w dokumentach zapytania. Dyrektywy `@deprecated`, znanej jako **dyrektywa na poziomie schematu**, używa się jedynie w definicjach schematów GraphQL. Pojawia się na końcu pola bądź definicji typu i wskazuje, że podane pole bądź typ nie są już obsługiwane.

Dyrektywa `@deprecated` ma opcjonalny argument `reason`, pozwalający programiście na zdefiniowanie komunikatu dla klientów bądź innych programistów, którzy będą próbowali używać danego pola. Ten komunikat będzie pojawiał się w miejscach takich jak odpowiedź udzielona na żądanie introspekcji bądź w sekcji dokumentacji narzędzi IDE GraphQL, takich jak GraphiQL Explorer i GraphQL Playground. Na listingu 3.9 możesz zobaczyć przykładowy schemat, w którym użyto dyrektywy `@deprecated`.

*Listing 3.9. Przykład użycia dyrektywy @deprecated w schemacie*

```
type PasteObject {
--cięcie--
  userAgent: String
  ipAddr: String @deprecated(reason: "We no longer log IP addresses")
  owner: OwnerObject
--cięcie--
}
```

Dodana niedawno dyrektywa `@specifiedBy` jest dyrektywą na poziomie schematu i pozwala na dostarczanie adresu URL prowadzącego do czytelnej dla człowieka specyfikacji niestandardowego typu skalarnego. W dalszej części rozdziału dowiesz się nieco więcej na temat jej używania.

Dyrektywy `@skip`, `@include`, `@deprecated` i `@specifiedBy` są wymagane, więc implementacja serwera GraphQL musi je obsługiwać, aby można było ją uznać za zgodną ze specyfikacją.

**Uwaga**

*Wraz z dopracowywaniem specyfikacji GraphQL i wprowadzaniem w niej kolejnych funkcjonalności można się spodziewać implementacji kolejnych dyrektyw domyślnych. Trwa dyskusja dotycząca dwóch kolejnych dyrektyw na poziomie zapytania, `@stream` i `@defer`, których klienty będą używały do przekazywania względnego priorytetu żądanych danych i podziału danych między wiele odpowiedzi. Więcej informacji na ten temat znajdziesz w sekcji RFC repozytorium GraphQL Working Group* (https://github.com/graphql/graphql-wg/blob/main/rfcs/DeferStream.md).

Dyrektywy niestandardowe pozwalają w implementacjach GraphQL na dodawanie kolejnych funkcjonalności bądź rozbudowę istniejących, które nie są powszechnie obsługiwane bądź używane w ekosystemie. Jednym z przykładów powszechnie przyjętej dyrektywy niestandardowej jest `@computed`. To oferująca potężne możliwości dyrektywa na poziomie schematu pozwala uniknąć konieczności tworzenia funkcji resolvera dla pól, których wartości można obliczyć na podstawie innych pól schematu. W kodzie zamieszczonym na listingu 3.10 pokazaliśmy, jak dyrektywa `@computed` może połączyć pola `firstName` i `lastName` w pole `fullName`.

*Listing 3.10. Dyrektywa @compute używana do złączania wartości dwóch pól*

```
type User {
  firstName: String
  lastName: String
```

```
  fullName: String @computed(value: "$firstName $lastName")
}
```

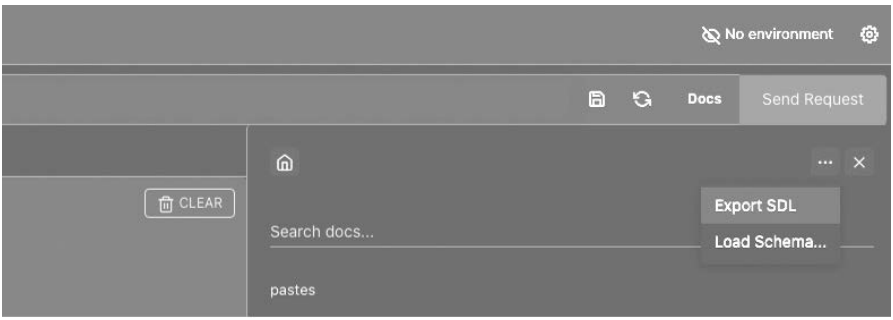
Potężne możliwości dyrektyw są zarazem ich największą słabością — pozostają właściwie nieuregulowane. Poza przedstawieniem ogólnej składni, specyfikacja GraphQL nie wspomina zbyt wiele na temat dyrektyw, co pozostawia twórcom implementacji serwerów pełną dowolność w zakresie projektowania architektury dyrektyw. Nie każda implementacja serwera GraphQL będzie obsługiwała te same dyrektywy. Jednak implementacje używające dyrektyw do modyfikacji sposobu działania języka GraphQL mogą wprowadzać ryzyko, o ile implementacja nie będzie przeprowadzona prawidłowo.

Używanie dyrektyw niestandardowych do rozbudowy możliwości GraphQL naraża implementacje na dodatkowe i nietypowe rodzaje ataków. Luka w zabezpieczeniach dyrektywy niestandardowej używanej przez popularną implementację GraphQL może mieć wpływ na setki organizacji. W rozdziale 5. wyjaśnimy, jak za pomocą dyrektyw można przeprowadzać ataki na serwery GraphQL.

# Typy danych

**Typ** w GraphQL definiuje obiekt niestandardowy i strukturę danych, wykorzystując do tego schemat GraphQL. Mamy sześć rodzajów typów: obiekt, skalar, wyliczenie, unię, interfejs i dane wejściowe. W tym podrozdziale przedstawimy poszczególne typy oraz wyjaśnimy ich przeznaczenie.

Przykładami będą typy zdefiniowane w schemacie aplikacji DVGA. Jeżeli chcesz otrzymać większy kontekst, za pomocą klienta Altair możesz pobrać pełny plik SDL dla aplikacji DVGA. W tym celu kliknij łącze *Docs* znajdujące się obok przycisku *Send Request*, a następnie kliknij wielokropek, aby uzyskać dostęp do opcji *Export SDL*, jak pokazaliśmy na rysunku 3.3.




*Rysunek 3.3. Funkcja Export SDL w narzędziu klienta Altair*

## Obiekt

Niestandardowe **typy obiektów** to grupa jednego lub więcej pól definiujących obiekty ściśle związane z domeną bądź aplikacją. Zapoznaj się z przedstawionym na listingu 3.11 fragmentem schematu aplikacji DVGA.

*Listing 3.11. Typ PasteObject w aplikacji DVGA*

```
type PasteObject {
  id: ID!
  title: String
  content: String
  public: Boolean
  userAgent: String
  ipAddr: String
  ownerId: Int
  burn: Boolean
  owner: OwnerObject ❶
}
```

Zdefiniowany został nowy niestandardowy obiekt typu o nazwie `PasteObject`. Pola tego obiektu zostały wymienione w nawiasie klamrowym. Niektóre być może rozpoznajesz, ponieważ były stosowane w zapytaniach GraphQL zaprezentowanych we wcześniejszej części rozdziału. Każde z tych pól używa standardowych w GraphQL typów skalarnych; wyjątkiem jest tutaj pole `owner`, które również jest niestandardowym typem obiektu.

Jeżeli przyjrzysz się polu `id`, dostrzeżesz na jego końcu wykrzyknik. Oznacza on, że każdy obiekt typu `Paste` wymaga pola `ID`, podczas gdy pozostałe pola mogą nie mieć przypisanych wartości. W przypadku pola wymaganego można się spotkać z określeniem *typ non-null*. Zwróć też uwagę na jednokierunkową relację między węzłami `Paste` i `Owner` ❶. Tego rodzaju relacje omówiliśmy w rozdziale 1. W praktyce ta relacja oznacza, że za pomocą obiektu `Paste` można żądać obiektu `Owner` i powiązanych z nim pól.

## Skalar

**Skalar** obejmuje wiele podstawowych, wbudowanych typów wartości, np. `ID`, `Int`, `Float`, `String` i `Boolean`. W przeciwieństwie do innych typów obiektów, skalar nie ma własnych pól.

Implementacja również może definiować własne typy skalarne. Spójrz na listing 3.12, na którym pokazaliśmy, jak aplikacja DVGA może wprowadzić w obiekcie `Paste` nowe pole o nazwie `createdAt`.

*Listing 3.12. Przykład definicji skalara*

```
scalar DateTime

type PasteObject {
  id: ID!
  title: String
  content: String
  public: Boolean
```

```
  userAgent: String
  ipAddr: String
  ownerId: Int
  burn: Boolean
  owner: OwnerObject
  createdAt: DateTime!
}
```

Podobnie jak w przypadku pola `ID`, także wartość pola `createdAt` może być przypisana automatycznie po utworzeniu obiektu, a jej wartością będzie niestandardowy typ skalarny o nazwie `DateTime`. Ten niestandardowy typ skalarny może pomóc w zapewnieniu poprawnej serializacji, formatowaniu i weryfikacji.

Niestandardowe typy skalarne mogą również używać wbudowanej dyrektywy `@specifiedBy` w celu dostarczenia klientom adresu URL zawierającego specyfikację danego typu. Na przykład niestandardowy typ skalarny `UUID` może zawierać adres URL specyfikacji IETF (ang. *internet engineering task force*).

```
scalar UUID @specifiedBy(url: "https://tools.ietf.org/html/rfc4122")
```

## Wyliczenie

**Wyliczenie** (ang. *enum*), znane również pod nazwą **typu wyliczeniowego**, to pola używane do zwrotu pojedynczej wartości tekstowej z listy dostępnych wartości. Na przykład aplikacja może pozwalać klientowi na wybór sposobu sortowania listy nazw użytkowników, która pojawi się w odpowiedzi udzielonej na żądanie. W tym celu można utworzyć wyliczenie o nazwie `UserSortEnum` przedstawiające rodzaje sortowania (np. według nazwy użytkownika, adresu e-mail, hasła bądź daty dołączenia).

```
enum UserSortEnum {
  ID
  EMAIL
  USERNAME
  DATE_JOINED
}
```

To wyliczenie `UserSortEnum` może być następnie użyte jako typ dla argumentu, takiego jak `order`, udostępnionego za pomocą typu danych wejściowych o nazwie `UserOrderedType`. (Typy danych wejściowych poznasz w dalszej części rozdziału). Na listingu 3.13 pokazaliśmy przykład schematu zawierającego wyliczenie.

*Listing 3.13. Sortowanie nazw użytkowników przeprowadzane na podstawie typu danych wejściowych używającego wyliczenia*

```
enum UserSortEnum {
  ID
  EMAIL
```

```
  USERNAME
  DATE_JOINED
}

input UserOrderType {
  sort: UserSortEnum!
}

type UserObject {
  id: Int!
  username: String!
}

type Query {
  users(limit: Int, order: UserOrderType): UserObject!
}
```

W omawianym przykładzie zdefiniowano wyliczenie razem z kilkoma polami: `ID`, `EMAIL`, `USERNAME` i `DATE_JOINED`. Następnie zdefiniowano typ danych wejściowych o nazwie `UserOrder➥Type`, zawierający pole o nazwie `sort` i typie `UserSortEnum`. Udostępnione zostaje zapytanie o nazwie `users`, które pobiera dwa argumenty, `limit` i `order`, przy czym typem argumentu `order` jest `UserOrderType`. To pozwala klientom na zwrot listy nazw użytkowników posortowanych na podstawie dowolnego elementu zdefiniowanego wyliczenia. Takie zapytanie może mieć przedstawioną tutaj postać.

```
query {
  users(limit: 100, order: {sort: ID})
}
```

Umożliwienie klientom sortowania za pomocą opcji zamieszczonych w `UserSortEnum` może być ryzykowne. Na przykład, jeśli klient będzie mógł sortować nazwy użytkowników według ich wartości `ID`, atakujący może uzyskać dostęp do pierwszego użytkownika utworzonego w systemie. Ten użytkownik będzie prawdopodobnie przedstawiał superadministratora lub wbudowane konto aplikacji. Wiedza o tym może pomóc w skoncentrowaniu wysiłku na ataku na cenne konta, które mogą mieć większe uprawnienia niż pozostałe konta użytkowników.

## Unia

**Unia** to typ zwracający jeden z wielu obiektów typu. Klient może wykorzystać unię w celu wykonania pojedynczego żądania do serwera GraphQL i pobrania listy obiektów. Spójrz na listing 3.14 przedstawiający zapytanie, w którym została użyta funkcjonalność wyszukiwania w aplikacji DVGA. Taka funkcjonalność pozwala klientowi na wyszukiwanie słowa kluczowego, którego użycie zwróci wiele obiektów `Users` i `Paste`.

*Listing 3.14. Funkcjonalność wyszukiwania w aplikacji DVGA*

```
query {
  search(keyword: "p") {
    ... on UserObject {
      username
    }
    ... on PasteObject {
      title
      content
    }
  }
}
```

Taka funkcjonalność wyszukiwania pomaga klientom w znajdywaniu zarówno fragmentów tekstu, jak i nazw użytkowników dopasowanych do słowa kluczowego z użyciem zaledwie jednego żądania. To okazuje się całkiem eleganckim rozwiązaniem. W kolejnym fragmencie kodu pokazaliśmy odpowiedź udzieloną na omawiane żądanie. Zapytanie zwraca listę dopasowanych pól fragmentów kodu, które mają literę *p* w tytule fragmentu bądź w nazwie użytkownika.

```
{
  "data": {
    "search": [
      {
        "title": "This is my first paste",
        "content": "What does your room look like?"
      },
      {
        "id": "2",
        "username": "operator"
      }
    ]
  }
}
```

Aby zaakceptować i wykonać zapytanie tego rodzaju, schemat może używać typu `union`. Na listingu 3.15 zamieściliśmy przykład unii (`union`) o nazwie `SearchResults`.

*Listing 3.15. Przykładowa definicja unii*

```
union SearchResults = UserObject | PasteObject

type UserObject {
  id: ID!
  username: String!
}

type PasteObject {
  id: ID!
  title: String
```

```
  content: String
--cięcie--
}

type Query {
  search(keyword: String): [SearchResults!]
}
```

Jak możesz zobaczyć, typ unii `SearchResults` powoduje złączenie obiektów użytkownika i fragmentu tekstu w pojedynczym typie obiektu. Następnie ten typ może być używany w pojedynczym zapytaniu wyszukiwania, które akceptuje argument `keyword` w postaci ciągu tekstowego.

## Interfejs

Innym sposobem na zwrot wielu typów w tym samym polu jest użycie tzw. **interfejsu**. Definiuje on listę pól, które muszą być dołączone do wszystkich implementujących je obiektów typu. W przypadku przedstawionego w poprzednim punkcie żądania wykorzystującego przykład unii pokazaliśmy, że można pobrać pole `username` dowolnego obiektu `User`, a także pola `title` i `content` dowolnego obiektu `Paset`, o ile zostały one dopasowane przez wzorzec wyszukiwania. Interfejs nie działa w taki sposób. Zamiast tego wymaga, aby te same pola znajdowały się w obu obiektach, i dopiero wtedy te obiekty mogą być złączone w odpowiedzi udzielanej klientowi.

Aby wspomnianą wcześniej funkcjonalność wyszukiwania zaimplementować za pomocą interfejsu zamiast unii, można skorzystać ze schematu przedstawionego na listingu 3.16.

*Listing 3.16. Przykładowa definicja interfejsu*

```
interface SearchItem {
  keywords: [String!]
}

type UserObject implements SearchItem {
  id: ID!
  username: String!
  keywords: [String!]
}

type PasteObject implements SearchItem {
  id: ID!
  title: String
  content: String
  keywords: [String!]
--cięcie--
}

type Query {
  search(keyword: String): [SearchItem!]!
}
```

W kodzie został utworzony interfejs typu `SearchItem` razem z polem listy `keywords` przechowującym wartości tekstowe. Każdy obiekt typu, który ma zaimplementować ten interfejs, będzie musiał dołączyć pole `keywords`. Następnie to pole zostanie zdefiniowane w obiektach `UserObject` i `PasteObject`. Teraz klient może wykonywać zapytania wyszukiwania podobnie jak w przypadku kodu przedstawionego na listingu 3.15 we wcześniejszej części rozdziału oraz pobierać wszystkie obiekty użytkowników i fragmentów tekstu, w których zostało użyte dane słowo kluczowe.

Interfejs może powodować problemy w aplikacjach, w których słabo zaimplementowano mechanizm autoryzacji. Jednym ze sposobów na implementację autoryzacji w GraphQL jest użycie niestandardowych dyrektyw na poziomie schematu. Skoro interfejs definiuje pola przeznaczone do używania przez inne obiekty, każde pole przechowujące dane wrażliwe i nieudekorowane poprawnie może zostać przypadkowo ujawnione. Ogromne pliki SDL mogą składać się z tysięcy wierszy i zawsze istnieje niebezpieczeństwo, że programista zapomni o dodaniu odpowiedniej dyrektywy autoryzacyjnej. Temat autoryzacji dokładnie omówimy w rozdziale 8.

## Dane wejściowe

Argument ma możliwość akceptowania wartości różnych typów, np. skalarów. Jednak gdy zachodzi potrzeba przekazania na serwer ogromnych i skomplikowanych danych wejściowych, wówczas można wykorzystać typ danych wejściowych, aby w ten sposób uprościć żądania. **Typ danych wejściowych** jest w zasadzie taki sam jak obiekt typu, ale może być używany jedynie jako dane wejściowe dla argumentów. Tego rodzaju typy pomagają w układaniu żądań klientów i ułatwiają klientom wielokrotne używanie danych wejściowych w wielu argumentach. Dopracowane wdrożenia GraphQL używają typów danych wejściowych w celu przygotowania lepszej struktury API i ułatwiają odczytywanie dokumentacji schematu.

Pokażemy teraz typ danych wejściowych w akcji. W kodzie zamieszczonym na listingu 3.17 zadeklarowano zmienną `$input`, której wartość jest typu `UserInput!`. Następnie tę zmienną danych wejściowych przekazano argumentowi `userData` mutacji `createUser`.

*Listing 3.17. Przykład użycia typu danych wejściowych w mutacji*

```
mutation newUser($input: UserInput!) {
  createUser(userData: $input) {
    user {
      username
    }
  }
}
```

Jak już wyjaśniliśmy we wcześniejszej części rozdziału, aby aplikacji przekazać dane wejściowe, trzeba utworzyć obiekt JSON przedstawiający typ `UserInput!` i przypisać go kluczowi danych wejściowych, jak pokazaliśmy na listingu 3.18.

*Listing 3.18. Definicja danych wejściowych*

```
{
  "input": {
    "username": "tom",
    "password": "secret",
    "email": "tom@example.com"
  }
}
```

W przypadku narzędzi takich jak klienty Altair lub GraphiQL Explorer kod JSON zamieszczony na listingu 3.18 będzie zdefiniowany w panelu *Variables* klienta.

Typy danych wejściowych zapewniają klientom możliwość pominięcia weryfikacji typu, co może, choć nie musi, uszkodzić logikę sprawdzania poprawności danych. Na przykład we wcześniejszej części rozdziału wyjaśniliśmy, jak nowy niestandardowy typ skalarny może nie poradzić sobie z weryfikacją wartości przekazywanych przez użytkownika, takich jak adresy IP lub adresy e-mail. Problemy związane ze sprawdzeniem poprawności adresu e-mail mogą pozwolić atakującemu na ominięcie formularza rejestracji i procesu logowania bądź też przeprowadzić operację wstrzyknięcia kodu.

# Introspekcja

Po zapoznaniu się z językiem GraphQL i jego systemem typów łatwiej możesz dostrzec różnice, jakie z perspektywy klienta występują w API REST i API GraphQL. Domyślnie GraphQL zapewnia klientom ogromne możliwości. Jednak na tym nie koniec.

Bezsprzecznie jedną z najpotężniejszych funkcjonalności GraphQL jest **introspekcja**, czyli wbudowane narzędzie pozwalające klientom na odkrywanie działań, które mogą być wykonywane z wykorzystaniem API GraphQL. Introspekcja umożliwia klientom wykonywanie do serwera GraphQL zapytań w celu pobierania informacji na temat schematu, które będą obejmowały dane dotyczące zapytań, mutacji, subskrypcji, dyrektyw, typów, pól itd. Dla hakera taka funkcjonalność może być prawdziwą kopalnią złota, wspomagającą etapy rozeznania, profilowania, zbierania danych i analizy kierunków ataku. Warto się dowiedzieć, jak można wykorzystać te informacje.

System introspekcji GraphQL ma siedem typów, które można wykorzystać w celu wykonywania zapytań do schematu. W tabeli 3.3 wymieniliśmy te typy.

W kodzie zamieszczonym na listingu 3.19 typ introspekcji `__Schema` został użyty w aplikacji DVGA.

*Listing 3.19. Przykład zapytania introspekcji*

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

*Tabela 3.3. Typy systemu introspekcji w GraphQL*

| Typ introspekcji | Opis |
|---|---|
| `__Schema` | Zapewnia wszystkie informacje dotyczące schematu usługi GraphQL |
| `__Type` | Zapewnia wszystkie informacje dotyczące typu |
| `__TypeKind` | Zapewnia informacje dotyczące różnych typów (skalar, obiekt, interfejs, unia, wyliczenie itd.) |
| `__Field` | Zapewnia wszystkie informacje dotyczące poszczególnych pól obiektu lub interfejsu typu |
| `__InputValue` | Zapewnia informacje dotyczące argumentów pól i dyrektyw |
| `__EnumValue` | Zapewnia informacje o jednej z możliwych wartości wyliczenia |
| `__DirectValue` | Zapewnia wszystkie informacje dotyczące dyrektyw, zarówno wbudowanych, jak i niestandardowych |

Zapytanie introspekcji do najwyższego poziomu pola spowoduje pobranie wszystkich informacji dostępnych za pomocą obecnie używanego schematu GraphQL. Możesz dalej uściślić interesujące Cię dane przez wskazanie zapytaniu, aby wyszukało wszystkie typy (`types`) i pobrało ich nazwy (`names`).

W kolejnym fragmencie kodu pokazaliśmy, jak GraphQL wyświetla odpowiedź na przedstawione wcześniej zapytanie introspekcji.

```
{
  "data": {
    "__schema": {
      "types": [
--cięcie--
        {
          "name": "PasteObject"
        },
        {
          "name": "ID"
        }
--cięcie--
        {
          "name": "String"
        },
        {
          "name": "OwnerObject"
        },
        {
          "name": "UserObject"
        }
--cięcie--
      ]
    }
}
```

W odpowiedzi udzielonej na żądanie można zobaczyć wiele nazw typów. Niektóre z nich, np. `ID`, `String` i `PasteObject`, powinny być Ci znane. Wiesz już, że `ID` i `String` to wbudowane w GraphQL typy skalarne. Z kolei nazwy `PasteObject`, `OwnerObject` i `UserObject` powinny natychmiast przyciągnąć uwagę hakera, ponieważ są to obiekty typów niestandardowych zdefiniowanych przez programistę. Warto dowiedzieć się nieco więcej na ich temat.

Typ introspekcji `__type` pozwala na pobieranie dalszych informacji na temat wskazanych typów. Kod na listingu 3.20 zawiera przykład zapytania o potężnych możliwościach, które będzie w stanie odkryć wszystkie pola i ich typy we wskazanym obiekcie typu niestandardowego.

*Listing 3.20. Zapytanie introspekcji pozwalające na odkrywanie pól we wskazanym obiekcie*

```
query {
  __type(name: "PasteObject") {
    name
    kind
    fields {
      name
      type {
        name
        kind
      }
    }
  }
}
```

W omawianym przykładzie postanowiliśmy dowiedzieć się nieco więcej na temat typu `PasteObject`. Zwróć uwagę, że podaliśmy nie tylko nazwę typu, ale również jego rodzaj (`kind`), co spowoduje zwrot introspekcji `__TypeKind` dla tego obiektu. Pobrane będą wszystkie pola `PasteObject`, ich nazwy, typy i rodzaje. Spójrz na odpowiedź udzieloną na omawiane żądanie.

```
"__type": {
      "name": "PasteObject",
      "kind": "OBJECT",
      "fields": [
        {
          "name": "id",
          "type": {
            "name": null,
            "kind": "NON_NULL"
          }
        },
        {
          "name": "title",
          "type": {
            "name": "String",
            "kind": "SCALAR"
          }
        },
```

```
        --cięcie--
        {
          "name": "content",
          "type": {
            "name": "String",
            "kind": "SCALAR"
          }
        },
        {
          "name": "owner",
          "type": {
            "name": "OwnerObject",
            "kind": "OBJECT"
          }
        }
      ]
}
```

Struktura wykonanego zapytania introspekcji odpowiada udzielonej na nie odpowiedzi. Dzięki temu można poznać całą listę pól, do których można wykonywać żądania, a także ich typów.

Pola zawierające dane wrażliwe, przeznaczone dla pracowników bądź jedynie do użytku wewnętrznego, łatwo ujawnić publicznie, jeśli są dołączone do schematu GraphQL, a introspekcja jest włączona. Jednak introspekcja to niejedyny sposób na odkrywanie informacji na temat pól. Można ją uznać za odpowiednik pliku definicji API REST Swagger (OpenAPI). Pozwala ustalać obsługiwane zapytania, mutacje i subskrypcje oraz akceptowane przez nie argumenty, a także ujawnia, jak je konstruować i wykonywać. To z kolei pozwala określać sposoby na przygotowywanie operacji o złośliwym działaniu.

Więcej na temat introspekcji dowiesz się w rozdziale 6., w którym skoncentrujemy się na technikach ujawniania informacji i przeznaczonych do tego narzędziach.

# Weryfikacja i wykonywanie zapytania

Wszystkie zapytania GraphQL są sprawdzane pod kątem ich poprawności ze schematem oraz z systemem typów, a dopiero później są przekazywane do serwera, który zajmuje się ich przetwarzaniem. Na przykład, gdy klient wykonuje zapytanie dotyczące określonych pól, wówczas implementacja GraphQL przeprowadza weryfikację schematu, aby sprawdzić, czy wszystkie żądane pola istnieją we wskazanym typie. Jeżeli pole nie istnieje w schemacie bądź też nie jest powiązane z danym typem, wówczas zapytanie zostanie oznaczone jako niepoprawne i nie będzie wykonane.

Specyfikacja GraphQL określa kilka typów weryfikacji. Obejmują one sprawdzanie dokumentu, operacji, pola, argumentu, fragmentu, wartości, dyrektywy i zmiennej. We wspomnianym przed chwilą przykładzie była przeprowadzana weryfikacja pola. Inne rodzaje weryfikacji, np. dyrektywy, mogą obejmować sprawdzenie, czy dyrektywa przekazana przez klienta została rozpoznana w schemacie i jest obsługiwana przez implementację.

Istnieją znaczne różnice w sposobach, w jakie implementacje GraphQL interpretują specyfikację GraphQL i zapewniają z nią zgodność, a w szczególności — w jakie zajmują się obsługą odpowiedzi na niepoprawne żądania. Wspomniane różnice mogą być wykrywane przez narzędzia typu Graphw00f (jest dokładnie omówione w następnym rozdziale). Skoro dokładny proces weryfikacji serwera ujawnia informacje w zakresie jego dojrzałości pod kątem zapewnienia bezpieczeństwa, bardzo ważne jest przeanalizowanie słabych punktów implementacji. W tym miejscu pomocna okazuje się tzw. macierz zagrożeń GraphQL (ang. *GraphQL Threat Matrix*).

GraphQL Threat Matrix (*https://github.com/nicholasaleks/graphql-threat-matrix*) to opracowany przez autorów książki framework bezpieczeństwa dla GraphQL. Używają go łowcy nagród, badacze zajmujący się kwestiami bezpieczeństwa oraz hakerzy do odkrywania luk w zabezpieczeniach istniejących w wielu różnych implementacjach GraphQL. Na rysunku 3.4 pokazaliśmy interfejs graficzny tego narzędzia.

**GraphQL Threat Matrix**

| Implementation | Validations | Field Suggestions | Query Depth limit | Query Cost Analysis | Automatic Persisted Queries | Introspection | Debug Mode | Batch Requests |
|---|---|---|---|---|---|---|---|---|
| wp-graphql | 38 | ✓ | ⚠ | ✗ | ✗ | ⚠ | ⚠ | ✓ |
| graphql-php | 37 | ✓ | ⚠ | ⚠ | ✗ | ✓ | ⚠ | ⚠ |
| Apollo | 34 | ✓ | ⚠ | ⚠ | ✓ | ✓ | ✓ | ✓ |
| graphql-yoga | 34 | ✓ | ⚠ | ✗ | ✗ | ⚠ | ⚠ | ⚠ |
| graphene | 34 | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ⚠ |
| Ariadne | 34 | ✓ | ⚠ | ⚠ | ✗ | ✓ | ⚠ | ✗ |
| Strawberry | 34 | ✓ | ⚠ | ✗ | ✗ | ✓ | ✗ | ✗ |
| graphql-ruby | 28 | ✓ | ✗ | ⚠ | ⚠ | ✓ | ✗ | ✓ |
| Sangria | 27 | ✓ | ⚠ | ⚠ | ✗ | ✓ | ✗ | ⚠ |
| Tartiflette | 26 | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ |
| graphql-java | 26 | ✓ | ⚠ | ⚠ | ✗ | ✓ | ✗ | ⚠ |
| gqlgen | 25 | ✓ | ✗ | ⚠ | ⚠ | ✓ | ⚠ | ⚠ |
| Dgraph | 25 | ✓ | ✗ | ✗ | ⚠ | ✓ | ✗ | ✗ |
| graphql-go | 24 | ✓ | ✗ | ✗ | ✗ | ✓ | ⚠ | ✗ |
| juniper | 24 | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ⚠ |
| Diana.jl | 10 | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ |
| gql-dart/gql | 9 | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ |
| Agoo | 0 | ✗ | ✗ | ✗ | ✗ | ✓ | ⚠ | ✗ |

*Rysunek 3.4. Interfejs graficzny frameworka GraphQL Threat Matrix*

Framework przeprowadza analizę, śledzenie i porównanie najczęściej używanych implementacji, wyszukuje obsługiwane przez nie operacje sprawdzenia, sprawdza domyślną konfigurację zabezpieczeń, dostępne funkcje oraz skanuje pod kątem najbardziej znanych luk w zabezpieczeniach. Używają go zarówno hakerzy, jak i obrońcy systemów. Wprawdzie

wiedza na temat sposobów zaatakowania implementacji ma krytyczne znaczenie, ale równie ważna jest możliwość podejmowania decyzji na podstawie informacji dotyczących danej implementacji.

Po zakończonej sukcesem weryfikacji żądanie GraphQL zostaje wykonane przez serwer. Za udzielenie odpowiedzi na żądanie są odpowiedzialne funkcje resolvera, omówione w rozdziale 1.

# Najczęściej spotykane słabe strony

W tym podrozdziale przedstawimy ogólne omówienie najczęściej spotykanych słabości w GraphQL. W późniejszych rozdziałach przeprowadzimy zaś testy penetracyjne pod kątem każdej z klas luk w zabezpieczeniach, a także omówimy kod exploitów pozwalających na wykorzystanie tych luk.

## Reguła specyfikacji i słaba strona implementacji

W specyfikacji GraphQL zdefiniowano reguły, zasady projektowe oraz praktyki standardowe. Jeżeli kiedykolwiek zechcesz samodzielnie opracować implementację GraphQL, wówczas powinna być ona zgodna z tym dokumentem, również pod względem sposobu formatowania odpowiedzi, przeprowadzania weryfikacji argumentów itd.

Zapoznaj się teraz z przykładami dwóch reguł pochodzących ze specyfikacji GraphQL.

> Argumenty mogą być podawane w dowolnej kolejności bez zmiany ich znaczenia.
>
> Element *data* w odpowiedzi udzielonej na żądanie będzie wynikiem wykonania wskazanej operacji.

Te reguły są dość proste. Zgodnie z pierwszą kolejność argumentów w zapytaniu nie powinna mieć wpływu na odpowiedź udzielaną przez serwer. Z kolei druga wyjaśnia, że odpowiedź serwera GraphQL musi być zwrócona jako część pola JSON *data*.

Zapewnienie zgodności z tymi regułami należy do obowiązków programisty. W tym miejscu mogą pojawiać się rozbieżności. W rzeczywistości twórców specyfikacji GraphQL nie interesuje, w jaki sposób implementacja będzie zapewniała zgodność ze specyfikacją.

> Wymagania zgodności wyrażone jako algorytmy mogą być spełnione przez implementację tej specyfikacji w dowolny sposób, o ile otrzymany wynik jest zgodny ze specyfikacją.

Żeby zobaczyć przykład różnic w zachowaniu poszczególnych implementacji, zapoznaj się z projektem graphql-php (*https://github.com/webonyx/graphql-php*). To implementacja typu open source stworzona w PHP i oparta na referencyjnej bibliotece implementacji GraphQL o nazwie GraphQL.js (*https://github.com/graphql/graphql-js*).

Jednak gdy przyjrzysz się dokładnie temu, jak graphql-php obsługuje aliasy, wówczas zauważysz różnicę względem wielu innych implementacji: klient ma możliwość stosowania aliasów zawierających znaki specjalne, takie jak $. Takie drobne różnice między implementacjami nie tylko pomagają hakerom w ustaleniu technologii kryjącej się za usługą API GraphQL (więcej informacji na ten temat znajdziesz w następnym rozdziale), ale również ułatwiają przygotowanie specjalnej zawartości, która będzie wpływała na

usługi wykorzystujące określone implementacje. Co więcej, taki zróżnicowany sposób działania oznacza, że luka w zabezpieczeniach znaleziona w jednej implementacji nie musi mieć wpływu na inne implementacje.

Haker często odwołuje się do dokumentu projektowego aplikacji, by lepiej zrozumieć, jak powinna ona działać w porównaniu z jej faktycznym działaniem w praktyce. Bardzo często będziesz odkrywać rozbieżności. Na przykład wyobraź sobie, że dokument projektowy aplikacji ma zdefiniowaną następującą regułę:

> Aplikacja musi mieć możliwość otrzymania adresu URL od klienta, pobrania go poprzez sieć i udzielenia odpowiedzi klientowi.

Ta reguła jest bardzo ogólnikowa, nie wyjaśnia sposobu zabezpieczenia tej funkcji oraz nie wskazuje, na co programista powinien zwrócić uwagę podczas jej implementowania. Jednak w przypadku funkcjonalności pobierającej treść z adresu URL wskazanego przez użytkownika wiele może pójść źle. Atakujący może zastosować dowolne z przedstawionych tutaj podejść.

- Określić w adresie URL prywatny adres IP (np. 10.1.1.1), co w praktyce pozwoli na zapewnienie dostępu do wewnętrznych zasobów serwera, na którym znajduje się aplikacja.
- Określić zdalny adres URL zawierający kod o złośliwym działaniu. Serwer będzie pobierał kod i w ten sposób trafi na niego oprogramowanie typu malware.
- Określić adres URL prowadzący do ogromnego pliku, wykorzystać wszystkie zasoby serwera oraz wpłynąć na możliwość używania aplikacji przez innych użytkowników.

To tylko wybrane ze szkodliwych rodzajów działań. Jeżeli programista nie uwzględni takich scenariuszy podczas opracowywania implementacji, wówczas każdy użytkownik tak zbudowanej aplikacji będzie narażony na wymienione luki w zabezpieczeniach.

Utworzenie oprogramowania wolnego od błędów jest trudnym zadaniem (praktycznie niemożliwe jest uniknięcie jakichkolwiek błędów). Im więcej wiesz na temat aplikacji i im bardziej się w nią zagłębisz, tym większe prawdopodobieństwo, że uda Ci się znaleźć w niej lukę w zabezpieczeniach.

## Odmowa usług

Jedna z dominujących klas luk w zabezpieczeniach GraphQL jest związana z atakami typu DoS. Tego rodzaju luki w zabezpieczeniach mogą prowadzić do degradacji wydajności działania zaatakowanego systemu bądź też do całkowitego wyczerpania dostępnych w nim zasobów. System może nie mieć możliwości udzielania odpowiedzi na żądania klientów, a nawet ulec całkowitej awarii. W rozdziale wyjaśniliśmy, jak pola i relacje obiektu, aliasy, dyrektywy i fragmenty potencjalnie mogą być użyte jako kierunki ataku na usługę GraphQL, ponieważ te możliwości zapewniają klientom API ogromną kontrolę nad strukturą zapytania i sposobami jego wykonywania.

W rozdziale 5. dowiesz się, że te potężne możliwości pozwalają również klientom na tworzenie bardzo skomplikowanych zapytań, które w praktyce doprowadzą do degradacji

wydajności działania serwera GraphQL, o ile nie będą zastosowane odpowiednie mechanizmy bezpieczeństwa. W książce przedstawimy cztery sposoby, na jakie klient może tworzyć kosztowne zapytania. Mogą one przeciążyć serwer GraphQL oraz prowadzić do powstania warunków przypominających atak typu DoS.

## Ujawnienie informacji

Powszechnie spotykaną słabością w wielu aplikacjach internetowych jest przypadkowe ujawnienie danych, które stają się dostępne publicznie bądź grupie użytkowników nieautoryzowanych, tak że ci mogą uzyskać dostęp do tych aplikacji. Ujawnienie informacji może mieć wiele przyczyn, a systemy, którym powierzono ochronę informacji wrażliwych, takich jak dane osobowe, powinny mieć wiele warstw mechanizmów wykrywania ataków i obrony przed nimi, aby nie dopuścić do ujawnienia informacji.

W przypadku technologii GraphQL hakerzy mogą sprawdzać i pobierać dane z API na wiele sposobów. W rozdziale 6. wyjaśnimy, jak można używać zapytań introspekcji do wyszukiwania pól, które mogą zawierać poufne informacje. Omówimy również narzędzia pozwalające wykorzystać sugestie nazw pól i komunikaty o błędach do poznania ukrytych modeli danych oraz manipulowania środowiskami GraphQL, w których nie można używać zapytań introspekcji.

## Błędy w mechanizmach uwierzytelnienia i autoryzacji

W każdym systemie architektury API uwierzytelnienie i autoryzacja to skomplikowane mechanizmy bezpieczeństwa. Na pewno nie pomaga to, że w specyfikacji GraphQL powstrzymano się od dostarczenia wskazówek związanych ze sposobami implementacji tych mechanizmów. Ten brak często będzie skłaniał inżynierów do implementacji własnych mechanizmów uwierzytelnienia i autoryzacji, opartych na rozwiązaniach typu open source, opracowanych przez firmę tworzącą daną aplikację bądź pochodzących z firm trzecich.

Większość luk w zabezpieczeniach związanych z uwierzytelnieniem i autoryzacją w GraphQL ma źródło w dokładnie tych samych problemach, które występują w tradycyjnych interfejsach API, czyli w braku odpowiedniej ochrony przed atakami typu brute force, błędach w logice i kiepskim sposobie tworzenia kodu, pozwalającym na zupełne pominięcie mechanizmów bezpieczeństwa. W rozdziale 7. omówimy kilka najczęściej spotykanych w GraphQL strategii dotyczących uwierzytelnienia i autoryzacji oraz wyjaśnimy, jak można je pokonać za pomocą aliasów, zapytań wsadowych oraz starych, dobrych błędów w logice.

## Wstrzykiwanie kodu

Związane ze wstrzykiwaniem kodu luki w zabezpieczeniach mogą mieć dewastujący wpływ na dane aplikacji. Wprawdzie frameworki są obecnie coraz lepiej chronione przed takimi zagrożeniami dzięki zastosowaniu wielu metod zabezpieczeń, ale niebezpieczeństwo wstrzyknięcia kodu wciąż istnieje. Podobnie jak REST i SOAP, także GraphQL jest uwzględniony na liście OWASP (ang. *open web application security project*), czyli liście najczęściej spotykanych luk w zabezpieczeniach aplikacji internetowych. Oparta na GraphQL

aplikacja może być podatna na ataki wstrzykiwania kodu, jeśli będzie akceptować i przetwarzać niesprawdzone dane pochodzące od klienta.

W przypadku języka GraphQL złośliwie działający klient ma wiele możliwości wstrzykiwania kodu do oprogramowania serwera, np. jako argumentów zapytania, argumentów pól i argumentów dyrektyw oraz jako mutacji. Poszczególne implementacje GraphQL różnią się poziomem zgodności ze specyfikacją GraphQL, co z kolei prowadzi do różnic w sposobach obsługi, oczyszczania i weryfikowania danych pochodzących od klientów. W rozdziale 8. dokładnie omówimy luki w zabezpieczeniach pozwalające na wstrzykiwanie kodu oraz przedstawimy różne sposoby na uzyskiwanie dostępu do systemów backendu.

# Podsumowanie

W tym momencie już wiesz, czym jest język GraphQL oraz jak można wykorzystać jego słabości za pomocą wybranych kierunków ataku. Prawdopodobnie czujesz się dość komfortowo w pracy z językiem GraphQL, znasz anatomię zapytań oraz jego komponentów wewnętrznych, takich jak operacje, pola i argumenty. Zacząłeś też korzystać ze zbudowanego w poprzednim rozdziale laboratorium GraphQL przez użycie klienta Altair w celu wykonania wielu zapytań do aplikacji DVGA. Z perspektywy serwera znasz już najważniejsze komponenty tworzące system typów GraphQL oraz role odgrywane przez te typy we wspomaganiu struktury schematów GraphQL i zapytań introspekcji.

Przygotowałeś sobie także grunt, z którego będziesz mógł przeprowadzić przyszłe ataki na GraphQL. Wskazaliśmy słabe strony i niedociągnięcia w specyfikacji GraphQL oraz wyjaśniliśmy, jak implementacje interpretują i rozszerzają funkcjonalność, której nie uregulowano w specyfikacji. Podążaj tym szlakiem i kontynuuj swoją działalność hakera atakującego GraphQL.


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

# *Lektura obowiązkowa dla każdego, kto zajmuje się bezpieczeństwem API!*

Corey Ball, autor książki ***Hakowanie interfejsów API***

GraphQL powstał jako alternatywa dla REST API. Charakteryzuje się większą elastycznością, wydajnością i prostotą użytkowania. Technologia ta skupiła na sobie uwagę wielu firm, gdyż umożliwia optymalizację wydajności działania, skalowanie i ułatwia wdrażanie nowych rozwiązań. Rosnąca popularność GraphQL nie idzie jednak w parze z wiedzą o lukach w zabezpieczeniach i exploitach zagrażających API GraphQL.

Dzięki tej książce dowiesz się, jak testować zabezpieczenia API GraphQL technikami ofensywnymi, takimi jak testy penetracyjne. Zdobędziesz i ugruntujesz wiedzę o GraphQL, niezbędną dla analityka bezpieczeństwa czy inżyniera oprogramowania. Nauczysz się skutecznie atakować API GraphQL, co pozwoli Ci wzmocnić procedury, stosować zautomatyzowane testy bezpieczeństwa w potoku ciągłej integracji i wdrażania, a ponadto efektywnie weryfikować mechanizmy zabezpieczeń. Zapoznasz się również z raportami o znalezionych lukach w zabezpieczeniach i przejrzysz kod exploitów, a także przekonasz się, jak wielki wpływ wywierają na działalność przedsiębiorstw.

**W książce między innymi:**

- zbieranie dokładnych informacji o celu ataku
- zabezpieczenie API przed atakami typu DoS i niebezpiecznymi konfiguracjami serwera GraphQL
- podszywanie się pod użytkownika z uprawnieniami administratora
- wykrywanie luk w zabezpieczeniach w celu atakowania techniką wstrzykiwania kodu
- ataki typu XSS i SSRF, przechwytywanie sesji WebSocket
- pozyskiwanie informacji wrażliwych

Autorzy są wybitnymi inżynierami bezpieczeństwa i współzałożycielami grupy DEFCON Toronto. **Nick Aleks** specjalizuje się w bezpieczeńswie ofensywnym i od ponad dekady hakuje wszystko, nawet inteligentne budynki. **Dolev Farhi** prowadzi zespoły inżynierów bezpieczeństwa w branży technologii finansowych. Hobbystycznie analizuje luki w zabezpieczeniach urządzeń IoT, uczestniczy w wyzwaniach CFT i dodaje exploity do bazy danych Exploit Database.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
*Sięgnij po więcej!* ▶
ISBN 978-83-289-1124-6
9 788328 911246
Cena: 79,00 zł
no starch press