helion.pl

» Idź do
- Spis treści
- Przykładowy rozdział

» Katalog książek
- Katalog online
- Zamów drukowany katalog

» Twój koszyk
- Dodaj do koszyka

» Cennik i informacje
- Zamów informacje o nowościach
- Zamów cennik

» Czytelnia
- Fragmenty książek online

Bezpieczeństwo aplikacji tworzonych w technologii Ajax

Autor: Billy Hoffman, Bryan Sullivan
Tłumaczenie: Robert Górczyński
ISBN: 978-83-246-1693-0
Tytuł oryginału: Ajax Security
Format: 168x237, stron: 496

Bezpieczeństwo aplikacji WWW w Twoich rękach!

- Jakie ataki Ci grożą?
- Jak zapewnić bezpieczeństwo Twojej aplikacji WWW?
- Jak testować aplikację AJAX?

Technologia AJAX zdobyła rynek aplikacji internetowych – to fakt. Siła i szybkość, z jaką tego dokonała, robią wrażenie. Niestety, wysoka wydajność w zdobywaniu rynku odbiła się negatywnie na jakości kodu i jego odporności na ataki. Oczywiście wiele rozwiązań spełnia najwyższe standardy jakości i bezpieczeństwa, ale istnieje jeszcze wiele serwisów, które powinny o to zadbać.

Jeżeli zastanawiasz się, czy Twój serwis korzystający z AJAX jest odpowiednio zabezpieczony, ta książka odpowie na to pytanie! Mało tego, dzięki „Bezpieczeństwo aplikacji tworzonych w technologii Ajax" dowiesz się, jak optymalnie zabezpieczyć Twoją aplikację, a także poznasz rodzaje ataków, na które ta aplikacja może być narażona. Poznasz także charakterystykę zagrożeń oraz nauczysz się efektywnie wykorzystywać mechanizm SSL (skrót od ang. Secure Sockets Layer). Po lekturze tego podręcznika będziesz w stanie poznać typowe błędy popełniane przez programistów. Zrozumiesz, w jaki sposób ktoś może wykonać atak na warstwę prezentacji oraz dane zgromadzone w przeglądarce. Ponadto nauczysz się testować aplikacje AJAX. Autorzy książki przedstawią również wady i zalety popularnych szkieletów aplikacji AJAX.

- Podstawy technologii AJAX
- Asynchroniczność
- Rodzaje ataków
- Wykorzystanie SSL
- Walidacja przyjmowanych danych
- Skomplikowanie kodu a bezpieczeństwo
- Typowe błędy programistów
- Przechwytywanie danych po stronie klienta
- Bezpieczeństwo w Google Gears oraz Dojo.Offline
- Zagrożenia związane z warstwą prezentacji
- Testowanie aplikacji AJAX
- Zalety i wady dostępnych szkieletów aplikacji AJAX

Zapewnij bezpieczeństwo Twojej aplikacji!

» Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

© Helion 1991-2008

Spis treści

Przedmowa 15

Przedmowa (ta rzeczywista) 17

Podziękowania 21

O autorach 23

Rozdział 1. Wprowadzenie do bezpieczeństwa technologii Ajax 25

Elementarz technologii Ajax 26
- Czym jest Ajax? 26
- Asynchroniczny 28
- JavaScript 30
- XML 34
- Dynamiczny HTML (DHTML) 35

Przechodzenie na architekturę Ajax 35
- Architektura grubego klienta 36
- Architektura cienkiego klienta 37
- Ajax: złotowłosa architektura 39
- Architektura grubego klienta z punktu widzenia bezpieczeństwa 40
- Architektura cienkiego klienta z punktu widzenia bezpieczeństwa 41
- Aplikacje Ajax z punktu widzenia bezpieczeństwa 42

Burza luk bezpieczeństwa 43
- Zwiększony poziom złożoności, przejrzystości i wielkości 43
- Kwestie socjologiczne 46
- Aplikacja Ajax: cel atrakcyjny i strategiczny 47

Podsumowanie 48

Rozdział 2. Włamanie ... 49

Ewa ... 49
Atak na witrynę HighTechVacations.net ... 50
  Złamanie systemu kuponowego ... 50
  Atak na dołączanie danych po stronie klienta ... 56
  Atak na API Ajax ... 61
Kradzież w jedną noc ... 66

Rozdział 3. Ataki sieciowe ... 69

Podstawowe kategorie ataków ... 69
  Wyliczanie zasobów ... 70
  Manipulacje parametrami ... 74
Inne rodzaje ataków ... 98
  Cross-Site Request Forgery (CSRF) ... 98
  Phishing ... 99
  Denial of Service (DoS) ... 100
Ochrona aplikacji sieciowej przed atakiem wyliczania zasobów lub manipulacji parametrami ... 101
  Secure Sockets Layer ... 102
Podsumowanie ... 102

Rozdział 4. Sposoby ataków na Ajax ... 105

Zrozumienie sposobów ataków na aplikacje Ajaksa ... 105
Sposoby ataków na tradycyjne aplikacje sieciowe ... 107
  Pola danych wejściowych formularzy sieciowych ... 107
  Mechanizm cookies ... 109
  Nagłówki ... 110
  Ukryte pola formularza sieciowego ... 110
  Parametry ciągu tekstowego zapytania ... 111
  Pliki przekazywane na serwer ... 114
Ataki na tradycyjne aplikacje sieciowe: ogólna ocena ... 115
Rodzaje ataków sieciowych ... 116
  Metody usługi sieciowej ... 117
  Definicje usługi sieciowej ... 118
Ataki na aplikacje Ajaksa ... 119
  Źródła ataków na aplikacje Ajaksa ... 120
  Najlepsze z połączenia obu światów — z punktu widzenia hakera ... 122
Prawidłowa weryfikacja danych wejściowych ... 123
  Problem z czarną listą oraz innymi określonymi poprawkami ... 124
  Leczenie objawów zamiast choroby ... 126
  Weryfikacja danych wejściowych na podstawie białej listy ... 129
  Wyrażenia regularne ... 133
  Dodatkowe przemyślenia dotyczące weryfikacji danych wejściowych ... 133

Weryfikacja rozbudowanych danych wejściowych użytkownika 135
Weryfikacja kodu znaczników 136
Weryfikacja plików binarnych 138
Weryfikacja kodu źródłowego JavaScript 138
Weryfikacja serializowanych danych 144
Mit dotyczący treści dostarczanej przez użytkownika 146
Podsumowanie 147

Rozdział 5. Złożoność kodu Ajaksa 149

Wiele języków i architektur 149
Indeksowanie tablicy 150
Operacje na ciągach tekstowych 152
Komentarze w kodzie 153
Problem innej osoby 154
Dziwactwa JavaScript 156
Interpretowany, a nie kompilowany 156
Słaba kontrola typów 157
Asynchroniczność 159
Problem tak zwanych wyścigów 159
Zakleszczenia i problem jedzących filozofów 163
Synchronizacja po stronie klienta 167
Zachowaj ostrożność podczas przyjmowania rad 168
Podsumowanie 169

Rozdział 6. Przejrzystość aplikacji Ajaksa 171

Czarne pudełka kontra białe pudełka 172
Przykład: witryna MyLocalWeatherForecast.com 174
Przykład: witryna MyLocalWeatherForecast.com wykonana z użyciem technologii Ajax 176
Podsumowanie porównania 179
Aplikacja sieciowa jako API 180
Rodzaje danych i sygnatury metod 181
Szczególne błędy dotyczące bezpieczeństwa 182
Nieprawidłowe uwierzytelnienie 182
Nadmierne rozdrobnienie API serwera 184
Przechowywanie stanu sesji w języku JavaScript 187
Ujawnianie użytkownikom poufnych danych 188
Komentarze i dokumentacja w kodzie działającym po stronie klienta 190
Transformacja danych wykonywana po stronie klienta 191
Bezpieczeństwo poprzez zaciemnianie 195
Techniki zaciemniania kodu 196
Podsumowanie 198

Rozdział 7. Przechwytywanie aplikacji Ajaksa 199

Przechwycenie struktury Ajaksa 200
  Przypadkowe nadpisanie funkcji 200
  Nadpisywanie funkcji dla rozrywki lub w celu osiągnięcia korzyści 202
Przechwytywanie technologii Ajax na żądanie 208
Przechwytywanie API JSON 213
  Przechwycenie obiektu 218
  Geneza przechwycenia danych JSON 218
  Obrona przed atakiem przechwycenia API 219
Podsumowanie 222

Rozdział 8. Ataki na magazyny danych po stronie klienta 223

Ogólny opis systemów magazynowania danych po stronie klienta 223
  Ogólne informacje dotyczące bezpieczeństwa magazynu danych po stronie klienta 225
Mechanizm HTTP cookies 226
  Reguły kontroli dostępu do cookie 229
  Pojemność cookie HTTP 234
  Czas życia cookie 237
  Dodatkowe informacje w zakresie bezpieczeństwa cookie 238
  Podsumowanie rozważań dotyczących używania cookie jako magazynu danych 239
Obiekty Flash Local Shared Objects 240
  Podsumowanie informacji o obiektach Local Shared Object 248
Magazyn danych DOM 249
  Magazyn danych sesji 250
  Globalny magazyn danych 252
  Diabelskie szczegóły dotyczące magazynu danych DOM 254
  Bezpieczeństwo magazynu danych DOM 256
  Podsumowanie informacji dotyczących magazynu danych DOM 257
Magazyn userData w przeglądarce Internet Explorer 258
  Podsumowanie dotyczące bezpieczeństwa 263
Ogólne informacje o atakach i obronie magazynów danych po stronie klienta 264
  Ataki typu cross-domain 264
  Ataki cross-directory 265
  Ataki cross-port 266
Podsumowanie 267

Rozdział 9. Ajaksowe aplikacje offline 269

Ajaksowe aplikacje offline 269
Google Gears 271
  Rodzime funkcje bezpieczeństwa i niedociągnięcia Google Gears 272
  Luki bezpieczeństwa w komponencie WorkerPool 276
  Ujawnienie danych LocalServer i skażenie danych 277
  Bezpośredni dostęp do bazy danych Google Gears 281
  Ataki typu SQL Injection i struktura Google Gears 282
  Jak niebezpieczny jest atak SQL Injection po stronie klienta? 287

Dojo.Offline ... 289
Bezpieczne przechowywanie klucza ... 290
Zapewnienie bezpieczeństwa danych ... 291
Dobre hasła dla dobrych kluczy ... 292
Weryfikacja danych wejściowych po stronie klienta stała się istotna ... 293
Inne podejścia do aplikacji offline ... 294
Podsumowanie ... 295

Rozdział 10. Kwestie związane z pochodzeniem żądań ... 297

Roboty, pająki, przeglądarki internetowe oraz inne koszmary ... 297
„Cześć! Nazywam się Firefox. Lubię programowanie, pliki PDF oraz długie spacery po plaży” ... 299
Niepewność dotycząca źródła pochodzenia żądania oraz JavaScript ... 300
Żądania Ajaksa z perspektywy serwera ... 300
Użytkownik lub ktoś podszywający się pod niego ... 304
Wysyłanie żądań HTTP za pomocą JavaScriptu ... 306
Ataki JavaScript z użyciem żądań HTTP w aplikacjach niestosujących technologii Ajax ... 308
Kradzież treści za pomocą obiektu XMLHttpRequest ... 310
Połączenie ataków XSS i XHR w praktyce ... 315
Sposoby obrony ... 317
Podsumowanie ... 319

Rozdział 11. Aplikacje mashup oraz agregatory ... 321

Dane z Internetu nadające się do użycia ... 322
Wczesne lata 90. ubiegłego wieku — początek współczesnego Internetu ... 322
Połowa lat 90. XX wieku — narodziny machiny sieciowej ... 323
XXI wiek — machina sieciowa stała się dojrzała ... 324
Publicznie dostępne usługi sieciowe ... 325
Aplikacja mashup: frankenstein Internetu ... 327
ChicagoCrime.org ... 328
HousingMaps.com ... 329
Inne aplikacje typu mashup ... 330
Budowanie aplikacji mashup ... 331
Aplikacje mashup i Ajax ... 332
Pomost, proxy, brama — och nie! ... 334
Alternatywy dla proxy Ajax ... 335
Ataki na proxy Ajax ... 336
Halo, HousingMaps.com? ... 338
Weryfikacja danych wejściowych w aplikacji mashup ... 341
Witryny agregujące ... 343
Zdegradowane bezpieczeństwo i zaufanie ... 351
Podsumowanie ... 354

Rozdział 12. Ataki na warstwę prezentacji ... 357

Szczypta prezentacji może zabić treść ... 357
Ataki na warstwę prezentacyjną ... 361
Analiza danych w kaskadowych arkuszach stylów ... 362
Sztuczki dotyczące wyglądu i działania ... 365
    Zaawansowane sztuczki dotyczące wyglądu i działania ... 369
Osadzona logika programu ... 372
Ataki na kaskadowe arkusze stylów ... 374
    Modyfikacja bufora przeglądarki internetowej ... 376
Uniemożliwianie ataków na warstwę prezentacyjną ... 379
Podsumowanie ... 381

Rozdział 13. Robaki JavaScript ... 383

Opis robaków JavaScript ... 383
    Tradycyjne wirusy komputerowe ... 384
    Robaki JavaScript ... 387
Konstrukcja robaka JavaScript ... 389
    Ograniczenia języka JavaScript ... 391
    Rozprzestrzenianie się robaków JavaScript ... 392
    Operacje wykonywane przez robaki JavaScript ... 392
    Podsumowanie części teoretycznej ... 401
Studium przypadku: robak Samy ... 401
    Jak działał robak Samy? ... 402
    Działalność robaka Samy ... 405
    Podsumowanie analizy robaka Samy ... 407
Studium przypadku: robak Yamanner (JS/Yamanner-A) ... 408
    W jaki sposób działał robak Yamanner? ... 409
    Działalność robaka Yamanner ... 412
    Podsumowanie analizy robaka Yamanner ... 413
Lekcje wyciągnięte po analizie rzeczywistych przykładów robaków JavaScript ... 416
Podsumowanie ... 417

Rozdział 14. Testowanie aplikacji Ajaksa ... 419

Czarna magia ... 420
Nie każdy do przeglądania Internetu używa przeglądarki internetowej ... 424
    Paragraf 22 ... 427
Narzędzia służące do sprawdzania bezpieczeństwa, czyli dlaczego prawdziwe życie nie przypomina Hollywood? ... 428
    Katalogowanie witryny ... 429
    Wykrywanie luk bezpieczeństwa ... 431
    Narzędzie analizy: Sprajax ... 433
    Narzędzie analizy: Paros Proxy ... 435
    Narzędzie analizy: LAPSE (Lightweight Analysis for Program Security in Eclipse) ... 437
    Narzędzie analizy: WebInspect™ ... 439
Dodatkowe przemyślenia dotyczące testów pod kątem bezpieczeństwa ... 440

Rozdział 15. Analiza struktur Ajaksa 443

ASP.NET 443
ASP.NET AJAX (dawniej Atlas) 444
ScriptService 448
Ostateczna rozgrywka bezpieczeństwa: UpdatePanel kontra ScriptService 449
ASP.NET AJAX i WSDL 450
ValidateRequest 454
ViewStateUserKey 455
Konfiguracja i usuwanie błędów w ASP.NET 456
PHP 457
Sajax 458
Struktura Sajax i ataki Cross-Site Request Forgery 459
Java EE 461
Direct Web Remoting (DWR) 461
Struktury JavaScript 464
Ostrzeżenie dotyczące kodu działającego po stronie klienta 464
Prototype 465
Podsumowanie 467

Dodatek A Kod źródłowy robaka Samy 469

Dodatek B Kod źródłowy robaka Yamanner 477

Skorowidz 483

2 Włamanie

Mit: *Hakerzy rzadko atakują przedsiębiorstwa, wykorzystując do tego celu ich aplikacje wykonane w Ajaksie.*

Poznaj bezpośredniego świadka obalającego ten mit, czyli Ewę.

Ewa

Nie poznasz jej nawet wtedy, gdy pojawi się ponownie. To nie tak, że dwudziestokilkuletnia kobieta siedząca w rogu lokalu pozostaje niewidzialną osobą — przecież fizycznie się tam znajduje. Jednak ona celowo była ubrana zupełnie przeciętnie, nie wypowiedziała więcej niż dziesięć słów do wszystkich i nie zrobiła niczego, co przyciągnęłoby uwagę innych osób. Oprócz tego, lokal Caribou Coffee znajduje się na rogu dziesiątej ulicy i Piedmont, czyli w samym modnym centrum Atlanty. Można więc znaleźć tutaj znacznie więcej interesujących rzeczy do podziwiania niż obserwacja noszącej okulary kobiety, która używa laptopa ThinkPad i siedzi przy stole stojącym w rogu lokalu.

Po przyjściu do lokalu zamówiła kawę i rogala, a godzinę później kolejną kawę. Oczywiście, zapłaciła gotówką, nie pozostawiając żadnego elektronicznego śladu, że znajdowała się w danym miejscu o określonej godzinie. Dokonany przez nią zakup był wystarczający na tyle, aby upewnić sprzedawcę, że kobieta nie jest pasożytem, który chciał jedynie wyłudzić bezpłatny dostęp do bezprzewodowego Internetu WiFi. Sygnał bezprzewodowy przenika przez ściany budynku, więc równie dobrze mogłaby skorzystać z Internetu, pozostając w samochodzie zaparkowanym w pobliżu lokalu. Jednak widok kobiety w volkswagenie jetta zaparkowanym na oblężonym parkingu i trzymającej laptop w rękach mógłby wydawać się podejrzany — znacznie lepszym rozwiązaniem było wejście do lokalu i rozpłynięcie się w tłumie. Ucieszyła się, gdy na środku lokalu zobaczyła siedzącego dzieciaka o blond włosach w czarnej podkoszulce. Chłopak pisał coś na przeciętnym laptopie marki Dell,

na obudowie którego znajdowały się naklejki „FreeBSD 4 Life”, „2600” oraz „Free Kevin!”. W głębi duszy była bardzo zadowolona, script kiddies zawsze wyglądają kiepsko, podobnie jak ich tani sprzęt komputerowy. Nawet przy założeniu, że jej obecna działalność pozostawi ślady prowadzące do tego lokalu (w co szczerze wątpiła), za hakera zostanie uznany osobnik w koszulce zespołu Metallica, czyli jedyna osoba, którą inni będą pamiętali.

Nikt nie będzie nawet podejrzewał Ewy. I to bardzo jej odpowiadało.

Atak na witrynę HighTechVacations.net

Tego dnia jej celem była witryna internetowa HighTechVacations.net[1] należąca do biura podróży. O witrynie dowiedziała się z artykułu zamieszczonego w Ajaxianie, popularnej witrynie zawierającej informacje na temat technologii Ajax. Ewa lubi aplikacje sieciowe. Cały Internet jest dla niej terenem, na którym może polować. Kiedy skreśli daną witrynę jako cel ataku, po prostu używa wyszukiwarki Google w celu znalezienia tysięcy kolejnych. Ewa szczególnie lubi aplikacje Ajaksa. Istnieje cała masa implikacji bezpieczeństwa powiązanych z procesem tworzenia aplikacji żywo reagujących na działania użytkownika, które mają bogate funkcje działające po stronie klienta. A ponieważ technologia jest na tyle nowa, posługujący się nią programiści nadal popełniają podstawowe błędy i wydaje się, że nikt nie jest w stanie zaproponować dobrych rozwiązań w zakresie bezpieczeństwa tego rodzaju aplikacji. Co najlepsze, cała rzesza nowych, podekscytowanych programistów sieciowych codziennie wydaje mnóstwo aplikacji w Ajaksie, wypełniając tym samym przestrzeń niebezpiecznymi programami. Ewa jest zachwycona, uwielbia środowisko pełne potencjalnych celów ataku!

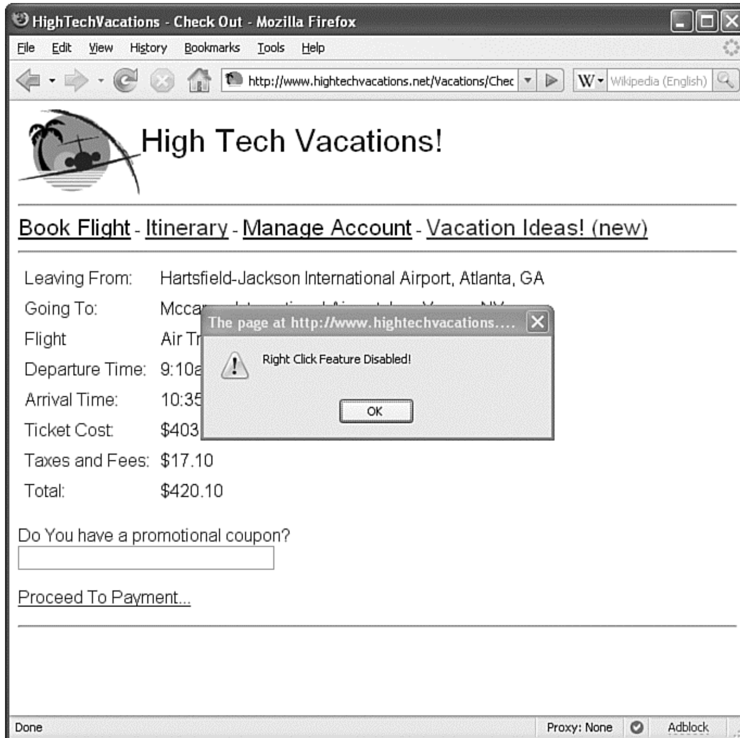
Do witryny HighTechVacations.net Ewa podeszła w taki sam sposób jak do innych celów. Przede wszystkim upewniła się, że jej cały ruch sieciowy odbywał się poprzez proxy HTTP na jej komputerze lokalnym i rozpoczęła przeglądanie witryny. Następnie utworzyła konto użytkownika, użyła funkcji wyszukiwania, wprowadziła dane w formularzu przeznaczonym do wysłania i rozpoczęła proces rezerwacji biletów na lot z Atlanty do Las Vegas. Zwróciła uwagę, że na witrynie został zastosowany protokół SSL. Przeanalizowała więc certyfikat SSL i uśmiechnęła się — został utworzony samodzielnie przez autorów witryny. Takie podejście to nie tylko duży błąd w trakcie wdrażania bezpiecznej witryny internetowej, ale również sygnał zaangażowania niechlujnych administratorów lub działu IT wynikający prawdopodobnie z pogoni za pieniądzem, a dla Ewy to bardzo dobry znak.

Złamanie systemu kuponowego

Ewa kontynuowała korzystanie z witryny, a po dotarciu do fazy finalizacji zamówienia zauważyła coś interesującego — pole *Coupon Code* w formularzu zamówienia. Wpisała więc słowo *FREE* i nacisnęła klawisz *Tab*, przechodząc tym samym do kolejnego pola formularza.

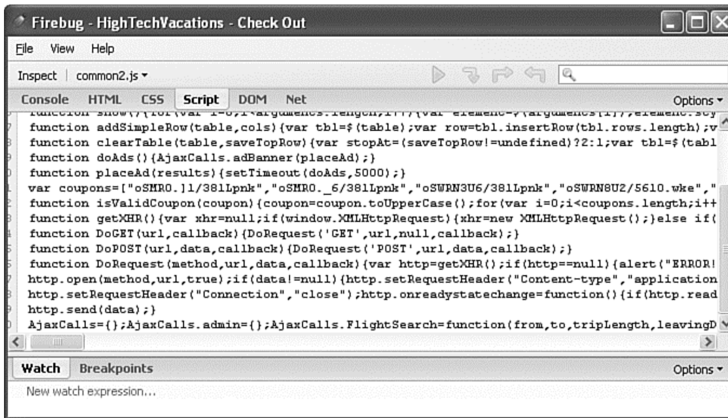
[1] Witryna testowa autorów książki — *przyp. red.*

Przeglądarka internetowa natychmiast wyświetliła komunikat błędu informujący Ewę, że podany kod kuponu jest nieprawidłowy. To dziwne. W jaki sposób witryna internetowa tak szybko określiła, że podany kod kuponu jest nieprawidłowy? Prawdopodobnie w celu wysłania żądania do serwera została użyta technologia Ajax. Ewa zdecydowała się na przejrzenie kodu źródłowego, aby zorientować się, co naprawdę zachodzi na stronie. Po kliknięciu w oknie prawym przyciskiem myszy w celu wyświetlenia kodu źródłowego na ekranie pojawił się komunikat pokazany na rysunku 2.1.



Rysunek 2.1. Strona finalizacji zamówienia na witrynie HighTechVacations.net uniemożliwia klikanie prawym przyciskiem myszy

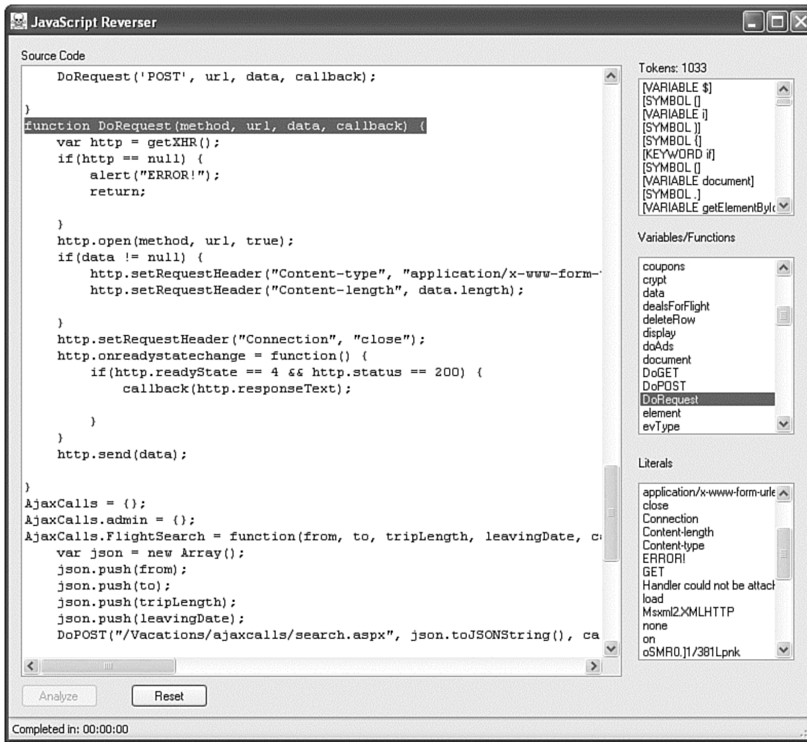
Ewa jest oszołomiona. Czy autorzy witryny HighTechVacations.net naprawdę uważają, że nie będzie mogła wyświetlić kodu źródłowego HTML? To naprawdę niedorzeczne! Przecież to jej przeglądarka internetowa wygenerowała kod HTML, tak więc nie może zostać ukryty. Odrobina kodu JavaScript przechwytującego kliknięcie prawym przyciskiem myszy i blokującego menu kontekstowe nie zatrzyma Ewy. Jej kolejnym krokiem jest uruchomienie rozszerzenia przeglądarki Firefox o nazwie Firebug. Jak pokazano na rysunku 2.2, to podręczne narzędzie wyświetla cały kod JavaScript dołączony do bieżącej strony.



Rysunek 2.2. Firebug, czyli między innymi debuger JavaScript, wyświetla zaciemniony kod na witrynie HighTechVacations.net

W tym miejscu wystąpił problem — kod JavaScript został zaciemniony. Oznacza to, że usunięto z niego wszystkie spacje, a niektóre nazwy zmiennych i funkcji zostały celowo skrócone, tak aby maksymalnie utrudnić człowiekowi zrozumienie zasady działania kodu. Ewa wie, że ten kod JavaScript, choć trudny do odczytania przez człowieka, pozostaje doskonale czytelny dla interpretera JavaScript w jej przeglądarce internetowej. Uruchomiła więc własne narzędzie o nazwie JavaScript Reverser. Wymieniony program pobiera kod JavaScript (zaciemniony lub nie), a następnie przetwarza go w taki sam sposób jak interpreter JavaScript wbudowany w przeglądarkę internetową. Narzędzie wyświetla nazwy wszystkich zmiennych i funkcji, miejsce ich występowania i częstotliwość używania, a także informacje, które funkcje są wywoływane przez inne funkcje oraz argumenty stosowane w tych wywołaniach. Dodatkowo narzędzie JavaScript Reverser umieszcza w kodzie spacje, przez co staje się łatwiejszy do odczytania przez człowieka. Ewa jest bardzo zaciekawiona tym, co zobaczy w kodzie JavaScript, ponieważ programista podjął wiele kroków, aby uniemożliwić innym jego poznanie. Na rysunku 2.3 pokazano kod wygenerowany przez narzędzie Ewy o nazwie JavaScript Reverser.

Ewa szybko odnajduje funkcję o nazwie `addEvent()`, która dołącza nasłuch zdarzeń JavaScript w sposób niezależny od używanej przeglądarki internetowej. Po przejrzeniu wszystkich wystąpień wywołań `addEvent()` Ewa zauważyła, że funkcja jest używana do dołączenia funkcji `checkCoupon()` zdarzenia `onblur` pola tekstowego, w którym użytkownik podaje kod kuponu. To zatem jest funkcja, która została wywołana, gdy Ewa nacisnęła klawisz *Tab*, opuszczając tym samym pole tekstowe służące do podania kodu kuponu. Funkcja ta określiła również, że ciąg tekstowy *FREE* nie jest prawidłowym kodem kuponu. Zadaniem funkcji `checkCoupon()` jest po prostu wyodrębnienie kodu podanego przez użytkownika w polu tekstowym i wywołanie kolejnej funkcji o nazwie `isValidCoupon()`. Poniżej przedstawiono niezaciemniony fragment kodu obejmujący między innymi funkcję `isValid` `↳Coupon()`:



RYSUNEK 2.3. Narzędzie JavaScript Reverser analizuje kod JavaScript znajdujący się na witrynie HighTechVacations.net i pomaga w zrozumieniu jego działania

```
var coupons = ["oSMR0.]1/381Lpnk",
"oSMR0._6/381LPNK",
"oSWRN3U6/381LPNK",
"oSWRN8U2/561O.WKE",
"oSWRN2[.0:8/O15TEG",
"oSWRN3Y.1:8/O15TEG",
"oSWRN4_.258/O15TEG",
"tQOWC2U2RY5DkB[X",
"tQOWC3U2RY5DkB[X",
"tQOWC3UCTX5DkB[X",
"tQOWC4UCTX5DkB[X",
"uJX6,GzFD",
"uJX7,GzFD",
"uJX8,GzFD"];

function crypt(s) {
   var ret = '';
   for(var i = 0; i < s.length; i++) {
      var x = 1;
      if( (i % 2) == 0) {
         x += 7;
```

```
      }
      if( (i % 3) ==0) {
         x *= 5;
      }
      if( (i % 4) == 0) {
         x -= 9;
      }
      ret += String.fromCharCode(s.charCodeAt(i) + x);
   }
   return ret;
}

function isValidCoupon(coupon) {
   coupon = coupon.toUpperCase();
   for(var i = 0; i < coupons.length; i++) {
      if(crypt(coupon) == coupons[i])
         return true;
   }
   return false;
}
```

Kod kuponu wprowadzony w formularzu przez Ewę został przekazany funkcji `isValid↪Coupon()`, w której wszystkie jego znaki zostały zmienione na duże, zaszyfrowane, a następnie porównane z listą zaszyfrowanych wartości. Ewa spojrzała na funkcję `crypt()` i roześmiała się. Funkcja szyfrująca to po prostu kilka podstawowych operacji matematycznych, które używają pozycji znaku w ciągu tekstowym w celu obliczenia pewnej wartości. Wartość ta zostaje następnie dodana do kodu ASCII znaku, a wynikiem jest kod ASCII zaszyfrowanego znaku. Ten algorytm „szyfrowania" jest klasycznym przykładem **trywialnego algorytmu szyfrowania**, czyli takiego, którego działanie można bardzo łatwo odwrócić i złamać (na przykład tak zwana „świńska łacina" jest trywialnym algorytmem szyfrowania w języku angielskim). Proces szyfrowania i deszyfrowania kodu kuponu jest bardzo prosty i sprowadza się do odjęcia wartości kodu ASCII znaku od wartości zaszyfrowanego znaku. W swoim komputerze Ewa szybko skopiowała do nowego pliku HTML utworzoną tablicę kodów kuponów oraz funkcję `crypt()`, a następnie zmodyfikowała ją, tworząc nową funkcję `decrypt()`. Dokument HTML opracowany przez Ewę przedstawia się następująco:

```
<html>
<script>

var coupons = ["oSMRO.]1/381Lpnk",
"oSMRO._6/381LPNK",
"oSWRN3U6/381LPNK",
"oSWRN8U2/561O.WKE",
"oSWRN2[.0:8/O15TEG",
"oSWRN3Y.1:8/O15TEG",
"oSWRN4_.258/O15TEG",
"tQOWC2U2RY5DkB[X",
"tQOWC3U2RY5DkB[X",
```

```
"tQOWC3UCTX5DkB[X",
"tQOWC4UCTX5DkB[X",
"uJX6,GzFD",
"uJX7,GzFD",
"uJX8,GzFD"];

function decrypt(s) {
   var ret = '';
   for(var i = 0; i < s.length; i++) {
      var x = 1;
      if( (i % 2) == 0) {
         x+=7;
      }
      if( (i%3) ==0) {
         x *=5;
      }
      if( (i%4) == 0) {
         x -=9;
      }
      ret += String.fromCharCode(s.charCodeAt(i) - x);
   }
   return ret;
}

for(var i = 0; i < coupons.length; i++) {
   alert("Coupon " + i + " is " + decrypt(coupons[i]));
}

</script>
</html>
```

Po wyświetleniu w przeglądarce internetowej powyższego dokumentu HTML w serii wyskakujących okien zostały podane wszystkie prawidłowe kody kuponów dostępne podczas rezerwacji lotów na witrynie HighTechVacations.net. Pełna lista kodów jest następująca:

- PREM1—500.00—OFF
- PREM1—750.00—OFF
- PROMO2—50.00—OFF
- PROMO7—100.00—OFF
- PROMO13—150.00—OFF
- PROMO14—200.00—OFF
- PROMO21—250.00—OFF
- PROMO37—300.00—OFF
- UPGRD1—1ST—CLASS

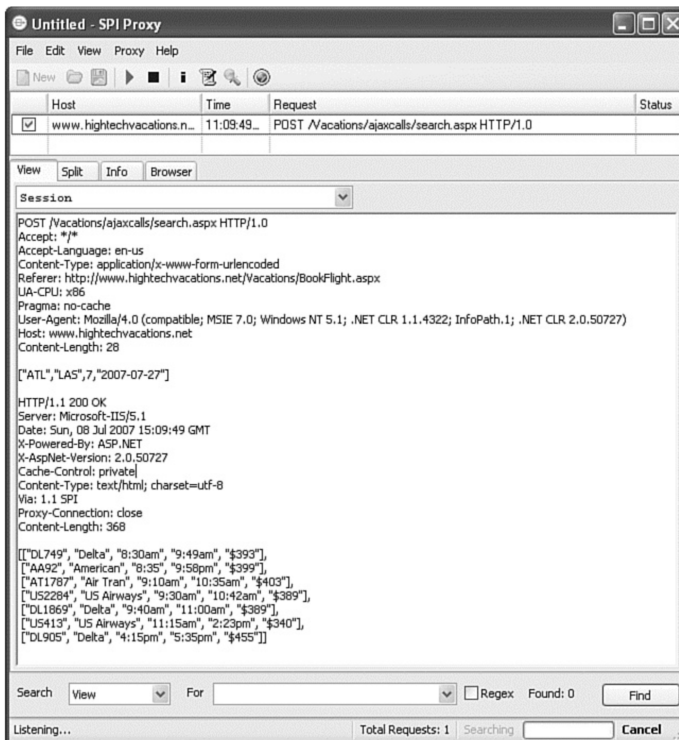
- UPGRD2—1ST—CLASS
- UPGRD2—BUS—CLASS
- UPGRD3—BUS—CLASS
- VIP1—FREE
- VIP2—FREE
- VIP3—FREE

Ewa zanotowała wszystkie powyższe kody. Zebrane informacje może wykorzystać na własny użytek bądź sprzedać je w Internecie. Teraz Ewa doskonale wie, że w tym roku nie będzie musiała płacić za podróż do Las Vegas.

Atak na dołączanie danych po stronie klienta

Ewa, wciąż żądna jeszcze większej ilości cennych danych, zdecydowała się na przeanalizowanie funkcji wyszukiwania zaimplementowanej na witrynie HighTechVacations.net. Przeprowadziła więc kolejną operację wyszukania lotu z Atlanty do Las Vegas. Zwróciła również uwagę na fakt, że strona z wynikami wyszukiwania nie zostaje odświeżona lub przekierowana na inny adres URL. Stało się oczywiste, że funkcja wyszukiwania używa technologii Ajax w celu komunikacji z usługą sieciową i wykonania pewnego rodzaju dynamicznego pobrania wyników wyszukiwania. Ewa dwukrotnie sprawdziła i upewniła się, że cały jej ruch internetowy nadal odbywa się poprzez proxy HTTP, który jest uruchomiony na jej komputerze. W ten sposób może podglądać żądania i odpowiedzi Ajaksa. Ewa zapisała więc kopię całego przechwyconego dotychczas ruchu poprzez proxy HTTP i ponownie uruchomiła serwer. Następnie powróciła do przeglądarki internetowej i wykonała wyszukiwanie lotów z międzynarodowego lotniska Hartsfield-Jackson w Atlancie do międzynarodowego lotniska McCarran w Las Vegas na 27 lipca. Po chwili opóźnienia otrzymała listę lotów. Wróciła więc z powrotem do serwera proxy i rozpoczęła analizę żądań i odpowiedzi Ajaksa, co pokazano na rysunku 2.4.

Ewa zauważyła, że w witrynie HighTechVacations.net zastosowano format JSON (JavaScript Object Notation) w charakterze warstwy prezentacji danych, co jest rozwiązaniem dość często spotykanym w aplikacjach ajaksowych. Szybkie przeszukanie Internetu za pomocą Google pomogło określić, że *ATL* i *LAS* to kody lotnisk w Atlancie oraz Las Vegas. Pozostała część tablicy JSON jest łatwa do rozszyfrowania: *2007-07-27* określa datę, a cyfra 7 wskazuje liczbę dni, które Ewa zamierzała spędzić w Las Vegas. Na tym etapie Ewa rozumiała już format żądania wysłanego do usługi sieciowej wykonującej wyszukiwanie lotów. Ponadto Ewa wiedziała, że port wylotu, port docelowy oraz numer lotu są prawdopodobnie przekazywane do bazy danych, w której jest wykonywane pewnego rodzaju wyszukiwanie w celu dopasowania lotu. Postanowiła przeprowadzić prostą próbę i sprawdzić, czy baza danych jest podatna na atak typu SQL Injection. Dlatego też skonfigurowała proxy z użyciem reguł typu „znajdź i zastąp”. Kiedy proxy wychwyci wartości *ATL*, *LAS*

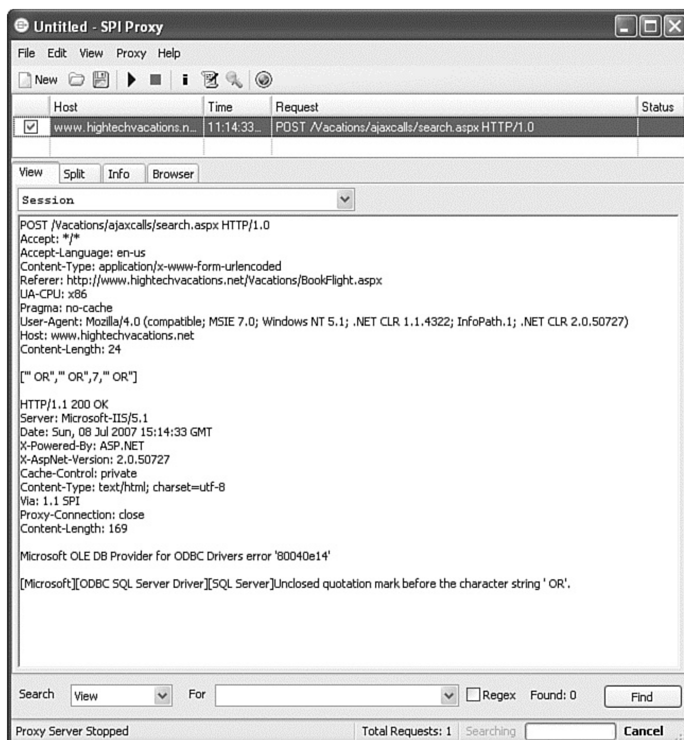


Rysunek 2.4. Żądania i odpowiedzi Ajaksa występujące podczas przeprowadzonej przez Ewę operacji wyszukiwania lotów

lub *2007-07-27* w wychodzących żądaniach HTTP, będzie je przed wysłaniem żądania do witryny HighTechVacations.net zastępowało wartościami `' OR`. Ciąg tekstowy `' OR` w każdej wartości może wygenerować błąd składni w zapytaniu bazy danych i wyświetlić komunikat błędu bazy danych. Szczegółowy komunikat błędu pochodzący z bazy danych jest najlepszym przyjacielem Ewy!

Ewa powraca do przeglądarki internetowej i ponownie inicjuje wyszukiwanie lotów z Atlanty do Las Vegas. Czeka… i czeka… i nic się nie dzieje. To dziwne. Ewa sprawdza więc proxy i otrzymuje wynik pokazany na rysunku 2.5.

Wysłane przez Ewę żądanie z próbą ataku SQL Injection zostało przekazane do serwera, który z kolei odpowiedział eleganckim i szczegółowym komunikatem błędu. Funkcja JavaScript wywołania zwrotnego obsługująca odpowiedź Ajaksa z informacjami o dostępnych lotach najwyraźniej zatrzymała dane przekazane przez serwer. Źle się stało, że czytelny komunikat błędu bazy danych został wysłany przez Internet, tak aby Ewa mogła się z nim zapoznać! Powyższy komunikat błędu ujawnia, że serwer bazy danych to Microsoft SQL Server. Ewa wie, że ma do czynienia z klasycznym przykładem ataku SQL Injection, ale podejrzewa również, iż po stronie klienta są przetwarzane pewne dane.

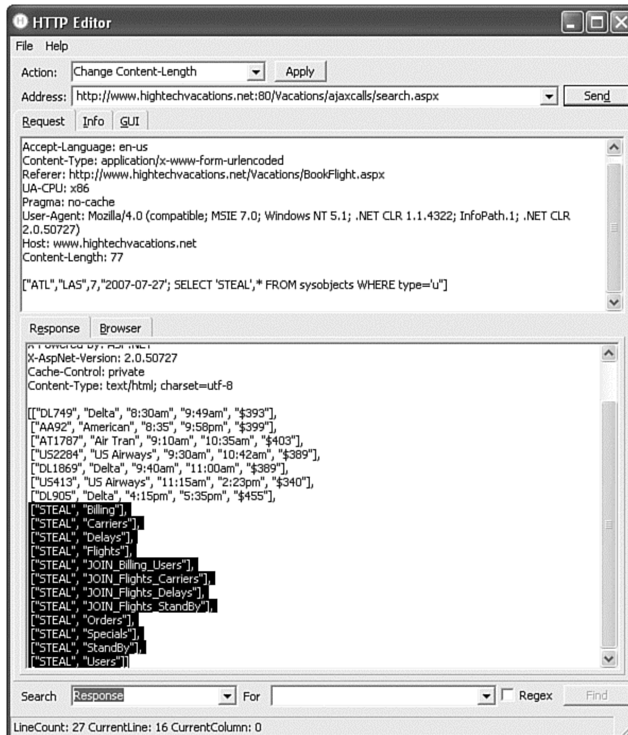


RYSUNEK 2.5. Próba przeprowadzona przez Ewę spowodowała wygenerowanie błędu ODBC. Kod JavaScript działający po stronie klienta przechwycił ten komunikat błędu, tak aby nie został wyświetlony w oknie przeglądarki internetowej

Serwer WWW witryny HighTechVacations.net pobiera dane lotów zwrócone przez zapytanie do bazy danych, a następnie bezpośrednio odsyła je do klienta, który z kolei formatuje dane i wyświetla użytkownikowi. Kiedy dane są przetwarzane po stronie serwera, wyniki zapytania do bazy danych są zbierane i formatowane po stronie serwera, a nie u klienta. Oznacza to, że dodatkowe dane lub dane sformatowane nieprawidłowo zwrócone z bazy danych są odrzucane przez serwer podczas dołączania ich do warstwy prezentacyjnej. Zatem Ewa ich nie pozna. Natomiast podczas przetwarzania danych po stronie klienta, które zwykle jest wykorzystywane w aplikacjach Ajaksa, atakujący — taki jak Ewa — może umieścić w zapytaniu wrogi kod SQL. Następnie napastnik może przechwycić pełne wyniki operacji w bazie danych, które są wysyłane do skryptu JavaScript formatującego dane dla użytkownika końcowego.

Ewa uruchamia kolejne narzędzie, którym jest edytor HTTP. Narzędzie to pozwala jej na utworzenie czystego żądania HTTP do serwera (zamiast polegania na regułach „znajdź i zastąp" zdefiniowanych w proxy) oraz wstawienie wrogich danych. Po chwili prób i błędów Ewie udaje się odkryć, że może wstawić wrogie polecenie SQL na początku parametru

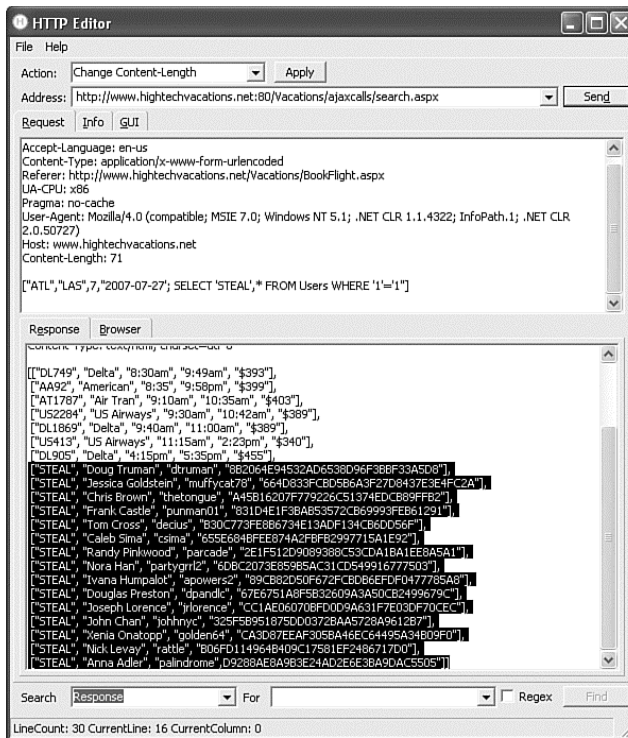
danych wewnątrz formatu JSON jej żądania. Ponieważ atakowany serwer to Microsoft SQL Server, zapytanie jest kierowane do tabeli `SYSOBJECTS`, jak pokazano na rysunku 2.6. Celem zapytania jest otrzymanie pełnej listy wszystkich tabel zdefiniowanych przez programistów w bazie danych witryny HighTechVacations.net.



RYSUNEK 2.6. Za pomocą tylko jednego polecenia Ewa otrzymuje listę wszystkich tabel zdefiniowanych przez programistów w bazie danych witryny HighTechVacations.net

W bazie danych istnieje wiele interesujących Ewę tabel, między innymi `Specials`, `Orders`, `Billing` oraz `Users`. Ewa postanawia wyświetlić wszystkie dane z tabeli `Users`, co pokazano na rysunku 2.7.

Cudownie! Za pomocą pojedynczego zapytania Ewie udało się pobrać informacje o wszystkich użytkownikach. Witryna HighTechVacations.net jest więc podatna na atak typu SQL Injection. Fakt, że autorzy użyli przetwarzania danych po stronie klienta zamiast po stronie serwera, oznacza, iż Ewa może ukraść całą bazę danych, korzystając z kilku zapytań. Nie musi poświęcać dużej ilości czasu na używanie zautomatyzowanego narzędzia ataków SQL Injection, takiego jak Absinthe.



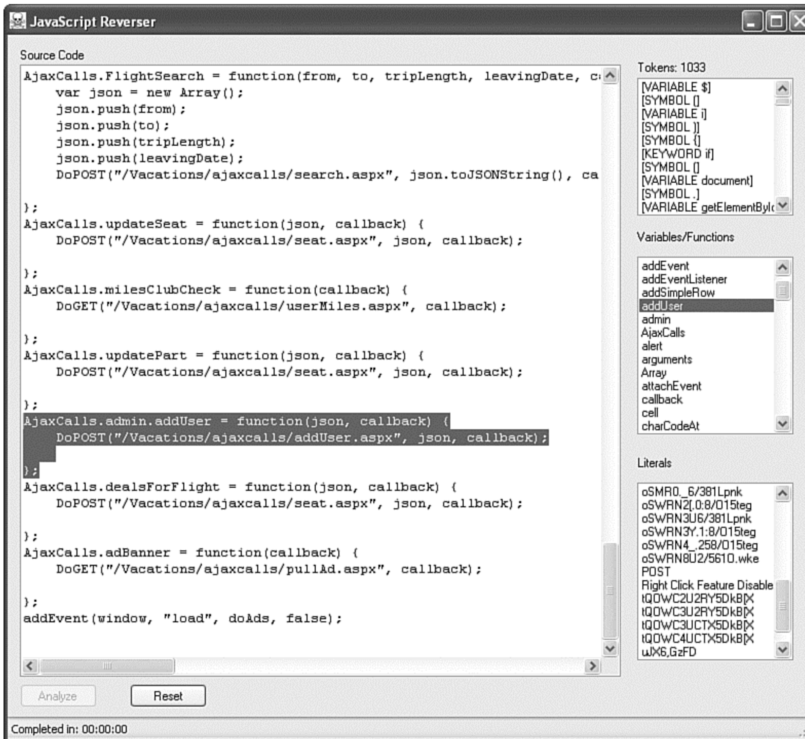
RYSUNEK 2.7. Przy użyciu pojedynczego zapytania Ewa otrzymała wszystkie kolumny każdego rekordu tabeli Users

Ewa jest zadowolona, otrzymała bowiem listę nazw użytkowników oraz ich haseł dostępu. Warto wspomnieć, że użytkownicy bardzo często używają tych samych nazw i haseł na wielu różnych witrynach internetowych. Dane uzyskane w wyniku tego ataku Ewa może wykorzystać podczas kolejnych. Udany atak na witrynę HighTechVacations.net spowodował, że Ewa uzyskała możliwość włamania się na nieokreśloną ilość zupełnie innych witryn internetowych, niepowiązanych z tą, której zabezpieczenia właśnie przełamała. Kto wie, być może jeszcze przed świtem Ewa będzie mogła uzyskać dostęp do czyjegoś konta bankowego, pożyczki studenckiej, kredytu hipotecznego lub nawet pracowniczego funduszu emerytalnego. Wyodrębnienie z wyników zapytania nazw użytkowników i zaszyfrowanych haseł zabiera chwilę. Wprawdzie Ewa nie jest pewna, czy hasła są zaszyfrowane, ale każde z nich jest liczbą szesnastkową o długości dokładnie trzydziestu dwóch znaków. Prawdopodobnie jest to wartość MD5 wygenerowana na podstawie rzeczywistego hasła. Kolejnym krokiem Ewy jest uruchomienie narzędzia John the Ripper służącego do łamania haseł i rozpoczęcie łamania listy haseł potrzebnych do rzucenia się na tabele `Billing` oraz `JOIN_Billing_Users`. Wymienione tabele zawierają informacje finansowe użytkowników, takie jak numery kart kredytowych, daty ich wygaśnięcia oraz adresy wszystkich użytkowników witryny HighTechVacations.net.

Atak na API Ajax

Ewa postanowiła przyjrzeć się bliżej stronom, które dotąd odwiedziła. Po sprawdzeniu zauważyła, że każda strona zawiera odwołanie do skryptu o nazwie *common.js*. Jednak nie w każdej stronie internetowej użyto wszystkich funkcji zdefiniowanych w wymienionym skrypcie. Przykładowo skrypt *common.js* zawiera definicję funkcji `isCouponValid()`, mimo że jest używana jedynie na stronie finalizującej zamówienie. Ewa uznała więc za możliwe, iż inne funkcje znajdujące się w skrypcie *common.js* są używane przez strony internetowe, których dotąd nie odwiedziła.

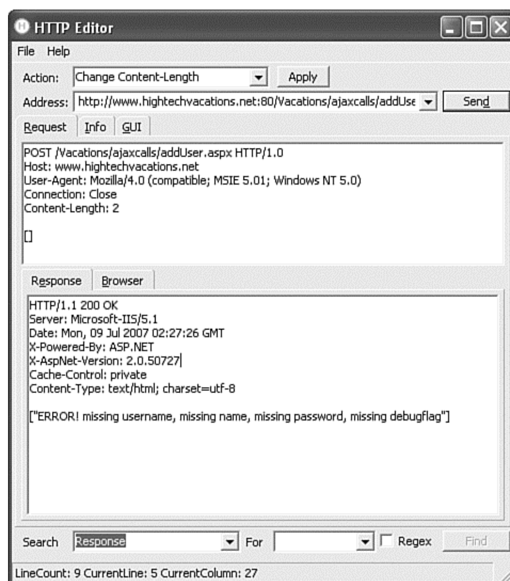
Przecież mogą to być funkcje administracyjne, których użytkownicy witryny nie powinni poznać! Ewa przejrzała więc dokładnie listę zmiennych i funkcji wyświetloną przez narzędzie JavaScript Reverser i niemal przeoczyła tę właściwą. W środku listy nudnych funkcji Ajaksa znajdowało się coś ciekawego — funkcja o nazwie `AjaxCalls.admin.addUser()` (zobacz środkową część rysunku 2.8).



Rysunek 2.8. Odniesienie w pliku common.js prowadzi do nieużywanej funkcji administracyjnej o nazwie AjaxCalls.admin.addUser()

Sama nazwa funkcji nie przekazywała zbyt wielu informacji. Stanowiła opakowanie, które wywołuje usługę sieciową wykonującą pozostałą część zadania. Jednak nazwa funkcji wskazuje, że należy ona do zbioru funkcji administracyjnych. Ewa szybko przeanalizowała wszystkie odpowiedzi przechwycone przez proxy HTTP. W danych nie znalazła żadnego odniesienia do funkcji `addUser()` na którejkolwiek z dotąd odwiedzonych stron. Była już bardzo zaintrygowana. Dlaczego definicja tej funkcji znajduje się w skrypcie *common.js*? Czy to zwykła pomyłka?

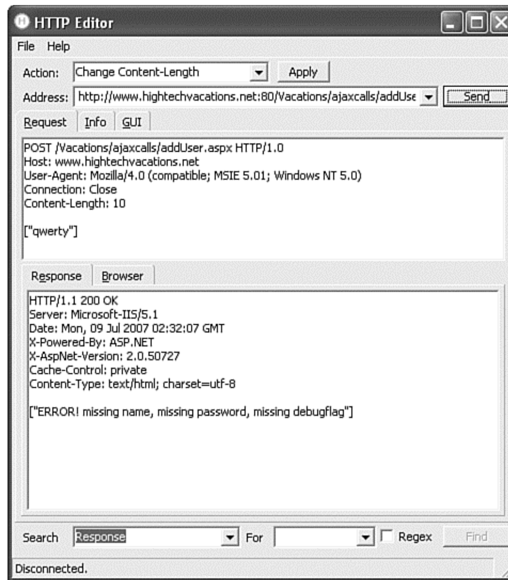
Ponownie uruchomiła edytor żądań HTTP. Wprawdzie znała adres URL usługi sieciowej, z którą kontaktuje się funkcja `addUser()`, i wiedziała, że podczas wysyłania żądań trzeba zastosować metodę `POST`, ale na tym koniec. Wszystkie pozostałe usługi używają formatu JSON, tak więc Ewa wysłała żądanie `POST` do strony */ajaxcalls/addUser.aspx* wraz z pustą tablicą JSON, jak pokazano na rysunku 2.9.



RYSUNEK 2.9. Na nieprawidłowo sformatowane żądanie usługa sieciowa addUser.aspx odpowiada komunikatem błędu

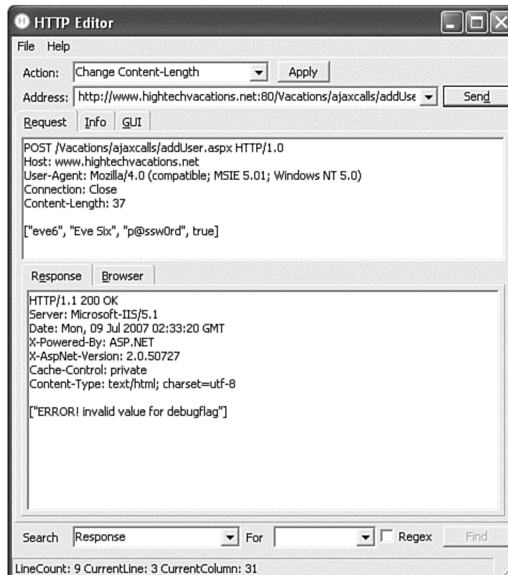
Interesujące. Witryna internetowa odpowiedziała komunikatem błędu, informującym Ewę, że w jej żądaniu zabrakło niektórych parametrów. Ewa podała więc zmyślony parametr i ponownie wysłała żądanie. Wynik drugiej próby został pokazany na rysunku 2.10.

Ewa usiadła na krawędzi krzesła. Jej strzał na oślep w rzeczywistości spowodował pewną reakcję. Wprawdzie usługa sieciowa nie utworzyła nowego użytkownika, ale poinformowała o braku trzech parametrów zamiast — jak dotąd — czterech. Ewa zaczęła się zastanawiać. Doskonale wiedziała, że do usługi sieciowej musi przekazać cztery parametry w formacie JSON. Mogła jedynie zgadywać, jakiego rodzaju dane są potrzebne: prawdopodobnie nazwa konta użytkownika, prawdziwe imię i nazwisko, hasło oraz pewien



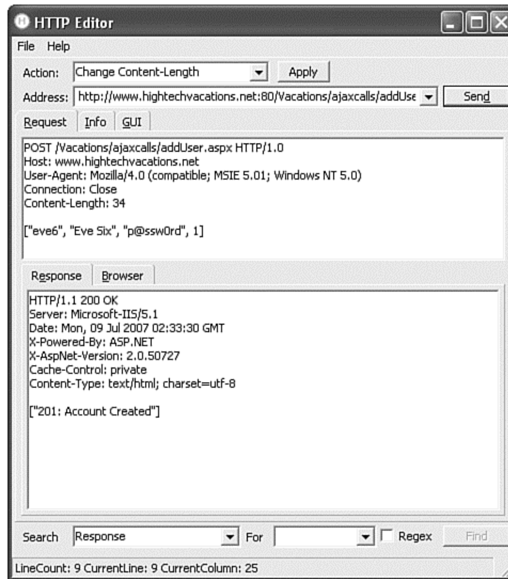
Rysunek 2.10. Wymyślony przez Ewę parametr spowodował wygenerowanie przez usługę sieciową nieco innego komunikatu błędu

rodzaj opcji. Ewa wiedziała, że opcje to najczęściej wartości typu Boolean, ale nie była pewna, jaki format powinien zostać zastosowany. Szybko utworzyła żądanie z prawdopodobnymi wartościami i wysłała je, co pokazano na rysunku 2.11.



Rysunek 2.11. Usługa sieciowa odrzuciła żądanie utworzone przez Ewę z powodu nieprawidłowej wartości debugflag

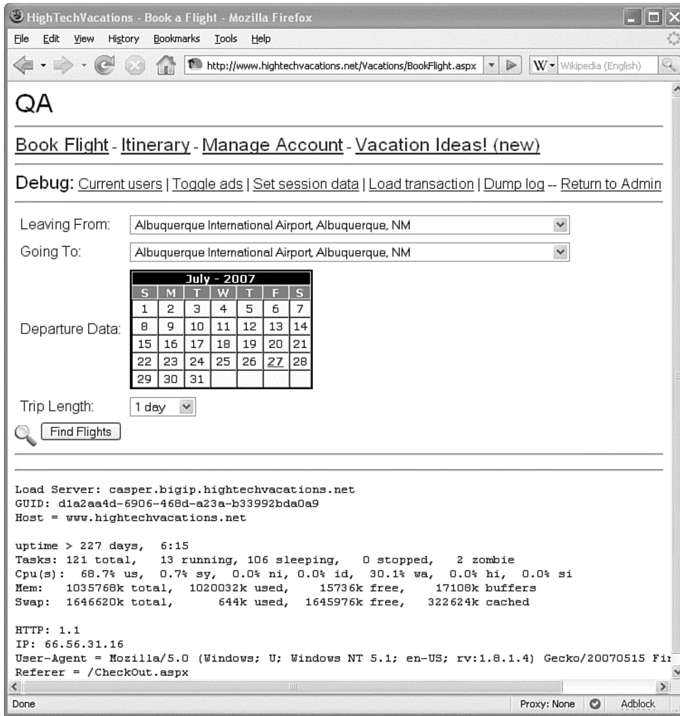
Och tak. Wystąpiła sytuacja, której Ewa się obawiała. Wysyła parametry w poprawnej formie, ale wygląda na to, że ostatni z nich — `debugflag` — jest nieprawidłowy. Opcja zwykle jest włączona bądź wyłączona. Ewa pomyślała, że wysłanie wartości `true` powinno zadziałać, ale myliła się. Spróbowała zastosować jeszcze kilka innych wartości, między innymi `true` umieszczone w cudzysłowie, `true` napisane dużymi literami, wartość `false`, ale wszystkie próby okazały się nieudane. W końcu podała `1` jako wartość opcji `debugflag`. Niektóre języki programowanie, takie jak C, nie posiadają rodzimych wartości `true` i `false`, ale zamiast nich używają odpowiednio `1` oraz `0`. Wynik tej operacji został pokazany na rysunku 2.12.



RYSUNEK 2.12. Ewa użyła 1 jako wartości opcji debugflag, po czym jej żądanie zostało zaakceptowane

Ewa nie wierzyła własnym oczom. Zadziałało! Nie była do końca pewna, jakiego rodzaju konto i gdzie zostało utworzone, ale na pewno utworzyła konto o nazwie *eve6*. Następnie powróciła do edytora HTTP i przeprowadziła kolejny atak typu SQL Injection, ponownie pobierając listę użytkowników. W ten sposób przekonała się, że na liście znajduje się nowe konto o nazwie *eve6*. Nadal jednak nie wiedziała, do czego służy flaga `debugflag` oraz gdzie jest przechowywana jej wartość. Chociaż mogła bardziej zagłębić się w bazę danych, poszukując tej opcji, to jednak zdecydowała się na utworzenie kolejnego konta. W przeglądarce internetowej otworzyła kolejną kartę i zalogowała się do nowo utworzonego konta *eve6*. Na rysunku 2.13 pokazano witrynę HighTechVacations.net po zalogowaniu się do konta *eve6*.

Wszystko było zupełnie inne! Ewa zobaczyła dane dotyczące używanej przez nią usługi sieciowej, obciążenie serwera oraz informacje dotyczące wysłanego żądania. Ewę najbardziej zainteresował pasek menu o nazwie *Debug*. Chociaż zawierał kilka opcji, Ewa natychmiast skoncentrowała się na łączu *Return to Admin*. Na bieżącą stronę nie dostała się



RYSUNEK 2.13. Witryna HighTechVacations.net prezentuje inny interfejs dla kont użytkowników debug

ze strony administracyjnej, więc zastanawiała się, co będzie, gdy spróbuje kliknąć łącze i powrócić na stronę administracyjną. Kliknęła wymienione łącze i zobaczyła stronę pokazaną na rysunku 2.14.

RYSUNEK 2.14. Obszar administracyjny dostępny z poziomu wersji debug witryny HighTechVacations.net

Znakomicie! Ewie wydawało się, że spowodowała wygenerowanie pewnego rodzaju wyjątku. Ponadto poznała położenie obszaru administracyjnego witryny. Często używała narzędzi, takich jak Nikto, w celu siłowego odgadnięcia często spotykanych katalogów typu *admin* lub *logs*, ale na liście katalogów nie znalazła */SiteConfig/*, więc mogła pominąć ten portal administracyjny. Dziwne, że w niektórych częściach witryny internetowej konto *eve6* było traktowane jak konto należące do administratora lub osoby odpowiedzialnej za zapewnienie jakości, podczas gdy w innych uniemożliwiało dostępu. Wygenerowanie wyjątku mogło nastąpić wtedy, gdy kod wewnętrzny próbował pobrać informacje o koncie *eve6*, których tam nie było, ponieważ w rzeczywistości konto *eve6* nie jest kontem administratora. Prawdopodobnie programiści tworzący witrynę HighTechVacations.net popełnili błąd, sądząc, że usługi administracyjne, takie jak `addUser()`, będą dostępne jedynie z poziomu portalu administracyjnego. Z tego powodu proces uwierzytelniania i autoryzacji zachodził najprawdopodobniej tylko wtedy, gdy użytkownik próbował uzyskać dostęp do portalu. Poprzez bezpośrednie działanie z `addUser()` lub inną usługą sieciową Ewa mogła wykonać wszystkie operacje zarezerwowane dla administratorów, ale bez faktycznego wykorzystania portalu administracyjnego.

Kradzież w jedną noc

Ewa ziewnęła, wypiła ostatni łyk kawy i przeciągnęła się. Jej włamanie zakończyło się pełnym sukcesem. Udało się zdobyć wszystkie kody promocyjne pozwalające na bezpłatną rezerwację biletów. Zdobyła listę wszystkich nazw użytkowników oraz złamała ich hasła dostępu. Uzyskała kopię danych kart kredytowych każdego, kto kiedykolwiek rezerwował lot za pomocą witryny HighTechVacations.net. Utworzyła sobie tylną furtkę w postaci (niestabilnego) konta z uprawnieniami administratora lub osoby odpowiedzialnej za zapewnienie jakości produktu. Wreszcie określiła login portalu administracyjnego, który prawdopodobnie może jej dać dostęp do innych witryn poza HighTechVacations.net.

Jednak nadal pozostały pewne możliwości, które mogłaby sprawdzić, jeśli miałaby na to ochotę. Przykładowo zwróciła uwagę, że podczas rezerwacji lotu następuje wywołanie serii usług sieciowych o nazwach `startTrans`, `holdSeat`, `checkMilesClub`, `debitACH`, `pushItinerary`, `pushConfirmEmail` oraz `commitTrans`. Jaki byłby skutek, gdyby Ewa wywołała wymienione usługi w innej kolejności? Czy nadal zostałaby obciążona płatnością, gdyby pominęła funkcję `debitACH`? Czy może przeprowadzić atak typu odmowa dostępu (ang. DoS — *Denial of Service*) poprzez rozpoczęcie tysięcy transakcji bazy danych, które nigdy nie zostaną sfinalizowane? Czy usługę `pushConfirmEmail` mogłaby wykorzystać do wysłania ogromnej ilości spamu lub do phishingu? Możliwości te pozostają do sprawdzenia w innym terminie, teraz zdobyła listę haseł. Najlepszym rozwiązaniem będzie sprzedaż spamerom zdobytych danych i poszukanie kolejnego celu ataku. Co można zrobić za pomocą portalu administracyjnego? Ewa pomyślała o połowicznie gotowym skrypcie w języku Perl, który napisała w celu siłowego łamania sieciowych formularzy logowania. Być może dzisiejszy atak będzie dobrym powodem, aby dokończyć pracę nad tym skryptem?

Ewa spogląda na zegarek, prawie dochodzi dwudziesta pierwsza. W czasie gdy wraca do domu, ktoś z jej partnerów biznesowych na Ukrainie właśnie kończy długą noc w klubie. Ewa uśmiecha się do siebie, ponieważ wie, że posiada dane, które mogą go zainteresować. Partner zawsze dobrze płaci, to tylko kwestia negocjacji.

Ewa wyłącza laptop, pakuje plecak i wyrzuca kubek po kawie do kosza znajdującego się przy wyjściu. Nie zdążyła przejechać nawet jednego kilometra, gdy na jej miejscu w lokalu siada kolejny klient i również wyciąga laptop. Nierzucająca się w oczy kobieta, która siedziała przy stole stojącym w rogu lokalu, znika w mroku nocy, nie pamiętają jej inni klienci i obsługa lokalu Caribou.

Nikt nie będzie pamiętał Ewy.

I taki stan rzeczy bardzo jej odpowiada.