helion.pl

Idź do

- Spis treści
- Przykładowy rozdział

Katalog książek

- Katalog online
- Zamów drukowany katalog

Twój koszyk

- Dodaj do koszyka

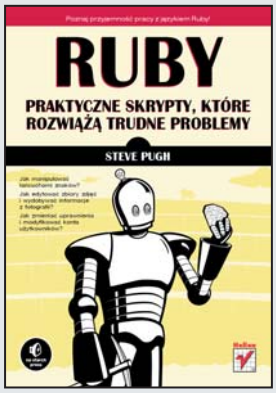
Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

Czytelnia

- Fragmenty książek online

Ruby. Praktyczne skrypty, które rozwiążą trudne problemy



Autor: Steve Pugh
Tłumaczenie: Tomasz Walczak, Marek Kowalcze
ISBN: 978-83-246-2254-2
Tytuł oryginału: Wicked Cool Ruby Scripts: Useful Scripts that Solve Difficult Problems
Format: 170x230, stron: 232

Poznaj przyjemność pracy z językiem Ruby!

- Jak manipulować łańcuchami znaków?
- Jak edytować zbiory zdjęć i wydobywać informacje z fotografii?
- Jak zmieniać uprawnienia i modyfikować konta użytkowników?

Ruby to obiektowy język programowania, przeznaczony do użytku ogólnego, udostępniający bogaty zestaw narzędzi służących do pisania skryptów. Jedną z największych zalet tego języka jest fakt, że wyjątkowo dobrze nadaje się on do tworzenia efektywnych programów proceduralnych. Dzięki tej książce nauczysz się wykorzystywać niezwykłe właściwości języka Ruby, aby zautomatyzować większość zadań i wykonywać swoją pracę bez trudu i z przyjemnością.

Książka „Ruby. Praktyczne skrypty, które rozwiążą trudne problemy" zawiera 58 niezwykle przydatnych skryptów, które pozwalają szybko rozwiązać często spotykane problemy, między innymi z administrowaniem systemem, manipulowaniem rysunkami i zarządzaniem witryną. Przy każdym skrypcie znajdziesz kod, omówienie jego działania i wskazówki opisujące, jak dostosować rozwiązanie do własnych potrzeb. Z podręcznika dowiesz się, jak posługiwać się skryptami do automatyzacji zadań (takich jak archiwizowanie czy wypakowywanie plików), a także na czym polega szyfrowanie plików oraz przetwarzanie wiadomości RSS. Nauczysz się tworzyć zaawansowane roboty sieciowe, skrypty z obszaru zabezpieczeń oraz kompletne biblioteki i aplikacje.

- Skrypty do zarządzania witrynami
- Pisanie skryptów sieciowych
- Administrowanie systemami Linux i Unix
- Walidacja rozwiązań symbolicznych
- Narzędzia do manipulowania zdjęciami
- Tworzenie galerii fotografii
- Narzędzia do przetwarzania łańcuchów znaków
- Serwery i wydobywanie danych
- Argumenty i dokumentacja
- Algorytmy sortowania

Skorzystaj z praktycznych skryptów w języku Ruby i oszczędzaj czas, aby pracować szybko i wydajnie

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**

PODZIĘKOWANIA **17**

WPROWADZENIE **19**
Odjazdowe skrypty w języku Ruby 20
Musisz znać podstawy języka Ruby 20
Dokumentacja 20
Struktura książki i zastosowane podejście 21
Witryna WWW 23

1.
NARZĘDZIA DO UŻYTKU OGÓLNEGO **25**
1. Wyszukiwanie zmodyfikowanych plików 25
Kod 26
Uruchamianie kodu 27
Dane wyjściowe 27
Jak działa ten skrypt? 28
Modyfikowanie skryptu 29
2. Szyfrowanie pliku 30
Kod 30
Uruchamianie kodu 30
Dane wyjściowe 31
Jak działa ten skrypt? 31
Modyfikowanie skryptu 32
3. Odszyfrowywanie pliku 32
Kod 32
Uruchamianie kodu 33
Wyniki 33
Jak działa ten skrypt? 33

4. Dzielenie plików ... 34
Kod ... 34
Uruchamianie kodu ... 35
Dane wyjściowe ... 35
Jak działa ten skrypt? ... 36
Modyfikowanie skryptu ... 36
5. Scalanie plików ... 36
Kod ... 37
Uruchamianie kodu ... 37
Dane wyjściowe ... 37
Jak działa ten skrypt? ... 38
Modyfikowanie skryptu ... 38
6. Przeglądarka procesów w systemie Windows ... 38
Kod ... 39
Uruchamianie kodu ... 39
Dane wyjściowe ... 39
Jak działa ten skrypt? ... 40
Modyfikowanie skryptu ... 40
7. Narzędzie do kompresji plików ... 41
Kod ... 41
Uruchamianie kodu ... 42
Dane wyjściowe ... 42
Jak działa ten skrypt? ... 42
8. Rozpakowywanie plików ... 43
Kod ... 43
Uruchamianie kodu ... 44
Dane wyjściowe ... 44
Jak działa ten skrypt? ... 44
9. Kalkulator raty kredytu hipotecznego ... 45
Kod ... 45
Uruchamianie kodu ... 45
Dane wyjściowe ... 45
Jak działa ten skrypt? ... 46
Modyfikowanie skryptu ... 46

2.
SKRYPTY DO ZARZĄDZANIA WITRYNAMI ... 47
10. Weryfikator odnośników do stron w sieci ... 48
Kod ... 48
Uruchamianie kodu ... 49
Dane wyjściowe ... 49
Jak działa ten skrypt? ... 49
Modyfikowanie skryptu ... 50

11. Kontroler osieroconych plików 51
Kod 51
Uruchamianie skryptu 52
Dane wyjściowe 52
Jak działa ten skrypt? 53
12. Generator formularzy 53
Kod 54
Uruchamianie kodu 55
Dane wyjściowe 55
Jak działa ten skrypt? 56
Modyfikowanie skryptu 57
13. Parser kanałów RSS 57
Kod 57
Uruchamianie kodu 58
Dane wyjściowe 58
Jak działa ten skrypt? 59
Modyfikowanie skryptu 60
14. Grep dla giełdy papierów wartościowych 60
Kod 60
Uruchamianie kodu 61
Dane wyjściowe 61
Jak działa ten skrypt? 62
Modyfikowanie skryptu 62
15. Generowanie adresów IP 63
Kod 63
Uruchamianie kodu 64
Dane wyjściowe 64
Jak działa ten skrypt? 65
16. Kalkulator masek podsieci 66
Kod 66
Uruchamianie kodu 66
Dane wyjściowe 66
Jak działa ten skrypt? 67
Modyfikowanie skryptu 67

3.
ADMINISTROWANIE SYSTEMAMI LINUX I UNIX 69
17. Poprawianie nieodpowiednich nazw plików 69
Kod 70
Uruchamianie kodu 71
Dane wyjściowe 71
Jak działa ten skrypt? 71
Modyfikowanie skryptu 72

18. Dodawanie kont użytkowników ... 72
Kod ... 72
Uruchamianie kodu ... 73
Dane wyjściowe ... 74
Jak działa ten skrypt? ... 74
Modyfikowanie skryptu ... 75
19. Modyfikowanie kont użytkowników ... 75
Kod ... 75
Uruchamianie kodu ... 77
Dane wyjściowe ... 77
Jak działa ten skrypt? ... 78
Modyfikowanie skryptu ... 78
20. Usuwanie zablokowanych procesów ... 78
Kod ... 79
Uruchamianie kodu ... 79
Dane wyjściowe ... 79
Jak działa ten skrypt? ... 80
21. Walidacja dowiązań symbolicznych ... 81
Kod ... 82
Uruchamianie kodu ... 82
Dane wyjściowe ... 82
Jak działa ten skrypt? ... 82
Modyfikowanie skryptu ... 83

4.
NARZĘDZIA DO MANIPULOWANIA ZDJĘCIAMI ... 85
22. Masowe edytowanie ... 86
Kod ... 86
Uruchamianie kodu ... 86
Dane wyjściowe ... 86
Jak działa ten skrypt? ... 87
23. Pobieranie informacji o zdjęciach ... 87
Kod ... 88
Uruchamianie kodu ... 88
Dane wyjściowe ... 88
Jak działa ten skrypt? ... 89
Modyfikowanie skryptu ... 90
24. Tworzenie miniatur ... 90
Kod ... 90
Uruchamianie kodu ... 91
Dane wyjściowe ... 91
Jak działa ten skrypt? ... 91
Modyfikowanie skryptu ... 92

25. Zmienianie wielkości zdjęć 92
Kod 92
Uruchamianie kodu 93
Dane wyjściowe 93
Jak działa ten skrypt? 93
Modyfikowanie skryptu 94
26. Dodawanie znaków wodnych do zdjęć 94
Kod 95
Uruchamianie kodu 95
Dane wyjściowe 96
Jak działa ten skrypt? 96
27. Przekształcanie zdjęć na czarno-białe 97
Kod 97
Uruchamianie kodu 98
Dane wyjściowe 98
Jak działa ten skrypt? 98
28. Tworzenie galerii fotografii 98
Kod 99
Uruchamianie kodu 100
Dane wyjściowe 100
Jak działa ten skrypt? 100
Modyfikowanie skryptu 102

5.
GRY I NARZĘDZIA WSPOMAGAJĄCE UCZENIE SIĘ 103
29. Rozwiązywanie łamigłówek Sudoku 103
Kod 104
Uruchamianie kodu 105
Dane wyjściowe 105
Jak działa ten skrypt? 106
30. Fiszki 107
Kod 107
Uruchamianie kodu 108
Dane wyjściowe 108
Jak działa ten skrypt? 108
Modyfikowanie skryptu 109
31. Gra w zgadywanie numerów 109
Kod 109
Uruchamianie kodu 110
Dane wyjściowe 110
Jak działa ten skrypt? 111
32. Kamień, papier, nożyce 111
Kod 111
Uruchamianie kodu 112

Dane wyjściowe 112
Jak działa ten skrypt? 113
Modyfikowanie skryptu 113
33. Rozsypanka wyrazowa 113
Kod 114
Uruchamianie kodu 114
Dane wyjściowe 115
Jak działa ten skrypt? 115
34. Szubienica 116
Kod 116
Uruchamianie kodu 117
Dane wyjściowe 117
Jak działa ten skrypt? 118
35. Świnia — gra w kości 118
Kod 119
Uruchamianie kodu 120
Dane wyjściowe 120
Jak działa ten skrypt? 122
Modyfikowanie skryptu 123

6.
NARZĘDZIA DO PRZETWARZANIA ŁAŃCUCHÓW ZNAKÓW 125
36. Generator dokumentów PDF 125
Kod 126
Uruchamianie kodu 127
Dane wyjściowe 127
Jak działa ten skrypt? 127
37. Zliczanie wystąpień słów 129
Kod 129
Uruchamianie kodu 130
Dane wyjściowe 130
Jak działa ten skrypt? 130
Modyfikowanie skryptu 131
38. Parser plików CSV 131
Kod 132
Uruchamianie kodu 133
Dane wyjściowe 133
Jak działa ten skrypt? 133
Modyfikowanie skryptu 134
39. Przekształcanie plików CSV na XML 134
Kod 135
Uruchamianie kodu 135
Dane wyjściowe 135
Jak działa ten skrypt? 136
Modyfikowanie skryptu 137

40. Program grep napisany w języku Ruby 137
Kod 137
Uruchamianie kodu 138
Dane wyjściowe 138
Jak działa ten skrypt? 138
Modyfikowanie skryptu 139
41. Sprawdzanie siły hasła 139
Kod 139
Uruchamianie kodu 140
Dane wyjściowe 140
Jak działa ten skrypt? 140
Modyfikowanie skryptu 142

7.
SERWERY I WYDOBYWANIE DANYCH 143
42. Definicje 143
Kod 144
Uruchamianie kodu 144
Dane wyjściowe 145
Jak działa ten skrypt? 145
Modyfikowanie skryptu 146
43. Automatyczne wysyłanie SMS-ów 146
Kod 146
Uruchamianie kodu 147
Dane wyjściowe 147
Jak działa ten skrypt? 147
44. Wydobywanie odnośników 148
Kod 148
Uruchamianie kodu 149
Dane wyjściowe 149
Jak działa ten skrypt? 150
Modyfikowanie skryptu 151
45. Wydobywanie rysunków 151
Kod 151
Uruchamianie kodu 152
Dane wyjściowe 152
Jak działa ten skrypt? 152
Modyfikowanie skryptu 153
46. Narzędzie do wydobywania danych ze stron WWW 153
Kod 153
Uruchamianie kodu 154
Dane wyjściowe 154
Jak działa ten skrypt? 155

47. Szyfrowanie po stronie klienta 155
Kod 155
Uruchamianie kodu 156
Dane wyjściowe 156
Jak działa ten skrypt? 156
48. Szyfrowanie po stronie serwera 157
Kod 157
Uruchamianie kodu 158
Dane wyjściowe 159
Jak działa ten skrypt? 159

8.
ARGUMENTY I DOKUMENTACJA 161
49. Bezpieczeństwo plików 162
Kod 162
Uruchamianie kodu 164
Dane wyjściowe 164
Jak działa ten skrypt? 165
50. Wydobywanie danych ze stron WWW 166
Kod 166
Uruchamianie kodu 168
Dane wyjściowe 169
Jak działa ten skrypt? 169
51. Narzędzia do zarządzania zdjęciami 170
Kod 170
Uruchamianie kodu 175
Dane wyjściowe 175
Jak działa ten skrypt? 176
Wnioski 176

9.
ALGORYTMY SORTOWANIA 177
52. Sortowanie bąbelkowe 178
Kod 178
Uruchamianie kodu 179
Dane wyjściowe 179
Jak działa ten skrypt? 179
53. Sortowanie przez wybieranie 180
Kod 181
Uruchamianie kodu 181
Dane wyjściowe 181
Jak działa ten skrypt? 182
54. Sortowanie Shella 182
Kod 182
Uruchamianie kodu 183

Dane wyjściowe 183
Jak działa ten skrypt? 184
55. Sortowanie przez scalanie 184
Kod 184
Uruchamianie kodu 186
Dane wyjściowe 186
Jak działa ten skrypt? 186
56. Sortowanie stogowe 186
Kod 187
Uruchamianie kodu 188
Dane wyjściowe 188
Jak działa ten skrypt? 188
57. Sortowanie szybkie 189
Kod 189
Uruchamianie kodu 190
Dane wyjściowe 190
Jak działa ten skrypt? 190
58. Sortowanie przez wycinanie 191
Kod 191
Uruchamianie kodu 193
Dane wyjściowe 193
Jak działa ten skrypt? 193
Komentarz na temat wydajności 194

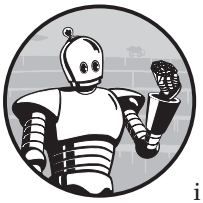
10.
TWORZENIE W JĘZYKU RUBY
MODUŁU PLATFORMY METASPLOIT 3.1 197
Wprowadzenie do platformy Metasploit 198
Instalowanie 198
Pisanie modułu 199
Budowanie exploita 201
Obserwowanie ataku w czasie rzeczywistym 202
Omówienie modułu powłoki platformy Metasploit 205
Określanie długości danych użytkowych 207

POSŁOWIE 219

SKOROWIDZ 221

7

Serwery i wydobywanie danych



Mocną stroną języka Ruby jest możliwość użycia go do zautomatyzowania interakcji z zasobami sieci WWW. Ten rozdział zawiera krótkie omówienie procesu przetwarzania stron WWW i kończy się zbiorem działających po stronie klienta oraz serwera skryptów, które służą do bezpiecznego przekazywania i uruchamiania poleceń. Interakcja z siecią WWW i wydobywanie z niej informacji (czyli tak zwany *data mining*) są ważne z uwagi na bogactwo dostępnych w niej materiałów. Zamiast wydobywać złoto, użytkownicy szukają różnych sposobów na pobranie ważnych danych i przekształcenie ich na znaczące informacje.

42. Definicje

define.rb Ten skrypt kieruje zapytanie do witryny *http://www.dictionary.com/* i pobiera pierwszą definicję słowa wskazanego przez użytkownika. Podobnie jak w przypadku innych programów komunikujących się z siecią WWW istnieje ryzyko, że skrypt przestanie działać, jeśli projektanci witryny wprowadzą w niej zmiany.

Zadanie programu polega na wydobywaniu określonych danych. Użyłem witryny Dictionary.com tylko jako środka do zaprezentowania tego mechanizmu, jednak omawiany przykład to pomysłowe rozwiązanie.

Kod

```
❶ require "open-uri"

  unless ARGV[0]
      puts "Musisz podać definiowane słowo."
      puts "Sposób użycia: ruby define.rb <definiowane słowo>"
      exit
  end

❷ word = ARGV[0].strip

❸ url = "http://dictionary.reference.com/browse/#{word}"

  begin
❹     open(url) do |source|
      source.each_line do |x|
❺         if x =~ /No results found/
          puts "\nDefinicji nie znaleziono - sprawdź pisownię."
          exit
      end
❻         if x =~ /(1\.)<\/td> <td>(.*)<\/td/
          puts "\n#{$1} #{$2}"
          exit
      end
  end
❼     puts "Niestety, nie można znaleźć definicji."
  end
  rescue => e
      puts "Wystąpił błąd — spróbuj ponownie."
      puts e
  end
```

Uruchamianie kodu

Uruchom skrypt przez wpisanie następującej instrukcji:

```
ruby define.rb definiowane_słowo
```

Chciałem znaleźć definicję słowa *Ruby*. Niestety, pierwsze zwrócone objaśnienie nie brzmiało *najbardziej odjazdowy języka programowania*.

Dane wyjściowe

Skrypt wyświetla definicję podanego słowa. Jeśli nie można jej znaleźć, program prosi użytkownika o sprawdzenie pisowni. Możliwe, że podany wyraz w ogóle nie istnieje.

```
1.a red variety of corundum, used as a gem.
```

Jak działa ten skrypt?

Ponownie wykorzystałem tu fantastyczną bibliotekę `open-uri` ❶. Do komunikacji skryptu z siecią WWW można używać wielu przydatnych narzędzi tego typu. Ja lubię `open-uri`, ponieważ ukrywa więcej szczegółów procesu łączenia się z siecią niż inne biblioteki. Po dołączeniu biblioteki program sprawdza błędy. Mam nadzieję, że przyzwyczaiłeś się już do używanego bloku kodu. W pierwszej zmiennej (`word`) skrypt zapisuje słowo, którego definicję użytkownik chce znaleźć ❷. Następnie program przypisuje do zmiennej `url` zapisany na stałe adres URL serwisu Dictionary.com z dołączonym wyrazem podanym jako argument ❸. Budowa używanej witryny sprawia, że dołączenie słowa do adresu powoduje automatyczne zwrócenie definicji.

W dalszej części skryptu znajduje się instrukcja `begin/rescue`, potrzebna z uwagi na zmienną naturę żądań sieciowych. Odpowiedzi na żądania HTTP często zawierają komunikaty o błędach. Ich właściwa obsługa to klucz do prawidłowego działania tego skryptu. Po dodaniu siatki zabezpieczającej w postaci bloku `begin/`
`↪rescue` można wysłać do serwisu Dictionary.com prośbę o podanie definicji. Kiedy korzystasz z biblioteki `open-uri`, wystarczy, że wpiszesz instrukcję `open()` z adresem URL, a skrypt zwróci odpowiednią stronę WWW ❹. Pobieranie stron za pomocą tej metody jest tak proste, że zawsze kiedy jej używam, na mojej twarzy gości uśmiech.

Po metodzie `open` następuje blok, który przetwarza kod źródłowy zwrócony przez serwer sieciowy. Ponieważ chcemy pobrać specyficzny wiersz (definicję słowa), w następnym bloku skrypt dzieli kod źródłowy na wiersze. Jeśli serwis Dictionary.com nie znajdzie danego słowa, wyświetli komunikat *No results found*. Gdy skrypt w czasie analizowania kodu źródłowego wykryje podany wyraz, ale już nie jego definicję, wyświetli wskazówkę w postaci prośby o sprawdzenie pisowni słowa, a następnie zakończy działanie ❺. Jeżeli jednak definicja jest dostępna, skrypt stara się wyodrębnić ją z kodu źródłowego. Do pobrania jej tekstu służy wyrażenie regularne.

Ważną częścią użytego w skrypcie wyrażenia regularnego jest cyfra `1`. Witryna Dictionary.com używa jej jako wskaźnika pierwszej definicji, którą program ma pobrać. Użycie w wyrażeniu regularnym nawiasów umożliwia skryptowi pogrupowanie określonych fragmentów dowolnego wiersza, które pasują do wyrażenia ❻. Te grupy są zapisywane w zmiennych o nazwach od `$1` do `$n`. Wiersz pod wyrażeniem regularnym wyświetla definicję. Jeśli program nie znajdzie w kodzie źródłowym ani definicji, ani tekstu *No results found*, zwróci inny komunikat, który

informuje użytkownika o tym, że nie można znaleźć wyjaśnienia słowa ❼. Jeśli w czasie przetwarzania definicji pojawią się problemy, program uruchomi blok `rescue` i poinformuje, jaki błąd wystąpił.

Modyfikowanie skryptu

Jednym ze sposobów na zmodyfikowanie skryptu jest dodanie obsługi pośrednika między użytkownikiem a żądaniem do serwera sieciowego. Jeśli korzystasz z takiego pośrednika, musisz zastosować to rozwiązanie. Jeżeli interesuje Cię ruch w sieci generowany przez skrypt w języku Ruby, pośrednik pomoże Ci uzyskać dodatkowe informacje. Aby dowiedzieć się, jak je uzyskać, zajrzyj do dokumentacji biblioteki `open-uri`. Potrzebna składnia wygląda następująco: `open(url, :proxy => "http://127.0.0.1:8080")`. Zwykle nie korzystam z pośrednika przy przeglądaniu sieci WWW, jednak jeśli w trakcie tworzenia stron wykryję błędy, często pomocne jest przyjrzenie się ruchowi w sieci.

Do diagnozowania problemów używam bezpłatnego sieciowego pośrednika Paros (*http://www.parosproxy.org/*). Narzędzie to należy zainstalować lokalnie na własnej maszynie, a następnie można obserwować zgłaszane żądania sieciowe i odbierane odpowiedzi. Dzięki wykorzystaniu przy programowaniu narzędzia Paros zaoszczędziłem sobie wielu godzin diagnozowania. Jestem bardzo przywiązany do tego pośrednika, jednak istnieje też wiele innych podobnych programów, dlatego warto poszukać czegoś dla siebie.

43. Automatyczne wysyłanie SMS-ów

sms.rb Ten skrypt wysyła wiadomość SMS pod podany numer telefonu komórkowego. Ostrzegam przed nadużywaniem tej funkcji, jednak musisz wypróbować ten program[1]. Skrypt ma automatyzować korzystanie z witryny, która służy do wysyłania wiadomości SMS. Zamiast pobierać statyczną zawartość strony, skrypt ten automatyzuje wypełnianie i przesyłanie formularza.

Kod

```
require 'win32ole'

❶ ie = WIN32OLE.new('InternetExplorer.Application')
❷ ie.navigate("http://toolbar.google.com/send/sms/index.php")

  ie.visible = true
❸ sleep 1 until ie.readyState() == 4

❹ ie.document.all["mobile_user_id"].value ="5712013623"
```

[1] Serwis użyty w tym skrypcie zakończył działalność, jednak za pomocą tej samej techniki możesz przesyłać wiadomości w innych bramkach SMS — *przyp. tłum.*

```
ie.document.all["carrier"].value ="TMOBILE"
ie.document.all["subject"].value ="***Ruby rządzi***"
❺ ie.document.all.tags("textarea").each do |i|
    i.value = "Dzięki za dobrą robotę, Matz!"
end

❻ ie.document.all.send_button.click
```

Uruchamianie kodu

```
Aby uruchomić ten skrypt, wpisz następujące polecenie:
```

```
ruby sms.rb
```

Dane wyjściowe

Ten skrypt nie generuje żadnych danych, jednak jeśli udanie zakończy działanie, telefon o podanym numerze powiadomi Cię o odebraniu wiadomości. W przykładzie użyłem fikcyjnych danych, jednak możesz je zmienić, aby wypróbować program.

Jak działa ten skrypt?

Jeśli korzystasz z komputera z systemem Windows i nigdy nie używałeś biblioteki `win32ole`, powinieneś poświęcić trochę czasu na jej opanowanie, ponieważ automatyzowanie wykonywania zadań w tym systemie jest ciekawe i daje dużo satysfakcji. Możesz nie tylko manipulować przeglądarką Internet Explorer, co ilustruje ten skrypt, ale też dowolnym produktem z pakietu Microsoft Office i innymi aplikacjami systemu Windows.

UWAGA *Dostępnych jest też kilka bibliotek przeznaczonych do automatyzacji witryn. Narzędzia te są niezwykle przydatne do testów regresji i jakości aplikacji sieciowych. Jedną z popularnych bibliotek tego typu jest Watir. Szczegółowe informacje na jej temat znajdziesz na stronie* http://wtr.rubyforge.org/.

Skrypt tworzy nowy obiekt typu `win32ole` na podstawie argumentu w postaci uchwytu przeglądarki IE ❶. Uchwyt ten informuje bibliotekę `win32ole`, którą aplikację ma kontrolować. Przy użyciu wbudowanej w przeglądarkę IE metody `navigate` skrypt przechodzi pod określony adres URL — *http://toolbar.google.com/send/sms/index.php* ❷. Następny wiersz ustawia atrybut okna przeglądarki. Jeśli nie chcesz widzieć, jak skrypt wykonuje swe zadania, możesz użyć w tym wierszu wartości `false`, a okno przeglądarki będzie działać w tle. Wtedy jego obecność będziesz mógł wykryć tylko na liście zadań. Ponieważ lubię widzieć działanie programu, ustawiłem wspomniany atrybut na `true`. Aplikacja Internet Explorer pojawi się i zniknie w krótkim czasie, dlatego bacznie przyglądaj się ekranowi.

Następnie skrypt uruchamia pętlę, która warunkowo wczytuje stronę. Jak pewnie wiesz, strony witryn nie pojawiają się natychmiast. Aby skrypt nie przesłał informacji przedwcześnie, należy „uśpić" program na sekundę, a następnie sprawdzić, czy kod `readyState` ma właściwą wartość (4) ❸. Przedwczesne działania nigdy nie są korzystne, a w tym przypadku spowodują, że skrypt przestanie działać. Po zakończeniu wczytywania dokumentu przez przeglądarkę IE skrypt może wypełnić odpowiednie pola.

Program wykrywa odpowiednie pola na podstawie nazw atrybutów. Jeśli zajrzysz do kodu źródłowego witryny, znajdziesz obiekty o nazwach `mobile_user_id`, `carrier`, `subject` i tak dalej. Te informacje służą do określenia, gdzie skrypt ma zapisać poszczególne dane wejściowe ❹. Większość kodu HTML z omawianej witryny jest zgodna ze standardami, jednak z nieznanych przyczyn wartość atrybutu `name` obszaru tekstowego nie znajduje się w cudzysłowach. Oznacza to, że nie można użyć wcześniej zastosowanej metody, aby uzyskać dostęp do tego obszaru. Jednak ponieważ w kodzie źródłowym znajduje się tylko jeden obszar tekstowy, wystarczy go znaleźć i zapisać w nim odpowiednie dane. Nie jest to wymyślna technika, ale nieco różni się od zastosowanego wcześniej podejścia ❺.

Po dodaniu informacji trzeba tylko wirtualnie kliknąć przycisk do wysyłania formularza. Programiści firmy Google prawidłowo nazywają przyciski, dlatego wystarczy pobrać odpowiednią nazwę i użyć jej w metodzie `click` ❻. I to już wszystko — język Ruby jest fantastyczny!

44. Wydobywanie odnośników

linkScrape.rb

Wydobywanie odnośników ze stron WWW ma wiele zastosowań. Można to zrobić — podobnie jak w przypadku innych zadań — na wiele sposobów. W rozdziale 2. zobaczyłeś skrypt do sprawdzania poprawności odnośników w witrynie. Z uwagi na konieczność walidacji odsyłaczy program ten wymagał większej liczby wierszy kodu niż skrypt, który musi tylko wydobyć wszystkie odnośniki. Nie zamierzam tworzyć tu robota sieciowego, jednak opiszę kilka jego podstawowych komponentów. Pierwszy z nich to mechanizm wydobywania odnośników.

Kod

```
❶ require 'mechanize'

  unless ARGV[0]
      puts "Musisz podać adres witryny."
      puts "Sposób użycia: ruby linkScrape.rb <przetwarzany adres URL>"
      exit
  end

❷ agent = WWW::Mechanize.new
  agent.set_proxy('localhost',8080)
```

```
begin
❸   page = agent.get(ARGV[0].strip)

    page.links.each do |l|
        if l.href.split("")[0] =='/'
❹           puts "#{ARGV[0]}#{l.href}"
        else
            puts l.href
        end
    end
rescue => e
    puts "Wystąpił błąd."
    puts e
    retry
end
```

Uruchamianie kodu

W celu uruchomienia skryptu wpisz następujące polecenie:

```
ruby linkScrape.rb http://przetwarzany_adres_url.com/
```

Dane wyjściowe

Skrypt wyświetla listę wszystkich odnośników znalezionych na stronie o podanym adresie URL. Ja wydobyłem odsyłacze ze strony *http://www.nostarch.com/main_menu.htm*.

index.htm
catalog.htm
wheretobuy.htm
about.htm
jobs.htm
media.htm
http://www.nostarch.com/blog/
http://ww6.aitsafe.com/cf/review.cfm?userid=8948354
abs_bsd2.htm
openbsd.htm
freebsdserver.htm
debian.htm
howlinuxworks.htm
appliance.htm
lcbk2.htm
lme.htm
nongeeks.htm
lps.htm
interactive.htm
gimp.htm
inkscape.htm
js2.htm
eblender.htm
oophp.htm
wpdr.htm
webbots.htm
google.htm
growingsoftware.htm
rootkits.htm
hacking2.htm
voip.htm
firewalls.htm
securityvisualization.htm
silence.htm
stcb4.htm
scsi2.htm cisco.htm
cablemodem.htm

mug.htm
ubuntu_3.htm
imap.htm
pf.htm
postfix.htm
webmin.htm
endingspam.htm
cluster.htm
nagios.htm
nagios_2e.htm
pgp.htm
packet.htm
tcpip.htm
assembly.htm
debugging.htm
qt4.htm
vb2005.htm
vsdotnet.htm
codecraft.htm
hownotc.htm
idapro.htm
mugperl.htm
gnome.htm
plg.htm
ruby.htm
vbexpress.htm
wcj.htm
wcps.htm
wcphp.htm
wcruby.htm
wcss.htm
greatcode.htm
greatcode2.htm
wpc.htm
xbox.htm
insidemachine.htm
nero7.htm
wireless.htm
creative.htm
ebaypg.htm
ebapsg.htm
geekgoddess.htm
wikipedia.htm
indtb.htm
sayno.htm
networkknowhow.htm
sharing.htm
apple2.htm
newmac.htm
cult_mac.htm
ipod.htm
art_of_raw.htm
firstlego.htm
flego.htm
legotrains.htm
sato.htm
nxt.htm
nxtonekit.htm
zoo.htm
legobuilder.htm
nxtig.htm
vlego.htm
mg_databases.htm
mg_statistics.htm
eli.htm
index.htm

Jak działa ten skrypt?

Porównaj kod tego programu ze skryptem 10. — „Weryfikator odnośników do stron w sieci". Widać dużą różnicę, prawda? Zawsze warto przemyśleć problem i starać się rozwiązać go w najprostszy możliwy sposób. Niektóre z najbardziej eleganckich rozwiązań w ogóle nie są skomplikowane. Przedstawiony skrypt jedynie wydobywa odnośniki ze strony bez sprawdzania ich poprawności i wykonywania innych zadań. Biblioteka `mechanize` to następne narzędzie często używane do interakcji z internetem ❶. Skrypt po standardowej instrukcji do obsługi błędów tworzy nowy obiekt `agent` typu `mechanize` ❷. Aby dostosować ten obiekt do potrzeb programisty, program ustawia atrybut pośrednika na lokalny program Paros,

którego używam. Jeśli nie chcesz korzystać z narzędzia tego rodzaju, usuń ten wiersz kodu. Następnie skrypt używa metody `get` obiektu `agent` do pobrania zawartości strony ❸. Ciekawą cechą biblioteki `mechanize` jest automatyczne kategoryzowanie pobranych informacji. Wyszukiwanie specyficznych elementów strony za pomocą tej biblioteki znacznie ułatwia życie programistom języka Ruby.

W zmiennej `page` znajduje się tablica `links`. Dzięki bibliotece `mechanize` odnośniki są już przetworzone. Podobnie jak przy korzystaniu z każdej innej tablicy można użyć metody `each` do przejścia po wszystkich elementach tablicy `links`. Pamiętaj, że elementy `link` zawierają nie tylko adres URL każdego odnośnika, ale też inne atrybuty zdefiniowane w kodzie źródłowym. Tu potrzebny jest tylko atrybut `href`, aby skrypt mógł wyświetlić go w konsoli ❹. Jeśli chcesz wydobyć informacje z dużej witryny, powinieneś zapisać dane w pliku, ale wybór podejścia należy do Ciebie. Po wyświetleniu odnośników skrypt kończy działanie.

Modyfikowanie skryptu

Istnieje kilka innych ciekawych narzędzi sieciowych, które przetwarzają strony w podobny sposób. Są to na przykład Hpricot (*http://wiki.github.com/why/hpricot/*) i Rubyful Soup (*http://www.crummy.com/software/RubyfulSoup/*). Wypróbuj je, aby znaleźć narzędzie dostosowane do własnych potrzeb.

45. Wydobywanie rysunków

imageScrape.rb Ten skrypt wydobywa każdy rysunek ze strony o adresie URL podanym przez użytkownika. Pobierane pliki graficzne obejmują obrazki znajdujące się na serwerze danej strony, a także rysunki dołączane z innych serwerów sieciowych.

Kod

```
require "open-uri"
require "pathname"

unless ARGV[0]
    puts "Musisz podać adres URL, aby wydobyć rysunki."
    puts "Sposób użycia: ruby imageScrape.rb <przetwarzany adres URL>"
    exit
end

url = ARGV[0].strip
begin
❶   open(url, "User-Agent" => "Mozilla/4.0 (compatible; MSIE 5.5; Windows
    ↪98)")
do |source|
        source.each_line do |x|
❷           if x =~ /<img src="(.+.[jpeg|gif])"\s+/
                name = $1.split('"').first
```

```
❸                   name = url + name if Pathname.new(name).absolute?
❹                   copy = name.split('/').last

❺                   File.open(copy, 'wb') do |f|
                        f.write(open(name).read)
                    end
                end
            end
        end
rescue => e
    puts "Wystąpił błąd – spróbuj ponownie."
    puts e
```

Uruchamianie kodu

Uruchom skrypt przez wpisanie poniższej instrukcji:

```
ruby imageScrape.rb http://przetwarzany_adres_url.com/
```

Dane wyjściowe

Ten skrypt pobiera wszystkie rysunki znalezione na stronie o podanym adresie URL. Ja włączyłem przetwarzanie strony *http://www.ruby-lang.org/* i pobrałem dwa obrazki: *logo.gif* (logo języka Ruby) i *download.gif* (rysunek-odnośnik, który pozwala pobrać język Ruby).

Jak działa ten skrypt?

Pierwszy etap wydobywania rysunków polega na pobraniu całej strony. Przy użyciu metody `open` biblioteki `open-uri` można wygodnie zapisać kod źródłowy strony w zmiennej `source` ❶. Jak sobie pewnie przypominasz z czasów pisania kodu w języku HTML, rysunki w dokumentach sieciowych są zagnieżdżane za pomocą znaczników `<img src=plik.jpg>`. W tym skrypcie użyłem wyrażenia regularnego, które analizuje każdy wiersz kodu źródłowego i wyszukuje podobne znaczniki ❷. Dzięki danym zwróconym przez wyrażenie regularne program może określić lokalizację znalezionych obrazków.

Po ustaleniu lokalizacji rysunku trzeba określić, czy obrazek pochodzi z zewnętrznej, czy z przetwarzanej witryny. W kodzie HTML rysunki z lokalnego serwera sieciowego są zwykle poprzedzone ukośnikiem (ich adres ma postać *ścieżki bezwzględnej*). W kodzie ścieżkę do obrazka zawiera zmienna `name`. Jeśli jest to ścieżka bezwzględna, skrypt dołącza ją do adresu URL strony, aby utworzyć kompletny adres rysunku. Sprawdzanie rodzaju ścieżki odbywa się przy użyciu metody `absolute?` i ma miejsce przy tworzeniu nowego obiektu typu `Pathname` ❸. Choć ścieżka do rysunku może się zmienić, jego nazwa jest stała i znajduje się w zmiennej `copy` ❹.

Po utworzeniu poprawnego adresu rysunku skrypt wykorzystuje obsługę plików wirtualnych, którą zapewnia biblioteka `open-uri`, w celu wczytania obrazka

i zapisania go w pliku (jego nazwę zawiera zmienna `copy`) ❺. Ten proces należy powtórzyć dla każdego rysunku znalezionego w dokumencie. Efekt tych operacji skrypt zapisuje w katalogu, w którym został uruchomiony.

Modyfikowanie skryptu

Możesz użyć gotowego parsera kodu HTML, takiego jak `mechanize`, Hpricot lub Rubyful Soup. Te narzędzia działają jeszcze dokładniej niż wyrażenie regularne zastosowane w skrypcie. Ponadto możesz zapisać rysunki w takiej samej strukturze katalogów, w jakiej znajdowały się na serwerze sieciowym. Istnieje wiele różnych możliwości, a ten skrypt to dobry punkt wyjścia.

46. Narzędzie do wydobywania danych ze stron WWW

scrape.rb Scraping w swej najprostszej postaci polega na wydobywaniu danych z innych witryn za pomocą standardowych zapytań HTTP. Przeznaczony do tego skrypt to wzbogacona wersja poprzednich programów. Program ten łączy omówione wcześniej techniki i udostępnia kilka dodatkowych funkcji. Jest to uniwersalne narzędzie do prostego pobierania danych ze stron. Nie jest to bot, ponieważ wszystkie działania skryptu wymagają interakcji z użytkownikiem, jednak po wprowadzeniu kilku drobnych usprawnień program ten można w pełni zautomatyzować.

Kod

```
require 'rio'
require 'open-uri'
require 'uri'

unless ARGV[0] and ARGV[1]
    puts "Musisz określić operację i adres URL."
    puts "Sposób użycia: scrape.rb [strona|rysunki|linki] <przetwarzany
    ↪adres URL>"
    exit
end

❶ case ARGV[0]

when "strona"
❷     rio(ARGV[1]) > rio("#{URI.parse(ARGV[1].strip).host}.html")
    exit
❸ when "rysunki"
    begin
        open(url, "User-Agent" => "Mozilla/4.0 (compatible; MSIE 5.5;
        ↪Windows 98)") do |source|
        source.each_line do |x|
```

```
            if x =~ /<img src="(.+.[jpeg|gif])"\s+/
                name = $1.split('"').first

                name = url + name if Pathname.new(name).absolute?
                    copy = name.split('/').last

                File.open(copy, 'wb') do |f|
                    f.write(open(name).read)
                end
            end
        end
    end
    rescue => e
        puts "Wystąpił błąd – spróbuj ponownie."
        puts e
    end
    exit
when "linki"
    links = File.open("links.txt","w+b")
    begin
❹       open(ARGV[1], "User-Agent" => "Mozilla/4.0 (compatible; MSIE 5.5;
        ↪Windows 98)") do |source|
❺             links.puts URI.extract(source, ['http', 'https'])
        end
    rescue => e
        puts "Wystąpił błąd – spróbuj ponownie."
        puts e
    end
    links.close
    exit
else
    puts "Podałeś nieprawidłową instrukcję – spróbuj ponownie."
    puts "Sposób użycia: scrape.rb [strona|rysunki|linki] <przetwarzany
adres URL>"
    exit
end
```

Uruchamianie kodu

Aby uruchomić skrypt, wpisz następujące polecenie:

```
ruby scrape.rb [strona|rysunki|linki] http://przetwarzany_adres_url.com/
```

Dane wyjściowe

Dane wyjściowe skryptu są różne w zależności od wybranej instrukcji. Możesz zapoznać się z przykładem z poprzedniego programu.

Jak działa ten skrypt?

Skrypt udostępnia trzy opcje. Możesz pobrać dane za pomocą instrukcji `linki`, `rysunki` lub `strona`. Do obsługi tych opcji służy instrukcja `case` ❶. Można też zastosować blok `if/else`, jednak polecenie `case` jest bardziej przejrzyste. Jeśli użytkownik chce pobrać stronę, skrypt używa instrukcji `rio` do skopiowania kodu źródłowego dokumentu do pliku HTML na lokalnym komputerze ❷. Polecenie `rio` obsługuje tak wiele szczegółowych, żmudnych operacji, że zadanie to można wykonać w jednym wierszu kodu!

Następny blok wydobywa rysunki ❸. Ta sekcja to kopia kodu ze skryptu 45. — „Wydobywanie rysunków", dlatego nie będę szczegółowo omawiał tego fragmentu. Jeśli chcesz dowiedzieć się czegoś więcej, zajrzyj do opisu poprzedniego programu.

Ostatni fragment instrukcji `case` wydobywa odnośniki. W odróżnieniu od innych części skryptu w tej zrezygnowałem ze stosowania gotowych rozwiązań, aby przedstawić inny sposób pobierania adresów URL. W tym podejściu do pobierania kodu źródłowego służy metoda `open` biblioteki `open-uri` ❹, a następnie skrypt wywołuje metodę `URI.extract`, która wyszukuje odsyłacze HTTP i HTTPS ❺. Wyniki program zapisuje w pliku tekstowym *links.txt*.

47. Szyfrowanie po stronie klienta

RSA_client.rb

W opisach technologii informacyjnych i zabezpieczeń często powtarzają się trzy zasady: poufności, integralności i dostępności. Każdy z tych komponentów wpływa na sposób interakcji użytkownika z danymi. Dwa następne skrypty mają wbudowane szyfrowanie RSA, co zapewnia poufność, i kodowanie SHA1, gwarantujące integralność informacji. Do przesyłania danych w tym programach służy sieciowe połączenie TCP.

Kod

```
require 'socket'
require 'digest/sha1'

begin
    print "Uruchamianie klienta..."
❶   client = TCPSocket.new('localhost', 8887)

    puts "nawiązano połączenie!\n\n"

❷   temp = ""
    5.times do
        temp << client.gets
    end
    puts "Otrzymano 1024-bitowy klucz publiczny RSA!\n\n"

❸   public_key = OpenSSL::PKey::RSA.new(temp)
```

```
    msg = 'mpg123*"C:\Program Files\Windows Media
    ➥Player\mplayer2.exe"*ruby.mp3'
❹   sha1 = Digest::SHA1.hexdigest(msg)

❺   command = public_key.public_encrypt("#{sha1}*#{msg}")
    print "Przesyłanie polecenia..."

❻   client.send(command,0)

    puts "wysłane!"
rescue => e
    puts "Wystąpił poważny problem..."
    puts e
    retry
end

client.close
```

Uruchamianie kodu

Aby włączyć skrypt, wpisz następującą instrukcję:

```
ruby RSA_client.rb
```

Dane wyjściowe

Poniżej znajdują się dane wyjściowe, wyświetlane po udanym nawiązaniu połączenia i przesłaniu polecenia.

```
Uruchamianie klienta...nawiązano połączenie!

Otrzymano 1024-bitowy klucz publiczny RSA!

Przesyłanie polecenia...wysłane!
```

Jak działa ten skrypt?

Działanie klienta rozpoczyna się od próby nawiązania połączenia TCP przy użyciu określonego adresu IP i numeru portu ❶. Jeśli operacja ta się powiedzie, skrypt przekaże do standardowego wyjścia tekst *nawiązano połączenie*. Następnie klient oczekuje na otrzymanie 1024-bitowego publicznego klucza szyfrującego RSA z serwera. Ten klucz skrypt zapisuje w zmiennej `tmp`, ponieważ do czasu przekształcenia danych na obiekt klucza RSA OpenSSL są one tylko niezrozumiałym łańcuchem znaków ❷. Po zainicjowaniu zmiennej `public_key` i zapisaniu w niej publicznego klucza RSI program potwierdza jego otrzymanie oraz może przystąpić do szyfrowania informacji ❸.

Skrypt przesyła dane, które zawierają nazwę odtwarzacza plików muzycznych. W systemie Linux jest to mpg123, a na platformie Windows — klasyczny odtwarzacz Media Player (plik *mplayer2.exe*). Oprócz nazwy aplikacji skrypt przekazuje nazwę pliku muzycznego — *ruby.mp3*. Ten plik znajduje się już na serwerze, dlatego program po prostu nakazuje serwerowi odtworzenie danego utworu. Poszczególne części łańcucha znaków z poleceniem są rozdzielone gwiazdką (*). Możesz w dowolny sposób zmodyfikować to polecenie i same dane, ponieważ program zaszyfruje je oraz prześle na serwer.

Następny krok to szyfrowanie informacji. Skrypt zapisuje omówiony wcześniej łańcuch znaków z zapytaniem w zmiennej `msg` i szyfruje przy użyciu publicznego klucza RSA otrzymanego z serwera. Przed zaszyfrowaniem danych program koduje komunikat za pomocą algorytmu SHA1 i zapisuje wygenerowany kod w zmiennej `sha1` ❹. Ten kod zostanie użyty po przesłaniu danych na serwer. Pamiętaj, że algorytm SHA1 jest jednostronny, dlatego jeśli w trakcie przekazywania danych zostaną one zmodyfikowane, pierwotny i nowy kod będą się od siebie różnić.

Następnie skrypt scala zmienne `sha1` i `msg`, łącząc je gwiazdką. Do szyfrowania danych wyjściowych służy metoda `public_encrypt` obiektu klucza RSA ❺. Jak może zgadłeś, metoda ta szyfruje informacje przy użyciu publicznego klucza RSA. Tylko odpowiadający mu prywatny klucz RSA pozwala odszyfrować komunikat.

Na zakończenie skrypt przesyła zaszyfrowany komunikat na serwer i zamyka połączenie ❻. Jeśli w trakcie szyfrowania lub przesyłania danych wystąpią problemy, sytuację pozwoli uratować zaufany blok `begin/rescue`. Jeżeli wszystkie operacje zakończą się sukcesem, serwer odtworzy fantastyczne dźwięki związane z językiem Ruby! Czy jest w życiu coś piękniejszego od słuchania piosenek o języku Ruby?

48. Szyfrowanie po stronie serwera

RSA_server.rb

Po zapoznaniu się z klientem i jego kodem pora przyjrzeć się serwerowi. Przyjmuje on dane, sprawdza, czy kod SHA1 jest poprawny, odszyfrowuje informacje i wykonuje polecenie z przesłanego łańcucha znaków.

Kod

```
require 'openssl'
require 'socket'
require 'digest/sha1'

❶ priv_key = OpenSSL::PKey::RSA.new(1024)
pub_key = priv_key.public_key

host = ARGV[0] || 'localhost'
port = (ARGV[1] || 8887).to_i

❷ server = TCPServer.new(host, port)
```

```
❸ while session = server.accept
      begin
          puts "Nawiązano połączenie...trwa wysyłanie klucza
          ↪publicznego.\n\n"
          puts pub_key
❹         session.print pub_key
          puts "Przesłano klucz publiczny – trwa oczekiwanie na
          ↪dane...\n\n"

❺         temp = session.recv(10000)
          puts "Otrzymano dane..."

❻         msg = priv_key.private_decrypt(temp)
      rescue => e
          puts "Wystąpił poważny problem przy pobieraniu i odszyfrowywaniu
          ↪danych."
          puts e
      end

❼     command = msg.split("*")

      serv_hash = command[0]
      nix_app = command[1]
      win_app = command[2]
      file = command[3]

❽     if Digest::SHA1.hexdigest("#{nix_app}*#{win_app}*#{file}")==serv_hash
          puts "Potwierdzono integralność komunikatu..."
❾         if RUBY_PLATFORM.include?('mswin32')
              puts "Uruchamianie polecenia dla systemu Windows: #{win_app}
              ↪#{file}"
              `#{win_app} #{file}`
              exit
❿          else
              puts "Uruchamianie polecenia dla systemu Linux: #{nix_app}
              ↪#{file}"
              `#{nix_app} #{file}`
              exit
          end
      else
          puts "Nie można przeprowadzić walidacji komunikatu!"
      end
      exit
  end
```

Uruchamianie kodu

W celu uruchomienia skryptu wpisz poniższe polecenie:

```
ruby RSA_server.rb
```

Dane wyjściowe

Poniżej znajdują się dane wyjściowe, wyświetlane po udanym nawiązaniu połączenia i przesłaniu polecenia.

```
Nawiązano połączenie...trwa wysyłanie klucza publicznego.

-----BEGIN RSA PUBLIC KEY-----
MIGJAoGBAMe12IJIyVULS/OLlHeekhZNyh2YhuGfJSwEozw2Z6GfaRjZg7sOcwqb
B/Z+MMUPIjCmiH38pkKzh5GhA8zcRSWEFtssa8HcyIowA5ftZM27/6diYz9kNueI
NO2kvlkqwU5KUOKnLISJnrZAlTbJMqio24dn3PNm27kgae8+KdrHAgMBAAE=
-----END RSA PUBLIC KEY-----
Przesłano klucz publiczny – trwa oczekiwanie na dane...

Otrzymano dane...
Potwierdzono integralność komunikatu...
Wykonywanie polecenia dla systemu Windows: "C:\Program Files\Windows Media
Player\mplayer2.exe" ruby.mp3
```

Jak działa ten skrypt?

Skrypt generuje najpierw niepowtarzalny prywatny klucz RSA ❶. Na podstawie tego klucza powstaje publiczny klucz RSA. Do jego wygenerowania służy metoda `public_key` obiektu klucza RSA. Przy każdym uruchomieniu program tworzy nową parę kluczy. Jeśli klient prześle dane zaszyfrowane przy użyciu dawnego klucza publicznego, serwer nie zdoła odszyfrować komunikatu.

Po utworzeniu kluczy RSA skrypt inicjuje serwer TCP ❷. Serwer ten należy uruchomić przy użyciu argumentów w postaci nazwy serwera i portu, które można podać w wierszu poleceń. Jeśli te argumenty nie są dostępne, skrypt używa ustawień domyślnych. Kiedy serwer jest już gotowy, rozpoczyna oczekiwanie na połączenia. Pętla `while` umożliwia zarządzanie w skrypcie sesjami ❸. Ponieważ program nie jest wielowątkowy, w danym momencie obsługuje tylko jedno połączenie.

Kiedy użytkownik uruchomi klienta, ten nawiąże połączenie z serwerem. Spowoduje to rozpoczęcie nowej sesji, a pierwsza operacja w niej polega na przesłaniu przez serwer publicznego klucza RSA ❹. Klucz RSA jest mały, dlatego jego przekazywanie trwa krótko. Następnie skrypt oczekuje na przesłanie danych przez klienta. W tym czasie klient otrzymuje publiczny klucz RSA i szyfruje przesyłany komunikat. W zmiennej `temp` skrypt zapisuje dane odebrane przez serwer przy użyciu połączenia TCP. Limit tych informacji to 10 000 bajtów ❺. Dopiero po odebraniu danych program przechodzi do dalszych działań.

Przy użyciu metody `private_decrypt` obiektu klucza RSA skrypt odszyfrowuje dane ze zmiennej `temp` i zapisuje je w zmiennej `msg` ❻. Jeśli w czasie pobierania i odszyfrowywania łańcucha znaków z poleceniem wystąpią błędy, klauzula `rescue` przechwyci je i wyświetli przydatne informacje, które pomogą rozwiązać problem.

Jak pewnie sobie przypominasz z opisu skryptu 47. — „Szyfrowanie po stronie klienta”, łańcuch znaków z poleceniem jest ograniczony gwiazdką (*). Dlatego aby pobrać właściwy fragment tekstu, należy użyć metody `split` i podać gwiazdkę jako ogranicznik zastosowany w łańcuchu `msg` ❼. Dane wyjściowe skrypt zapisuje w zmiennej `command`, która przechowuje tablicę łańcuchów znaków. Ponieważ tekst powstaje w skrypcie klienckim, wiadomo, z jakich fragmentów się składa. Na początku znajduje się kod SHA1, następnie nazwa aplikacji dla systemu Linux, nazwa programu dla platformy Windows i przeznaczony do użycia plik.

Do utworzenia kodu SHA1 służy instrukcja dla systemu Linux, polecenie dla platformy Windows i nazwa pliku ❽. Między poszczególnymi częściami łańcucha znaków znajdują się gwiazdki, co pozwala odtworzyć pierwotny tekst. Efekt jego zakodowania jest porównywany z wartością zmiennej `serv_hash`, która zawiera kod SHA1 przesłany przez klienta. Jeśli porównywane kody nie są sobie równe, dane musiały ulec zmianie w trakcie ich przesyłania. Nie można ufać takim informacjom, dlatego program kończy działanie. Miejmy jednak nadzieję, że oba kody będą takie same, a skrypt wykona dalsze operacje.

Po potwierdzeniu integralności komunikatów program musi wybrać uruchamianą aplikację. Ruby umożliwia łatwe określenie używanego systemu. Wystarczy o to zapytać przy użyciu wyrażenia `RUBY_PLATFORM`. Jeśli korzystasz z systemu Windows, wartość tego wyrażenia to `i386-mswin32`. Przy użyciu wygodnej metody `include?` skrypt sprawdza, czy łańcuch znaków z wyrażenia `RUBY_PLATFORM` zawiera fragment `mswin32` ❾. Jeśli ten test zwróci wartość `true`, program wywoła polecenie dla systemu Windows. W przeciwnym razie uruchomiona zostanie aplikacja dla platformy Linux ❿. W obu przypadkach jeżeli wszystkie operacje przebiegną prawidłowo, system powinien uruchomić program muzyczny i rozpocząć odtwarzanie pliku *ruby.mp3*. Skrypt kończy działanie po zamknięciu aplikacji muzycznej. W ten sposób można komunikować się w poufny sposób, a jednocześnie zachować integralność danych.