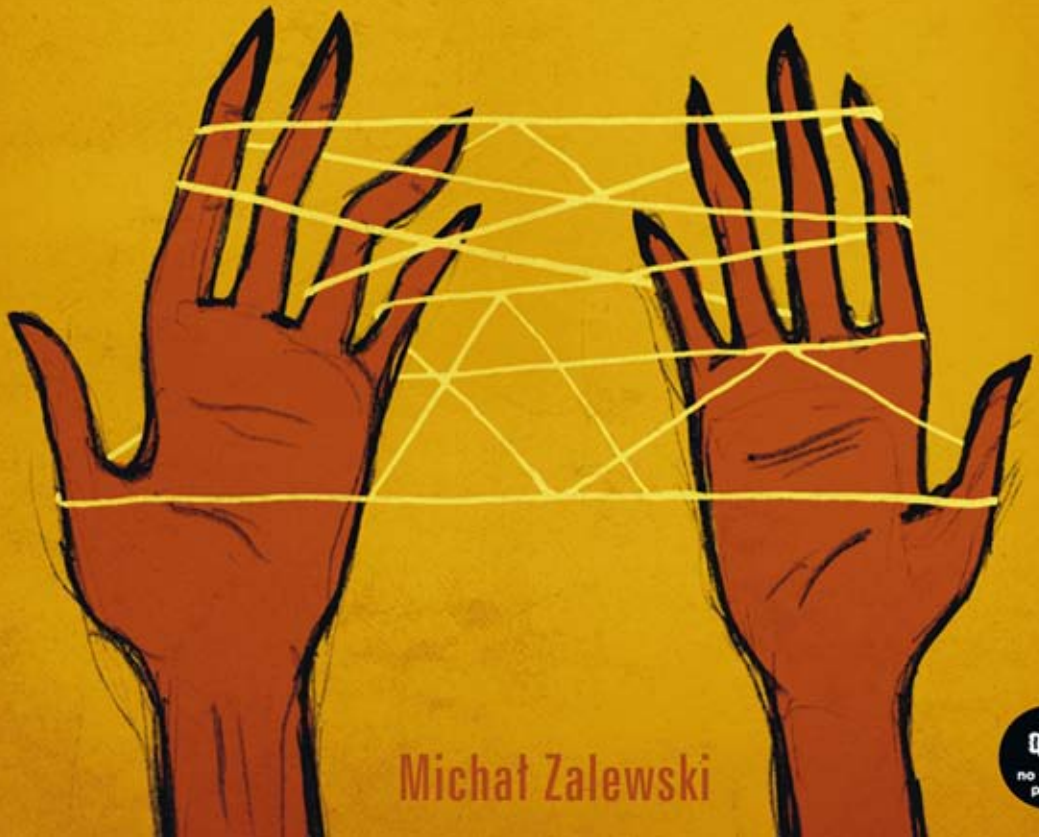
Helion

Splątana sieć

Przewodnik po bezpieczeństwie nowoczesnych aplikacji WWW



Michał Zalewski

no starch press

Tytuł oryginału: The Tangled Web: A Guide to Securing Modern Web Applications

Tłumaczenie: Wojciech Moch

ISBN: 978-83-246-4477-3

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

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

Printed in Poland.

Spis treści

6

Skrypty działające w przeglądarce

Pierwszy mechanizm skryptowy działający w przeglądarce pojawił się w Nestcape Navigatorze w roku 1995 jako wynik prac Brendana Eicha. Wbudowany język Mocha (bo taka była jego pierwotna nazwa) dawał twórcom stron możliwość manipulowania dokumentami HTML, wyświetlania prostych systemowych okien dialogowych, otwierania i przesuwania okien przeglądarki i wykorzystywania innych prostych typów automatyzacji działania strony klienta.

W kolejnych wydaniach beta swojej przeglądarki Netscape ostatecznie zmienił nazwę Mocha na LiveScript, a po dopięciu dziwacznych umów z firmą Sun Microsystems ostateczną nazwą języka stało się JavaScript. Było zaledwie kilka podobieństw między językiem Mocha Eicha Brendana a Javą Suna, ale w firmie Netscape wierzono, że takie marketingowe małżeństwo z pewnością pozwoli uzyskać JavaScriptowi dominację w lukratywnym świecie serwerów. Pragnienie to stało się jasne po pojawieniu się w 1995 roku słynnego i wprowadzającego w błąd ogłoszenia zapowiadającego nowy język, które od razu wiązało go z ogromną liczbą produktów komercyjnych[1]:

> **Netscape i Sun prezentują JavaScript, nowy, otwarty, międzyplatformowy i obiektowy język skryptowy dla sieci korporacyjnych i internetu.**
>
> […]
>
> Netscape Navigator Gold 2.0 pozwala na tworzenie i edytowanie skryptów w języku JavaScript, natomiast Netscape LiveWire umożliwia instalowanie, uruchamianie i zarządzanie programami napisanymi w tym języku, zarówno w sieci korporacyjnej, jak i w całym internecie. Netscape LiveWire Pro dodaje funkcje łączenia z językiem JavaScript wydajnych relacyjnych baz danych firm Illustra, Informix, Microsoft, Oracle i Sybase. Obsługa języków Java i JavaScript została wbudowana w produkty firmy Netscape w celu stworzenia zunifikowanego środowiska narzędzi klienckich i serwerowych przeznaczonych do tworzenia i wdrażania aplikacji sieciowych.

Mimo niewłaściwego przywiązania firmy Netscape do języka Java wartość języka JavaScript dla rozwiązań programistycznych działających po stronie klienta wydawała się całkiem spora, co było zrozumiałe również dla konkurencji. W 1996 roku Microsoft odpowiedział Internet Explorerem 3.0, w którym wprowadzona została niemal idealna kopia języka JavaScript oraz własna propozycja firmy: zmieniona wersja języka Visual Basic o nazwie VBScript. Zaproponowana przez Microsoft alternatywa nie zyskała jednak popularności, a jej obsługa nie została nawet zaimplementowana w innych przeglądarkach, co mogło wynikać ze spóźnionej prezentacji albo z nieeleganckiej składni nowego języka. Ostatecznie to JavaScript zagarnął lwią część rynku, a od tego czasu nawet nie próbowano wprowadzić nowych języków skryptowych do przeglądarek, co również może wynikać z pierwotnej porażki Microsoftu na tym polu.

Firma Netscape zachęcona popularnością języka JavaScript część odpowiedzialności za jego rozwój i konserwację przekazała niezależnemu stowarzyszeniu ECMA (*European Computer Manufacturers Association*). Nowy opiekun języka już w 1999 roku udostępnił trzecie wydanie specyfikacji ECMAScript[2], ale od tego momentu rozwój języka stał się dużo trudniejszy. Czwarte wydanie specyfikacji, które całkowicie przedefiniowało cały język, po kilku latach przepychanek między twórcami przeglądarek zostało odłożone na półkę, a ograniczone wydanie piąte[3] mimo opublikowania w 2009 roku nadal nie cieszy się pełną obsługą w przeglądarkach (choć ten stan ciągle się poprawia). W 2008 roku rozpoczęły się prace nad kolejną iteracją, nazywaną „Harmony", ale są one nadal dalekie od ukończenia. Z powodu braku obowiązującego wszystkich i stale ewoluującego standardu pojawiają się rozszerzenia języka tworzone przez poszczególnych producentów przeglądarek, ale takie dodatki zwykle powodują tylko problemy.

Podstawowe cechy języka JavaScript

JavaScript jest względnie prostym językiem przeznaczonym do interpretowania w momencie uruchomienia. Charakteryzuje się składnią podobną do używanej w języku C (z wyjątkiem arytmetyki wskaźników), prostym, bezklasowym modelem

obiektów, który podobno został zainspirowany mało znanym językiem programowania o nazwie Self, mechanizmem automatycznego zwalniania pamięci i słabym typowaniem dynamicznym.

JavaScript nie ma wbudowanych mechanizmów wejścia-wyjścia. W przeglądarkach otrzymał ograniczone możliwości interakcji ze środowiskiem komputera, realizowane za pomocą metod i właściwości odwołujących się do kodu samej przeglądarki. Jednak w przeciwieństwie do interfejsów znanych z innych języków programowania funkcje te mają bardzo ograniczone możliwości i zostały przygotowane do realizowania konkretnych zadań.

Większość głównych funkcji języka JavaScript nie jest niczym niezwykłym i powinny być one znane programistom używającym już języków C lub C++ oraz w mniejszym stopniu — Java. Prosty program JavaScript może wyglądać tak:

```
var text = "Cześć, mamo!";

function display_string(str) {
  alert(str);
  return 0;
}

// To wywołanie wyświetli tekst "Cześć, mamo!".
display_str(text);
```

Dokładniejsze wyjaśnienie semantyki języka JavaScript wykracza poza ramy tej książki, dlatego w tym rozdziale podsumuję tylko szczególne i wpływające na bezpieczeństwo cechy języka. Czytelnicy szukający bardziej systematycznego wprowadzenia do języka powinni zainteresować się książką Marjina Haverbeke'a *Eloquent JavaScript* (No Starch Press, 2011).

Model przetwarzania skryptów

Każdy dokument HTML wyświetlany w przeglądarce — niezależnie od tego, czy w osobnym oknie, czy też w ramce — otrzymuje osobne środowisko wykonania skryptów JavaScript razem z niezależną przestrzenią nazw dla wszystkich zmiennych globalnych oraz funkcjami tworzonymi przez ładowane skrypty. Wszystkie skrypty wykonywane w ramach kontekstu konkretnego dokumentu współdzielą między sobą jedną piaskownicę, ale mogą też kontaktować się z innymi kontekstami za pośrednictwem specjalnego API. Takie interakcje między dokumentami muszą być wykonywane w sposób otwarty, dlatego przypadkowe interferencje są bardzo mało prawdopodobne. Na pierwszy rzut oka reguły izolowania skryptów bardzo przypominają model szufladkowania procesów, znany z nowoczesnych systemów operacyjnych, ale nie są one aż tak rozbudowane.

W ramach jednego kontekstu wykonania wszystkie znalezione bloki JavaScriptu przetwarzane są po kolei, niemal zawsze w ściśle określonym porządku. Każdy blok kodu musi składać się z dowolnej liczby prawidłowo zbudowanych jednostek składniowych, które są przetwarzane w trzech następujących po sobie krokach o nazwach: parsowanie, rozwiązywanie funkcji i wykonanie kodu.

Parsowanie

Na etapie parsowania sprawdzana jest składnia bloku skryptu, który najczęściej konwertowany jest do pośredniej postaci binarnej. Wykonanie skryptu zapisanego w tej postaci jest zdecydowanie szybsze. Sam kod nie ma żadnych globalnych skutków do czasu całkowitego ukończenia prac na tym etapie. Jeżeli pojawią się błędy składniowe, to cały problematyczny blok kodu jest porzucany, a parser przechodzi do kolejnego dostępnego bloku kodu.

W ramach zobrazowania zachowania parsera zgodnego ze specyfikacją języka JavaScript zachęcam do przyjrzenia się poniższemu wycinkowi kodu:

```
blok nr 1:  <script>
            var my_variable1 = 1;
            var my_variable2 =
            </script>

blok nr. 2: <script>
            2;
            </script>
```

Doświadczenie zdobyte w pracy z językiem C podpowiada, że powyższy kod powinien zostać zinterpretowany tak samo jak poniższy. W języku JavaScript tak się jednak nie dzieje.

```
<script>
var my_variable1 = 1;
var my_variable2 = 2;
</script>
```

Wynika to z faktu, że bloki `<script>` nie są ze sobą łączone przed rozpoczęciem parsowania. W tym przypadku pierwszy segment skryptu spowoduje powstanie błędu składniowego (przypisanie z brakującą wartością po prawej stronie), przez co cały blok zostanie zignorowany i nie przejdzie do etapu wykonania. Fakt, że cały segment jest pomijany, zanim będzie mógł mieć jakikolwiek wpływ na stronę, oznacza też, iż pierwotny przykład nie jest równoważny poniższemu:

```
<script>
var my_variable1 = 1;
</script>

<script>
2;
</script>
```

Ta cecha odróżnia język JavaScript od wielu innych języków skryptowych, takich jak Bash, w których etap parsowania nie został tak mocno oddzielony od etapu wykonania.

W naszym oryginalnym przykładzie po zignorowaniu pierwszego bloku kodu parser przejdzie do drugiego dostępnego bloku (`<script>2;</script>`), który zostanie obsłużony bez żadnych błędów. Drugi blok kodu jest równoważny wykonaniu instrukcji pustej, ponieważ znajduje się w nim tylko jedna instrukcja będąca wartością numeryczną.

Rozwiązywanie funkcji

Po zakończeniu etapu parsowania w kolejnym kroku rejestrowane są wszystkie nazwane, globalne funkcje znalezione przez parsera w aktualnym bloku kodu. Od tego momentu każda znaleziona funkcja będzie dostępna w uruchomionym później kodzie. Dzięki zastosowaniu tego kroku poniższy kod zadziała bez żadnych problemów, ponieważ funkcja `hello_world()` zostanie zarejestrowana jeszcze przed wykonaniem pierwszego wiersza kodu, wywołującego tę właśnie funkcję. Jest to działanie odmienne od tego, do jakiego przyzwyczaili się programiści używający języków C lub C++.

```
<script>
hello_world();

function hello_world() {
  alert('Cześć, mamo!');
}
</script>
```

Z drugiej strony poniższy kod nie zachowa się tak, jak byśmy sobie tego życzyli:

```
<script>
hello_world();
</script>

<script>
function hello_world() {
  alert('Cześć, mamo!');
}
</script>
```

Tak zmieniony kod zgłosi błędy w czasie wykonania, ponieważ poszczególne bloki nie są obsługiwane razem, ale w kolejności, w jakiej zostają zaprezentowane mechanizmowi skryptowemu. W czasie gdy pierwszy blok jest już wykonywany, blok definiujący funkcję `hello_world()` nie został nawet sparsowany.

Żeby było jeszcze ciekawiej, okazuje się, że opisany tu nieco dziwny sposób rozwiązywania nazw dotyczy wyłącznie funkcji, a deklaracje zmiennych są z niego wyłączone. Zmienne są rejestrowane sekwencyjnie w czasie wykonania, podobnie jak dzieje się to we wszystkich innych językach interpretowanych. Oznacza to, że poniższy przykład, w którym funkcja `hello_world()` została zastąpiona funkcją anonimową przypisywaną do zmiennej globalnej, nie będzie działał tak, jak byśmy tego chcieli:

```
<script>
hello_world();

var hello_world = function() {
  alert('Cześć, mamo!');
}
</script>
```

W tym przypadku przypisanie wartości do zmiennej `hello_world` wykonywane jest dopiero po próbie wywołania `hello_world()`.

Wykonywanie kodu

Po zakończeniu procesu rozwiązywania nazw mechanizm skryptowy przystępuje do wykonywania wszystkich instrukcji znajdujących się poza blokami funkcji. Na tym etapie wykonywanie skryptu może zostać zatrzymane z powodu nieobsłużonego wyjątku lub z kilku innych, bardziej ezoterycznych powodów. Mimo wystąpienia takiego błędu nadal możliwe będzie wywołanie wszystkich funkcji znajdujących się w tym samym bloku kodu, a zmiany wprowadzone przez wykonany do tej pory kod pozostaną aktywne w ramach bieżącego kontekstu.

W poniższym (długawym, ale i interesującym) wycinku kodu prezentowanych jest kilka innych cech wykonania kodu JavaScript, w tym i mechanizm wznawiania pracy po wystąpieniu wyjątku:

```
<script>
function not_called() {
  return 42;
}

function hello_world() {
  alert("W tym programie wszystko jest możliwe!");
  do_stuff();
}

alert("Witamy w naszej przykładowej aplikacji.");

hello_world();

alert("Dziękujemy, zapraszamy ponownie.");
</script>


<script>
alert("Skoro już popracowaliśmy, to może partyjka szachów?");
</script>
```

Ta funkcja nie zostanie wykonana, ponieważ nigdy nie jest wywoływana.

Ta funkcja zostanie wykonana dopiero wtedy, gdy zostanie wywołana. Wyświetla ona okno dialogowe, a następnie rzuca wyjątek z powodu próby wywołania funkcji `do_stuff()`, której nazwa nie została rozwiązana.

Wykonywanie programu rozpoczyna się od tej instrukcji.

Następnie wyświetlany jest komunikat „W tym…”.

Ten kod nie zostanie wykonany z powodu nieobsłużonego wyjątku rzuconego w funkcji `hello_world()`.

Poprzedni wyjątek nie zablokuje jednak wykonania kodu znajdującego się w osobnym bloku.

Proszę samodzielnie wypróbować działanie tego przykładu i sprawdzić, czy podane przy nim komentarze zgadzają się z rzeczywistością.

Z tego małego ćwiczenia wynika, że nieoczekiwane i nieobsłużone wyjątki mają dość nietypowe konsekwencje. Mogą one wprowadzić aplikację w stan niespójny, a mimo to umożliwiający dalsze wykonywanie kodu. Zadaniem wyjątków jest uniemożliwienie rozprzestrzenienia się błędów wynikających z nieoczekiwanych sytuacji, a zatem taki sposób wykonywania skryptów należy uznać za dziwny, tym bardziej że na wielu innych frontach (na przykład w sprawie zakazu używania instrukcji `goto`) język JavaScript ma zdecydowanie twardsze zasady.

Zarządzanie wykonaniem kodu

Chcąc odpowiednio przeanalizować cechy bezpieczeństwa pewnych popularnych wzorców projektowych aplikacji WWW, musimy dokładnie poznać model koordynacji i zarządzania wykonywaniem kodu stosowany w języku JavaScript. Na szczęście model ten jest zadziwiająco rozsądny.

Niemal cały kod JavaScript, który znajduje się w określonym kontekście, wykonywany jest w sposób synchroniczny. Jeżeli kod skryptu jest wykonywany, to jego wykonywanie nie może zostać wznowione w wyniku działania zewnętrznego zdarzenia. Nie istnieje też mechanizm wątków, które mogłyby równocześnie modyfikować wspólną pamięć. W czasie gdy mechanizm skryptowy jest zajęty, wstrzymana zostaje obsługa wszystkich zdarzeń, czasomierzy, żądań nawigacji w stronach itp. W większości przypadków cała przeglądarka, a przynajmniej moduł rysujący HTML, przestaje odpowiadać. Obsługa zdarzeń zapisanych w kolejce wznawiana jest dopiero po zakończeniu wykonywania skryptu, gdy mechanizm skryptowy przechodzi do stanu wstrzymania. Od tego momentu ponownie można uruchomić kod JavaScript.

Co więcej, język JavaScript nie udostępnia funkcji `sleep(…)` lub `pause(…)`, która tymczasowo zwalniałaby procesor i po pewnym czasie wznawiałaby wykonywanie kodu od tego samego miejsca. Jeżeli programista chce opóźnić wykonywanie wykonania skryptu, musi zarejestrować czasomierz, który później ponownie uruchomi wykonywanie kodu. Wykonywanie to musi się jednak rozpocząć na początku funkcji obsługującej zdarzenie czasomierza (albo na początku anonimowej funkcji wprowadzonej w czasie konfigurowania czasomierza). Takie warunki można uznać za irytujące, ale trzeba przyznać, że prawie do zera redukują one ryzyko wystąpienia warunków wyścigu (ang. race condition) w uruchomionym kodzie.

UWAGA *W tym synchronicznym modelu wykonywania kodu istnieje kilka raczej przypadkowych luk. Jedną z nich jest możliwość wykonania kodu w czasie, gdy wykonanie innego bloku kodu zostaje wstrzymane w wyniku wywołania funkcji `alert(…)` lub `showModalDialog(…)`. Trzeba jednak pamiętać, że takie przypadki brzegowe nie wpływają zbyt często na sposób wykonywania kodu.*

Taki blokujący przeglądarkę sposób wykonywania pętli w kodzie JavaScript wymusza na samych przeglądarkach wprowadzanie pewnych mechanizmów łagodzących. Mechanizmy te opiszę dokładniej w rozdziale 14. Na razie wystarczy

powiedzieć, że wiążą się one z innymi, i to dość nieoczekiwanymi konsekwencjami. Wykonanie każdej nieskończonej pętli może zostać zakończone w sposób zbliżony do rzucenia nieobsłużonego wyjątku. Mechanizm skryptowy wróci wtedy do stanu wstrzymania, ale kod nadal będzie mógł być wykonywany, ponieważ wszystkie czasomierze i zdarzenia pozostaną aktywne.

Możliwość nieoczekiwanego przerwania dłużej działającego kodu, który przez autora traktowany jest jako zawsze wykonujący się w całości, może zostać wykorzystana przez atakującego do wprowadzenia aplikacji w stan niezdefiniowany. To nie wszystko, inna podobna konsekwencja takiej semantyki powinna być widoczna i zrozumiała po przeczytaniu punktu „JSON i inne metody serializacji danych".

Możliwości badania kodu i obiektów

Język JavaScript udostępnia podstawowe metody badania dekompilowanego kodu źródłowego dowolnych niestandardowych funkcji. Wystarczy wywołać metodę `toString()` lub `toSource()` na rzecz dowolnej funkcji, którą chcemy skontrolować. Poza tym możliwości skontrolowania przepływu działającego programu są bardzo ograniczone. Aplikacje mogą próbować wykorzystywać dostęp do zapisanej w pamięci reprezentacji głównego dokumentu i poszukiwać w niej bloków `<script>`, ale w ten sposób nie da się przejrzeć zdalnie ładowanego lub generowanego kodu. Za pomocą niestandardowej właściwości `caller` można też uzyskać wgląd w stos wywołań działającego skryptu, ale nie da się w ten sposób określić aktualnie wykonywanego wiersza kodu ani jego części, która zostanie wykonana w następnej kolejności.

Dynamiczne tworzenie nowego kodu JavaScript jest jedną z najważniejszych cech tego języka. Istnieje tu możliwość nakazania mechanizmowi skryptowemu, aby synchronicznie interpretował ciągi znaków przekazywanych poprzez wbudowaną funkcję `eval(…)`. Na przykład poniższa instrukcja spowoduje wyświetlenie okienka z komunikatem:

```
eval("alert(\"Cześć, mamo!\")")
```

Błędy składniowe wykryte w tekście przekazanym do funkcji `eval(…)` spowodują, że funkcja ta rzuci wyjątek. Jeżeli parsowanie się powiedzie, to wszystkie nieobsłużone wyjątki rzucone przez interpretowany kod również zostaną przekazane do funkcji wywołującej. W przypadku gdy interpretowany kod zostanie bezbłędnie przygotowany i wykonany, to jako wartość zwracana przez funkcję `eval(…)` potraktowana zostanie wartość ostatniej instrukcji znajdującej się w przekazanym jej kodzie.

Obok opisanej funkcji można też wykorzystać mechanizmy działające na poziomie przeglądarki, aby z ich pomocą zaplanować opóźnione parsowanie i wykonywanie nowych bloków kodu JavaScript, które to operacje zostaną wykonane, gdy tylko mechanizm skryptowy powróci do stanu wstrzymania. Przykładami takich mechanizmów mogą być czasomierze (`setTimeout`, `setInterval`), funkcje obsługi zdarzeń (`onclick`, `onload` itd.) oraz interfejsy umożliwiające dostęp do parsera HTML (`innerHTML`, `document.write(…)` i inne).

Co prawda możliwości badania kodu pozostawiają wiele do życzenia, ale już możliwości introspekcji obiektów działającego kodu zostały w języku JavaScript dość mocno rozbudowane. Aplikacje mogą wyliczać niemal wszystkie metody lub właściwości obiektu za pomocą prostych iteratorów `for … in` oraz `for each … in`, a także używać takich operatorów jak `typeof`, `instanceof` lub „jest dokładnie równy" (`===`) i właściwości `length`. W ten sposób programy mogą poznać szczegóły wszystkich znalezionych przez siebie elementów.

Wszystkie te mechanizmy sprawiają, że skrypty działające w tym samym kontekście nie mają prawie żadnych możliwości ukrycia przed sobą jakichkolwiek danych. Co więcej, utrudniają one ukrywanie danych nawet pomiędzy różnymi kontekstami dokumentów. Z tym problemem twórcy przeglądarek walczyli przez bardzo długi czas, a jak dowiemy się w rozdziale 11., nadal nie udało im się całkowicie go rozwiązać.

Modyfikowanie środowiska uruchomieniowego

Mimo względnej prostoty języka JavaScript wykonywane w nim skrypty mają wiele możliwości, żeby mocno wpłynąć na zachowanie własnej piaskownicy tworzonej przez język. W rzadkich przypadkach takie zachowania mogą wpływać nawet na inne dokumenty.

Pokrywanie wbudowanych elementów

Jednym z najbardziej zaskakujących narzędzi, jakimi mogą posługiwać się złośliwe skrypty, jest możliwość usunięcia, pokrycia albo przesłonięcia większości wbudowanych w język JavaScript funkcji oraz praktycznie wszystkich metod wejścia-wyjścia udostępnianych przez przeglądarkę. Proszę przyjrzeć się na przykład działaniu poniższego kodu:

```
// To przypisanie nie spowoduje powstania błędu.
eval = alert;

// To wywołanie spowoduje nieoczekiwane wyświetlenie okna z komunikatem.
eval("Cześć, mamo!");
```

A to dopiero początek zabawy. W Chrome, Safari i Operze możliwe jest całkowite usunięcie funkcji `eval(…)`. Wystarczy skorzystać z operatora `delete`. Dziwne jest też to, że próba wykonania tej samej operacji w Firefoksie spowoduje przywrócenie oryginalnej, wbudowanej funkcji i usunięcie wszystkich ewentualnych pokryć. Z kolei w Internet Explorerze próba usunięcia tej funkcji spowoduje opóźnione rzucenie wyjątku, który w tym miejscu raczej nie ma żadnego konkretnego celu.

Zgodnie z tą zasadą niemal wszystkie obiekty, włącznie z wbudowanymi typami `String` lub `Array`, są tylko prototypami, które można dowolnie modyfikować. Taki prototyp jest obiektem nadrzędnym, od którego wszystkie istniejące i przyszłe obiekty wywodzą swoje metody i właściwości. Jest to zgrubne odwzorowanie mechanizmu dziedziczenia klas istniejącego w bardziej rozbudowanych językach programowania.

Możliwość modyfikowania prototypów obiektów może powodować mało intuicyjne zachowania nowo tworzonych obiektów, o czym można się przekonać w poniższym przykładzie:

```
Number.prototype.toString = function() {
  return "Mam cię!";
};

// To wywołanie wyświetli napis "Mam cię!", a nie "42":
alert(new Number(42));
```

Funkcje get i set

Jeszcze bardziej interesującą cechą modelu obiektów w używanym powszechnie dialekcie języka JavaScript są funkcje `get` i `set`, które umożliwiają zdefiniowanie własnego kodu obsługującego odczytywanie i zapisywanie wartości do właściwości danego obiektu. Co prawda nie jest to mechanizm aż tak rozbudowany jak przeciążanie operatorów w języku C++, ale można go wykorzystać do zmuszenia istniejących obiektów lub prototypów do zachowań dalekich od standardu. W poniższym wycinku kodu operacje przypisania i odczytania wartości do właściwości zostały w prosty sposób podmienione:

```
var evil_object = {
  set foo() { alert("Mam cię!"); },
  get foo() { return 2; }
};

// To przypisanie spowoduje wyświetlenie komunikatu "Mam cię!".
// Poza tym nie stanie się nic więcej.
evil_object.foo = 1;

// To porównanie na pewno się nie uda.
if (evil_object.foo != 1) alert("Co się dzieje?!");
```

UWAGA *Funkcje `set` i `get` powstały początkowo jako rozszerzenie języka, ale zostały włączone do standardu ECMAScript wydanie 5. Funkcje te dostępne są we wszystkich nowoczesnych przeglądarkach (z wyjątkiem Internet Explorera w wersji 6 i 7).*

Efekty potencjalnych przypadków użycia języka

W wyniku zastosowania technik opisywanych w poprzednich dwóch podpunktach skrypt uruchomiony w danym kontekście, który został zmieniony przez niezaufane treści, nie ma żadnej pewnej metody na sprawdzenie swojego środowiska ani na wykonanie działań korygujących. Nie można zaufać nawet zachowaniu prostych wyrażeń warunkowych oraz pętli. Proponowane rozszerzenia języka najprawdopodobniej spowodują jeszcze większe skomplikowanie tej sytuacji. Na przykład

nieudana propozycja z 4. wydania języka ECMAScript zawierała opis rozbudowanego mechanizmu przeciążania operatorów i pomysł ten prawdopodobnie będzie powracał w przyszłości.

Bardzo ciekawe jest też to, że podjęte decyzje projektowe utrudniają teraz możliwości sprawdzania kontekstu wykonania spoza piaskownicy tworzonej dla danej strony. Na przykład ślepe zaufanie w działanie obiektu `location` w potencjalnie wrogim dokumencie doprowadziło już do wielu problemów z bezpieczeństwem, związanych z wtyczkami do przeglądarek, rozszerzeniami tworzonymi w języku JavaScript oraz w niektórych klasach aplikacji WWW działających po stronie klienta. Wszystkie te problemy spowodowały ostatecznie powstanie różnorakich obejść na poziomie przeglądarki, które mają choć częściowo chronić ten szczególny obiekt przed sabotażem. Niestety większość pozostałych obiektów nadal pozostaje niechroniona.

UWAGA *Możliwości modyfikowania własnego kontekstu wykonania zostały mocno ograniczone w trybie „strict" zdefiniowanym w 5. wydaniu specyfikacji ECMAScript. Niestety na razie ten tryb nie jest w pełni obsługiwany przez żadną przeglądarkę, a na dodatek ma być mechanizmem dobrowolnym, z którego nie każda strona będzie musiała korzystać.*

JSON i inne metody serializacji danych

Bardzo ważną strukturą składniową używaną w języku JavaScript jest bardzo kompaktowy i wygodny mechanizm serializacji danych, znany pod nazwą JavaScript Object Notation (notacja obiektów języka JavaScript) albo po prostu JSON (RFC 4627[4]). Ten format danych wykorzystuje możliwość przeciążenia znaczenia symbolu otwierającego nawiasu klamrowego ({). Nawias ten używany jest normalnie do oznaczenia początku zagnieżdżonego bloku kodu, co oznacza, że otwiera on nową instrukcję. Jeżeli jednak nawias ten pojawi się jako część wyrażenia, to traktowany jest jako początek zserializowanego obiektu. Poniższy przykład przedstawia prawidłowe zastosowanie takiej składni do wyświetlenia prostego komunikatu:

```
var impromptu_object = {
  "given_name" : "Jan",
  "family_name" : "Kowalski",
  "lucky_numbers" : [ 11630, 12067, 12407, 12887 ]
};

// W okienku pojawi się napis "Jan".
alert(impromptu_object.given_name);
```

Przeciążenie operatora otwierającego nawiasu klamrowego oznacza, że bloki JSON nie zostaną właściwie rozpoznane, jeżeli spróbujemy ich użyć jako osobnych instrukcji. Oznacza to, że zachowują się one inaczej niż liczby, ciągi znaków i tablice, których serializacja nie ma takich dwuznaczności. Może się to wydawać mało zna-

cząće, ale jest to całkiem spora zaleta. Oznacza to, że niemożliwe staje się dołączanie do stron odpowiedzi zgodnych z tą składnią, przesyłanych przez serwer z innej domeny, z wykorzystaniem znaczników `<script src=…>`*. Przedstawiony niżej listing spowoduje powstanie błędu składniowego z powodu wykrycia nieprawidłowo umiejscowionego znaku cudzysłowu (❶) w miejscu, które interpreter traktuje jako etykietę kodu†. Taki błąd nie będzie miał tutaj żadnych efektów ubocznych.

```
<script>
{
❶   "given_name" : "Jan",
  "family_name" : "Kowalski",
  "lucky_numbers" : [ 11630, 12067, 12407, 12887 ]
};
</script>
```

UWAGA *Brak możliwości dołączenia danych w formacie JSON za pomocą znaczników `<script src=…>` jest bardzo interesującą właściwością, ale nie jest to zabezpieczenie doskonałe. Wystarczy umieścić odpowiedź serwera w nawiasach okrągłych lub kwadratowych albo usunąć cudzysłowy otaczające nazwy etykiet, a tak poprawioną składnię będzie można wykonać jako osobny blok kodu, co może mieć już poważne efekty uboczne. Biorąc pod uwagę szybko rozwijającą się składnię języka JavaScript, nie można oczekiwać, że ten szczególny układ kodu będzie już zawsze powodować błędy parsowania. W wielu mało istotnych zastosowaniach ten poziom bezpieczeństwa z pewnością będzie wystarczający i można go traktować jako bardzo prosty mechanizm zabezpieczający.*

Dane w formacie JSON otrzymane przez dowolny kanał, taki jak na przykład żądanie `XMLHttpRequest`, można szybko i łatwo przekształcić w obiekty zapisane w pamięci. Wystarczy wywołać funkcję `JSON.parse(…)` dostępną we wszystkich popularnych przeglądarkach (z wyjątkiem Internet Explorera). Niestety w celu utrzymania zgodności z tą przeglądarką, a czasami po prostu z przyzwyczajenia, wielu programistów używa równie prostego wywołania, które jednak jest zdecydowanie bardziej niebezpieczne:

```
var parsed_object = eval("(" + json_text + ")");
```

* W przeciwieństwie do innych sposobów włączania treści do skryptów (takich jak `XMLHTTPRequest`) znaczniki `<script src=…>` nie podlegają ograniczeniom odwołań międzydomenowych, o których mówić będę w rozdziale 9. Oznacza to, że mechanizm ten stanowi pewne ryzyko dla bezpieczeństwa, w przypadku gdy pośrednie dane uwierzytelniające (na przykład ciasteczka) używane są przez serwer do dynamicznego generowania kodu JavaScript dla konkretnego użytkownika. Ta klasa podatności opisywana jest skrótem XSSI — *cross-site script inclusion*.

† Zaskakującą funkcją języka JavaScript jest obsługa etykiet znanych z języka C, takich jak `moja_etykieta: alert("Cześć, mamo!");`. Jest to o tyle interesujące, że język nie obsługuje instrukcji `goto`, a to oznacza, iż w większości przypadków takich etykiet po prostu nie da się wykorzystać.

Problem polega na tym, że funkcja `eval(…)` używana tutaj do wyliczania „wartości" danych w formacie JSON pozwala na przekazanie jej nie tylko zserializowanych danych, ale i pełnoprawnego kodu JavaScript, a to może mieć nieoczekiwane, globalne konsekwencje. Na przykład wywołanie funkcji, umieszczone w spreparowanej odpowiedzi JSON, zostanie oczywiście wykonane:

```
{ "given_name": alert("Cześć, mamo!") }
```

Takie zachowanie sprawia, że twórca aplikacji musi dodatkowo zadbać o to, żeby przyjmować dane JSON tylko z zaufanych źródeł, a wszystkie dane otrzymane od serwera poddawać dodatkowemu procesowi oznaczania znaków. Jak można się spodziewać, niedociągnięcia w tym zakresie powodowały już wiele błędów w bezpieczeństwie aplikacji WWW.

UWAGA *Kłopoty z prawidłową obsługą funkcji `eval(…)` widoczne są nawet w samej specyfikacji formatu JSON (RFC 4627). Rzekomo bezpieczna implementacja parsera dołączona do tego dokumentu umożliwia spreparowanym odpowiedziom w formacie JSON na dowolne inkrementowanie i dekrementowanie zmiennych programu, których nazwy składają się z liter* a, e, f, l, n, r, s, t, u *oraz cyfr. To wystarczy, żeby złożyć takie słowo jak „unsafe" (niebezpiecznie) oraz około tysiąca innych angielskich słów. Wadliwe wyrażenie regularne prezentowane w tym dokumencie RFC pojawia się w wielu miejscach w internecie i z pewnością szybko nie zniknie.*

Łatwość wykorzystania formatu JSON powoduje, że jest on powszechnie używany przez nowoczesne aplikacje WWW. Jedynymi rywalami tego formatu są mniej bezpieczne sposoby serializacji ciągów znaków i tablic oraz format JSONP*. Wszystkie te metody nie są jednak zgodne z funkcją `JSON.parse(…)`, a to oznacza, że wymuszają użycie niebezpiecznej funkcji `eval(…)` w celu konwersji danych do postaci pozwalającej na użycie w programie. Inną właściwością tych formatów jest to, że w przeciwieństwie do formatu JSON dane załadowane z zewnętrznej strony za pomocą znacznika `<script src=…>` zostaną prawidłowo przetworzone. W rzadkich przypadkach można to uznać za zaletę, ale zwykle należy traktować to jako niepotrzebne ryzyko. Jeżeli się zastanowić, to samo załadowanie danych serializowanej tablicy za pomocą znacznika `<script>` nie ma poważnych efektów ubocznych, ale atakujący mógłby zmodyfikować funkcję `set` przypisaną do prototypu `Array`, aby w ten sposób odczytywać wpisywane do niej dane. Powszechnie stosowane zabezpieczenie polegające na doklejeniu do początku odpowiedzi pętli `while(1);`, która ma zablokować taki atak, może mieć jednak interesujące skutki uboczne. Proszę sobie przypomnieć o możliwości przerwania nieskończonej pętli, jaką daje nam język JavaScript.

* Skrót JSONP oznacza „JSON with padding" czyli „JSON w opakowaniu". Jest to format standardowej serializacji JSON opakowanej w uzupełniający kod, który zmienia całość w niezależną instrukcję JavaScript. Wśród typowych przykładów można tu wymienić wywołania funkcji (np. `callback_function({…dane JSON…})`) albo przypisania wartości do zmiennych (`var return_value = {…dane JSON…}`).

E4X i inne rozszerzenia składni języka

Język JavaScript ewoluuje bardzo szybko, podobnie jak język HTML. Niektóre wprowadzane do niego zmiany były na tyle radykalne, że mogły powodować przekształcenie formatów tekstowych do tej pory odrzucanych przez parsery w prawidłowy kod JavaScript. To z kolei mogło prowadzić do nieoczekiwanego ujawniania danych, szczególnie w połączeniu z rozbudowanymi możliwościami wglądu w kod i obsługiwane przez niego obiekty, o których mówiłem już w tym rozdziale. Nie wspominając już o możliwości wykorzystania znacznika `<script src=…>` do załadowania kodu z zewnętrznych domen.

Jednym z najbardziej widocznych przykładów tego trendu jest język *ECMSScript for XML* (E4X)[5], będący zupełnie niepotrzebną, choć elegancką próbą włączenia składni XML do języka JavaScript i traktowania jej jako alternatywy dla formatu JSON. W rozwiązaniach zgodnych z E4X, takich jak Firefox, poniższe wycinki kodu są sobie mniej więcej równoważne:

```
// Normalna serializacja obiektów
var my_object = { "user": {
                    "given_name": "Jan",
                    "family_name": "Kowalski",
                    "id": make_up_value()
                  } };

// Serializacja E4X
var my_object = <user>
                    <given_name>Jan</given_name>
                    <family_name>Kowalski</family_name>
                    <id>{ make_up_value() }</id>
                  </user>;
```

Nieoczekiwaną konsekwencją obsługi rozszerzenia E4X jest to, że każdy prawidłowo zbudowany dokument XML może zostać załadowany za pomocą znacznika `<script src=…>` i przetworzony w ramach bloku wyrażenia traktowanego jak instrukcja. Co więcej, jeżeli atakujący zdoła strategicznie umieścić w dokumencie znaki nawiasów klamrowych ({ i }) albo zmodyfikować funkcję `set` prototypów odpowiednich obiektów, to może mu się udać pobrać tekst wyświetlany w atakowanym dokumencie. Takie niebezpieczeństwo prezentowane jest w poniższym przykładzie:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  ...
  { wykradnij_dane(                          <- ciąg znaków wstawiony przez atakującego
    ...
    <span>Tajne informacje użytkownika</span>
    ...
  ) }                                        <- ciąg znaków wstawiony przez atakującego
  ...
</html>
```

Trzeba przyznać, że po kilku latach istnienia tego błędu w Firefoksie jego twórcy zdecydowali się zakazać instrukcji E4X, które by obejmowały całość przetwarzanego skryptu, co częściowo załatało ten problem. Mimo to wyraźnie widać, że język jest w ciągłym ruchu, dlatego trudno jest mieć całkowitą pewność, iż odpowiedzi JSON będą dobrym zabezpieczeniem przed międzydomenowym dołączaniem skryptów. W momencie gdy pojawi się trzecie znaczenie otwierającego nawiasu klamrowego albo na początku nazwy etykiety, dopuszczony zostanie znak cudzysłowu, więc bezpieczeństwo tej metody wymiany danych między serwerem a klientem zostanie mocno zredukowane. Dobrze jest się na to przygotować.

Standardowa hierarchia obiektów

Środowisko wykonawcze języka JavaScript zbudowane jest wokół niejawnego obiektu podstawowego, który używany jest jako domyślna przestrzeń nazw dla wszystkich zmiennych i funkcji globalnych tworzonych przez program. Przestrzeń ta wypełniana jest hierarchią funkcji, które implementują operacje wejścia i wyjścia realizowane w środowisku przeglądarki, oraz kilkoma dodatkami wynikającymi ze specyfiki języka. Do wspomnianych operacji należą funkcje manipulowania oknami przeglądarki (`open(…)`, `close(…)`, `moveTo(…)`, `resizeTo(…)`, `focus(…)`, `blur(…)` itp.), konfigurowania czasomierzy języka JavaScript (`setTimeout(…)`, `setInterval(…)` itp.), wyświetlania okienek dialogowych interfejsu użytkownika (`alert(…)`, `prompt(…)`, `print(…)`) oraz wykonywania wielu innych działań przygotowanych przez producenta przeglądarki. Nierzadko chodzi tu o dość ryzykowne operacje, takie jak dostęp do schowka systemowego, tworzenie zakładek albo modyfikowanie strony startowej.

Ten podstawowy obiekt udostępnia też referencje głównych obiektów przypisanych do danego kontekstu, do których można zaliczyć obiekt nadrzędnej ramki (`parent`), głównego dokumentu w aktualnym oknie przeglądarki (`top`) oraz wszystkich ramek zdefiniowanych w tym dokumencie (`frames[]`). Znajdziemy w nim nawet kilka dodatkowych referencji do aktualnego obiektu głównego, takich jak `windows` i `self`. We wszystkich przeglądarkach (z wyjątkiem Firefoksa) do tej przestrzeni nazw dołączane są też elementy, którym przypisano parametry `id` lub `name`, dzięki czemu możliwe jest zastosowanie takich konstrukcji:

```
<img id="hello" src="http://www.example.com/">
...

<script>
  alert(hello.src);
</script>
```

Na szczęście w przypadku wystąpienia konfliktów nazw ze zmiennymi języka JavaScript lub elementami wbudowanymi w język elementy z parametrem `id` nie są traktowane priorytetowo. Dzięki temu unika się możliwych interferencji między prawidłowo zbudowanym kodem HTML dostarczonym przez użytkownika a znajdującymi się w dokumencie skryptami.

Pozostała część tej wysokopoziomowej hierarchii składa się przede wszystkim z kilku obiektów-dzieci tematycznie grupujących różne API udostępniane przez przeglądarkę:

Obiekt `location`

Jest to kolekcja właściwości i metod pozwalających programowi na odczytywanie adresu URL aktualnego dokumentu oraz inicjowanie przejścia do nowego. W większości przypadków ta ostatnia operacja źle się kończy dla wywołującego dokumentu, ponieważ jego kontekst jest niszczony i krótko potem zastępowany nowym. Wyjątkiem od tej zasady jest aktualizowanie samego identyfikatora fragmentu (`location.hash`), co zostało dokładniej opisane w rozdziale 2.

Trzeba pamiętać o tym, że podczas konstruowania ciągów znaków zawierających dane z obiektu `location` (kod HTML, a w szczególności kod JavaScript) nie wolno zakładać, iż niebezpieczne znaki zostaną w tych danych prawidłowo oznaczone. Internet Explorer nie zmienia znaków nawiasów ostrych na właściwości `location.search` (przechowuje ona tekst zapytania). Z drugiej strony Chrome prawidłowo je oznacza, ale za to pomija znaki cudzysłowu i lewego ukośnika. Większość przeglądarek w ogóle nie stosuje oznaczania znaków w identyfikatorze fragmentu.

Obiekt `history`

Ten obiekt udostępnia kilka rzadko używanych metod pozwalających na korzystanie z historii danego okna przeglądarki w sposób zbliżony do klikania przycisków *Wstecz* i *Naprzód*. Nie ma tu możliwości bezpośredniego skontrolowania poprzednio odwiedzonych adresów URL, a jedyną opcją jest ślepe nawigowanie w historii stron przez określenie liczbowego przesunięcia, na przykład za pomocą wywołania `history.go(-2)`. W rozdziale 17. omówię też kilka nowych funkcji dodanych ostatnio do tego obiektu.

Obiekt `screen`

Proste API pozwalające określić wielkość ekranu i okna przeglądarki, rozdzielczość monitora, głębię kolorów i inne parametry. Udostępniane jest ono witrynom, aby mogły jak najlepiej dopasować swój wygląd do konkretnego urządzenia wyświetlającego.

Obiekt `navigator`

Interfejs umożliwiający sprawdzenie wersji przeglądarki, systemu operacyjnego oraz listy zainstalowanych w przeglądarce wtyczek.

Obiekt `document`

To zdecydowanie najbardziej złożony jeden z obiektów głównych. Można go traktować jako wejście do modelu DOM[6] aktualnej strony (na temat tego modelu mówić będę w następnym punkcie). Dostępnych jest w nim też kilka funkcji zupełnie niezwiązanych ze strukturą dokumentu, które dodane zostały

w wyniku niezależnych decyzji projektowych. Jako przykład można tu podać właściwość `document.cookie` pozwalającą manipulować plikami cookie, funkcję `document.write(…)` przeznaczoną do dopisywania kodu HTML do aktualnej strony albo funkcję `document.execCommand(…)` umożliwiającą edycję dokumentu w trybie WYSIWYG.

UWAGA *Co ciekawe, informacje dostępne w obiektach `navigator` i `screen` całkowicie wystarczają do pobrania unikalnego „odcisku palca" przeglądarki wielu użytkowników. Ta dobrze znana właściwość tych obiektów prezentowana jest obrazowo na stronie projektu* Panopticlick *prowadzonego przez fundację Electronic Frontier Foundation (https://panopticlick.eff.org).*

Kilka innych obiektów zdefiniowanych w języku JavaScript udostępnia proste funkcje obsługi ciągów znaków lub funkcje arytmetyczne. Na przykład funkcja `math.random()` realizuje niebezpieczny, przewidywalny generator liczb pseudolosowych. Niestety bezpieczna alternatywa PRNG nie jest w tej chwili dostępna w większości przeglądarek*. Z kolei za pomocą funkcji `String.fromCharCode()` można przekształcać wartości liczbowe w ciągi znaków Unicode. W uprzywilejowanych kontekstach wykonania (niedostępnych dla normalnych aplikacji WWW) pojawia się też całkiem sporo obiektów przeznaczonych do realizacji konkretnych działań.

UWAGA *Korzystając z obiektów wbudowanych w przeglądarkę, trzeba pamiętać, że choć język JavaScript nie korzysta z ciągów znaków ASCIZ, których koniec oznaczany jest znakiem NUL, to sama przeglądarka czasami ich używa (ponieważ została napisana w języku C lub C++). Oznacza to, że przypisywanie właściwościom modelu DOM ciągów znaków, które zawierają znaki NUL, albo przekazywanie takich ciągów znaków do funkcji udostępnianych przez przeglądarki może powodować nieprzewidywalne i niejednoznaczne zachowania. Niemal wszystkie przeglądarki obcinają ciągi znaków przypisywane do właściwości obiektu `location` w miejscu pierwszego wystąpienia znaku NUL, ale tylko niektóre z nich postępują podobnie w przypadku właściwości `innerHTML` obiektów modeli DOM.*

Model DOM

Model DOM (*Document Object Model*) danego dokumentu dostępny jest poprzez obiekt `document`, w którym znajduje się reprezentacja dokumentu HTML przygotowana przez parser. W wyniku jego pracy tworzone jest drzewo obiektów zawierające wszystkie elementy HTML znajdujące się na danej stronie, uzupełnione o metody i właściwości odpowiednie dla danego rodzaju znacznika oraz powiązane z nimi dane CSS. To właśnie taka reprezentacja dokumentu, a nie jego źródłowy

* Do Chrome dodano ostatnio nowe API `window.crypto.getRandomValues(…)`, a w Firefoksie pojawiło się API `window.crypto.random(…)`, które na razie nie realizuje żadnych funkcji.

kod HTML, używana jest przez przeglądarki do rysowania i aktualizowania wyświetlanego dokumentu.

Język JavaScript może bardzo łatwo uzyskać dostęp do struktur DOM i korzystać z nich tak jak z dowolnych innych obiektów. Na przykład poniższy wycinek kodu przejdzie do piątego znacznika umieszczonego w bloku `<body>` aktualnego dokumentu, wyszuka pierwszy zagnieżdżony w nim znacznik i za pomocą stylu CSS nada mu kolor czerwony:

```
document.body.children[4].children[0].style.color = "red";
```

Chcąc uniknąć konieczności przedzierania się przez całe drzewo elementów w celu dotarcia do szczególnie głęboko umieszczonego w nim elementu, możemy skorzystać z udostępnianych przez przeglądarki funkcji wyszukiwania, takich jak `getElementById(…)` i `getElementsByTagname(…)`. Można też wykorzystać różne mechanizmy grupujące takie jak `frames[]`, `images[]` lub `forms[]`. Wszystkie te rozwiązania umożliwiają stosowanie takiego kodu jak przedstawione poniżej instrukcje odwołujące się do pewnego elementu bez uwzględniania jego dokładnej pozycji w hierarchii obiektów:

```
document.getElementsByTagName("input")[2].value = "Cześć, mamo!";
document.images[7].src = "/example.jpg";
```

Ze względu na konieczność utrzymania wstecznej zgodności bezpośrednio w przestrzeni nazw `document` dostępne są również nazwy niektórych elementów HTML (`<img>`, `<form>`, `<embed>`, `<object>` i `<applet>`), co pozwala na stosowanie takiego kodu jak w poniższym przykładzie:

```
<img name="hello" src="http://www.example.com/">

<script>
  alert(document.hello.src);
</script>
```

W przeciwieństwie do całkiem rozsądnego odwzorowania właściwości `name` i `id` w globalnej przestrzeni nazw (mówiłem o tym w poprzednim punkcie), takie dodatki do obiektu `document` mogą tylko przesłonić wbudowane w ten obiekt funkcje i właściwości typu `getElementById` lub `body`. Z tego powodu tworzenie nazw znaczników na podstawie danych pobranych od użytkownika, choćby w celu dynamicznego konstruowania formularzy, należy uznać za bardzo niebezpieczne.

Podstawową funkcją węzłów modelu DOM jest tworzenie abstrakcyjnej reprezentacji dokumentu, ale wiele z nich udostępnia dodatkowo takie właściwości jak `innerHTML` i `outerHTML`, które umożliwiają odczytanie części drzewa dokumentu w postaci prawidłowo zbudowanego ciągu znaków z kodem HTML. Co ciekawe, tym samym właściwościom można przypisywać wartości, aby w ten sposób podmienić dowolną część drzewa DOM poprzez parsowanie wycinka kodu HTML

przygotowanego przez skrypt. Oznacza to, że w kodzie JavaScript mogą znaleźć się takie instrukcje:

```
document.getElementById("output").innerHTML = "<b>Cześć, mamo!</b>";
```

Każdy przypisywany do właściwości `innerHTML` ciąg znaków musi zawierać prawidłowo zbudowany i niepodzielny blok kodu HTML, który nie wpływa na hierarchię dokumentu poza podmienianym fragmentem. Jeżeli wprowadzany kod nie spełnia tych warunków, to przed dokonaniem wstawienia zostanie on przymusowo przekształcony do spełniającej je postaci. Oznacza to, że poniższy przykład nie zadziała zgodnie z naszym oczekiwaniem, czyli nie wyświetli pogrubionego tekstu „Cześć, mamo!" i nie spowoduje, iż pozostała część dokumentu będzie wyświetlana kursywą.

```
some_element.innerHTML = "<b>Cześć";
some_element.innerHTML += " mamo!</b><i>";
```

W rzeczywistości obie operacje przypisania zostaną przetworzone osobno i skorygowane niezależnie od siebie, w wyniku czego uzyskamy efekt równoważny poniższej instrukcji:

```
some_element.innerHTML = "<b>Cześć</b> mamo!<i></i>";
```

Należy tu nadmienić, że mechanizm właściwości `innerHTML` powinien być używany z zachowaniem ostrożności. Oprócz tego, że pozwala on na wstrzykiwanie kodu HTML do strony, to w przypadku niewłaściwego oznaczenia niebezpiecznych znaków można się spodziewać dziwnych zachowań przeglądarek, ponieważ stosowane w nich algorytmy serializacji DOM-do-HTML są dalekie od doskonałości. Poniżej prezentuję niedawny (teraz już poprawiony) błąd w przeglądarkach z rodziny WebKit[7]:

```
<textarea>
  &lt;/textarea&gt;&lt;script&gt;alert(1)&lt;/script&gt;
</textarea>
```

Z powodu nieporozumień związanych z semantyką znacznika `<textarea>` ten pozornie jednoznaczny zapis po umieszczeniu w drzewie DOM i odczytaniu za pomocą właściwości `innerHTML` zostanie nieprawidłowo zwrócony jako:

```
<textarea>
  </textarea><script>alert(1)</script>
</textarea>
```

W takiej sytuacji nawet wykonanie niczego niezmieniającego przypisania do tej właściwości (takiego jak `jakiś_element.innerHTML += ""`) spowoduje nieoczekiwane wstrzyknięcie skryptu do kodu strony. Poza tym programiści pracujący w Internet Explorerze nad kodem właściwości `innerHTML` nie wiedzieli, że w języku MSHTML znaki lewego apostrofu (`) są równoważne cudzysłowom, a przez to przeglądarka rozpoznaje je nieprawidłowo w tym kontekście. Okazuje się, że Internet Explorer serializuje poniższy kod:

```
<img src="test.jpg" alt="``onload=alert(1)">
```

do postaci:

```
<img src=test.jpg alt=``onload=alert(1)>
```

Nawet bez takich drobnych błędów sytuacja właściwości `innerHTML` nie jest ciekawa. W sekcji 10.3 aktualnego szkicu specyfikacji HTML5 potwierdza, że struktury DOM tworzone za pomocą skryptów nie dają się prawidłowo serializować do tekstu HTML i w związku z tym nie wymaga się od przeglądarek specjalnych zachowań w tym zakresie. *Caveat emptor!*

Dostęp do innych dokumentów

Skrypty mogą też uzyskać dostęp do uchwytów obiektów wskazujących główne hierarchie obiektów innych kontekstów skryptów. Na przykład domyślnie każdy kontekst może swobodnie odwoływać się do obiektów `parent`, `top`, `opener` oraz `frames[]`, które dostępne są w obiekcie najwyższego poziomu. Wywołanie funkcji `window.open(…)` spowoduje otworzenie nowego okna i zwróci referencję do niego. Podobnie zachowa się próba wyszukania istniejącego już okna o określonej nazwie, wykonana za pomocą poniższej instrukcji:

```
var window_handle = window.open("", "nazwa_okna");
```

Jeżeli dany program uzyska uchwyt wskazujący kontekst innego okna, może podjąć próby interakcji z tym kontekstem, co wiąże się z kolejnymi mechanizmami kontroli bezpieczeństwa, o których będę mówił w rozdziale 9. Przykład prostej interakcji między dokumentami może wyglądać tak:

```
top.location.path = "/new_path.html";
```

albo tak:

```
frames[2].document.getElementById("output").innerHTML = "Cześć, mamo!";
```

Interakcja między niezwiązanymi ze sobą dokumentami na poziomie skryptu JavaScript nie jest możliwa, jeżeli skrypt nie ma dostępu do prawidłowego uchwytu kontekstu. Szczególnie nie istnieje żaden sposób na wyszukanie nienazwanych okien przeglądarki otwartych w niezależnych przepływach nawigacyjnych. Nie jest to możliwe do czasu, aż któraś z odwiedzonych stron nada oknu jakąś nazwę (umożliwia to właściwość `window.name`).

Kodowanie znaków w skryptach

Implementacje języka JavaScript obsługują kilka metod oznaczania znaków za pomocą lewego ukośnika, które można wykorzystać do oznaczenia znaków cudzysłowu, znaczników HTML oraz innych problematycznych elementów umieszczanych w tekście. Poniżej przedstawiam wszystkie te metody:

- Skrócona notacja języka C umożliwiająca zapisanie pewnych znaków sterujących. Na przykład `\b` oznacza znak Backspace, `\t` to tabulacja, `\v` to tabulacja pozioma, `\f` to wysunięcie strony, `\r` to znak powrotu karetki, a `\n` to wysunięcie wiersza. Dokładnie ten zestaw kodów rozpoznawany jest zarówno przez specyfikację ECMAScript, jak i dokument RFC dotyczący formatu JSON.
- Ośmiobitowe, ósemkowe, trzycyfrowe kody bez żadnego przedrostka (na przykład `\145` zamiast litery *e*). Ta inspirowana językiem C składnia nie została włączona do specyfikacji ECMAScript, ale w praktyce obsługiwana jest przez wszystkie mechanizmy skryptowe zarówno w normalnym kodzie, jak i w funkcji `JSON.parse(…)`.
- Ośmiobitowe, szesnastkowe, dwucyfrowe kody poprzedzane znakiem *x* (literę *e* zapisuje się jako `\x65`). Tutaj również opis składni nie pojawia się ani w specyfikacji ECMAScript, ani w dokumencie RFC 4627, ale z powodu swoich korzeni wywodzących się z języka C jest ona w praktyce obsługiwana prawie wszędzie.
- Szesnastobitowe, szesnastkowe, czterocyfrowe wartości Unicode poprzedzane znakiem *u* (litera *e* zamienia się w `\u0065`). Ten format został zdefiniowany w specyfikacji ECMAScript oraz dokumencie RFC 4627 i w związku z tym obsługiwany jest we wszystkich nowoczesnych przeglądarkach.
- Lewy ukośnik, za którym znajduje się dowolny znak (z wyjątkiem cyfry ósemkowej, liter *b*, *t*, *v*, *f*, *r* lub *n* używanych w innej metodzie kodowania znaków oraz liter *x* lub *u*). W tym przypadku znak znajdujący się za lewym ukośnikiem traktowany jest jako literał. Specyfikacja ECMAScript dopuszcza stosowanie tego kodowania jedynie do oznaczania znaków cudzysłowu oraz lewego ukośnika, ale w praktyce akceptowane są również wszystkie inne wartości.

Rozwiązanie to jest jednak podatne na błędy, ponieważ w przypadku stylów CSS nie powinno się go używać do oznaczania znaków nawiasów ostrych i innych znaków stosowanych w składni języka HTML. Wynika to z faktu, że parsowanie kodu JavaScript wykonywane jest po zakończeniu parsowania kodu HTML, a znak lewego ukośnika nie ma dla parsera HTML żadnego specjalnego znaczenia.

UWAGA *Nieco zaskakujący jest fakt, że Internet Explorer nie rozpoznaje sekwencji kodującej znak pionowej tabulacji (\v), co umożliwia zastosowanie bardzo wygodnej (choć i niezwykle paskudnej!) metody rozpoznawania tej szczególnej przeglądarki:*

```
if ("\v" == "v") alert("Wygląda jak Internet Explorer!");
```

Zaskakujący jest natomiast fakt, że metody oznaczania znaków Unicode (i tylko one) rozpoznawane są również poza ciągami znaków. Pomysł może wydawać się nieco dziwny, ale takie zachowanie sprawdza się lepiej niż w przypadku stylów CSS. Kody znaków mogą być używane tylko w identyfikatorach i nie da się ich zastosować do podmiany znaków wpływających na składnię skryptu. Oznacza to, że poniższy wiersz kodu jest jak najbardziej prawidłowy:

```
\u0061lert("A tu mały komunikat!");
```

Z drugiej strony wszystkie próby zastąpienia w ten sposób znaków cudzysłowu lub nawiasów na pewno się nie powiodą.

Żaden mechanizm skryptowy JavaScriptu nie toleruje wielowierszowych literałów tekstowych, co jest działaniem zgoła odmiennym od tego, co znamy z niektórych implementacji języków C lub C++. Trzeba jednak zaznaczyć, że mimo wyraźnego sprzeciwu zapisanego w specyfikacji ECMAScript od zasady tej istnieje jeden wyjątek: lewy ukośnik umieszczony na końcu wiersza może zostać użyty do połączenia wielu wierszy jednego literału tekstowego. Zachowanie to przedstawiam w poniższym przykładzie:

```
var text = 'Ta składnia
           jest nieprawidłowa.';

var text = 'Z drugiej strony tę składnię \
           rozpoznają wszystkie przeglądarki.';
```

Tryby dołączania kodu i ryzyko zagnieżdżania

Z opisów podanych w poprzednich podrozdziałach jasne już jest, że istnieje kilka sposobów na wykonanie skryptów w kontekście aktualnej strony. Dobrze by było wymienić tutaj te najczęściej używane sposoby:

- Bloki `<script>` w kodzie strony.

- Zdalne skrypty ładowane znacznikiem `<script src=…>`*.
- Pseudoadresy *javascript:* umieszczane w różnych parametrach HTML i CSS.
- Składnia `expression(…)` używana w stylach CSS i wiązaniach XBL dostępnych w niektórych przeglądarkach.
- Funkcje obsługi zdarzeń (`onload`, `onerror`, `onclick` itd.).
- Czasomierze (`setTimeout`, `setInterval`).
- Wywołania funkcji `eval(…)`.

Łączenie ze sobą tych metod wydawać się może całkiem naturalne, ale takie działanie może tworzyć nieoczekiwane i niebezpieczne efekty podczas parsowania. Na przykład w poniższym kodzie przyjrzyjmy się transformacjom, jakie trzeba wykonać na wartości zwracanej przez serwer, którą wstawiamy tutaj w miejsce tekstu `dane_użytkownika`:

```
<div onclick="setTimeout('do_stuff(\'dane_użytkownika\')', 1000)">
```

Często można nawet nie zauważyć, że taka wartość będzie trzykrotnie poddawana parsowaniu! Po pierwsze parser HTML pobierze parametr `onclick` i umieści go w modelu DOM. Po drugie po kliknięciu przycisku pierwsze parsowanie kodu JavaScript pobierze wszystkie informacje związane z funkcją `setTimeout(…)`. Po trzecie sekundę po kliknięciu rozpocznie się parsowanie i wykonywanie właściwej funkcji `do_stuff(…)`.

Oznacza to, że powyższy przykład musi zostać odpowiednio przygotowany na cały ten proces. Ciąg znaków `dane_użytkownika` musi zostać podwójnie zakodowany z wykorzystaniem JavaScriptowych sekwencji z lewym ukośnikiem, a następnie zakodowany ponownie za pomocą encji HTML. Wszystkie operacje kodowania muszą być wykonane w podanej tu kolejności, gdyż jakiekolwiek odstępstwo od tej zasady może doprowadzić do wstrzyknięcia kodu.

Poniżej prezentuję kolejną ciekawą sytuację związaną z kodowaniem:

```
<script>
var some_value = "dane_użytkownika";
...
setTimeout("do_stuff('" + some_value + "')", 1000);
</script>
```

* W przypadku obu rodzajów bloków `<script>` Microsoft dopuszcza zastosowanie pseudodialektu, nazywanego `JScript.Encode`. Ten tryb można włączyć, podając parametr `language` w znaczniku `<script>`, a pozwala on na zakodowanie skryptu za pomocą trywialnego mechanizmu podstawiania liter, który ma sprawić, że jego kod stanie się nieczytelny dla zwykłych użytkowników. Takie działanie jest całkowicie bezużyteczne z punktu widzenia bezpieczeństwa, ponieważ takie „szyfrowanie" można złamać bez żadnego problemu.

Co prawda początkowe przypisanie do zmiennej `some_value` wymagałoby tylko jednokrotnego zakodowania ciągu znaków `dane_użytkownika`, ale następująca później konstrukcja, która tworzy kolejny skrypt w parametrze funkcji `setTimeout(…)`, może wprowadzić podatność na wstrzyknięcie kodu, jeżeli wcześniej dane nie zostaną odpowiednio zakodowane.

Takie wzorce bardzo często powtarzają się w programach JavaScript, dlatego łatwo można je przeoczyć. Znacznie lepiej byłoby, gdyby ich użycie było utrudnione, ponieważ wyszukiwanie ich w gotowym kodzie jest niezwykle trudne.

Żywy trup: Visual Basic

Po omówieniu najważniejszych zagrożeń związanych z językiem JavaScript możemy oddać honory dawno zapomnianemu konkurentowi w wyścigu o tron języków skryptowych. Mimo że od 15 lat nie jest on prawie w ogóle używany, język VBScript nadal jest obsługiwany przez kolejne wersje Internet Explorera. Pod wieloma względami język Microsoftu miał być funkcjonalnym odpowiednikiem języka JavaScript, dlatego również może korzystać z tego samego API modelu DOM oraz innych wbudowanych funkcji. Jak się jednak można spodziewać, istnieje w nim kilka drobnych zmian i rozszerzeń. Na przykład zamiast funkcji wbudowanych w JavaScript dostępnych jest kilka funkcji przeznaczonych wyłącznie dla języka VBScript.

Na temat bezpieczeństwa skryptów VBScript nie przeprowadzono prawie żadnych badań. Podobnie niewiele wiadomo na temat skuteczności parsera oraz potencjalnych niezgodności z nowoczesnym modelem DOM. Z anegdot dowiadujemy się, że nawet w samej firmie Microsoft język ten nie jest poddawany pełnej analizie. Na przykład wbudowana funkcja `MsgBox`[8] może zostać użyta do wyświetlenia modalnych okien z komunikatami, których możliwości i elastyczność nie mają swojego odpowiednika w świecie JavaScript. Funkcja `alert(…)` jest w tym porównaniu tylko ubogim krewnym.

Trudno przewidywać, jak długo język VBScript obsługiwany będzie jeszcze w Internet Explorerze i jakie konsekwencje będzie to miało dla użytkowników oraz dla bezpieczeństwa aplikacji WWW. Czas pokaże.

Ściąga

Podczas ładowania zdalnego skryptu

Podobnie jak w przypadku stylów CSS, łączysz bezpieczeństwo swojej witryny z domeną, z której pochodzi skrypt. W razie wątpliwości lepiej zrobić lokalną kopię kodu. W przypadku witryn HTTPS wymagaj, żeby wszystkie skrypty były przesyłane za pomocą tego protokołu.

Podczas parsowania danych JSON otrzymanych od serwera

Jeżeli masz taką możliwość, korzystaj z funkcji `JSON.parse(…)`. Nie stosuj funkcji `eval(…)` ani żadnej korzystającej z niej implementacji pochodzącej z dokumentu RFC 4627. Nie są one bezpieczne, szczególnie w trakcie przetwarzania danych pochodzących z zewnętrznych źródeł. Prawdopodobnie bezpieczna jest natomiast późniejsza implementacja wykonana przez autora dokumentu RFC 4627 o nazwie *json2.js*[9].

Gdy umieszczasz dane przesłane przez użytkownika w blokach JavaScriptu

☑ **Samodzielne ciągi znaków w blokach `<script>`**: Koduj za pomocą lewego ukośnika wszystkie znaki sterujące (`0x00` – `0x1F`), znaki lewego ukośnika, nawiasów ostrych oraz cudzysłowu, podając ich kody liczbowe. Dobrze jest też podobnie traktować znaki z górnego zakresu.

Nie korzystaj z ciągów znaków podanych przez użytkownika do dynamicznego tworzenia kodu HTML. Zawsze używaj bezpiecznych funkcji i właściwości modelu DOM, takich jak `innerText` albo `createTextNode(…)`. Nie korzystaj z ciągów znaków podanych przez użytkownika do konstruowania skryptów parsowanych z opóźnieniem. Unikaj stosowania funkcji `eval(…)`, `setTimeout(…)` itp.

☑ **Samodzielne ciągi znaków w skryptach przesyłanych oddzielnie**: Stosuj reguły dotyczące bloków `<script>`. Jeżeli Twoje skrypty zawierają wrażliwe informacje opisujące użytkownika, upewnij się, że bierzesz pod uwagę ryzyko międzydomenowego dołączania skryptów. W pobliżu początku pliku stosuj przedrostki oszukujące parsery, takie jak `)}]'\n`, a przynajmniej użyj właściwej serializacji JSON, bez wykorzystania dodatków i innych zmian. Oprócz tego przejrzyj rozdział 13., w którym znajdziesz wskazówki zapobiegające umieszczaniu skryptów międzydomenowych w treściach niezwiązanych z kodem HTML.

☑ **Ciągi znaków w funkcjach obsługi zdarzeń definiowanych w kodzie HTML, pseudoadresy `javascript:` itd.**: W takich sytuacjach konieczne jest zastosowanie wielu poziomów kodowania. Nie próbuj tego, ponieważ jest to proces bardzo podatny na błędy. Jeżeli nie ma innego wyjścia, stosuj prezentowane wyżej zasady kodowania dla języka JavaScript, następnie w razie potrzeby w wynikowym ciągu znaków użyj kodowania parametrów języka HTML lub adresów URL. Nigdy nie stosuj w połączeniu z funkcjami i właściwościami typu `eval(…)`, `setTimeout(…)`, `innerHTML` i podobnymi.

☑ **Treści niebędące ciągami znaków**: Pozwalaj na pojawianie się wyłącznie alfanumerycznych słów kluczowych z białej listy i ostrożnie sprawdzaj poprawność wartości liczbowych. Nie próbuj metody odrzucania znanych, niebezpiecznych wzorców.

Podczas interakcji z obiektami przeglądarki po stronie klienta

- ☑ **Generowanie treści HTML po stronie klienta**: Nie korzystaj z takich narzędzi jak `innerHTML`, `document.write(…)` i podobnych, ponieważ z ich pomocą można wprowadzić do aplikacji podatności na międzydomenowe wykonanie skryptów. Do konstruowania dokumentu używaj raczej bezpiecznych metod, takich jak `createElement(…)` i `appendChild(…)`, oraz takich właściwości jak `innerText` lub `textContent`.
- ☑ **Korzystanie z danych tworzonych przez użytkownika**: Nie czyń żadnych założeń co do reguł kodowania zastosowanych wobec wartości odbieranych od przeglądarki, we właściwości `location` oraz innych zewnętrznych źródłach adresów URL. Reguły te są niespójne i różnią się w poszczególnych implementacjach. Zawsze samodzielnie wykonuj kodowanie zawartości ciągów znaków.

Jeżeli chcesz pozwolić na działanie skryptów użytkownika na swojej stronie

Zadania tego nie da się wykonać w sposób bezpieczny. Eksperymentalne próby przepisywania języka JavaScript, takie jak Caja (*http://code.google.com/p/google-caja/*), są chyba jedynym przenośnym rozwiązaniem. Przejrzyj też rozdział 16., w którym znajdą się informacje na temat ramek umieszczanych w piaskownicy, które w przyszłości mogą stać się alternatywą dla osadzania na własnych stronach kodu niezaufanych gadżetów.

Skorowidz

D

E

F

G

N

O

P

Q

R

S

Ś

T

U

W

X

Z

Ż

„Dokładna i wyczerpująca analiza, przygotowana przez jednego z najpoważniejszych ekspertów od bezpieczeństwa przeglądarek".

Tavis Ormandy, Google Inc.

Nowoczesne aplikacje WWW są jak splątany kłębek, złożony z powiązanych wzajemnie technologii, które powstawały w różnym czasie i których współpraca nie przebiega całkiem gładko. Użycie w stosie aplikacji WWW dowolnego elementu — od żądań HTTP aż po skrypty działające w przeglądarce — pociąga za sobą ważne, choć subtelne konsekwencje związane z bezpieczeństwem. Twórcy aplikacji chcący chronić użytkowników muszą pewnie poruszać się w tym środowisku.

Michał Zalewski, jeden z czołowych ekspertów od bezpieczeństwa przeglądarek, prezentuje w *Splątanej sieci* porywające objaśnienie metod działania przeglądarek i powodów niedostatecznego poziomu ich bezpieczeństwa. Nie podaje uproszczonych porad dotyczących różnych podatności, ale przegląda cały model bezpieczeństwa i wskazuje jego słabe punkty. Pokazuje też sposoby poprawienia bezpieczeństwa aplikacji WWW.

Z książki dowiesz się, jak:

- wykonać powszechne, a mimo to bardzo złożone zadania, takie jak parsowanie adresów URL i oczyszczanie kodu HTML
- używać nowoczesnych funkcji bezpieczeństwa, takich jak Strict Transport Security, Content Security Policy oraz Cross-Origin Resource Sharing
- wykorzystywać warianty reguły tego samego pochodzenia do bezpiecznego rozdzielania złożonych aplikacji WWW i ochrony danych użytkownika w przypadku wystąpienia błędów XSS
- tworzyć aplikacje hybrydowe i wstawiać na stronę gadżety bez wpadania w pułapki wynikające z reguł nawigacji w ramkach
- osadzać na stronie i udostępniać treści tworzone przez użytkowników bez uciekania się do mechanizmów wykrywania rodzajów tych treści

Unikalny podręcznik poświęcony bezpieczeństwu!

Michał Zalewski jest uznanym na całym świecie ekspertem ds. bezpieczeństwa informacji. Może poszczycić się wykryciem setek różnego rodzaju podatności i często wymieniany jest wśród osób mających największy wpływ na bezpieczeństwo w sieci. Jest autorem *Ciszy w sieci*, dostępnego na stronach *Google Browser Security Handbook* oraz wielu ważnych artykułów.

ISBN 978-83-246-4477-3

Cena: 54,90 zł