[JavaScript] Ukrywanie ciała funkcji

Spis treści

Niestety, w nowych wersjach Chrome ta metoda już nie działa i dostajemy pełen dostęp do ciała funkcji.

Na wstępie uprzedzę, że technika ta nie powstała w mym płodnym umyśle. Została znaleziona przez **developera Opery (http://web.archive.org/web/20130907212310/http://my.opera.com/hallvors/blog/2012/03/20/debugging-maps-google-maps)** w czasie debugowania pewnego upierdliwego błędu w działaniu Google Maps w tejże przeglądarce. Jak nietrudno zgadnąć, zostało to w bólach znalezione w źródle tego zaawansowanego webappa. Ja jedynie postanowiłem to wykorzystać na szerszą skalę. Trzeba też zaznaczyć, że technika ta jest średnio przydatna jeśli nie zaciemnimy równocześnie źródła skryptu (np. poprzez uglify.js czy też GCC).

O co wgl chodzi? A no chodzi o pewną magiczną właściwość poniższego fragmentu kodu:

```
Function.prototype
```

Niby co w tym takiego nadzwyczajnego? Już mówię!

```
console.log( Function.prototype.call.apply( Function.prototype.bind, [ function(){
} ] ) );
```

Zapewne w większości powyższy kod wzbudza podświadomy lęk. Przyznać trzeba, że naprawdę wygląda dość dziwacznie. Ale postaram się troszkę przybliżyć jego sens.

Zacznijmy od `Function.prototype`. Jak już kiedyś wspomniałem, JS jest takim mocno zboczonym językiem, gdzie wszystko jest obiektem. Nic zatem dziwnego, że funkcje są tak naprawdę… obiektami "klasy" `Function` ("klasy", bo jak wiadomo ECMAScript to jedyna grupa języków prototypowanych). Zatem każda zmiana w prototypie tego globalnego obiektu będzie dziedziczona przez wszystkie funkcje. Jednak tutaj skrypterzy Google poszli o krok dalej i wywołują poszczególne metody prosto z prototypu (czasami, ale to bardzo rzadko, okazuje się to przydatne)!

Zatem jakie metody się tu wywołuje? `call`, `apply` i `bind`. Wszystkie działają dość podobnie i ich głównym zadaniem jest zmienienie kontekstu wywołania funkcji (`this`). Już wyjaśniam na przykładzie:

```
var d = { n: 1 };

function g() {
    console.log( this );
}

g.apply( d );
```

To wywołanie funkcji wyrzuci do konsoli obiekt `d` zamiast standardowego `window`.Dość użyteczne, pod warunkiem, że pisze się obiektowy JS i trochę kojarzy pojęcie kontekstu wywołania. Dla nas jednak nie jest to aż tak ważne. Wystarczy wiedzieć, że `apply` przyjmuje jako pierwszy parametr obiekt, który ma zastąpić nam `this`, a drugim parametrem jest tablica parametrów wywoływanej funkcji. Przykład, bo brzmi to dziwnie:

```
function g( r ) {
    console.log( r );
};

g.apply( window, [ 'whatever' ] );
```

W konsoli ujrzymy wspaniały napis `whatever`. `call` w zasadzie działa tak samo, ale zamiast tablicy parametrów, przyjmuje ich listę:

```
function g( r ) {
    console.log( r );
};

g.call( window, 'whatever' );
```

Efekt identyczny jak w poprzednim przykładzie. Trochę inaczej działa `bind`, ponieważ nie wywołuje funkcji w podanym kontekście, a zwraca jej "klona", działającego w danym scope. Dla kompletności przykład, z wiadomym wynikiem:

```
function g( r ) {
    console.log( r );
};
( g.bind( window, [ 'whatever' ] ) )();
```

Zatem gdy trochę się wysili szare komórki, można dojść do pewnego uogólnienia, że w bardzo zawoalowany sposób po prostu wywołujemy `Function.prototype.bind`.

OK, a teraz gwóźdź programu: co nam daje połączenie tych trzech metod? A no, przy dobrze zaciemnionym kodzie bardzo utrudnia podejrzenie działania naszych funkcji i ukrywa to przed debuggerami. Kod

```
console.log( Function.prototype.call.apply( Function.prototype.bind, [ function()
{} ] ) );
```

zwróci nam bowiem nic nie mówiące

```
function() { [native code] }
```

Chyba nie muszę mówić jaką konsternację na twarzy młodocianego hakiera zrobi informacja, że nasza super-hiper-tajna funkcja, w której zaimplementowaliśmy czekający na opatentowanie algorytm, uparcie twierdzi, że jest funkcją wbudowaną i to na dodatek bez nazwy!

I zanim polecisz opakowywać wszystkie swoje funkcje w ten sposób, wiadro zimnej wody dla ochłody: **KOMPATYBILNOŚĆ**. Tak, wiem, że wiesz, że ja wiem, że wiesz, że nie działa to w IE < 9 – przecież to logiczne! Ale niestety rynek tych przeglądarek wciąż jest dośc spory. Toteż – dla własnej wygody i kompatybilności – napiszmy sobie prostą funkcję pomocniczą. Nazwijmy ją (a jakże!) `obfuscate`. Użyjemy tu dwóch bardziej zaawansowanych elementów JS: funkcji natychmiastowego wywołania oraz closures (i w tym miejscu większość mniej zdeterminowanych postanowiła zamknąć tą kartę przeglądarki). Niech Was nie zwiodą pozory – wcale to (aż tak) trudne nie będzie! Na początku zadeklarujmy sobie swoją funkcję:

```
var obfuscate = ( function() {
}() );
```

To jest właśnie tzw. funkcja natychmiastowego wywołania (IIFE – Immediately Invoked Function Expression). Jak nietrudno zgadnąć, nazywa się tak, bo od razu po zadeklarowaniu jest wywoływana (to tak naprawdę jest wywołanie funkcji anonimowej). Na razie nasz kod jest bezużyteczny, bo zmienna `obfuscate` będzie miała wartość `undefined`. Toteż trzeba dodać do naszej funkcji `return`:

```
var obfuscate = ( function() {
    return function( func ) {
        return Function.prototype.call.apply( Function.prototype.bind, [ func ] );
    };
}() )
```

Tak, to jest właśnie closure – funkcja zwracana przez inną funkcję, zamknięta w jej obszarze (no ej, w końcu to technika od samego Google; nikt nie mówił, że łatwo będzie). Tak naprawdę, gdy JS się wczyta, przeglądarka otrzyma coś takiego:

```
var obfuscate = function( func ) {
    return Function.prototype.call.apply( Function.prototype.bind, [ func ] );
}
```

Czemu zatem nie zapisałem tego w taki sposób? Otóż przez IE 8 tak nie zapisałem! Trzeba przecież sprawdzić czy ta metoda ukrywania kodu jest wspierana i w zależności od tego funkcja `obfuscate` przyjąć może dwie postaci:

- zaciemni nam kod,
- zwróci niezaciemniony kod.

A jak sprawdzić czy technika jest obsługiwana? Można sprawdzić czy przeglądarka wysypie się na wywołaniu zaciemnionej w taki sposób funkcji. Ok, ale to nam zatrzyma wykonywanie skryptu… chyba że wrzucimy do tego obsługę wyjątków!

```
try {
    Function.prototype.call.apply( Function.prototype.bind, [ function() {} ] )();
} catch ( err ) {}
```

Dobra, zwracanie niezaciemnionego kodu jest łatwe – wystarczy zwrócić parametr `func`. Tyle. Zatem po połączeniu wszystkiego razem, otrzymamy coś takiego:

```
var obfuscate = ( function() {
    var a = Function.prototype;

    try {
        a.call.apply( a.bind, [ function() {} ] )();
    } catch ( err ) {
        return function( func ) {
            return func;
        };
    }

    return function( func ) {
        return a.call.apply( a.bind, [ func ] );
    };
}() );
```

- Najpierw przypisujemy sobie `Function.prototype` do zmiennej `a` (żeby się później nie napisać i oszczędzić te parę bajtów).
- Sprawdzamy, czy obfuskacja w ogóle działa w bloku `try`/`catch`. Jeśli nie, jak gdyby nigdy nic zwracamy funkcję zwracającą parametr `func` (nieźle to brzmi).
- W innych przypadkach po prostu zwracamy funkcję zwracającą zaciemnioną funkcję

Przykład:

```
var f = obfuscate( function( n, r ) {
    return n + ':' + r;
} );
console.log( f( 'r', 't' ) ); // "r:t"
console.log( f ); // function () { [native code] }
```

Demo online nie wstawiam, bo raczej za dynamiczne by nie było.

Aaaaa, i taka uwaga na koniec: moja funkcja `obfuscate` bardzo lubi strict mode!