

実行プロセス分離による JIT シェルコード実行防止

市川 顕†

松浦 幹太‡

† 東京大学生産技術研究所
153-8505 東京都目黒区駒場 4-6-1
ichik@iis.u-tokyo.ac.jp

‡ 東京大学生産技術研究所
153-8505 東京都目黒区駒場 4-6-1
kanta@iis.u-tokyo.ac.jp

あらまし JIT コンパイラを悪用した JIT Spraying という攻撃を利用すれば、DEP や ASLR といったセキュリティ機構を突破し、バッファオーバーフロー攻撃などが可能になるとして話題となっている。JIT Spraying は JIT コンパイラが仕様としてデータ領域に実行属性を付加することにより生成コードがシェルコードとして使用され得ることを利用する。本稿では、JIT コンパイラにより生成されたコードを走らせるためにデータ領域に実行属性を付けるプロセスをメインのプロセスとは分離し、メインのプロセスではデータ領域に実行属性を付加させず、容易に攻撃ができないような JIT コンパイラの実装方法について紹介する。

Preventing execution of JIT shellcode by isolating running process

Ken Ichikawa†

Kanta Matsuura‡

†Institute of Industrial Science, The University of Tokyo
4-6-1 Komaba Meguro-ku, Tokyo 153-8505, JAPAN
ichik@iis.u-tokyo.ac.jp

‡Institute of Industrial Science, The University of Tokyo
4-6-1 Komaba Meguro-ku, Tokyo 153-8505, JAPAN
kanta@iis.u-tokyo.ac.jp

Abstract JIT spraying attacks which abuse JIT compilers bypass security structures such as DEP and ASLR, and they enable memory corruption attacks such as buffer overflow attacks again. JIT compilers have a property that makes some data areas executable. Due to this property, generated native codes can be used as shellcodes for JIT spraying attacks. In this paper, we introduce the implementation technique that can't be attacked easily by isolating the process in which JIT compiled codes run. Main process therefore doesn't have to add executable attributes to its data areas.

1 はじめに

JIT(Just-In-Time) コンパイラを悪用した JIT spraying という攻撃が D.Blazakis の発表 [1] により広く知られるようになった。JIT Spraying はセキュリティ機構である DEP(Data Execu-

tion Prevention) と ASLR(Address Space Layout Randomization) を回避することができる。DEP は本来実行する必要のないスタックやヒープなどのデータ領域を不正に実行されることのないように実行不可とすることができるが、JIT コンパイラには、実行時に JavaScript などの

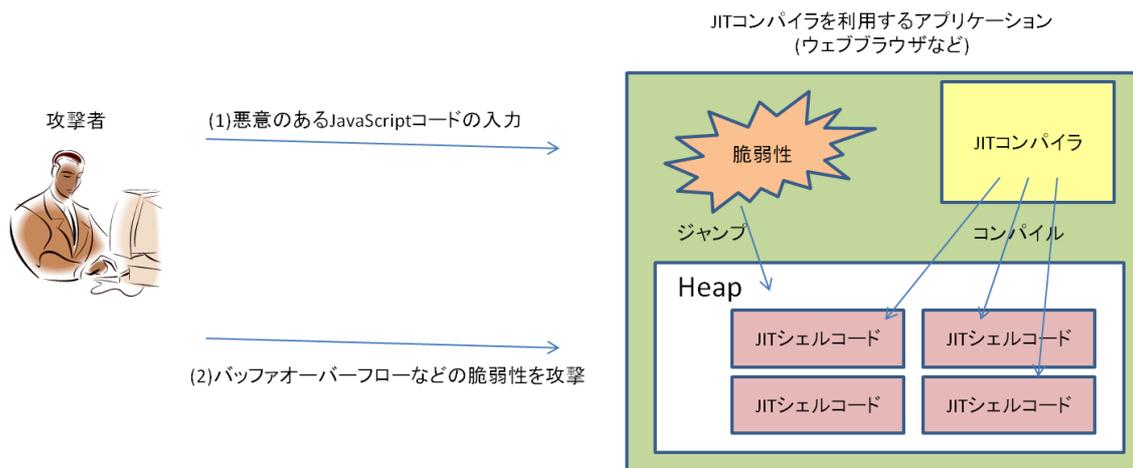


図 1: JIT spraying 攻撃の概要

コードをネイティブコードにコンパイルして実行するという特性がある。そのため、JIT コンパイルされたコードが格納されるデータ領域については実行属性を付加して DEP を無効にせざるを得ない。ASLR は仮想メモリのアドレスをある程度ランダムにするが、JIT コンパイラに入力されたコードが大量の悪性の機械語を生成するものであった場合、Heap Spray[2] と同じような状態に陥り、攻撃コードが実行される可能性が高まる。このようにして、DEP と ASLR によって防がれていたバッファオーバーフローなどの脆弱性を利用した攻撃が再び可能になってしまう。

本稿では、まず JIT spraying 攻撃の概要について説明する。さらに V8 JavaScript エンジン [3] で JIT シェルコードを構築し、JIT コンパイラの危険性を示す。本稿では JIT コンパイルによって生成されるシェルコードのことを JIT シェルコードと呼ぶ。また、そのような危険を排除するため、JIT コンパイルされたコードを実行するプロセスをメインのプロセスと分離させる手法を提案する。これにより、メインのプロセスのデータ領域には実行属性を付加する必要がなくなり、JIT シェルコードを容易に実行することができなくなる。

2 JIT spraying 攻撃の概要

図.1 に JIT spraying 攻撃の例を示す。まず、攻撃者は悪意のある JavaScript コードを JIT コンパイラを利用しているアプリケーションへ入力する。すると、入力したコードは JIT コンパイラにより JIT シェルコードを含むネイティブコードへコンパイルされ、それが大量にそのアプリケーションのメモリのヒープ領域に格納される。その領域は本来実行される必要があるため、実行属性がつけられる。次に、攻撃者はさらにアプリケーションの脆弱性を攻撃するための入力をする。それにより、プログラムの実行位置はヒープ上へジャンプさせられる。ヒープ上には大量の JIT シェルコードがあり、当てずっぽうでジャンプさせられても JIT シェルコードが実行されてしまう。

3 JIT コンパイラの危険性

Flash Player 10.0.42.34 の危険性については既に D.Blazakis により示されている [1] が、本稿では V8 という JavaScript エンジンで実際に JIT シェルコードが構築され得ることを示す。本稿で言及する V8 のバージョンは 2.1.10 である。V8 は Google により開発されているオープンソースの JavaScript エンジンであり、JIT コンパイラを搭載している。ウェブブラウザである

```
//spray.js
function func(
  a1, a2, a3, a4, a5,
  a6, a7, a8, a9, a10,
  a11, a12, a13, a14, a15,
  a16, a17, a18, a19){}
func(
  0x1e6018c8,
  0x1e345a48,
  0x1e6d98c8,
  0x1e00dbc8,
  0x1e71fbc8,
  0x1e39da48,
  0x1e17d848,
  0x1e375a28,
  0x1e34d848,
  0x1e71fbc8,
  0x1e71fbc8,
  0x1e315a48,
  0x1e17d848,
  0x1e2daa28,
  0x1e6498c8,
  0x1e6918c8,
  0x1e6018c8,
  0x1e05d848,
  0x1e4066c8
);
```

図 2: V8 へ入力する JavaScript コード

Google Chrome[4] やサーバサイド JavaScript の Node.js[5] などで利用されている。

次に、V8 でシェルコードを構築する具体的な方法について示す。例として、図.2 に示す JavaScript コードを V8 へ入力する。このコードは JIT コンパイルされ、その結果のネイティブコードがメモリに置かれる。そのメモリ領域の保護属性は `rwX` に設定されているため、実行が可能である。図.3 に、そのネイティブコードの一部を示す。このままでは特に害のあるコードではないが、もしこのコードを 1 バイトずらして解釈したとすると図.4 のようになる。最初の `push` 命令 (`0x68`) のオペランドであった `0x3cc03190` は

address	binary(hex)	instruction
0x3aac75	689031c03c	push 0x3cc03190
0x3aac7a	6890b4683c	push 0x3c68b490
0x3aac7f	689031db3c	push 0x3cdb3190
0x3aac84	6890b7013c	push 0x3c01b790
0x3aac89	6890f7e33c	push 0x3ce3f790
0x3aac8e	6890b4733c	push 0x3c73b490
0x3aac93	6890b02f3c	push 0x3c2fb090
0x3aac98	6850b46e3c	push 0x3c6eb450
0x3aac9d	6890b0693c	push 0x3c69b090
0x3aaca2	6890f7e33c	push 0x3ce3f790
0x3aaca7	6890f7e33c	push 0x3ce3f790
0x3aacac	6890b4623c	push 0x3c62b490
0x3aacb1	6890b02f3c	push 0x3c2fb090
0x3aacb6	6850545b3c	push 0x3c5b5450
0x3aacbb	689031c93c	push 0x3cc93190
0x3aac0	689031d23c	push 0x3cd23190
0x3aac5	689031c03c	push 0x3cc03190
0x3aacca	6890b00b3c	push 0x3c0bb090
0x3aaccf	6890cd803c	push 0x3c80cd90

図 3: JIT コンパイルされて生成されたネイティブコード

それぞれ `0x90` が `nop` 命令、`0x31c0` が `xor` 命令、`0x3c` が `cmp` 命令となり、その `cmp` 命令は次の `push` 命令のオペコードであったはずの `0x68` をオペランドにとる。これにより本来の `push` 命令は表に出てこられなくなる。そして、本来の命令から 1 バイトずらして解釈したこのコードは最終的に `/bin/sh` を起動することができる。さらに、頭に `nop` 命令を大量に生成するようにすれば ASLR を回避する確率が高まる。このようにして、V8 を使って JIT spraying 攻撃に利用できる JIT シェルコードを実際に構築することができた。もし、このような JIT コンパイラを利用しているアプリケーションにバッファオーバーフローなどの脆弱性が存在した場合、その脆弱性が悪用されてプログラムの実行位置を操作され、実際に JIT シェルコードを実行される可能性がある。

address	binary(hex)	instruction
0x3aac76	90	nop
0x3aac77	31c0	xor eax, eax
0x3aac79	3c68	cmp al, 0x68
0x3aac7b	90	nop
0x3aac7c	b468	mov ah, 0x68
0x3aac7e	3c68	cmp al, 0x68
0x3aac80	90	nop
0x3aac81	31db	xor ebx, ebx
...		

図 4: JIT コンパイルされて生成されたネイティブコードを 1byte ずらして解釈した結果

4 実行プロセス分離の提案

JIT コンパイラは実行時にコードをコンパイルする。そのため、コンパイルしたネイティブコードをデータ領域に置き、それを実行する。データ領域に書き込んだコードを実行させるにはそのメモリ領域へ実行属性を付ける必要がある。その際、前節の例のように生成したネイティブコードにシェルコードとして働く部分が存在していると、もし JIT コンパイラやそれを利用しているアプリケーションにバッファオーバーフローなどメモリ破壊の脆弱性が存在していた場合、その脆弱性について JIT シェルコードを実行される場合がある。

そこで、我々は JIT コンパイルしたネイティブコードを走らせる専用の子プロセスを作り、それをそれ以外の処理をするメインのプロセスとは分離させることを提案する。子プロセスでは、JIT コンパイルされたネイティブコードの格納されているメモリ領域に実行属性を付け、それを実行する。脆弱性の入り込む余地を減らすため、それ以外の処理はなるべく行わない。メインのプロセスは JIT コンパイルしたネイティブコードを走らせる以外の全ての処理を行う。そのため、脆弱性の入り込む余地も大きい。しかし、JIT コンパイルしたネイティブコードを走らせる必要はなく、また、プロセス間では仮想メモリ空間が独立しているため、データ領域に実行属性を付ける必要はない。そのため、

もし脆弱性が存在したとしても DEP や ASLR などのセキュリティ機構が適切に設定されていれば JIT シェルコードを実行することは困難である。

5 実装方針の詳細

JIT コンパイルされたコードの実行プロセスを分離させる手法を JavaScript エンジンである V8 に実装することについて考える。プラットフォームは Linux 2.6 を想定する。

5.1 V8 の仕組み

V8 の JIT コンパイルやコード実行時の動作について説明する。

V8 はまず、変数やオブジェクトなどのデータを格納する領域や JIT コンパイルされたコードを格納する領域をメモリに確保し、データを格納する領域には保護属性 `rw`、コードを格納する領域には `rx` を指定する。そして、入力されたコードのグローバル領域をコンパイルし、それを実行する。実行されたコードは、実行中に関数呼び出しに遭遇した場合、V8 に組み込まれているコンパイルのための関数を読んでその関数の中の処理をコンパイルし、さらにそれを実行していく。また、JavaScript の変数などの読み込みや保存も V8 に組み込まれているロードストアの関数を JIT コンパイルされたコードから呼び出す。

5.2 実行プロセス分離処理実装の方針

まず、親プロセスがネイティブコードを入れる領域をメモリに確保するが、本来 `rx` 属性で確保しているところを `rw` で確保するようにする。そして、親プロセスが最初のコンパイルを終え、そのコンパイルしたネイティブコードを実行する直前で `fork` を行う。その `fork` によって子プロセスが生成される。JIT コンパイルされたネイティブコードの実行はこの子プロセスで行う。子プロセスには、`fork` により親プロセスのメモリの情報が複製されているので、子プ

プロセスはまず JIT コンパイルされたネイティブコードが格納されている領域に実行属性を付加し、そのコードの実行を開始させる。実行時に関数呼び出しに遭遇し、さらにコンパイルが必要になるなどした場合もそのまま子プロセスに実行させる。JIT コンパイルされたネイティブコードの実行が終了したら子プロセスは終了させる。親プロセスは子プロセスが終了するまで待機し、子プロセスが終了したら実行を再開する。そのため、親プロセスと子プロセスでの排他制御は必要ない。

さて、ここで問題となるのはデータの受け渡しである。このままでは最初に入力された JavaScript コードは子プロセスで最後まで実行されるが、そのあと変数やオブジェクトなどの情報やコンパイルされた関数などのコードの情報が親プロセスへ受け渡されていないため、続けてまた JavaScript コードを入力して実行する必要がある場合、親プロセスの実行が正しく継続できなくなる。そこで、JavaScript の変数などのデータや JIT コンパイルされたコードが格納される領域を共有メモリとして確保するようにし、親と子で共有する。共有メモリは、データは共有するが保護属性は共有しない。そのため、親では実行属性は付けず、子では実行属性を付けるということが可能である。子プロセスの実行中に新しく領域を確保した場合には、その共有メモリの id とアドレスをパイプで親に渡す。親はその情報を元に子で生成された共有メモリを自分の仮想メモリ領域の適切な場所へアタッチする。これにより、親プロセスは再び JavaScript コードを受け取り、実行を継続することができるであろう。

6 考察

本手法では、JIT コンパイルされて生成されるネイティブコード自体に変更は加えないため、主に生成されたコードの性能を測るようなベンチマークにはほとんど影響を及ぼさないであろう。アプリケーション全体としての実行速度は fork や共有メモリの操作、データの受け渡しの処理が加わるため多少落ちるが、実用的な範囲

内だと思われる。

また、本手法は共有メモリを多用する。Linux 2.6.38 のデフォルトの共有メモリ最大値は 32M バイトである。そのため、JavaScript コードが大きなデータを扱う場合や JavaScript コード自体が非常に大きい場合、共有メモリの最大値を設定によって引き上げる必要が出てくるかもしれない。

7 関連研究

既にいくつかの JIT spraying 攻撃を防ぐための研究が存在する。

Piotr Bania[6] は JIT シェルコードを検知するアルゴリズムを提案した。32bit の即値をオペランドに持つ mov 命令とそれに続いて 32bit の即値をオペランドに持つ命令が指定数以上継続したらそれを JIT シェルコードとして検知する。しかし、例えば本稿で示したものはそのアルゴリズムでは検知されない。本稿で示したのも含め様々な形の JIT シェルコードがあり得ることから、全ての JIT シェルコードのためのアルゴリズムを網羅することは難しい。そのため、本手法では JIT シェルコードを検知するという手段は取っていない。

Willem De Groef らの JITSec[7] はカーネルモジュールを組み込み、それによりシステムコールを呼び出すルーチン中で戻りアドレスをチェックし、それが .text セクション以外だった場合にはプログラムを強制終了させる。しかし、この手法を使うと JIT コンパイルしたネイティブコードがシステムコールを呼ぶ可能性のある JIT コンパイラは機能しなくなる。本手法にはそのような制限はない。

Tao Wei らの INSeRT[8] は JIT コンパイラを修正し、JIT シェルコードとして機能しないようなネイティブコードを生成するようにする。しかし、即値、レジスタ、関数引数やローカル変数位置のランダム化、ランダム性増加や攻撃検知のためのスニペットの挿入など、本手法と比較して考慮すべきことが多い。また、JIT コンパイルにより生成されるコードが変更され、コード量も増加するため、実行速度が 5%程度落ち

る。一方、本手法では生成されるコードは変わらないため、JavaScript コード自体の実行速度はほとんど変わらないものと思われる。

Ping Chen らの JITDefender[9] は、JIT コンパイルされたコードを実行するそのときだけその領域に実行属性を付け、それ以外のときには実行属性を外しておく。しかし、そのコード実行時に攻撃されてしまう可能性がある。本手法では、コード実行するプロセスを分離させ、その欠点を克服している。

8 おわりに

本稿では実在する JavaScript エンジンである V8 を用いて実際に JIT シェルコードを構築し、JIT コンパイラの危険性を示した。そして、そのような JIT シェルコードの実行を防ぐために、メインのプロセスから JIT コンパイルされたネイティブコードを実行させる処理を専用の子プロセスに分離させる手法を提案した。提案した手法は実用的な速度で動作することが期待される。

参考文献

- [1] D. Blazakis, “Interpreter exploitation: Pointer inference and jit spraying,” *Black Hat DC*, 2010.
- [2] SkyLined, “Internet explorer iframe src&name parameter bof remote compromise,” 2004. http://skypher.com/wiki/index.php?title=Www.edup.tudelft.nl/~bjwever/advisory_iframe.html.php.
- [3] Google, “v8 - v8 javascript engine - google project hosting,” <http://code.google.com/p/v8/>.
- [4] Google, “Google chrome - get a fast new browser. for pc, mac, and linux,” <http://www.google.co.jp/chrome>.
- [5] “node.js,” <http://nodejs.org/>.
- [6] P. Bania, “Jit spraying and mitigations,” 2010. <http://arxiv.org/abs/1009.1038v1>.
- [7] W. De Groef, N. Nikiforakis, Y. Younan, and F. Piessens, “Jitsec: Just-in-time security for code injection attacks,” in *Benelux Workshop on Information and System Security (WISSEC 2010)*, November 2010.
- [8] T. Wei, T. Wang, L. Duan, and J. Luo, “Insert: Protect dynamic code generation against spraying,” in *Information Science and Technology (ICIST), 2011 International Conference on*, pp. 323–328, march 2011.
- [9] P. Chen, Y. Fang, B. Mao, and L. Xie, “Jit-defender: A defense against jit spraying attacks,” in *Future Challenges in Security and Privacy for Academia and Industry*, vol. 354 of *IFIP Advances in Information and Communication Technology*, pp. 142–153, Springer Boston, 2011.