JIT コンパイラに対する 1 バイト定数攻撃とその防御手法

鈴木悠希 †　中山智之 †
味曽野雅史 †　品川高廣 †

1. はじめに

JIT (Just-In-Time) コンパイラは多くの Web ブラウザに実装されており，JavaScript プログラムを動的に機械語に変換することで高速化を図る機能である．しかし JIT コンパイラが処理する JavaScript プログラムはインターネットからダウンロードされるものであり，悪意のある JavaScript プログラムによる攻撃を受ける危険性がある．本稿では，JavaScript プログラムの定数が JIT コンパイラによって機械語の命令の一部として埋め込まれることを利用して，定数にガジェットと呼ばれる機械語のバイナリを埋め込んで ROP (Return-Oriented Programming) で連結して任意コード実行をおこなう攻撃を対象とする．

このような攻撃に対する防御手法として，constant blinding がある[2)]．Constant blinding は JavaScript プログラム中の定数をランダム値と XOR するコードを機械語に挟むことで，攻撃者の意図した機械語が実行可能領域に置かれないようにする手法である．Constant blinding は実際の Web ブラウザでも用いられており，Microsoft Edge の JavaScript エンジンである Chakra のコア部分である ChakraCore に実装されている．ChakraCore の実装では実行速度を維持するため，2 バイトより大きい定数にのみ constant blinding をおこなっているが，2 バイト以下の定数と関数末尾の ret 命令を組み合わせることで任意コード実行が可能であることが知られている[2)]．しかし，この攻撃では，システムコールを発行するために少なくとも 2 バイトの定数が必要であり，1 バイトの定数で攻撃をおこなう手法は知られていない．

本稿では，まず Intel x86/x64 において，JavaScript プログラム中の 1 バイト定数のみでシステムコールの発行に必要な 2 バイトガジェットを生成可能なことを示す．この手法では，1 バイトの定数 2 つが 1 つの機械語命令にまとめられることを利用して，2〜3 バイトのガジェットを生成して ROP によるシステムコールの発行を可能にする．次に，この 1 バイト定数攻撃に対する防御手法として，定数の大きさではなく値に基づく constant blinding をおこなうことで意図せず制御フローが変更されることを防ぎつつ，既存の関数の末尾の ret 命令を利用した攻撃を防ぐために，JIT コンパイルされた関数に CFI を導入する手法を提案する．

† 東京大学

2. 脅威モデル

本稿では，文献 2) と同じ脅威モデルを想定する．

- ASLR (Address Space Layout Randomization) 及び DEP (Data Execution Prevention) が有効
- G-Free[1)] 等により，JIT バッファ以外にはライブラリも含めて ROP ガジェットは存在しない
- 攻撃者は stack canary などをバイパスして制御フローを変えられる
- 攻撃者はメモリ公開バグなどで ASLR を破って攻撃者のデータを予測可能な場所に置ける

3. 1 バイト定数攻撃

Intel x64 において，1 バイト定数を 2 つとる機械語命令としては，以下のものがある．

```
movb $0x58, 0x41(%rax)
```

これは 1 バイトの定数 0x58 をレジスタ rax が示すアドレスからのオフセットが 0x41 バイトのメモリにストアする命令である．ここで，0x58 と 0x41 はそれぞれ 1 バイトの定数であり，Intel x86/x64 では以下にようなバイナリにエンコーディングされる．

```
c6 40 41 58
```

ここで，0x41 が格納されたアドレスに ret することにより，以下のような機械語として解釈して実行させることができる．

```
41 58           pop %r8
```

すなわち，スタック上に設定した任意の値をレジスタ r8 に格納することができる．このように，1 バイトの定数とオフセットを持つ機械語命令を生成させて関数末尾の ret と組み合わせることで，任意のレジスタに値を設定したうえで syscall 命令を呼び出すガジェッ

表 1 blind するべき命令とその機械語

blind するべき命令	機械語
ret	0xc2, 0xc3, 0xca, 0xcb
jmp/call	0xff

トが生成可能である．

このような機械語命令を生成させるための JavaScript プログラムとしては，以下のものが考えられる．

```
var a = new Uint8Array;
a[0x58] = 0xc3;
```

これはバイト配列に値を代入するものであり，実際に以前のバージョンの ChakraCore では，最適化により上記の機械語命令が生成される．ただし，最近の ChakraCore では，Spectre 対策の副作用で上記の機械語命令が生成されなくなっている．これは，以下のように WebAssembly を使うことで回避できる．この手法は最新の ChakraCore でも有効である．

```
(i32.store8 offset=0x41
      (get_global $g) (i32.const 0x58))
```

4. 防御手法

1 バイト定数攻撃を防ぐために，1 バイトの定数に対しても constant blinding を適用するというアプローチを採用する．ただし，このままでは適用する定数の数が多くなってオーバーヘッドが非常に大きくなってしまう．そこで本稿では，constant blinding を適用する定数の数を減らすために，まず (1) 定数の値に基づく constant blinding を提案する．これは，ROP ガジェットを構成するためには，制御フローを変更して攻撃者が指定したアドレスにジャンプさせるコードが必須であることに着目して，機械語として解釈したときに制御フローを変更する命令となりうる定数の値のみを blind するという手法である．制御フローを変える命令は ret 命令の他に pop rax, jmp rax のように jmp 命令や call 命令も使えることが知られている[1]ので，これらの機械語が含まれた定数を blind する．**表 1** に blind するべき命令とその機械語を示す．

一方，文献 2) の攻撃手法でも本稿の攻撃手法でも，2 バイトの命令と最後に ret 命令を含むガジェットを生成するには最低 3 バイト必要であり，2 バイトだけでは生成できるガジェットの種類が制限されてしまう．そこで文献 2) では，関数の最後に元々存在する ret 命令を活用することで完全なガジェットとする手法が用いられている．この手法は，本稿の防御手法 (1) でも有効であり，制御命令を変更しない命令と関数末尾の ret 命令を組み合わせることでバイパスできてしまう．

そこで，本稿では (2)JIT コンパイルした関数に CFI (Control Flow Integrity) をおこなう手法と併用することで，オーバーヘッドが低くかつ安全な防御手法を実現する．この手法では，関数の最初と最後でリターンアドレスをランダム値と XOR するコードを追加することで，ROP で関数の途中に入ってきたときに，攻撃者が指定したアドレスに戻ることを防止する．

この 2 つの手法を併用することにより，1 バイト定数のみを用いてシステムコールを発行する 3 バイトガジェットを生成することを防止することができる．

5. 関連研究

G-Free[1] はコンパイル時に ROP ガジェットをすべて排除することで ROP を防ぐという研究である．文献 2) は G-Free の環境下でかつ 2 バイトより大きい定数に constant blinding をおこなう JavaScript エンジンに対し，2 バイト以下の定数のみで ROP ガジェットを作り，攻撃が可能であることを示した．本研究は文献[2] を発展させたものである．

文献 3) は ARM における JIT コンパイラへの攻撃法を 2 つ示した研究である．1 つは JIT コンパイラによって置かれた正規の機械語を用いて JIT バッファ上の機械語を書き換えるというもので，本稿では JIT バッファが実行時には書き込み不可能であるという前提のため，この攻撃は本稿の対象外である．もう 1 つは JIT コンパイラによって生成された 32bit モードの機械語を 16bit モードで解釈することで攻撃をおこなうというもので，制御フローを変える機械語の入った定数を constant blinding することで防ぐことができる．提案手法の ARM への適用は現在検討中である．

6. おわりに

本稿では，1 バイト定数のみを用いて JIT コンパイラに 2〜3 バイトの ROP ガジェットを生成させる攻撃手法と，定数の値に基づいた constant blinding と JIT コンパイルした関数に CFI を組み合わせた防御手法を提案した．現在，Firefox の JIT コンパイラに constant blinding の実装をおこなっている．

参考文献

1) Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, “G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries, ” ASCAS ’10 2010.
2) Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, Sotiris Ioannidis, “The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines, ” NDSS ’15, 2015.
3) Wilson Lian, Hovav Shacham, Stefan Savage, “A Call to ARMs: Understanding the Costs and Benefits of JIT Spraying Mitigations, ” NDSS ’17, 2017.