情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2011-DPS-146 No.46
Vol.2011-CSEC-52 No.46
2011/3/11

実行監視によるJIT Spraying攻撃検知

市川 顕[†1] 松浦幹太[†1]

近年のウェブアプリケーションの興隆により、ウェブブラウザ上で動作する言語のエンジンにJITコンパイラが搭載される例が増えている。しかし、JITコンパイラで生成された実行コードはWindowsのセキュリティ機構であるDEPとASLRを回避することができるJIT Spraying攻撃に悪用される例が示されており、対策が必要とされている。本稿では、ユーザレベルでの実行監視によりJIT Spraying攻撃を検知し、攻撃コードが実行される前にプログラムを停止させる手法について述べる。

Detecting JIT Spraying Attacks by Monitoring Execution

Ken Ichikawa [†1] and Kanta Matsuura[†1]

Many web browsers use JIT compiler as a language processor because of the rise of web applications in recent years. However, it has been demonstrated that execution codes generated by JIT compiler are abused for JIT Spraying attacks that can bypass Windows security structures such as DEP and ASLR. Aiming at defending against the threat above, this paper describes and examines a user-level execution monitoring method which detects JIT Spraying attacks so that the defender can stop programs before attack codes are executed.

1. はじめに

現在、マイクロソフト社のOSであるWindowsには様々なセキュリティ機構が搭載されているが、その中で特にバッファオーバーフロー攻撃に有効であるものにDEPとASLR[1)]がある。DEP(Data Execution Prevention)が有効になると、メモリ上のデータは実行属性が付加されていない限り実行ができなくなる。ASLR(Address Space Layout Randomization)は各メモリセグメントの位置やDLLがロードされるアドレスをランダムにずらすことにより、攻撃者にメモリアドレスを推測されにくくする。DEPとASLRを同時に有効にすることにより、バッファオーバーフロー攻撃は一筋縄では行かなくなった。しかし、それはいまだに不可能とはなっていない。DEPやASLRが適切に設定されていたとしても、バッファオーバーフロー攻撃を成功させる手法が発明されている。そのような巧妙な手法の一つにJIT Spraying[2)]がある。JIT SprayingはJIT(Just-In-Time)という名前にあるようにJITコンパイラを悪用した手法である。また、Sprayingという名前にあるように、メモリ上に攻撃コードをまき散らしてASLRを回避するHeap Sprayingと似た手法でもある。

Heap SprayingやJIT Sprayingといった手法は、ウェブブラウザの脆弱性を突く場合に使用されることが多い。ウェブブラウザならば、攻撃者のウェブサイトを利用者に開かせさえすれば任意のスクリプトを容易に動かすことができ、それを利用して利用者のメモリに任意のデータをSprayすることができる。また、近年のウェブブラウザはスクリプトを実行するエンジンにインタプリタではなくJITコンパイラを採用している場合が多い。JITコンパイラは実行時にスクリプトやバイトコードをネイティブコードへコンパイルする。ネイティブコードが実行されるため、インタプリタと比較して非常に高速に動作する。そのような特性から、近年ますますリッチになっていくウェブアプリケーションの動作速度の向上のためにJITコンパイラがこれからも採用されていくと思われる。JIT SprayingはそのJITコンパイラを利用して行われる。JITコンパイラはメモリ上にネイティブコードを生成するが、そのコードが保持されているメモリ領域には当然ながら実行のために実行属性が付加される。そのため、DEPで実行を防止することはできない。また、JIT SprayingはHeap Sprayingのようにたくさんのコードをメモリへばらまく。メモリを多数の攻撃コードが占めることによって攻撃者はメモリのアドレスを細かく推測する必要がほとんどなくなる。そのため、JIT SprayingはDEPとASLRを同時に回避することができると言える。

JIT SprayingはJITコンパイラが意図していなかったアドレスから命令を実行することにより、JITコンパイラが生成したコードを攻撃コードとして実行する。本研究ではユーザレベルの実行監視により、JITコンパイラによって生成されたコードの実行時の命令アドレスを監視してJIT Sprayingを検知する。

2. JIT Spraying攻撃

JIT Spraying攻撃はJITコンパイラを利用した攻撃である。例えば、攻撃者は自身のウェブサーバへ巧みに細工したバイトコードやスクリプトコードを用意する。利用者がウェ

†1 東京大学
University of Tokyo

Vol.2011-DPS-146 No.46
Vol.2011-CSEC-52 No.46
2011/3/11

ブブラウザでそのサーバにアクセスすると、それらのコードは JIT コンパイラにより実行時にコンパイルされ、利用者のマシンのメモリへ多数配置される。その状態でバッファオーバーフロー攻撃などによりウェブブラウザの脆弱性が突かれ、ウェブブラウザの現在の実行位置が JIT コンパイルされたコードのあるメモリ領域へ移動させられると、そのマシンは攻撃者に制御を奪われる可能性がある。以上のように、JIT Spraying はバッファオーバーフロー攻撃などの脆弱性攻撃を補佐して攻撃を成功させる目的に使われる。

2.1 攻撃コードが生成される仕組み

通常、ウェブブラウザ上でスクリプト言語などを動かす場合には安全のためにできることが限られている。例えば、JavaScript をブラウザ上で動作させる場合、通常はウェブブラウザの動作しているマシンのファイルを自由にいじったり勝手に新たなプログラムを実行することはできない。このように、JIT コンパイルされたコードは普通に実行される限りにおいて、元の言語の制約に反することはできない。しかし、JIT Spraying 攻撃は任意のコード実行というクリティカルな危険をはらむ。それは JIT コンパイラの意図した正規の命令とは違う、意図してない不正な命令が実行されてしまうことによる。

図 1 は、ActionScript のソースコードである。これはウェブサーバなどに置かれる前に一度コンパイルされてバイトコードとなる。**図 2** はそのバイトコードが JIT コンパイラによってさらにコンパイルされた結果を示す。これらのコードは普通に解釈すればただひたすらにある変数へある数値を XOR していくものである。しかしここで、コンパイル後のコードの命令アドレスを一つずらしてみるとどうだろうか。そうすると、**図 3** のようになる。図 2 にあった命令が姿を変え、別の命令になってしまっている。CMP 命令を意味的に NOP と考えると、これは攻撃コードの一部としてよく使われる NOP スレッドであると言える。元々あった XOR 命令を示す 35 は CMP 命令のオペランドとなってしまい表に出てこない。そのため、元のコードは実行されず、JIT コンパイラが意図していなかった命令が実行されてしまう。文献[3] では、このような JIT Spraying に使われる命令を**図 4** のようにヘッダ、ペイロード、ボルトの三つの部分に分けている。ヘッダは JIT Spraying 攻撃として実行されるときに前の命令によって無効にされる部分であり、1 バイト以上の大きさがある。ペイロードはヘッダとボルトの間にあり、ここに攻撃者が実行させたい任意のコードであるシェルコードが置かれる。ボルトは元の命令の最後に位置し、次のペイロードにあるコードを実行するためにヘッダを無効にするという重要な役割を負っている。

```
var y = (
    0x3c909090 ^
    0x3c909090 ^
    0x3c909090 ^
    0x3c909090 ^
    0x3c909090 ^
    0x3c909090 ^
    ...
```

図 1 ActionScript のコード例
Fig. 1 Example of an ActionScript code

address	binary	disassemble
03470069	b8 9090903c	MOV EAX,3c909090
0347006e	35 9090903c	XOR EAX,3c909090
03470073	35 9090903c	XOR EAX,3c909090
03470078	35 9090903c	XOR EAX,3c909090
0347007d	35 9090903c	XOR EAX,3c909090
03470082	35 9090903c	XOR EAX,3c909090
...		

図 2 図 1 の JIT コンパイル結果
Fig. 2 JIT Compiled result of Fig. 1

3. 実行監視手法

本研究では、JIT コンパイルされたコードが JIT コンパイラが意図していなかったアドレスから実行されることにより攻撃コードとなる点に注目し、ユーザレベルでの実行時監視によって実行される命令アドレスを監視し、JIT Spraying 攻撃を検知する。ユーザレベルでの監視には、OS の書き換えを必要としないという利点がある。また、実行監視という手法ならば JIT エンジンの書き換えを行わずに JIT Spraying を防ぐことができる。今回対象とする OS は Windows、JIT エンジンは Internet Explorer 上で動く Flash Player である。実行監視には Windows で用意されているデバッグ用 API を利用する。

3.1 正規の命令アドレスの確定

Windows には VirtualProtect 関数というメモリの属性を変更するための API 関数があ

Vol.2011-DPS-146 No.46
Vol.2011-CSEC-52 No.46
2011/3/11

```
address     binary          disassemble
0347006a    90              NOP
0347006b    90              NOP
0347006c    90              NOP
0347006d    3c 35           CMP AL,35 ; semantic NOP
0347006f    90              NOP
03470070    90              NOP
03470071    90              NOP
03470072    3c 35           CMP AL,35 ; semantic NOP
03470074    90              NOP
03470075    90              NOP
...
```

図 3 図 2 を 1 バイトずらした結果
Fig. 3 Fig. 2 with 1 byte offset

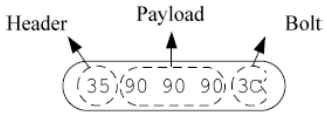

図 4 JIT シェルコードの構造[3)]
Fig. 4 Structure of JIT shellcode[3)]

る。**図 5** に VirtualProtect 関数のフォーマットを示す。Flash Player は JIT コンパイルしたコードをメモリに配置し、実行可能にするために VirtualProtect 関数を呼び出してそのコードのあるメモリ領域に読み込み可能かつ実行可能属性を付加する[4)]。図 5 に示したように VirtualProtect 関数の引数には対象とするメモリ領域の先頭アドレス (lpAddress) や領域のサイズ (dwSize)、付加する属性 (flNewProtect) などが指定されている。それらの引数を利用すると JIT コンパイラが意図した正規の命令アドレスがわかる。VirtualProtect 関数を捕捉して引数を調べ、付加する属性が読み込み可能かつ実行可能属性であった場合、さらに引数から対象とするメモリ領域の先頭アドレスを取得する。その先頭アドレスは JIT コンパイラが実行されることを意図した命令の先頭アドレスである。その先頭アドレスから引数で指定されている領域サイズの分だけ命令を逆アセンブルしていけば、そのメモリ領域

```
BOOL VirtualProtect(
  LPVOID lpAddress,    // コミット済みページ領域のアドレス
  DWORD dwSize,        // 領域のサイズ
  DWORD flNewProtect,  // 希望のアクセス保護
  PDWORD lpflOldProtect
                       // 従来のアクセス保護を取得する変数のアドレス
);
```

図 5 VirtualProtect 関数のフォーマット[5)]
Fig. 5 Format of VirtualProtect function[5)]

の正規の命令アドレスが全て判明する。

3.2 不正な命令の停止

VirtualProtect 関数により読み込み可能かつ実行可能属性を付加された領域には breakpoint を設置する。breakpoint で実行が停止したときには、次に実行する命令のアドレスが入っている EIP レジスタの値を調べる。EIP レジスタが正規の命令アドレスであればそのまま実行を続ける。EIP レジスタが正規の命令アドレスでなかった場合、それは不正な命令であるとして実行を停止させる。

4. breakpoint の設置手法

VirtualProtect 関数により読み込み可能かつ実行可能属性を付加された領域の全てのアドレスに breakpoint を設置すれば JIT Spraying による不正な命令の実行を検知することができるが、それではあまりにもアプリケーションの動作速度が遅くなりすぎてしまう。そのため、本研究では breakpoint の設置位置にある程度インターバルを持たせることとした。また、規則的なインターバルは安全性の低下を招くため、breakpoint 設置位置のランダム化を行った。

4.1 breakpoint の設置インターバル

攻撃を防ぐためにはシェルコードの実行が完了する前に不正な命令の実行を検知し、プログラムを停止できさえすればよいので、breakpoint を全ての命令について設置する必要はない。そのため、全ての命令に breakpoint を仕掛けるのではなく、ある程度のインターバルを置いて breakpoint を仕掛けることを考える。例えば、サイズが最小であると言われているシェルコードの大きさは 25 バイトである。[6)7)] このようなシェルコードの大きさを考慮して breakpoint を設置するインターバルを決めると良い。ただし、シェルコードを JIT Spraying を用いて実行させたい場合、図 4 で示したヘッダやボルトがオーバヘッドとして

Vol.2011-DPS-146 No.46
Vol.2011-CSEC-52 No.46
2011/3/11

かかるため、それを考慮して単純に計算すると 25 バイトのシェルコードは JIT シェルコードになると 42 バイトの大きさになる。さらに、今回示している形の JIT シェルコードでは 3 バイトの大きさのペイロードまでしか許されない。それにも関わらず、紹介した 25 バイトのシェルコードは 5 バイト命令を含んでいる。そのため、この 25 バイトのシェルコードはそのままの命令では JIT シェルコードにすることができない。JIT シェルコードにするためには、5 バイト命令を 3 バイト以下のいくつかの命令に分ける必要がある。したがって、実際には 25 バイトのシェルコードは 42 バイトよりも大きくなる。

インターバルを長くすればするほど breakpoint の設置箇所は少なくなる。そのため、breakpoint によって実行が止められる回数や breakpoint の着け外しの手間が減り、結果として実行速度は向上するはずである。しかし、インターバルを長くすればその分 JIT シェルコードの入り込む余地が生まれる。よって、インターバルの長さと安全性はトレードオフであると言える。

4.2 breakpoint 設置位置のランダム化

breakpoint をインターバルに設定した距離ごとの命令に素直に置くと、攻撃者は breakpoint の設置箇所の推定をすることが可能である。すると、攻撃者は breakpoint の設置されている命令をうまく回避するようなシェルコードを仕込んでくるかもしれない。そのような攻撃を防ぐため、**図 6** のように breakpoint を設置する命令はインターバルに設定されている距離の中にある命令からランダムに選択することとした。ただし、そのようにすると隣り合う breakpoint 間の距離は最大でインターバルの長さの二倍近く空くことがある。そのため、最小と思われる JIT シェルコードのサイズをそのままインターバルの長さに設定してしまうと JIT シェルコードが入り込む余地を与えてしまうことに注意されたい。例えば、breakpoint の設置インターバルを 50 とし、breakpoint を設置する命令がたまたま 1 バイトの長さであった場合、**図 7** のように最大で 98 バイトの JIT シェルコードが入り込む余地を与える。

5. 実　　装

実装は、Python2.5 と Python から簡単に Windows のデバッグ関連の API を操作することができる PyDbg というライブラリを用いて行った。PyDbg は PaiMei[8] というリバースエンジニアリングフレームワークのコンポーネントの一つである。

実装したプログラムの動作の概略を**図 8** のフローチャート図に示す。まず最初に、VirtualProtect 関数の開始アドレスに breakpoint を仕掛け、VirtualProtect 関数を検知できるようにする。VirtualProtect 関数を検知するとその引数を調べ、付加する属性が読み込み可能かつ実行可能であればその引数に指定されているメモリ領域の先頭アドレスとその領域のサイズを用いて先頭アドレスから逆アセンブルを行い、正規のアドレスを特定し記録する。そして、設定された breakpoint の設置インターバルの値を基にその領域に breakpoint を仕掛ける。実行を続け、そこで設置した breakpoint に到達した場合には、EIP の値を調べる。EIP の値が記録しておいた正規のアドレスになかった場合、不正な命令の実行とし、それを JIT Spraying 攻撃として検知する。EIP の値が正規のアドレスであった場合には、実行を続けるために一度その領域の breakpoint を全部取り除く。ただし、その breakpoint はまたあとで復帰させなければならないため、取り除いた breakpoint の情報を記録しておく。さらに実行を続け、取り除かれた breakpoint は次に別の breakpoint に到達した最に再設置される。

JIT Spraying 攻撃を検知できることの確認には、文献[9] で示される simple_sploit を用い、攻撃が検知されることを確認した。

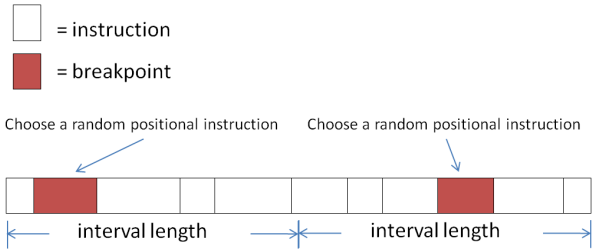

図 6 breakpoint 設置位置のランダム化
Fig. 6 Randomize breakpoint setup place

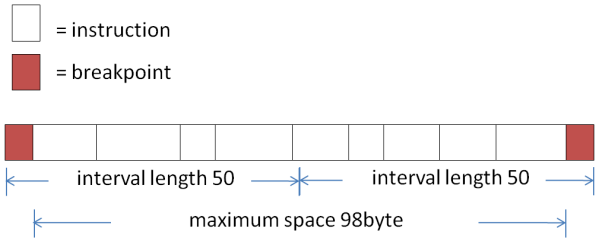

図 7 インターバル 50 で最大 98 バイトの余地ができる場合
Fig. 7 Case of maximum 98 byte space in interval 50

Vol.2011-DPS-146 No.46
Vol.2011-CSEC-52 No.46
2011/3/11

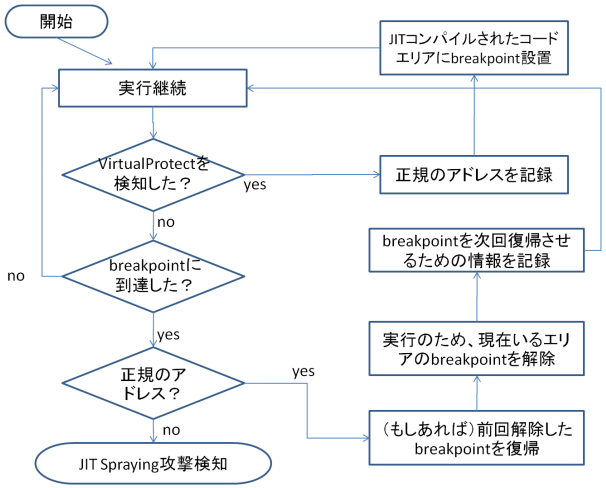

図 8 作成したプログラム動作のフローチャート図
Fig. 8 Flowchart of the program behaver

6. 評　　価

実験環境は、VMware Player 3.1.3 上の Windows XP SP2 32-bit 版である。実験に用いたウェブブラウザは Internet Explorer 8.0.6001.18702、JIT エンジンは Flash 10.0.42.34 である。

6.1 Flash Player の制限

Flash Player には実行時間についての制限があり、場合によっては実行途中に強制的に実行が停止してしまう。途中で実行が停止させられたかどうかは Flash Player の代わりに Flash Debug Player[10] を用いるとわかる。もし途中で実行が停止させられた場合、Flash Debug Player を使っていればエラーを知らせるウィンドウが出現する。

6.2 計測結果 1

まず最初に、Adobe の Flash Player を紹介しているウェブページ[11] において、ページを開いてから Flash アプリケーションのムービーが終了するまでの時間を計測した。結果を**表 1** に示す。interval length は breakpoint 設置インターバルの長さ、time はページを開いてから Flash アプリケーションのムービーが終了するまでの時間、time overhead は実行監視をしていないときと比較したときの time のオーバーヘッド、#breakpoints は breakpoint を設置した数、memory overhead は実行監視をしていないときと比較したときのメモリオーバーヘッドである。データは各 interval length について 5 回ずつ計測を行った平均である。前述の Flash Player の制限により、interval length が 0 から 200 のときには実行が途中で停止してしまった。250 のときには途中で停止してしまうときとムービーが終了するまで実行できることがあった。そのため、データは interval length が 300 のときからとなっている。interval length のカラムの no monitoring は実行監視を行っていない場合にかかる時間である。interval length を大きくしていくと実行速度が向上していくことがわかる。しかし、interval length が 500 のときにおいても time overhead は 180%もかかっている。interval leength が 300 から 500 になると time overhead は 376%から 180%と、約 1/2 になり、#breakpoints も 1933.4 個から 1006.8 個になり、約 1/2 となる。このことから、設置した breakpoint の数は実行にかかる時間のオーバーヘッドに比例しているように見える。メモリのオーバーヘッドもインターバルが長くなるに連れ減少するが、時間のオーバーヘッドと比較するとあまり大きな変化はない。

6.3 計測結果 2

前述の Flash Player の制限により、6.2 節で計測した Flash アプリケーションではインターバルの長さが短いと実行時間を計測できなかった。そこで、インターバルの長さが短いときのデータも得られるように、非常に単純な Flash アプリケーションを作成した。作成した Flash アプリケーションは、起動してから現在までの実行時間を表示するだけのものである。その結果が**表 2** である。表 1 と比べて、time の単位がミリ秒になっていることに注意してほしい。データは各 interval length について 5 回実行した平均の値である。この Flash アプリケーションの場合、interval length が 0、すなわち breakpoint の設置インターバルを設けなかったときにおいても Flash Player の制限にとらわれず、実行時間を計測することができた。interval length が 0 のときと 25 のときの time を比較すると、breakpoint の設置インターバルを設けることによる実行時間のパフォーマンス向上効果が非常に大きいことが見て取れる。しかし、それでも満足のいく実行時間になっているとは言いがたい。

7. 関 連 研 究

本研究が JIT Spraying 攻撃を検知する際の判断材料として JIT エンジンが意図していない不正なアドレスが実行されてしまうことに着目しているのに対し、文献[4] は、一般的な JIT シェルコードは 32 ビット即値オペランドを持つ mov 命令のあとに 32 ビット即値

Vol.2011-DPS-146 No.46
Vol.2011-CSEC-52 No.46
2011/3/11

表 1 Adobe Flash Player 紹介ウェブページの Flash ムービーの計測結果
Table 1 Measuring results of a Flash movie on introduction webpage of Adobe Flash Player

interval length(byte)	time(s)	time overhead(%)	#breakpoints	memory overhead(%)
no monitoring	5.0	0	0	0
300	23.8	376	1933.4	51.00
350	19.8	296	1616.6	49.52
400	18.0	260	1372.4	47.95
450	15.8	216	1164.4	46.59
500	14.0	180	1006.8	45.30

表 2 単純な Flash アプリケーションの計測結果
Table 2 Measuring results of a simple Flash application

interval length(byte)	time(ms)	time overhead(%)	#breakpoints	memory overhead(%)
no monitoring	19.0	0	0	0
0	13565.6	71298	2322.0	17.84
25	1815.8	9457	275.0	16.18
50	275.2	1348	122.0	16.08
100	209.4	1002	55.6	16.05
150	159.4	739	28.6	16.18
200	165.4	771	21.0	16.18
250	175.0	821	16.2	16.09
300	156.0	721	13.8	16.09
350	187.4	886	9.6	16.07
400	150.0	689	7.8	15.96
450	125.0	558	7.8	15.95
500	118.6	524	6.2	15.94

オペランドを持つなんらかの命令が継続するという形を取ることに着目している。JIT コンパイルされたコードのあるメモリ領域でそのような形のコードを発見したらそれを JIT Spraying 攻撃として検知する。また、この対策による Flash アプリケーションのパフォーマンスへの影響はほとんどないと記されている。しかし、この手法は JIT シェルコードの形をある程度限定してしまっているという点で本研究よりも柔軟性に欠ける。文献[4] の中でも JIT シェルコードの変形については記されているが、例えばそれよりもさらに多様な変形が発明されたときに、この手法では検知アルゴリズムをそれぞれのパターンに適用できるよう変化させていかなければならない。

文献[3] は、JIT エンジンレベルの JIT Spraying 攻撃対策である。本研究が JIT エンジンに手を加えていないのに対し、この研究では INSeRT(INstruction Space Randomization & Trapping) と呼ばれる仕組みを V8 JavaScript Engine に加えることによって改良を行っている。INSeRT は、生成コードのレジスタの割り当てをランダム化したり、即値のオペランド、パラメータ、ローカル変数をランダムに変形させる。さらに、exploit を捉えるための snippet を多数注入する。それにより、攻撃者は任意のコードを生成することが困難になる。この対策により生成コードのサイズは 5.9%上昇するが、攻撃の成功確率を 100 万分の 1 に減らすことができる。また、パフォーマンスオーバーヘッドは 5%以下に抑えられている。

8. おわりに

ユーザレベルの実行監視により、JIT コンパイルされたコードの実行時のアドレスを確認することで JIT Spraying 攻撃を検知することができた。ただし、ただ単純に JIT エンジンが読み込み可能かつ実行可能属性を付加したメモリ領域に breakpoint を仕掛けただけでは、アプリケーションの実行速度は耐え難く低速になってしまう。それについては、breakpoint の設置インターバルを設けることにより実行速度の高速化が図れる。しかし、それでも十分な実行速度が出ているとは言えない。よって、本研究の実行監視をユーザレベルで行う手法は十分に実用的な実行速度を出せなかったと結論づける。

9. 今後の課題と方針

我々は、JIT エンジンが意図していない不正な命令アドレスのコードが実行されてしまうことが JIT Spraying 攻撃の根本的な問題点であると考えている。さらに、そのような問題の解決を十分な実行速度の上で目指したい。ただし、JIT エンジンレベルでの対策の場合、その実装は各ベンダに依存してしまい根本的な解決にならないように思える。そのため、今後はカーネルレベルでの JIT Spraying 防御技術にも視野を広げて取り組んでいき、今回の経験を生かしていきたい。

参 考 文 献

1) Howard, M., Miller, M., Lambert, J. and Thomlinson, M.: Windows ISV Software Security Defenses (2010). `http://msdn.microsoft.com/en-us/library/bb430720.aspx`.
2) Blazakis, D.: Interpreter Exploitation: Pointer Inference and JIT Spraying, *Black Hat DC* (2010).
3) Tao, W., Tielei, W., Lei, D. and Jing, L.: Secure Dynamic Code Generation Against Spraying, *In Proceedings of the 17th ACM conference on Computer and communi-*

Vol.2011-DPS-146 No.46
Vol.2011-CSEC-52 No.46
2011/3/11

cations security, Vol.19, No.3, pp.738–740 (2010).
4) Bania, P.: JIT spraying and mitigations (2010). http://arxiv.org/abs/1009.1038v1.
5) Microsoft: VertualProtect 関数. http://msdn.microsoft.com/ja-jp/library/cc430214.aspx.
6) Gadaleta, F., Younan, Y. and Joosen, W.: BuBBle: A Javascript Engine Level Countermeasure against Heap-Spraying Attacks, *LNCS*, Vol.5965, pp.1–17 (2010).
7) PacketStorm: 25byte-execve (2009). http://packetstormsecurity.org/shellcode/25bytes-execve.txt.
8) Amini, P.: paimei - Project Hosting on Google Code. http://code.google.com/p/paimei/.
9) Sintsov, A.: Writing JIT Shellcode for fun and profit (2010). http://dsecrg.com/files/pub/pdf/Writing%20JIT-Spray%20Shellcode%20for%20fun%20and%20profit.pdf.
10) Adobe: Flex 3 - Using the debugger version of Flash Player. http://livedocs.adobe.com/flex/3/html/logging_03.html.
11) Adobe: rich internet applications — Adobe Flash Player. http://www.adobe.com/products/flashplayer/.

（URL 記載の文献はすべて 2011 年 2 月 13 日閲覧）