修士論文

JIT Spraying攻撃防止手法に関する研究

A Study on Prevention Methods for Combating JIT Spraying Attacks

指導教員　　松浦幹太　准教授

東京大学大学院 情報理工学系研究科 電子情報学専攻

48-106403　　市川 顕

平成24年2月7日提出

内容梗概

JIT コンパイラは従来のインタプリタと比較して高速に動作するため，近年様々なアプリケーションに採用されるようになってきている．しかし，JIT Spraying 攻撃という JIT コンパイラを悪用した攻撃が 2010 年に公表され，話題になっている．JIT Spraying 攻撃を利用すると，従来バッファオーバーフローなどの脆弱性を利用した攻撃の対策に効果的であったセキュリティ機構である Data Execution Prevention(DEP) と Address Space Layout Ramdomization(ASLR) を同時に回避することができてしまう．本論文では，JIT Spraying 攻撃を防止するための手段として二つの手法を提案する．

一つ目の提案は外部のプログラムから防御対象プログラムを実行監視する手法である．この手法では JIT コンパイラが実行されることを意図していたアドレスを正当なアドレス，そうでないアドレスを不正なアドレスとして区別し，不正なアドレスが実行されたときにはプログラムの実行を停止させる．この手法はオーバーヘッドが大きいが OS の修正も JIT エンジンの修正も必要としないという利点がある．

二つ目の提案は JIT コンパイルにより生成されたコードを実行する専用の子プロセスを用意する手法である．これにより親プロセスでは生成されたコードの格納領域に実行属性をつける必要がなくなり JIT シェルコードの実行が不可能となる．また，子プロセスでは生成コード実行時に実行属性を付加する必要があるが，プロセス間のメモリ空間は独立しているため親プロセスへの攻撃により子プロセスの JIT シェルコードが実行されることはなく，また子プロセスでも脆弱なコードを実行する可能性のあるスレッドを排除すれば JIT シェルコードは実行されない．この手法は JIT エンジンの修正が必要となる代わりに十分に実用的な速度で動作する．

目次

内容梗概 **1**

1 序論 **4**

1.1 メモリ破壊を利用した脆弱性攻撃とその対策 4
1.1.1 バッファオーバーフロー攻撃の歴史 4
1.1.2 バッファオーバーフロー攻撃の仕組み 4
1.1.3 対策技術 6
1.2 JIT Spraying 攻撃の出現 7
1.3 本研究の貢献 8
1.4 本論文の構成 8

2 JIT Spraying 攻撃 **9**

2.1 JIT コンパイラ 9
2.2 JIT shellcode 生成の仕組み 10
2.3 JIT Spraying 攻撃の例 12
2.4 DEP と ASLR の回避 13
2.5 Heap Spraying 攻撃との相違 13

3 関連研究 **15**

3.1 JIT Spraying 攻撃対策に関する研究 15
3.2 Heap Spraying 攻撃対策に関する研究と JIT Spraying 攻撃に対する有効性 16
3.3 提案手法への要件 17

4 外部のユーザレベルプログラムを用いた実行監視手法 (提案手法 1) **18**

4.1 提案手法 18
4.2 実装 18
4.2.1 正当な命令アドレスの確定 19
4.2.2 不正な命令の停止 19
4.2.3 ブレークポイントの設置手法 20
4.2.4 検知アルゴリズム概略 21
4.3 評価 22
4.3.1 Flash Player の制限 22
4.3.2 計測結果 1 23
4.3.3 計測結果 2 24
4.4 議論 25
4.4.1 メモリブレークポイントの利用 25

4.4.2 実装言語の変更 …… 26
4.4.3 JIT エンジン上への実装 …… 26
4.4.4 一部処理のハードウェア上への実装 …… 27
4.5 まとめ …… 27

5 生成コード実行プロセスを分離する手法 (提案手法 2) 28
5.1 プロセスの分離に関して …… 28
5.2 提案手法 …… 29
5.3 実装 …… 29
5.3.1 プロセス分離処理の実装 …… 30
5.3.2 データ共有処理の実装 …… 31
5.3.3 動作確認 …… 32
5.4 評価 …… 35
5.4.1 実験環境 …… 35
5.4.2 性能評価 …… 36
5.4.3 既存研究との比較 …… 39
5.5 議論 …… 41
5.5.1 .data セグメント, .bss セグメントの共有メモリ化 …… 41
5.5.2 コールバック関数の問題 …… 42
5.5.3 複数スレッドからの生成コード実行呼び出しの問題 …… 42
5.5.4 他の手法との組み合わせについて …… 43
5.5.5 ウェブブラウザに用いられるプロセス分離との組み合わせ …… 44
5.6 まとめ …… 44

6 結論 46
6.1 本論文のまとめ …… 46
6.2 展望 …… 47

謝辞 49

参考文献 50

発表文献 53

受賞リスト 54

Chapter 1　序論

1.1　メモリ破壊を利用した脆弱性攻撃とその対策

1.1.1　バッファオーバーフロー攻撃の歴史

メモリ破壊を利用した脆弱性攻撃の代表的なものにバッファオーバーフロー攻撃がある．バッファオーバーフロー攻撃を一躍有名にしたのが 1988 年に Robert Tappan Morris の放った Morris Worm と呼ばれるワームである．このワームは初めて世界で大きな被害をもらたしたワームであり，数千台のホストに侵入し，インターネットへの接続を何日にも渡って困難にした．このワームには複数の侵入手法があったが，その手法の一つが BSD Unix に実装されていた fingerd というデーモンのバッファオーバーフローの脆弱性を突くものであった [33][24]．さらに 2001 年には Code-Red と呼ばれるワームが大流行し，359000 台以上のコンピュータがそのワームに感染したと言われる．このワームはマイクロソフトの IIS ウェブサーバにあったバッファオーバーフローの脆弱性を利用していた [24]．近年においてもバッファオーバーフロー攻撃は利用され続けている．例えば Sony の PSP や Apple の iPhone は TIFF という画像形式を処理するライブラリのバッファオーバーフローの脆弱性 [1] により root 権限を奪われた [27]．また，2009 年に流行した Gumblar も Adobe Reader などのバッファオーバーフローの脆弱性を利用していた [22]．Gumblar はウェブ感染型のウィルスである．ウェブサイトを改ざんし，ウェブサイトを閲覧しただけで被害者の知らぬ間にマルウェアをダウンロードしてきて実行してしまう drive-by-download 攻撃を行うコードを仕込む．バッファオーバーフロー攻撃はその drive-by-download 攻撃のきっかけとして利用されることもよくある．

1.1.2　バッファオーバーフロー攻撃の仕組み

ここでは典型的なバッファオーバーフロー攻撃としてスタックセグメントを利用したバッファオーバーフロー攻撃について簡単に紹介する．バッファオーバーフローの脆弱性はアセンブリや C 言語など，プログラマが直接ポインタを操作することができるプログラミング言語により作られたソフトウェアの中に存在する[1]．

図 1.1 に典型的な脆弱性を持った C 言語のソースコードを示す．また，図 1.2 にはそのときに生成されるスタックフレームの例を示す．このソースコードではまず最初にバッファを 100 バイト分確保し，引数に与えられた文字列をそのバッファにコピーする．その際に使用されている strcpy という関数はバッファの境界チェックを自身では行

[1]プログラマが直接ポインタを操作することができないプログラミング言語でもその言語の処理系が C 言語で書かれていることは多く，どのようなプログラミング言語でもバッファオーバーフローの脆弱性は潜在的に存在するとも言える．

```
void main(int argc, char *argv[]){
    char buffer[100];
    strcpy(buffer, argv[1]);
}
```

図 1.1: 典型的な脆弱性を持った C 言語のソースコードの例

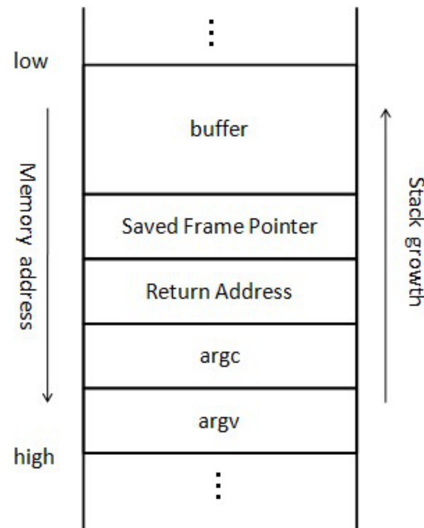

図 1.2: スタックフレームの例

わない．そのため，もしこのプログラムの引数に 100 バイト以上の文字列が指定されていた場合，このコードで用意されたバッファはオーバーフローを起こす．この例では，バッファオーバーフローが起こるとまずバッファに隣り合う Saved Frame Pointer が上書きされる．溢れが大きい場合，リターンアドレスや argc, argv, さらに前のスタックフレームも上書きされるかもしれない．ここで，リターンアドレスが上書きされた場合を考える．もしも悪意のある攻撃者がプログラムの引数にシェルコード[2]を入力し，さらにリターンアドレスをそのシェルコードが格納される領域を指すよう上書きしたとする．すると，関数が終了するときにプログラムはリターンアドレスを参照して関数が呼ばれた地点へプログラムの実行位置を戻すわけだが，リターンアドレスがシェルコードの格納位置に変更されてしまっているため，プログラムの実行位置はシェルコードの格納位置となる．それによりシェルコードが実行される．このような攻撃はスタックスマッシング攻撃とも呼ばれる．

同様にヒープセグメントの場合でも，ヒープセグメントに置かれたバッファを溢れさせてプログラムの制御に関わる変数を書き換えることができれば攻撃が可能である．

[2]攻撃者が最終的に被害者のマシンで実行させたい任意のコードであり，通常は機械語で記述される．

1.1.3　対策技術

バッファオーバーフロー攻撃などのメモリ破壊の脆弱性を利用した攻撃に対して，今まで様々な対策手法が考えられてきた．それらの中で実際によく使われているものとして，データ実行防止とアドレス空間配置のランダム化がある．

データ実行防止 (DEP)

多くのバッファオーバーフロー攻撃では，通常シェルコードは実行コードが置かれる場所ではなくスタックセグメントやヒープセグメントなどデータが置かれるメモリセグメントに格納される．データは通常実行される必要はないため，データが格納されるメモリ領域を実行不可能にすればシェルコードは実行されないはずである．そのようなことを可能にする仕組みは CPU の NX ビット (No eXecute bit) と呼ばれる機能やその機能のソフトウェアエミュレーションを利用して OS で実装されており，それは Windows OS の場合 Data Execution Prevention(DEP) などと呼ばれる [18][20]．また，Linux など他の多くの OS でも類似の技術が組み込まれている．DEP を有効にすると，プログラム内で明示的に実行属性を付加しない限りデータが格納される.stack, .heap, .bss, .data などのメモリセグメントは実行が不可能になる．ただし，DEP の弊害として，DEP の登場前に書かれたようなデータを格納する領域が実行可能であることを当然のように思っているプログラムは，データ領域に実行コードを生成してそれを実行させようとするとき明示的に実行属性を付加しないため，正常に動作を行うことができなくなる．

DEP は単体で用いても大きな効果はない．なぜなら，データ領域を実行不可能にしたとしてもコード領域に置かれている共有ライブラリなどのコードを実行させて攻撃が可能であるためである．その場合，攻撃者はリターンアドレスを実行させたい共有ライブラリの関数のアドレスを指すように上書きする．スタック上には関数の引数も置かれるため，攻撃者は所定の位置を上書きすることにより通常の関数呼び出しと同じように引数を付けて関数を呼ぶことも可能である．このような攻撃の場合，攻撃者はデータ領域には攻撃に必要なデータを置いただけであり実際にデータ領域を実行したわけではない．そのため，データ領域に実行属性が付いていなくとも攻撃ができる．このような攻撃は return-into-libc 攻撃などと呼ばれる．また，攻撃者はデータ領域にシェルコードを置き，さらメモリのアクセス保護属性を変更する関数を呼んでにシェルコードの置かれたデータ領域を実行可能にすることができる．すると，データ領域に置かれているシェルコードを実行することもできるようになる．

アドレス空間配置のランダム化 (ASLR)

攻撃者はシェルコードや任意の関数を実行するためにリターンアドレスをそれらのアドレスで上書きする必要がある．そのため，攻撃者はシェルコードが格納されるアドレスや関数の位置するアドレスを知っていなければならない．しかし，従来の環境ではプログラムは一度コンパイルしてしまえば使われるアドレスが変化しなかったため，それらのアドレスを把握することは容易であった．それらのアドレスの把握を困難に

するための技術がアドレス空間配置のランダム化である．アドレス空間配置のランダム化は Address Space Layout Randomization(ASLR) と呼ばれ，これも DEP と同様 OS で実装される [20]．ASLR が有効であると，マシンの起動時やプログラムの起動時などのタイミング[3]でメモリの仮想アドレスが変化する．それにより，攻撃者はアドレスの把握が困難になり，結果として攻撃ができなくなる．ただし，アドレスは完全にランダムになるわけではない．例えば，ある関数とある関数の相対的なアドレスは変わらない場合がある．そのため，どこか一箇所のアドレスが漏洩すると他の箇所のアドレスを知られてしまい攻撃が可能となるケースがある [12]．また，攻撃者は非常に大きな NOP スレッド[4]を用いたり，シェルコードを大量にばらまいたりして ASLR を回避することもある．これは Heap Spraying 攻撃などと呼ばれる．Java や JavaScript, ActionScript などの言語処理系を持つアプリケーションではデータを撒き散らすのが容易であるため，Heap Spraying 攻撃はそのようなアプリケーションを攻撃する際によく使われる．

DEP と ASLR を同時に用いた場合

DEP と ASLR はそれぞれの対策を個別に用いると，DEP は return-into-libc 攻撃，ASLR は Heap Spraying 攻撃により容易に破られる．しかし，DEP と ASLR を組み合わせることで，それらの攻撃は困難なものとなる．例えば DEP を容易に破ることのできる return-into-libc 攻撃は ASLR を同時に用いることで攻撃者が関数のアドレスを把握できなくなり攻撃は困難になる．また，ASLR を破ることのできる Heap Spraying 攻撃はシェルコードの置かれるデータ領域を実行不可能にすればシェルコードが実行できず攻撃が成功しなくなる．

1.2　JIT Spraying 攻撃の出現

近年，プログラミング言語を処理するエンジンに Just-In-Time (JIT) コンパイラが用いられることが多くなっている．JIT コンパイラは実行時にソースコードや中間言語をネイティブコードにコンパイルする．コンパイルしたコードを再利用することによってコードで頻繁に実行される部分の度重なる解釈が不要となったり，コンパイルによりある程度の最適化が可能となり，従来のインタプリタよりも高速に動作する．そのため，特に実行速度競争の激しいウェブブラウザなどの分野では，JavaScript などを動作させるためのエンジンに JIT コンパイラが採用される例が増加している．

　しかし，2010 年に Dionysus Blazakis により JIT コンパイラを悪用する攻撃手法である JIT Spraying 攻撃が発表された [9]．JIT Spraying 攻撃を利用すると，攻撃者は DEP と ASLR を同時に回避して再びバッファオーバーフローなどのメモリ破壊の脆弱性を利用した攻撃が可能となる．JIT Spraying 攻撃については 2 章で解説する．

---

[3] どのタイミングでアドレスが変わるかは ASLR の実装による．

[4] 何もしない NOP 命令を大量に並べたもの．これを利用すれば詳細なアドレスがわからなくともある程度の目星を付けるだけで攻撃ができる．

1.3　本研究の貢献

本論文では JIT Spraying 攻撃を防ぐための手法を二つ提案する．

一つ目に外部のユーザレベルプログラムを利用して対象プログラムの実行監視をし，JIT Spraying 攻撃を防ぐ手法を提案する．この手法では通常では実行されるはずのないメモリアドレスが実行されないか監視を行う．この手法は外部のユーザレベルプログラムとして実装できるため OS の修正も JIT エンジンの修正も必要としないことが利点として挙げられるが，実行時間のオーバーヘッドが大きい．

二つ目に JIT エンジンの実装方法としてプロセス分離を用いた手法を提案する．JIT コンパイラにより生成されたコードの実行時に専用のプロセスを生成し，そのプロセス上で生成コードを実行させる．この手法を用いると，脆弱なコードが実行される可能性のあるプロセス上で生成コードを実行しなくて済む．そのため，脆弱なコードが実行される可能性のあるプロセスでは生成コードが格納されているメモリ領域へ実行属性を付加することが不要となる．これにより脆弱性を突かれても JIT シェルコードの実行は不可能となる．この手法は JIT エンジンの修正を必要とするが，実行時間のオーバーヘッドは非常に少ないものとなった．

1.4　本論文の構成

以下，2 章では JIT Spraying 攻撃についての解説を例を交えて行う．3 章では JIT Spraying 攻撃対策と JIT Spraying 攻撃と類似した攻撃である Heap Spraying 攻撃の対策についての関連研究を紹介する．4 章では我々の提案手法の一つである外部のユーザレベルプログラムを用いて実行監視をし，JITSpraying 攻撃を防止する手法を紹介し，その実装と評価について述べる．5 章では我々の二つ目の提案手法として生成コードを実行するプロセスを JIT エンジンのメインのプロセスから分離する手法を紹介し，その実装と評価について述べる．6 章では本論文のまとめとして結論を述べる．

なお，4 章の提案手法の一部は査読無し国内会議 CSEC52(発表文献 [vi]) において発表した．また，5 章の提案手法の一部は国際会議 IWSEC2011 のポスターセッション (発表文献 [ii])，国際会議 ACSAC2011 の Works-in-Progress セッション (発表文献 [iii])，査読無し国内会議 CSS2011(発表文献 [iv])，査読無し国内会議 SCIS2012(発表文献 [v]) において発表した．

Chapter 2　JIT Spraying攻撃

本章では，JIT コンパイラを悪用した攻撃である JIT Spraying 攻撃について解説する．2.1 節では，JIT Spraying 攻撃に悪用される JIT コンパイラについて説明する．2.2 節では，JIT コンパイルによって生成されるネイティブコードがシェルコードとして動作してしまう仕組みについて解説する．2.3 節では，JIT Spraying 攻撃について例を交えて解説する．2.4 節では，JIT Spraying 攻撃が DEP と ASLR を同時に回避してしまう仕組みについて解説する．2.5 節では JIT Spraying 攻撃と似た攻撃である Heap Spraying 攻撃と JIT Spraying 攻撃の相違点について解説する．

2.1　JIT コンパイラ

JIT コンパイラは C 言語などのコンパイルに用いられる事前コンパイル方式と違い，実行時にプログラムをコンパイルする．そのため，一度コンパイルしたコードは再解釈の必要なく再利用でき，さらにコンパイル時に最適化を施すことが可能であるため，従来のインタプリタと比較して高速に動作する．ただし，事前コンパイル方式と違って実行時にコンパイルを行うため，最適化のオーバーヘッドがそのまま実行時間のオーバーヘッドへ繋がる．したがって，あまり高度で時間のかかる最適化を行うことはできない．また，実行時にコンパイルを行うため，事前にコンパイルする手間がないことやプラットフォーム間の違いを吸収するといった従来のインタプリタの利点も持つ．

JIT コンパイラは現在様々なアプリケーションで利用されている．例えば Java はその代表例である．Java はまずソースコードを事前コンパイルして中間言語であるバイトコードを生成する．実行時にはそのバイトコードを JIT コンパイルする．Java はクライアント，サーバ問わず様々なアプリケーションで使用されている．また，.net にも JIT コンパイラが採用されている．.net では C#や Visual Basic などのソースコードを共有の中間言語に事前コンパイルし，その中間言語を JIT コンパイルして実行する．近年では様々な Windows アプリケーションで.net が利用されている．さらに，.net は Windows だけでなく，オープンソース実装である mono などによって Linux など様々なプラットフォームで利用されている．ECMAScript の実装である JavaScript や ActionScript でも JIT コンパイル方式が採用されるようになってきている．多くの主要なウェブブラウザでは JavaScript が実行可能であり，それぞれのウェブブラウザが異なる JavaScript エンジンを搭載して動作速度を競っている．また，JavaScript はウェブブラウザだけでなく pdf リーダなどスクリプトが必要とされる様々なアプリケーションで利用されている．近年では，JavaScript はクライアントサイドに留まらず，サーバサイドで利用される例も目立ってきている．ActionScript は Flash や Air など主に Adobe 製品で使用されている．特に Flash は主要なウェブブラウザのほとんどに標準で搭載されている．

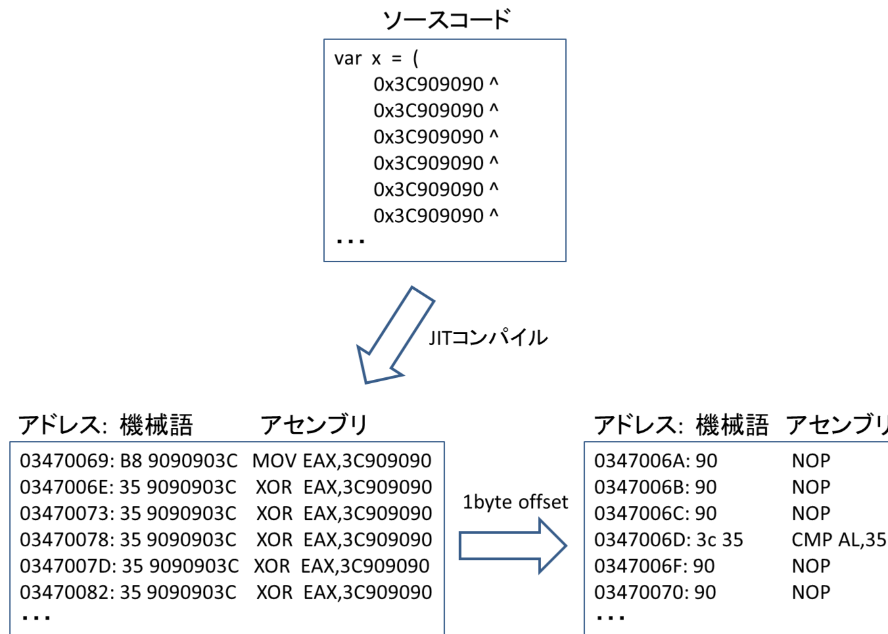

図 2.1: JIT コンパイラにより生成されたコードが JIT シェルコードとなる仕組み

JIT コンパイラによる高速化の例として，Internet Explorer で初めて JIT コンパイラが搭載されるようになった Internet Explorer9 は Internet Explorer8 と比較して，SunSpider JavaScript Benchmark で 16.1 倍高速である [37]．また，JIT コンパイラを搭載した Python の処理系である PyPy 1.7 は標準的な Python の処理系である CPython 2.7.2 と比較して 4.7 倍高速である [34]．

2.2　JIT shellcode 生成の仕組み

JIT コンパイラにより生成されたネイティブコードは，正常に実行される限りにおいて当然ながらコンパイル元の言語の制約によりできることが限られる．しかし，JIT Spraying 攻撃ではコンパイル元の言語の制約を無視し，システムに悪影響を与えるような攻撃者の任意のコード[1]を実行することが可能である．

本論文では JIT コンパイラが生成したシェルコードとなり得るネイティブコードを JIT シェルコードと呼ぶ．図 2.1 に JIT コンパイラの生成したコードが JIT シェルコードとなる例を示す．まず，図 2.1 の上方のコードは JIT コンパイルがなされる前のソースコードである．このコードでは “0x3C909090” という定数をいくつも XOR していき変数 x に代入している．これが JIT コンパイルされると図 2.1 の左のネイティブコードが生成される．このコードは一見したところソースコードと同様にただ単に XOR 演算を繰り返すコードに見える．しかし，このコードを 1 バイトだけずらして解釈すると図 2.1 の右のコードと見なせる．すると，本来オペランドであった機械語の

[1]利用できる個々の命令の長さに制限はあるかもしれない．

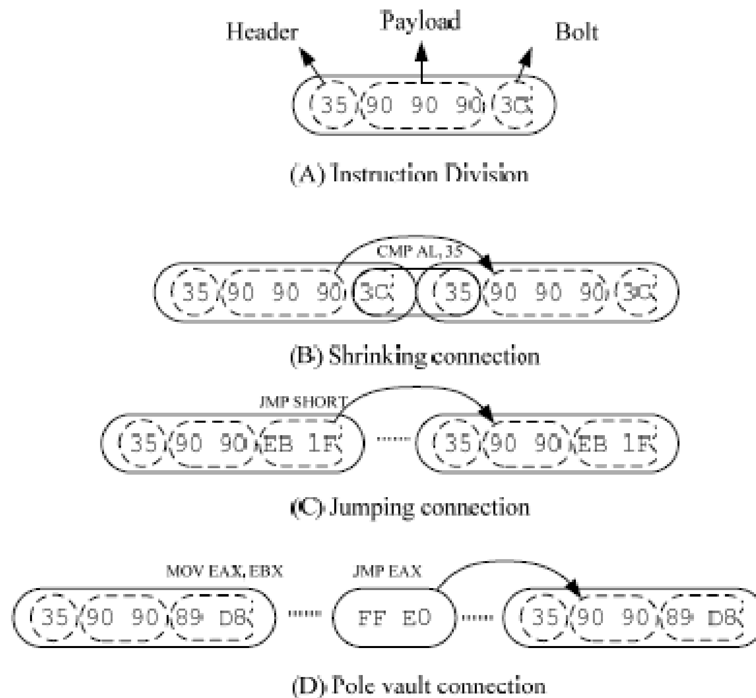

図 2.2: DCG-Spraying Model(文献 [35] より引用)

“3C” がオペコードとなって CMP 命令として解釈され，その CMP 命令のオペランドに本来 XOR 命令であった “35” が取られ，XOR 命令が消える．また，本来オペランドにあった “90” もオペコードとして解釈され NOP 命令となる．このコードの “CMP AL, 35” を特に意味のない命令と考えると，このコードは NOP スレッドとして解釈できる．この例で NOP となる部分を好きな命令に置き換えれば任意のシェルコードを作成することも可能である．このようにして JIT コンパイラが意図していなかった命令を実行することが可能である．

もし，この例の NOP スレッドの後にシェルコードが続いているとすれば，攻撃者はプログラムの実行アドレスをこの例ならば “0x347006A”, “0x347006B”, “0x347006C”, “0x347006D”, “0x347006F”, “0x3470070” などに変更することができればシェルコードの実行に成功する．逆に， “0x3470069” や “0x347006E” などに変更すると JIT コンパイラの意図していた命令が実行されてしまい，シェルコードは実行されない[2]．

また，JIT シェルコードはこのような形だけでなく，図 2.2 のようないくつかの形式が考えられる．文献 [35] によると Tao Wei らは図 2.2 に示すような形式の JIT シェルコードを用いるなどして GCJ 4.3.3，JDK 7 build b95 (HotSpot)，.Net 4.0.30319, SilverLight 4.0.50401.0，Adobe Flash 10.1.53.64，Firefox 3.6.3 (TraceMonkey)，Squirrelfish Extreme 2010-06-01 rev 60524，Chrome 5.0.375.70 (V8) が JIT Spraying 攻撃に脆弱であることを確認したとのことである．

[2] ただし，DOS という意味では攻撃は成功するかもしれない．

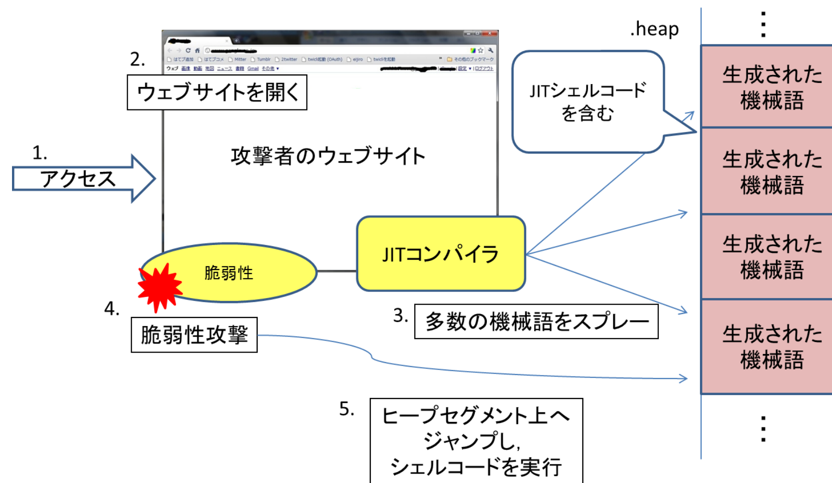

図 2.3: JIT Spraying 攻撃の例

2.3　JIT Spraying 攻撃の例

ウェブブラウザを介した JIT Spraying 攻撃の例を図 2.3 に示す．まず，被害者がウェブブラウザを用いて攻撃者のウェブサイトへアクセスする．すると，ウェブサーバから必要なファイルをダウンロードしてきてウェブブラウザがそれを読み込む．ウェブブラウザはダウンロードしてきた JavaScript や Flash などを実行する際に JIT エンジン[3]に処理を委託する．JIT エンジンは JIT コンパイラを用いてソースコードやバイトコードからネイティブコードを生成するが，攻撃者のウェブサイトからダウンロードしてきた JavaScript や Flash などは JIT コンパイルによってメモリの広大な領域に JIT シェルコードがばらまかれるような長大なコードが生成されるよう細工がされている．JIT コンパイラはその通りにネイティブコードを生成してヒープセグメント上にばらまく．その状態で，ウェブブラウザが何らかの不正な入力により脆弱性を突かれる．これにはバッファオーバーフローなどが利用されることとなる．それによりプログラムカウンタが書き換えられ，プログラムの実行位置がヒープセグメント上に移される．ヒープセグメント上には JIT コンパイラにより生成されたネイティブコードが待ち構えているが，2.2 節で説明したようにそのネイティブコードはアドレスによってはシェルコードとして解釈されることがあり，その場合にはユーザのマシンの制御が攻撃者のコードに奪われることもある．

　このような方法は，近年マルウェアに感染させる方法としてよく使われる drive-by download 攻撃に利用することも可能である．

[3]JIT エンジンはコンパイル時にブラウザに直接リンクされていたり，後からプラグインとして組み込まれていたりする．

2.4　DEP と ASLR の回避

JIT コンパイラは実行時に中間言語やソースコードをコンパイルしてネイティブコードを生成する．そのネイティブコードはメモリのヒープセグメントに置かれる．生成されたネイティブコードは実行しなければならないため，JIT エンジンはそのプログラム中で明示的にヒープセグメント上の生成したネイティブコードが置かれる位置へ実行属性を付加する．シェルコードはそのネイティブコードの中に埋め込まれている．そのため，JIT Spraying 攻撃においては DEP が有効になっていたとしても JIT シェルコードの格納位置に実行属性が付加されてしまっているため JIT シェルコードは実行可能である．

DEP と ASLR は組み合わせて使うからこそ効果が高いが，もし何らかの形で片方が無効化できればもう一方も回避できる場合が多い．JIT Spraying 攻撃の場合には上記のように DEP を回避しているので残った ASLR を回避するのは容易である．攻撃者はコンパイル後 JIT シェルコードとなるような中間言語やソースコードを JIT コンパイラに大量に入力すれば良い．そうすると，ヒープセグメント上の広大な領域が JIT シェルコードで埋められる．その後，攻撃者はバッファオーバーフロー攻撃などを用いてプログラムカウンタを書き換え，プログラムの実行位置を JIT シェルコードの格納位置にジャンプさせなければならないが，ヒープセグメント上の広大な領域が JIT シェルコードで占められている．そのため，攻撃者はメモリアドレスの情報を詳細に知らなくとも攻撃者の指定したアドレスがたまたま JIT シェルコードの格納アドレスに当たってしまう可能性は高い．

このように，ASLR によりメモリアドレスがランダム化されていても攻撃者はメモリアドレスの詳細を知ることなく攻撃をすることができてしまい，結果として DEP と ASLR を同時に回避することができてしまう．

2.5　Heap Spraying 攻撃との相違

Heap Spraying 攻撃は文字列や音楽や画像などのデータ部分にシェルコードを埋め込み，それをメモリ上に拡散させる．広大なメモリ領域にシェルコードを拡散させることにより，ASLR を回避することが可能になる．シェルコードをメモリ上にばらまくという点は JIT Spraying 攻撃と似ているが，JIT Spraying 攻撃との大きな違いはシェルコードが埋め込まれる部分が本来コードとして機能するわけではないという点である．そのため，Heap Spraying 攻撃でシェルコードが埋め込まれるメモリ領域には本来実行属性は付ける必要がなく，Heap Spraying 攻撃は ASLR に加えてさらに DEP が有効に機能していれば防御が可能である[4]．それに比べ，JIT Spraying 攻撃は DEP と ASLR を同時に回避されてしまうため，さらなる対策が必要とされる．

ただし，Heap Spraying 攻撃はデータとして使われるオブジェクトにシェルコードを埋め込むため，従来のインタプリタのような動的コード生成を行わないアプリケーションでも可能である．一方 JIT Spraying 攻撃は実行コード部分へシェルコードを埋

[4] アドレス情報の漏洩により ASLR が回避されないと仮定する．

め込むため，JIT Spraying 攻撃が可能なアプリケーションは JIT コンパイラのような動的コード生成を行うアプリケーションに限定される．しかし，動的コード生成は様々なアプリケーションで行われている．現在よく攻撃が行われるウェブブラウザは非常に複雑でバッファオーバーフローなどの脆弱性も多く，主要なウェブブラウザには大抵 JavaScript エンジンに JIT コンパイラが採用されている．さらに，そのウェブブラウザ上でプラグインとして動かすことの多い Java や Flash の実行環境にも JIT コンパイラが使用されている場合がほとんどである．

Chapter 3　関連研究

本章では JIT Spraying 攻撃対策や Heap Spraying 攻撃対策に関連する先行研究について紹介する．まず，3.1 節で JIT Spraying 攻撃対策に関する先行研究について紹介する．3.2 節では JIT Spraying 攻撃に類似した攻撃である Heap Spraying 攻撃への対策に関する研究について紹介し，JIT Spraying 攻撃への有効性について考察する．

3.1　JIT Spraying 攻撃対策に関する研究

既にいくつかある JIT Spraying 攻撃を防ぐための研究について紹介する．本研究と既存研究との詳細な比較については 5.4.3 節で述べる．JIT Spraying 攻撃に関する研究はまだ日が浅く，それぞれが本質的に異なる手法で JIT Spraying 攻撃の防御を試みている．

JIT シェルコードの検知　Piotr Bania は JIT シェルコードを検知するようなアルゴリズムを提案している [7]．このアルゴリズムは 32bit 即値をオペランドにとる mov 命令を見つけると，さらにそのあと 32bit 即値をオペランドにとるなんらかの命令が指定回数以上継続しているかどうかを検査する．もし指定回数以上の命令が継続していた場合にはそれを JIT シェルコードとして検知するというものである．ただし，全ての JIT シェルコードがこのような形式をとるわけではないため，このアルゴリズムでは検知できない JIT シェルコードのパターンも存在する．もし，他の JIT シェルコードのパターンにも順次対応していくとしても，全ての JIT シェルコードのパターンを網羅するのは困難だと思われる．

不正なシステムコール呼び出しの検知　Willem De Groef らによる JITSec[11] は Linux へカーネルモジュールを組み込むことにより，システムコールを呼び出すインターフェイスへ呼び出し元アドレスを調べる処理を加える．システムコールの呼び出し元が，通常の実行コードが格納されている.text セグメントや共有ライブラリであったら呼び出しを許可するが，ヒープセグメントなどの通常ではデータを格納するメモリセグメントからの呼び出しであったらそれは JIT コンパイルを悪用した攻撃である可能性があるとしてプログラムを強制終了させる．ただし，この手法ではもし JIT コンパイラが生成したコードやその他動的コード生成を行うアプリケーションが生成したコードが直接システムコールを呼ぶ場合に攻撃と誤検知され，正常に動作できないことが予想される．

生成コードの無害化　Tao Wei らは INSeRT[35][36] という防御手法を提案している．この手法は，JIT エンジンを修正し，吐き出されたネイティブコードで使われる即値がラ

ンダムな値から生成されるようにしたり，使用されるレジスタをランダム化したりする．また，関数の引数やローカル変数の配置もランダム化する．これにより，JIT シェルコードの構築が困難になる．さらに，生成コードが不正に実行されたときのため，それを検知するためのトラッピングスニペットを挿入し，不正な実行を報告する．生成コードの変更が必要となるため，生成コードのサイズが増加することや JavaScript コードの実行性能に数%のオーバーヘッドが生じてしまうこと，開発者にとっては実装の手間やデバッグ時に生成コードの挙動がわかり難くなることが欠点として挙げられる．

不要な実行属性の排除　Ping Chen らによる JITDefender[10] という手法は，生成されたネイティブコードを実行するときにだけに生成コードが格納されたメモリ領域に実行属性を付加し，それ以外のときには実行属性を付加しないという非常にシンプルな手法である．これにより生成コード実行時以外には JIT シェルコードは実行できなくなる．しかし，その生成コード実行時に限り生成コードが格納してある領域に実行属性を付加しなければならないため，そのタイミングを狙って攻撃されてしまう可能性がある．また，この対策手法は行われるのが当たり前のように思えるかもしれないが，多くの JIT エンジンが JIT コンパイルされたコードの格納領域に読み書き実行が可能である RWX のパーミッションを終始付けたままにしているのが現状である [30]．

3.2　Heap Spraying 攻撃対策に関する研究と JIT Spraying 攻撃に対する有効性

本節では Heap Spraying 攻撃対策に関する研究について紹介するが，Heap Spraying 攻撃は JIT Spraying 攻撃と同様にメモリの広範囲にシェルコードを撒き散らす攻撃である．そのため，Heap Spraying 攻撃対策研究が JIT Spraying 攻撃においても有効である可能性があることが考えられる．そのため，それぞれの手法が JIT Spraying 攻撃に適応可能かどうかについても議論する．

シェルコード (NOP スレッド) の検知　Paruj Ratanaworabhan らによる NOZZLE[29] はヒープセグメント上のオブジェクトを逆アセンブルして NOP スレッドを探す．しかし，この手法をそのまま JIT Spraying 攻撃対策に用いても JIT シェルコードが解釈されずに JIT コンパイラが意図していた本来の意味のコードだけが解釈されてしまい，JIT シェルコードを素通りしてしまうと考えられる．スキャンしているオブジェクトが機械語として解釈されたらさらにそこから 1 バイト, 2 バイトとアドレスをずらした場合についても逆アセンブルして NOP スレッドかどうかの判定を行えば JIT シェルコードを検知できるかもしれないが，その分オーバーヘッドが増えることとなるであろう．

Manuel Egele らの研究 [13] では，ヒープセグメント上のオブジェクトというよりも JavaScript インタプリタによって割り当てられた文字列に限定してシェルコードの

スキャンを行う．JIT Spraying 攻撃の場合にはシェルコードが埋め込まれるのは実行コードであるため，この手法では JIT Spraying 攻撃に対応することはできない．

一部文字列の CPU 割り込み発生コードへの置き換え　Francesco Gadaleta らの BuBBle[15] では，JavaScript により生成される文字列の一部分を INT3 割り込み命令を発生させる "CC" というコードへ置き換える．また，文字列が使用されるときには置き換えられた文字列を元に戻し，文字列の使用が終わったら再び文字列の一部分を割り込み命令を発生させるコードへ置き換える．これにより，攻撃者が文字列に埋め込んだシェルコードを実行させようとしても途中で割り込みが発生し，Heap Spraying 攻撃は検知される．JIT Spraying 攻撃の場合，シェルコードが埋め込まれるのは実行コード部分であるため，文字列を割り込み発生コードへ置き換えるこの手法では JIT Spraying 攻撃へ直接対応することはできない．しかし，割り込み発生コードを生成コードへ埋め込むという応用は可能である．そのような手法は本論文の 4 章で紹介する提案手法 1 や 3.1 節で紹介した INSeRT でも一部使用されている．

不正なシステムコール呼び出しの検知　Fu-Hau Hsu らの HSP[21] は，glibc を修正してシステムコールの呼び出し元のアドレスを一つに定める．また，システムコールが呼び出されたときにリターンアドレスを検証できるよう OS をを修正する．リターンアドレスがその定められたアドレスでなかったら不正なシステムコール呼び出しとして攻撃を検知する．この手法は 3.1 節で紹介した JITSec と類似している．Heap Spraying 攻撃の場合はシェルコードが埋め込まれるのは文字列などのデータであり，これは通常実行されるものではない．そのため，正常な実行において文字列などのデータからは直接システムコールが呼ばれることはない．したがって，Heap Spraying 攻撃を誤検知することはない．しかし，JIT Spraing 攻撃の場合はシェルコードが埋め込まれるのは生成された実行コード部分であり，これは正常な実行であっても JIT コンパイラの実装によってはシステムコールを直接呼ぶ可能性がある．つまり，この手法を JIT Spraying 攻撃に適用すると正常な実行を攻撃として誤検知することが考えられる．

3.3　提案手法への要件

本章で紹介した JIT Spraying 攻撃対策に関する研究では，それぞれの手法に欠点があった．それらを踏まえると，新たに提案する手法では

- JIT シェルコードのパターンに依存しない
- 生成コードからのシステムコールの呼び出しを許可する
- 生成コードに変更を加えない
- 脆弱な時間が発生しない

といった項目が達成されることが望ましい．

Chapter 4 外部のユーザレベルプログラムを用いた実行監視手法 (提案手法 1)

本章では，JIT Spraying 攻撃を防止するために我々の提案した一つ目の手法である外部プログラムを用いて実行監視を行う手法について述べる．

4.1 提案手法

JIT コンパイラにより生成されたコードは，JIT コンパイラが実行を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シェルコードとして解釈されてしまう．我々の提案手法は，JIT コンパイラが実行されることを意図していなかった不正なアドレスが実行されたとき，それを検知し，JIT Spraying 攻撃を止めることである．プログラムの実行中のアドレスを検査するため，外部のユーザレベルプログラムを用いて JIT Spraying 攻撃から守りたいアプリケーションにアタッチする．また，どのアドレスが不正なアドレスであるかを知る必要もある．そのためには，メモリのアクセス保護属性を変更する関数を捉える．メモリのアクセス保護属性を変更する関数の引数には属性を変更するアドレスの位置や付加する属性が指定されているはずである．もし実行属性を付加する場合には，その実行属性が付加されるメモリ領域に JIT コンパイラにより生成された実行コードが格納されていると考えられる．JIT コンパイラが実行を意図していた正当な命令のアドレスの一つは通常その領域の先頭アドレスであることが予想されるため，そこから逆アセンブルを行うことによりその領域の全ての正当な命令のアドレスがわかる．不正な命令のアドレスはその領域の正当な命令のアドレス以外の全てのアドレスであるため，その領域へブレークポイントを仕掛けておいてその不正なアドレスを実行しようとした際にはプログラムを停止すればよい．

この手法では，ユーザレベルプログラムを用いるため，OS の書き換えを必要としない．また，外部のプログラムを用いて対象プログラムをアタッチするという形で実行監視を行うため，JIT エンジンの書き換えも必要ない．

4.2 実装

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[28] を利用する．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[6] 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．

```
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 4.1: VirtualProtect 関数のフォーマット [23]

4.2.1　正当な命令アドレスの確定

Windows には VirtualProtect 関数というメモリのアクセス保護属性を変更するための API 関数がある．図 4.1 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でかつ実行可能となる属性を付加する [7]．図 4.1 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) などを指定することになっている．これらの引数を調べることにより JIT コンパイラが意図した正当な命令アドレスがわかる．その手順を次に示す．まず，VirtualProtect 関数を捕捉して引数を取得し，指定されたアクセス保護属性を調べる．その属性が読み込み可能かつ実行可能であった場合，JIT コンパイラにより生成されたコードの格納領域に実行属性を付けようとしていることが考えれる．そのため，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調べる．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．また，その領域のサイズも VirtualProtect 関数の引数によって指定されているため簡単に知ることができる．よって，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くことにより，その生成コード格納領域の正当な命令アドレスが全て判明する．

4.2.2　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である属性を付加された領域にはソフトウェアブレークポイントを設置する[1,2]．そのブレークポイントで実行が停止したときには，EIP レジスタの値を調べる．EIP レジスタには次に実行する命令のアドレスが入っている．EIP レジスタが正当な命令アドレスであればそのまま実行を続

[1]ハードウェアブレークポイントは通常の CPU では数個しか設置できないため本手法では使用できない．

[2]このとき，不正な命令のオペコードだけをブレークポイント (CC) にすれば正常な実行のときにはブレークポイントで止まらずに実行できると思うかもしれない．しかし，不正な命令のオペコードは正当な命令のオペランドである．そのため，確かに止まらずに実行はできるかもしれないが，少なくとも計算がおかしくなる (最悪クラッシュする)．したがって，正当な命令のオペコードもブレークポイントにし，正常な実行のときにはブレークポイントを付け外ししながら実行をしなければならない．

ける．EIP レジスタが正当な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．

4.2.3　ブレークポイントの設置手法

VirtualProtect 関数により読み込み可能かつ実行可能である属性を付加された領域の全てのアドレスにブレークポイントを設置すれば JIT Spraying 攻撃による不正な命令の実行を確実に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，本手法ではブレークポイントの設置位置にある程度のインターバルを持たせることにした．また，規則的なインターバルは安全性の低下を招くため，ブレークポイント設置位置のランダム化を行った．

ブレークポイント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，ブレークポイントを全ての命令について設置しなくとも多くの攻撃を防ぐことが可能である．そのため，全ての命令にブレークポイント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ブレークポイント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 [15][25]．このようなシェルコードの大きさを考慮してブレークポイント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、JIT シェルコードの形式が図 2.1 と同様の形であるときには図 2.2 の (A) で示されているヘッダやボルトがシェルコードのサイズにオーバヘッドとしてかかるため，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，図 2.1 で示している形の 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 JIT シェルコードにすると 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ブレークポイントの設置箇所は少なくなる．そのため，ブレークポイントによって実行が止められる回数やブレークポイントの付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．

また，シェルコードが実行の途中でジャンプを伴うものであると，運悪くブレークポイントの位置を飛び越えて実行が行われてしまう可能性がある．そのため，あらゆる形式の JIT シェルコードを防ぐことができるのはインターバルが 0 バイト，つまり生成コードの全ての命令へブレークポイントを仕掛けたときだけである．仮にサイズが最小の JIT シェルコードを 42 バイトとし，さらにシェルコードがジャンプを伴うも

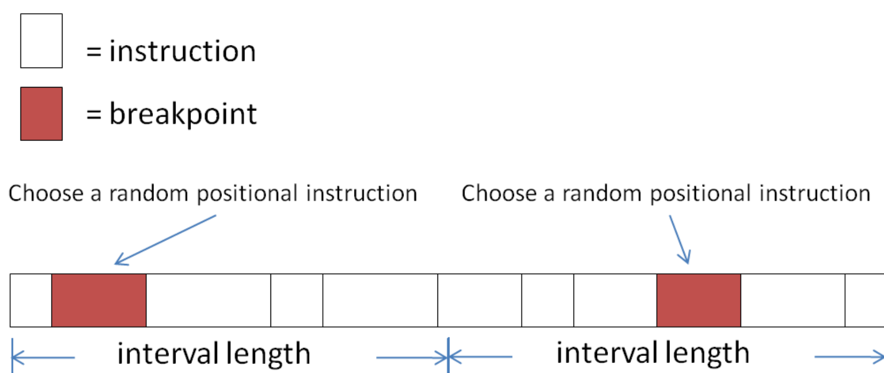

図 4.2: ブレークポイント設置位置のランダム化

のでないと仮定すると，インターバルが 42 バイト以下であればその JIT シェルコードは確実に検知することができる．

ブレークポイント設置位置のランダム化

ブレークポイント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，攻撃者はブレークポイントの設置箇所の推定をすることが可能である．すると，攻撃者はブレークポイント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，ブレークポイントの設置位置をランダム化する必要がある．本実装では，図 4.2 のように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一つの命令を選択し，その命令へブレークポイントを設置することにした[3]．ただし，そのようにすると隣り合うブレークポイント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，ブレークポイントの設置インターバルを 50 バイトとし，ブレークポイント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 4.3 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可能性がある．

4.2.4　検知アルゴリズム概略

実装したプログラムの動作の概略を図 4.4 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ブレークポイントを仕掛け，VirtualProtect 関数を捕捉できるようにする．VirtualProtect 関数を捕捉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，その

[3]本実装で命令にブレークポイントを設置するという意味は，例えば命令長が 3 バイトの命令にブレークポイントを設置する場合にはその 3 バイトをブレークポイントに置き換えるということであり，オペコードだけをブレークポイントにするのではない．

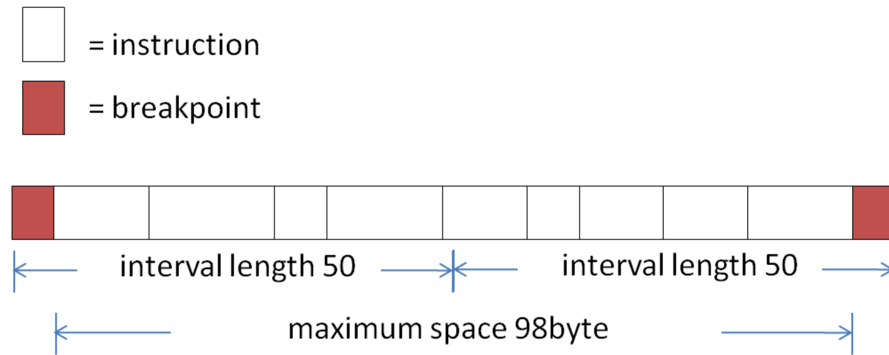

図 4.3: インターバル 50 で最大 98 バイトの余地ができる場合

領域の全ての正当なアドレスを特定し記録する．そして，設定されたブレークポイントの設置インターバルの値を基にその領域にブレークポイントを仕掛ける．実行を続け，そこで設置したブレークポイントに到達した場合には，EIP レジスタの値を調べる．EIP レジスタの値が記録しておいた正当なアドレスのリストに存在しなかった場合，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 レジスタの値が正当な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ブレークポイントを全部取り除く．ただし，そのブレークポイントはまたあとで復帰させなければならないため，取り除いたブレークポイントの情報を記録しておく．さらに実行を続け，取り除かれたブレークポイントは次に別のブレークポイント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 [32] 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．

4.3　評価

本手法を実装したものについてパフォーマンス計測を行った．実験は VMware Player 3.1.3 上の Windows XP にて行った．詳細な環境を表 4.1 に示す．また，既存研究との比較については 5.4.3 節において本章の提案手法が提案手法 1 として述べられている．

4.3.1　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ことがある．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[4] 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．

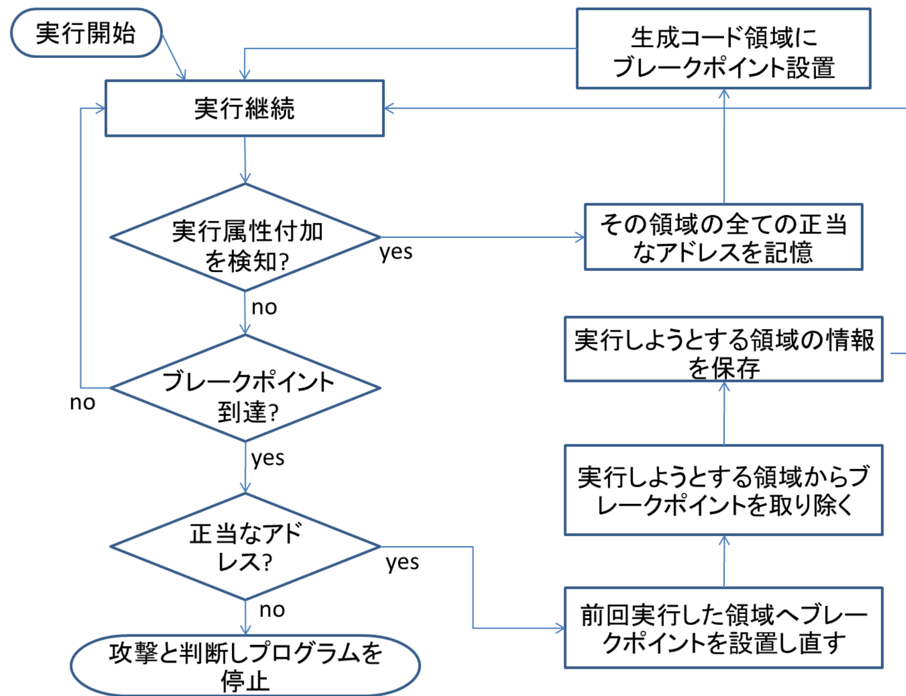

図 4.4: 作成したプログラム動作のフローチャート図

4.3.2　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 [5] 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 4.2 に示す．interval length はブレークポイント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 はブレークポイントを設置した数，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使用量の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 length が 300 バイトから 500 バイトになると time overhead は 376%から 180%と，約 1/2 になり，#breakpoints についても 1933.4 個から 1006.8 個と，約 1/2 となる．このことから，設置したブレークポイントの数は実行にかかる時間のオーバーヘッドにほぼ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

表 4.1: 実験環境

CPU	Intel Core 2 Duo P8800 (2.66GHz x2)
仮想化ソフト	VMware Player 3.1.3
OS	Windows XP SP2 32-bit
割り当てプロセッサコア数	2 個
割り当てメモリ	1024MB
ウェブブラウザ	Internet Explorer 8.0.6001.18702
JIT エンジン	Flash Player 10.0.42.34

表 4.2: Adobe Flash Player 紹介ウェブページの Flash ムービーの計測結果

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

ない．

4.3.3　計測結果 2

前述の Flash Player の制限により，4.3.2 節で計測に利用した Flash アプリケーションではインターバルの長さが短い場合の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，パフォーマンスの計測を行った．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 4.3 である．表 4.2 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，すなわちブレークポイント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 のときが本実装では最も安全性が高いわけであるが，実行時間のオーバーヘッドは 71298%と非常に大きなものとなってしまっている．

表 4.3: 単純な Flash アプリケーションの計測結果

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

interval length が 0 のときと 25 のときの time を比較すると，ブレークポイント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，インターバルを設けたとしても満足のいく実行速度になっているとは言いがたい．

4.4　議論

我々の提案手法は OS の修正や JIT エンジンの修正の必要がないということが大きな利点となっている．また，我々の提案手法にはフォルスポジティブはなく，フォルスネガティブについてもブレークポイントの設置インターバルの長さによっては発生しないという利点もある．しかし，大きな欠点となったのが実行速度の遅さである．他にいくら素晴らしい点があっても，実際に使用するときに実行速度が遅くては使い物にならない．ここでは主に本章の提案手法の実行速度を向上する方法について議論する．

4.4.1　メモリブレークポイントの利用

Windows ではメモリ領域に PAGE_GUARD というアクセス保護属性を設定すると，そのメモリ領域の各ページはガードページとなり，アクセスがあった際に例外が発生する [23]．そのため，メモリ領域に PAGE_GUARD というアクセス保護属性を設定してブレークポイントのように利用することができる．本論文ではそれをメモリブレーク

ポイントと呼ぶ．メモリのアクセス保護属性は CPU のメモリ管理ユニットにより 1 ページ (4096 バイト) ごとに管理されている．そのため，メモリブレークポイントを利用した場合には，もしブレークポイントの設置にインターバルを設けるとしたら最低でも 4096 バイトの間隔ができることになる．よって，メモリブレークポイントの設置にはインターバルを設けるべきではない．

我々はメモリブレークポイントを利用した実装を試作し，4.3.2 節で利用した Flash Player を紹介しているウェブページのアプリケーションと 4.3.3 節で利用した単純な Flash アプリケーションについて実行時間計測を行った．しかし，Flash Player を紹介しているウェブページのアプリケーションについては残念ながら Flash Player の制限により最後まで実行できず，実行時間を計測することはできなかった．また，単純な Flash アプリケーションについては実行時間を計測することができ，ブレークポイントの設置インターバルが 0 のときにおいて実行時間は 212.4ms となった．これを表 4.3 のソフトウェアブレークポイントを利用したときの結果と比較すると，ソフトウェアブレークポイントの場合のブレークポイントの設置インターバルを 100 バイトに設定した場合と同等程度の実行速度となる．安全性としてはソフトウェアブレークポイントを用いた場合のインターバル 0 バイトの場合と同じであり，その場合には実行時間が 13565.6ms かかっているため，メモリブレークポイントはソフトウェアブレークポイントを用いるより高速に動作すると言えるであろう．しかし，Flash Player を紹介しているウェブページが動作速度の問題で動作しなかったことから，その実行速度が十分であるとは言いがたい．

4.4.2　実装言語の変更

我々が実装に使用したプログラミング言語は Python である．各プログラミング言語のベンチマーク実行速度比較ができる The Computer Language Benchmarks Game[3] というウェブサイトにて，X86 Ubuntu Intel Q6600 quad-core を用いたときの Python3 と C GNU gcc の速度を比較すると，Python の方が 39 倍程度遅いということがわかる．そのため，もし C 言語などの高速なプログラミング言語を使用すれば実行速度がいくらか向上するはずである．しかし，我々の実装の実行速度の遅さの主要因はブレークポイントにより幾度もプログラムが停止することやその際にブレークポイントの付け外しの処理が行われるためと考えられる．そのため，実装言語を変更したからといってそれほど大きな速度向上がなされるとは限らない．

4.4.3　JIT エンジン上への実装

さらなる速度向上方法として，JIT エンジン上へ実装することを考えてみる．この場合には，外部プログラムから JIT エンジンを含むプロセスへアタッチする必要がないため，Windows のデバッグ API を使用して外部のプログラムを操作する分のオーバーヘッドがなくなる．そのため，実装言語を C 言語などの高速なプログラミング言語に変更することと組み合わせればいくらかの速度向上が望めるかもしれない．しかし，JIT エンジン上へ実装したからといって生成コード実行時のブレークポイントの付け

外しなど時間がかかるであろう処理をすることは同じであり，劇的な速度向上は望めないかもしれない．また，JIT エンジン上へ実装する場合，当然ながら JIT エンジンの修正が必要になることが大きな欠点となる．

さらに，JIT エンジン上へ実装し，ブレークポイントにソフトウェアブレークポイントではなくメモリブレークポイントを使用する場合を考えて見る．この場合には，JITDefender とほとんど差異のない手法となる．JITDefender が生成コード格納領域から実行属性を排除するのに対し，メモリブレークポイントでは生成コード格納領域をガードページとする．ただし，実行属性の排除ではなくメモリブレークポイントを用いた場合には，DEP が無効な環境であっても JIT Spraying 攻撃を検知可能であるという利点がある．JITDefender が実用的な速度で動作することから，同様にメモリブレークポイントを利用した場合にも実用的な速度で動作することが予想される．

4.4.4　一部処理のハードウェア上への実装

もう一つの実装方法として，一部処理をハードウェアで処理してもらう方法を考える．もしハードウェアが正当な命令の先頭アドレスと領域を指定する命令を持ち，その命令で指定された領域について全ての正当な命令を把握し，正当な命令のオペランドがオペコードとして不正に実行されることを禁止する機構を持てば，ソフトウェア側でブレークポイントの付け外しなどの処理をする必要がなくなり，高速な動作が可能になると考えられる．ただし，現状でそのような処理を行なってくれるハードウェアは存在しないと思われる．

4.5　まとめ

本章では，外部のユーザレベルプログラムを用いて実行監視を行うことにより JIT Spraying 攻撃を検知し実行を停止する手法を提案した．メモリのアクセス保護属性を変更する関数を捕捉し，そこから正当な命令アドレスと不正な命令アドレスを割り出す．それから，JIT コンパイラにより生成された実行コードが格納されている領域へソフトウェアブレークポイントを仕掛ける．そして，ブレークポイントでプログラムが止まったときに不正な命令アドレスを実行しようとしていた場合にはプログラムの実行を停止するよう実装を行った．本章の提案手法は OS や JIT エンジンの修正の必要がないことが大きな利点であるが，実行時間の計測により実行速度のオーバーヘッドが大きいという欠点があることがわかった．

また，3.3 節で述べた提案手法に求められる項目のうち，“脆弱な時間が発生しない” という項目が達成できなかった．本章の提案手法では，実行しようとする領域のブレークポイントを一時的に外して実行しなければならない．そのため，一時的に攻撃可能な隙が生まれる．

Chapter 5　生成コード実行プロセスを分離する手法(提案手法2)

本章では，JIT Spraying 攻撃を防止するために我々の提案した二つ目の手法である JIT コンパイラにより生成されたコードを本来のプロセスから分離したプロセス上で実行させる手法について述べる．4 章ではソフトウェアブレークポイントを利用した JIT Spraying 攻撃防止手法を提案したが，実行速度のオーバーヘッドが大きいことが欠点となった．しかし，本章で紹介する提案手法はブレークポイントを利用しないため高速な動作が期待できる．

5.1　プロセスの分離に関して

プロセスの分離は，近年よくウェブブラウザなどで用いられる．近年のウェブブラウザは HTML パーサーや JavaScript エンジン，Document Object Model(DOM) など様々な要素を含んでおり，大変複雑である．そのため，バグの混入を完全に避けることは非常に困難である．もし全ての要素を一つのプロセス上で動かすと，どれか一つの要素でエラーが起きただけでブラウザ全体がクラッシュしてしまう可能性がある．したがって，近年のウェブブラウザでは例えば各タブごとに一つのプロセスを立ち上げるなどして，それぞれのプロセスをブラウザカーネルが管理する，といった方法をとっていることがある．これにより，一つのタブがクラッシュしてもそのタブがクラッシュするだけで済み，ブラウザカーネル自体がクラッシュしない限りウェブブラウザ全体がクラッシュすることはない．プロセスの分離を用いているウェブブラウザには Internet Explorer8 や Google Chrome，また，実験的なものとしては The OP web browser[19] などがある．しかし，これらのプロセス分離手法は JIT Spraying 攻撃への考慮がなされているわけではない．

Google Chrome では分離したプロセスに更にサンドボックス化を施す．これにより，脆弱性攻撃によりシステムに被害を及ぼすことは一見困難に思えるが，サンドボックスは完全な物ではない．例えば，サンドボックス内であっても FAT32 のファイルシステムには依然として読み込み・書き込みが行えるなど制限がいくつかある [8]．そのため，もし Google Chrome で分離されているプロセスに JIT Spraying 攻撃が行われれば，FAT32 ファイルシステムに攻撃者が任意の読み込み・書き込みを行うことが可能である．

5.2　提案手法

我々は JIT Spraying 攻撃対策に主眼を置いたプロセス分離手法を提案する．我々の提案手法では，JIT エンジンを修正し，JIT コンパイラが生成したコードの実行を開始するときにそのコード実行のための専用のプロセスを立ち上げる．生成コードはそのプロセス上で実行させるようにする．そのようにした場合，プロセス分離元である親プロセスは自身のメモリの生成コード格納領域に実行属性を付加する必要がない．そのため，ASLR と DEP が適切に動作していれば，JIT エンジンを利用するプログラム側に脆弱性があろうと親プロセスで JIT シェルコードが実行されることはない．ただし，生成したコードを実行させるための子プロセスでは実際に生成されたネイティブコードを実行する必要があるため，メモリの生成コード格納領域に実行属性を付加する必要がある．しかし，そのコードの実行をするために必要な JIT エンジンのスレッド以外の不要なスレッドを全て排除すれば，JIT エンジンに脆弱性がない限りは攻撃を行うことができず，子プロセス上の実行属性の付いた JIT シェルコードのアドレスへ処理を移されることはない．また，プロセス間のメモリ空間は独立なので，もし親プロセスが攻撃されたとしても，子プロセスには影響を及ぼさない．そのため，親プロセスが攻撃によりプログラムカウンタを操作されようとも，そこから子プロセスのメモリ上の JIT シェルコードへ処理を移されるということはない．

ただし，ここで必要な仮定が三つある．

- 少なくとも親プロセスで DEP が有効であり，生成コード格納領域に実行属性が付けられていない
- ASLR により return-into-libc 攻撃が確実に防げる
- JIT エンジンには脆弱性がない

もし親プロセスの生成コード格納領域に実行属性が付けられていたり，DEP がそもそも無効に設定されていた場合には，問題なく JIT シェルコードが動作してしまう．なお，子プロセスでは DEP が無効であっても結局のところ実行属性を付けて生成コードを実行するため問題ない．また，もし return-into-libc 攻撃が可能であると，JIT エンジンを利用しているアプリケーションの脆弱性を攻撃され実行属性を付加することができる関数を呼ばれてしまう可能性がある．これにより親プロセスの生成コード格納領域へ実行属性を付加されると，そこに含まれるシェルコードが実行されてしまうかもしれない．さらに，JIT エンジンに脆弱性がある場合には，子プロセスで不要なスレッドを全て排除したとしても，有害な JavaScript コードが入力されることによりメモリ破壊を利用した攻撃が行われてしまう可能性がある．

5.3　実装

我々は提案手法を V8 JavaScript エンジン (バージョン 2.1.10)[16] に実装した．また，今回ターゲットとしているプラットフォームは Ubuntu11.04(32-bit) である．V8 は Google

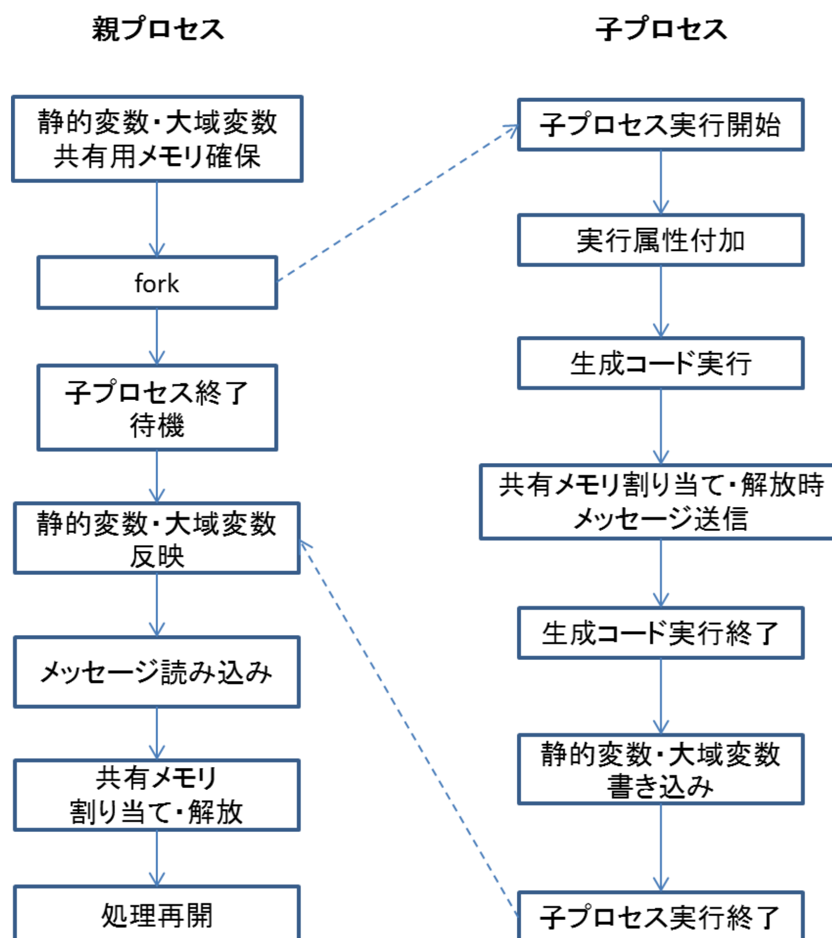

図 5.1: プロセス分離に必要な処理の流れ

Chrome の JavaScript エンジンに採用されていることで有名であるが，Google Chrome に特化した JavaScript エンジンというわけではなく，JavaScript エンジンを必要とする様々なアプリケーションに組み込んで利用することが可能である．ウェブブラウザ以外で利用されている具体的な例では，サーバサイドでも利用できる JavaScript 実行環境である Node.js などが挙げられる．

提案したプロセス分離に必要な処理の流れを図 5.1 にまとめる．本章ではこの処理について説明していく．

5.3.1　プロセス分離処理の実装

親プロセスは最初の JIT コンパイルが終わり，コンパイルされたネイティブコードを実行する直前に fork 関数を呼び，子プロセスを生成する．fork 関数を使い生成された子プロセスは親プロセスと同じ内容のメモリを持つが，プロセス間の仮想メモリ空間は独立である．また，fork 関数で生成された子プロセスは開始時点ではその fork を呼

んだ一つスレッドだけを持つ．そのため，分離前のプロセスにあったその他の不要なスレッドは自動的に排除された状態になる．子プロセスは実行開始と共に JIT コンパイラにより生成されたネイティブコードが格納されている領域へ実行属性を付加し，生成コードの実行を開始する．子プロセスが終了するまでの間，親プロセスの fork 関数を呼んだスレッドは何もせずに待ち続ける．子プロセスは，処理が終了したら自身に kill シグナルを送信して自らを強制終了させる．これは，後述する共有メモリにより親プロセスと共有しているデータがあるためである．もし子プロセスでそのまま通常の終了処理を行うと，共有メモリ上にあるオブジェクトのデストラクタが呼ばれて共有メモリ上のデータが変更されたり，共有メモリの解放が行われることがある．そうなると，親プロセスで main 関数終了後に再び終了処理が行われるときに不整合が生じて正常に終了できなくなる．そのため，子プロセスで通常の終了処理を行わないことが必要になる．子プロセスの終了後，親プロセスは後述する共有メモリの割り当て・解放処理などを行い処理を再開する．

5.3.2 データ共有処理の実装

子プロセスでは生成コードの実行中に様々な変数が更新される．また，JIT コンパイラなので実行時にさらにコンパイルが行われることもあり，その際にも様々な変数が更新されたり新たにメモリを確保することもあるであろう．そうなると，子プロセスと親プロセスの間では当然ながらメモリの内容に差異が出てくる．子プロセスの終了後にも親プロセスの実行は継続され，その後さらにまた生成コードを実行するために子プロセスが生成されることもあり得る．そのため，親プロセスは子プロセスで行われたメモリの変更を反映し，データの整合性を保つ必要がある．データを共有する必要があるメモリセグメントは.heap，.data，.bss である．.stack については特殊な利用をされない限り fork 関数を呼んだスコープの変数についてだけ共有すれば良い．

ヒープセグメントの共有については，共有メモリを利用した．ページ単位でメモリを割り当て・解放する mmap・munmap を使用している箇所については POSIX で定義されている共有メモリの取得・配置・分離を行うための関数である shmget・shmat・shmdt を用いて共有メモリの取得・解放処理への置き換えを行った．任意の大きさでメモリを割り当て・解放する箇所については OSSP mm[14] というライブラリを用いて実装を行った．OSSP mm は malloc や free のように共有メモリを扱うことができるように共有メモリの管理をしてくれるライブラリである．これを用いて，malloc と free を使用している箇所を OSSP mm の関数である MM_malloc・MM_free に置き換え，共有メモリを利用できるようにした．また，C++の予約語である new，delete，new[]，delete[] をオーバーロードしてそれらの予約語が使われたときには MM_malloc，MM_free が呼び出されるようにし，ヒープの代わりに共有メモリが使われるようにした．

子プロセスでページ単位での共有メモリの割り当て・解放が行われたときは，その情報をメッセージキューを用いて親プロセスへ送信する．子プロセスの終了後，親プロセスはメッセージキューの内容を読み取り，必要な共有メモリの割り当てと不要になった共有メモリの解放を行う．OSSP mm を用いて実装した任意サイズのメモリ割り当てと解放についてはこの処理は必要ない．なぜなら，OSSP mm は初期化時に指

定したサイズでメモリプールを作成し，その範囲内でメモリの管理を行うからである．[1]

さらに，初期値を持つ静的変数や大域変数を格納する.data セグメント，初期値を持たない静的変数や大域変数を格納する.bss セグメントの共有のため，プロセス分離前にまず.data, .bss セグメントの変数共有用の共有メモリを取得しておくようにした．子プロセスはプロセスの終了直前に V8 で定義されている大域変数や静的変数を取得し，その値を共有メモリへ書きこむ．親プロセスは子プロセスの終了後，その共有メモリを参照して大域変数や静的変数の値を書き戻して自分のプロセスへ反映させる．

5.3.3　動作確認

我々の提案手法を実装した V8 が実際に JIT Spraying 攻撃を防げるかどうか検証を行った．まず最初に，オリジナルの V8 で JIT シェルコードが実行可能であることを確認した．次に，我々の提案手法を実装した V8 が生成コード実行終了後に JIT シェルコードが実行不能であることを確認した．最後に，我々の提案手法を実装した V8 が生成コード実行中にも JIT シェルコードが実行不能であることを確認した．

動作確認のために，まず JIT コンパイル後に JIT シェルコードが生成される JavaScript コードが必要になるが，V8 用の公開されているコードが発見できなかったため，図 5.2 に示す JIT シェルコード生成用の JavaScript コードを我々で作成した．*jit_shellcode_str1* は何もしない関数 (func) の定義とその関数を呼び出すコードの一部を文字列として持つ．*jit_shell_code_str2* は JIT コンパイル後シェルコードとなるような定数を関数の引数に入れるコードを文字列として持つ．*nop_str* が持つ文字列は JIT コンパイル後に NOP スレッドとなるような定数である．次の for ループで指定回数 (この例では 1000 回)，NOP スレッドとなる定数をシェルコードとなる定数の前に付け足している．最後に文字列変数を組み合わせてコードを完成させ，eval 関数で実行する．

これを JIT コンパイルすると，func 関数呼び出し時に引数をスタックに push する部分のコードが図 5.3 に示したものとなる．図 5.3 の左側のコードは機械語，右側のコードはそのアセンブリ表記である．当然ながら実際にこのような機械語が生成されるかどうかは V8 のバージョンによって異なる．我々の用いた V8 のバージョンは 2.1.10 であり，このバージョンの場合には実際に図 5.3 に示した JIT シェルコードが生成される．このバージョンの V8 では JavaScript コードで関数の引数に指定された定数は JIT コンパイルされたネイティブコード上では値が二倍された状態で処理されているようである．具体的には，図 5.2 で示した JavaScript コードの func 関数の (for 文で *nop_str* を付け足さなかった場合の) 第二引数にあたる “0x1e6018c8” は JIT コンパイル後の図 5.3 では “0x3cc03190”，func 関数の第三引数にあたる “0x1e345a48” は “0x3c68b490” になっている．そのため，JIT シェルコードを構成する上で JIT コンパイル後の定数の最下位 1 バイトに位置する値は必ず 2 の倍数であるという制約が生まれる．そのため，シェルコードは全て 2 バイト以下の命令で構成し，定数の最下位 1 バイトの多くは簡

[1] OSSP mm が作成できるメモリプールの最大サイズはコンパイル時に決められる．デフォルトでは約 32MB である．V8 に付属している V8 ベンチマークの実行や，後述する動作確認時に用いたプログラムでは，デフォルトの最大サイズで問題が出ることはなかった．

```
var jit_shellcode_str1 = 'function func(){}func(';

var jit_shellcode_str2 =
'0x1e484848,0x1e6018c8,0x1e345a48,0x1e6d98c8,
0x1e00dbc8,0x1e71fbc8,0x1e39da48,0x1e17d848,
0x1e375a28,0x1e34d848,0x1e71fbc8,0x1e71fbc8,
0x1e315a48,0x1e17d848,0x1e2daa28,0x1e6498c8,
0x1e6918c8,0x1e6018c8,0x1e05d848,0x1e4066c8,
0x11111111);';

var nop_str = '0x1e484848,';

for(var i=0; i<1000; i++){
    jit_shellcode_str1 =
        jit_shellcode_str1 + nop_str;
}

jit_shellcode_str1 =
    jit_shellcode_str1 + jit_shellcode_str2;

eval(jit_shellcode_str1);
```

図 5.2: JIT シェルコードが生成される JavaScript コード

単のため NOP(90) 命令でパディングしている．図 5.3 のコードを 1 バイトずらして解釈すると元の push 命令 (68) が cmp 命令 (3c) のオペランドとなって表に出てこなくなり，シェルコードとして動作する．そのときに解釈されるシェルコードを図 5.4 に示す．わかりやすいように元の push 命令を無効にするための cmp 命令 (3c 68) とパディングのための NOP 命令 (90) を省略している．このシェルコードは全て 2 バイト以下の命令で構成されており，これが実行されると/bin/sh が起動する．

V8 を利用するユーザプログラムとしては，V8 に付属しているサンプルプログラムの一つである shell を利用した．このプログラムは起動時に引数に JavaScript コードの書かれたファイルを指定することでその JavaScript コードを実行する．また，起動時に引数なしで実行すると対話的に JavaScript を実行することができる．今回は JIT シェルコードの実行可否を確かめるため，shell に変更を加えた．具体的には JIT シェルコードの置かれるアドレス[2]を調べ，生成コードの実行終了直後に JIT シェルコードの置かれているアドレスへジャンプする命令を挿入した．

ASLR をオフにし，オリジナルの V8 をリンクしている shell の引数に作成した

[2]V8 のコンパイル時に"disassembler=on" オプションをつけ，さらにサンプルプログラムである shell を実行するときに"–print_code" オプションをつけることで JIT コンパイルされて生成されたコードの情報がメモリアドレスを含め表示される．

```
689031c03c      push 0x3cc03190
6890b4683c      push 0x3c68b490
689031db3c      push 0x3cdb3190
6890b7013c      push 0x3c01b790
6890f7e33c      push 0x3ce3f790
6890b4733c      push 0x3c73b490
6890b02f3c      push 0x3c2fb090
6850b46e3c      push 0x3c6eb450
6890b0693c      push 0x3c69b090
6890f7e33c      push 0x3ce3f790
6890f7e33c      push 0x3ce3f790
6890b4623c      push 0x3c62b490
6890b02f3c      push 0x3c2fb090
6850545b3c      push 0x3c5b5450
689031c93c      push 0x3cc93190
689031d23c      push 0x3cd23190
689031c03c      push 0x3cc03190
6890b00b3c      push 0x3c0bb090
6890cd803c      push 0x3c80cd90
6822222222      push 0x22222222
```

図 5.3: 生成される JIT シェルコード

JavaScript ファイルを指定し実行した．すると，/bin/sh が実行されたため，オリジナルの V8 は V8 を利用するユーザプログラムの脆弱性により JIT シェルコードを実行されてしまう危険性のあることが確認できた．次に，我々の提案手法を実装した V8 をリンクさせた shell で同様の試行を行った．すると，/bin/sh は実行されず，(実行属性のない場所を実行しようとしたため) セグメンテーションフォルトが発生してプログラムは終了した．ASLR がオフのときに JIT シェルコードが動作しなかったため，当然ながらメモリアドレスがランダム化されてしまう ASLR がオンのときにおいても同様に JIT シェルコードは動作しない．これにより，我々の提案手法により JIT シェルコードが実行できなくなったことが確認できた．

さらに，生成コードの実行中に JIT シェルコードの置かれているアドレスへジャンプするよう，shell に変更を加えた．具体的には，生成コード実行開始直後にスレッドを生成し，5 秒後に JIT シェルコードの置かれているアドレスへジャンプするようにした．また，JavaScript コードには処理を終えたあと 10 秒間待ってから終了するようなコードを付け加えた．ただし，JavaScript の組み込み関数には sleep する関数はないため，sleep を行う関数をユーザプログラムでコールバック関数として作成し，JavaScript からそれを呼ぶようにした．これにより，生成コードを実行している最中に他のスレッドが JIT シェルコードの置かれているアドレスへジャンプしてくることになる．オリジナルの V8 をリンクした shell の場合，やはり/bin/sh が起動したので JIT

```
31 c0           xor    eax,eax     ;zero clear
b4 68           mov    ah, "h"
31 db           xor    ebx,ebx
b7 01           mov    bh, 0x01    ;ebx = 0x00000100
f7 e3           mul    ebx         ;eax = eax * ebx (shift left 1byte)
b4 73           mov    ah, "s"
b0 2f           mov    al, "/"
50              push   eax         ;push "/sh\0"
b4 6e           mov    ah, "n"
b0 69           mov    al, "i"
f7 e3           mul    ebx
f7 e3           mul    ebx
b4 62           mov    ah, "b"
b0 2f           mov    al, "/"
50              push   eax         ;push "/bin"
54              push   esp         ;push address of "/bin/sh\0"
5b              pop    ebx         ;ebx = address of "/bin/sh\0"
31 c9           xor    ecx,ecx
31 d2           xor    edx,edx
31 c0           xor    eax,eax
b0 0b           mov    al, 0x0b    ;system call num 11 is execve()
cd 80           int    0x80        ;execve(ebx, ecx, edx)
                                   ;execve("/bin/sh\0", null, null)
```

図 5.4: JIT シェルコードに含まれるシェルコード

シェルコードを実行することができた．しかし，我々の提案手法を実装した V8 をリンクした shell の場合には/bin/sh が起動せずにセグメンテーションフォルトが起こった．以上により，生成コードの実行中であっても我々の提案手法ならば JIT シェルコードの実行を防止できることが確認できた．実験結果を表 5.1 にまとめる．

5.4　評価

5.4.1　実験環境

VMware Player3.1.5 上に構築した Ubuntu 11.04(32-bit) にて実験を行った．ホストマシンの環境を表 5.2 に，ゲストマシンの環境を表 5.3 に示す．また，コンパイル時に用いた gcc のバージョンは 4.5.2，最適化オプションは-O3 を用いた．kernel パラメータの randomize_va_space は 2 に設定し，ASLR を有効にした．

計測時に実行したアプリケーションは V8 に付属している shell というサンプルアプリケーションであり，V8 はそれにリンクされ使用されている．

表 5.1: JIT シェルコードの動作可否

	JIT shellcode work or not
original V8(after execution)	work
original V8(while execution)	work
process isolated V8(after execution)	not work
process isolated V8(while execution)	not work

表 5.2: ホストマシンの環境

OS	Windows 7 Professional 64-bit
プロセッサ	Intel(R) Core(TM) i7 CPU M640 2.80GHz
メモリ	8.00GB (7.80GB 使用可能)

5.4.2　性能評価

V8 ベンチマーク

V8 のソースコードにデフォルトで付属している V8 ベンチマーク [17] を利用してスコアを計測した．使用した V8 ベンチマークのバージョンは 5 である．表 5.4 にその結果を示す．表 5.4 の値は各ベンチマークを 1000 回実行したときのスコアの平均と標準偏差である．スコアは高いほど性能が良いことを示す．TotalScore は各ベンチマークのスコアを合計した値の平均である．オリジナルの V8 と我々の提案手法を実装した V8 のスコアを見比べると，TotalScore ではオリジナルの V8 のほうが勝っている．しかし，割合にしてみるとわずか 0.2%の違いであり，ほとんど差はない．これは我々の提案手法の場合，JIT コンパイラにより生成されるコードに変更を加えておらず，生成コード自体のオーバーヘッドがほぼないためと考えられる．わずかなオーバーヘッドは共有メモリ取得・解放とそのときのメッセージ送信により生じるものだと考えら

表 5.3: ゲストマシンの環境

OS	Ubuntu 11.04 (natty) 32-bit
Kernel	Linux 2.6.38-13-generic
割り当てプロセッサコア数	2 個
割り当てメモリ	1024MB

表 5.4: V8 ベンチマークのスコア

Benchmarks	Original V8	Our V8
Richards	5268.61±97.46	5259.58±78.74
DeltaBlue	6606.79±168.44	6436.28±108.99
Crypto	4747.20±59.29	4746.51±85.56
RayTrace	9626.65±297.98	9696.67±164.62
EarleyBoyer	20074.41±388.34	20176.15±272.83
RegExp	2952.24±48.87	2929.658±50.34
Splay	12457.39±202.28	12354.53±270.16
TotalScore	61733.30±543.22	61599.39±460.61

れる．

また，個々のベンチマークを見ると RayTrace と EarleyBoyer について，オリジナルの V8 より我々の提案手法を実装した V8 の方がスコアがわずかに高くなっている．これらのスコアは t 検定によるとオリジナルの V8 と提案手法を実装した V8 の間で有意差があると判定されたが，それは細かな実装の違いによるものであると考えられ，気にする必要はない．実際，オリジナルの V8 と提案手法を実装した V8 の間のスコアの高低は誤差により容易に逆転してしまう程度のものであり，実質的な差はほとんどない．

実行速度

V8 ベンチマークの各ベンチマークについて実行時間の計測を行った．計測は bash 組み込みの time コマンドで行った．結果を表 5.5 に示す．表 5.5 の値はベンチマークを 1000 回実行したときの実時間[3]の平均である．我々の提案手法を実装した V8 ではそれぞれのベンチマークについて 0.005s∼0.026s，平均して 0.011s のオーバーヘッドがあり，割合にすると 0.20%∼1.77%，平均では 0.80%である．これは実使用において十分無視できる値であろう．

実行速度についてはベンチマークスコアよりもオリジナルの V8 と提案手法を実装した V8 の間で有意差があることがより明らかである．これはベンチマークスコアの計測の場合，オーバーヘッドは生成コード実行中にかかるオーバーヘッド，すなわち共有メモリ取得・解放とそのときのメッセージ送信程度であるのに対し，実行速度の計測では V8 をリンクしたアプリケーションの実行開始から実行終了までの速度を計測しているため，それに加えて実行コード生成前後のプロセス分離に必要な処理 (fork, 実行属性付加，静的変数・大域変数の受け渡し，割り当て・解放された共有メモリの反映など) がオーバーヘッドとしてかかるためである．

[3]bash の組み込みの time コマンドで表示される real の値

表 5.5: V8 ベンチマークの実行時間

Benchmarks	Original V8	Our V8	Overhead(s)	Overhead(%)
Richards	1.024s	1.035s	0.011s	1.07%
DeltaBlue	1.025s	1.037s	0.012s	1.17%
Crypto	2.036s	2.040s	0.004s	0.20%
RayTrace	1.030s	1.042s	0.012s	1.17%
EarleyBoyer	2.043s	2.050s	0.007s	0.34%
RegExp	1.042s	1.047s	0.005s	0.48%
Splay	1.472s	1.498s	0.026s	1.77%
Average	1.382s	1.393s	0.011s	0.80%

表 5.6: ほとんど処理のない JavaScript コードの実行時間

Original V8	Our V8	Overhead
0.023s	0.026s	13.04%

また，変数を一つ宣言するだけという非常に単純な JavaScript コードを作成し，それについても実行時間を計測した．結果を表 5.6 に示す．値は 1000 回実行したときの平均である．オリジナルの V8 と比べて我々の提案手法を実装した V8 では 0.003s のオーバーヘッドがあり，割合にすると 13.04%であるが，時間を尺度として見ると非常に小さいものであるためユーザにとってはまったく気にならない程度である．しかし，もしこのような極度に短い JavaScript コードが何度も連続して実行されるという状況においては大きなオーバーヘッドがかかるかもしれない．ただし，例えばウェブブラウジングのような実利用においてそのような特殊な状況はほとんど現れないと思われるため，大きな問題はない．

共有メモリ使用量

Ubuntu 標準のシステムモニタである gnome-system-monitor により V8 ベンチマーク実行時の共有メモリ使用量を監視したところ，V8 ベンチマークの Splay 実行時に最大で約 60MiB [4]程度の共有メモリを使用するようであった．参考までに表 5.7 に共有メモリ使用量を他のアプリケーションのデータと共に示す．なお，他のアプリケーションは特に大きな負荷がかかっていない状態であることに注意してほしい．これによると，他のアプリケーションと比較して我々の提案手法を実装した V8 が桁違いに共有

[4]MiB(メビバイト) は 2^{20} バイト，つまり 1048576 バイトのことである．gnome-system-monitor では KiB(キビバイト) や MiB といった単位でメモリ使用量が表示される．

表 5.7: 共有メモリ使用量

アプリケーション	共有メモリ使用量
提案手法を実装した V8	最大約 60MiB
gnome-system-monitor	18.4MiB
gedit	14.1MiB
nautilus	13.7MiB

表 5.8: 既存研究との比較

	JIT エンジンの修正	OS の修正	生成コードの変化	フォルスポジティブ	フォルスネガティブ[5]	脆弱な時間	実行速度
提案手法 1	不要	不要	なし	なし	パラメータによってはなし	あり	遅い
提案手法 2	必要	不要	なし	なし	なし	なし	速い
JIT shellcode detection[7]	不要	不要	なし	あり	あり	なし	速い
JITSec[11]	不要	必要	なし	あり	なし	なし	速い
INSeRT[36]	必要	不要	あり	なし	あり	なし	速い
JITDefender [10]	必要	不要	なし	なし	なし	あり	速い

メモリを消費するということはなく，特に問題はなさそうである．

5.4.3　既存研究との比較

表 5.8 に既存研究との比較を示す．提案手法 1 は 4 章で紹介した提案手法であり，提案手法 2 が本章で紹介した提案手法である．項目の” 脆弱な時間” とは，他のタイミングと比べて著しく脆弱となるタイミングが存在するかどうかである．“実行速度” は実用的な速度で動作すれば “速い” とし，そうでなければ “遅い” とした．

まず JIT shellcode detection についてであるが，この手法は文献 [7] 中の記載より外部プログラムとして実装されているものと考えられる．そのため，JIT エンジンの修正は不要である．攻撃の意思のないコードも JIT シェルコードとなり得るため，少なくともフォルスポジティブは存在する．検知アルゴリズムで検知できない JIT シェルコードが存在し得るため，フォルスネガティブも存在する．

[5]脆弱な時間は考慮に入れない

JITSec はシステムコールの呼び出しに処理を挟み込むため，OS の修正 (より正しくはカーネルモジュールの組み込み) が必要になる．この手法ではヒープセグメントからのシステムコール呼び出しがブロックされるが，生成されたコードが直接システムコールを呼び出すことがあり得ると誤検知が発生するため，この手法のフォルスポジティブは存在する．

INSeRT は JIT エンジンを修正して JIT シェルコードとならないようなコードを生成するため，生成されるコードは変化する．正常な実行なら実行され得ない場所に検知用のトラッピングスニペットを置くため，実際に攻撃されない限り攻撃は検知しない．そのためフォルスポジティブはない．一方フォルスネガティブについてであるが，生成コードの様々なランダム化を行なっても少なからず JIT シェルコードが生成されてしまう可能性がある．この手法の場合，攻撃成功確率を 20 万分の 1 にすると文献 [36] 中に記述されており，フォルスネガティブはわずかながら存在する．

JITDefender は生成コード実行時以外にメモリの実行属性を外して攻撃を防ぐため，JIT エンジンの修正が必要である．生成コード実行時には生成コード格納メモリに実行属性を付けなければならず，全く無防備な状態となるため脆弱な時間が存在する．また，脆弱な時間を考慮に入れなければフォルスネガティブはない．

提案手法 1 は JIT エンジンの修正も OS の修正も不要であるが，ブレークポイントの設置位置にインターバルを設けた場合にはフォルスネガティブが発生する可能性がある．また，実行しようとする領域のブレークポイントを一時的に外す必要があり，その際に攻撃が可能な隙が生まれる．そのため，脆弱な時間が存在する．また，実行速度のオーバーヘッドは他の研究と比較して大きい．

これらと比較した提案手法 2 の利点は，まず OS の修正が必要でないことが挙げられる．これは JITSec に対するアドバンテージとなるが，その代わり，INSeRT や JITDefender と同様に JIT エンジンの修正が必要となる．また，生成コードには手を加えないため生成コードの変化はない．これは，INSeRT に対するアドバンテージとなる．そして，我々の手法では積極的に異常を検知しようとすることはなく，攻撃が行われない限りプログラムの実行は妨げられない．そのため，フォルスポジティブはない．これは JIT shellcode detection，JITSec に対するアドバンテージとなる．さらに，攻撃が行われてもそもそも親プロセスの JIT シェルコード格納領域には常時実行属性がついておらず，攻撃はいかなるときにも妨げられる．また，子プロセスでは余計なスレッドが排除されており脆弱性がないと仮定している JIT エンジンのコードしか実行されない．そのため，そもそも攻撃は行われず，フォルスネガティブも存在しない．これは JIT shellcode detection，INSeRT に対するアドバンテージとなる．それから，親プロセスの生成コード格納メモリ領域に実行属性を付ける必要はないため，生成コード実行中にも安全性が低下することはなく，脆弱な時間は存在しない．これは JITDefender，提案手法 1 に対するアドバンテージとなる．脆弱な時間についての比較図を図 5.5 に示す．通常の JIT エンジンは常に脆弱なのに対し，JITDefender を適用すると生成コード実行時のみが脆弱な時間となる．また，提案手法 1 の脆弱な時間は JITDefender とほぼ同様となる．提案手法 2(図中の Process Isolated JIT engine) を適用すると常に安全となる．最後に実行速度についてであるが，提案手法 2 の実行速度

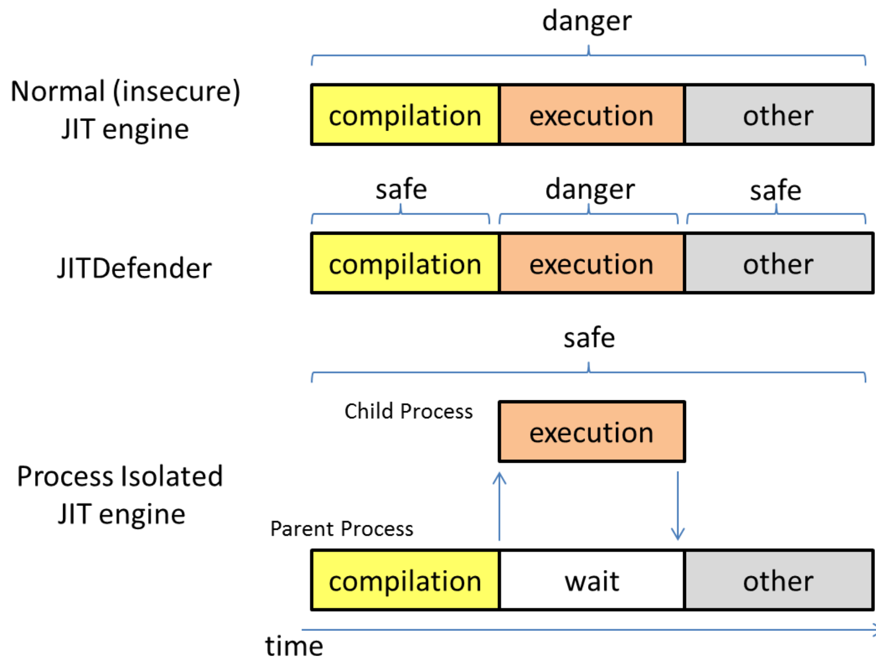

図 5.5: 脆弱な時間についての比較

は十分に実用的であり，これは提案手法 1 に対するアドバンテージとなる．また，実行速度については環境の違いにより一概には比較できないが，INSeRT は生成コードに手を加えることによってベンチマークのスコアに数%のオーバーヘッドがかかることから，提案手法 2 の方が高速に動作することが期待できる．

5.5　議論

5.5.1　.data セグメント，.bss セグメントの共有メモリ化

.data セグメント，.bss セグメントには静的変数や大域変数が格納されており，それらはアプリケーションの開始時から存在するため，ヒープセグメントと違ってこれらを直接共有メモリ化することはできない．よって，現在の実装では，.data セグメントと.bss セグメントは共有メモリ化しているわけでなく，それらに格納されている変数の内容を共有メモリを使って子プロセスから親プロセスへ受け渡すという処理をし，データの共有を実現している．しかし，この処理では実際には変更されていない変数も含め全ての静的変数，大域変数を子プロセスでは共有メモリへコピーし，親プロセスでは共有メモリから自プロセスの変数へコピーしているため，無駄が多い．この無駄を無くすため，共有メモリを.data セグメント，.bss セグメントの代わりに用いるという実装方法が考えられる．具体的には，アプリケーションの開始とともに静的変数と大域変数を全て共有メモリ上にコピーし，静的変数や大域変数へのアクセスは全てその共

有メモリ上へアクセスするようコードを書き換えるという方法が考えられる．既存のJIT エンジンを修正して提案手法を実装する場合，この方法を採ると現実装よりも修正すべき点が多くなるが，子プロセス終了時と親プロセス復帰時の無駄な処理がなくなりオーバーヘッドが減るはずである．ただし，現実装でもそれほど大きなオーバーヘッドは発生していないため，大きく実行速度が改善されるわけではなさそうである．

5.5.2　コールバック関数の問題

JIT エンジンを利用するプログラム側でコールバック関数を定義し，これを JIT コンパイラにより生成されたコードから呼ぶことが可能な JIT エンジンがある．例えば V8 はそのようなことが可能で，JavaScript コードから呼び出したい関数をユーザプログラム側で定義しておくことができる．この場合，ユーザプログラム側で定義されたコールバック関数に脆弱性があると，我々の提案手法にとって大きな問題となる．なぜなら，そのコールバック関数はユーザ側のプログラムであるにも関わらず子プロセス上で動作してしまう．すると，JavaScript コードからそのコールバック関数に不正な引数を渡して呼び出すことにより脆弱性が突かれ，プログラムの制御を奪われて JIT シェルコードが子プロセス側で実行されてしまうかもしれない．

この問題の解決策としては，JIT エンジン側でコールバック関数を呼ぶためのインターフェイスとなる関数を用意し，その関数でユーザプログラムに定義されたコールバック関数を呼ぶ前にデータ格納用領域にある生成コードから実行可能属性を取り除き，コールバック関数から返るときには再び実行可能属性を付加する，という方法が考えられる．この方法を採れば，もしコールバック関数に不正な入力をして攻撃を行おうとしてもコールバック関数実行中には JIT シェルコードが置かれている可能性のあるデータ格納領域には実行属性が付加されておらず，攻撃は成功しない．

5.5.3　複数スレッドからの生成コード実行呼び出しの問題

JIT エンジンを利用するプログラムは多くの場合複数のスレッドを持っている．これらのスレッドのいくつかが同時に生成コードの実行開始関数を呼び出した場合に問題が発生すると考えられる．なぜなら，生成コード実行時に fork で新たに生成されたプロセスは fork が呼ばれた時点での親プロセスのメモリの内容をコピーする．そして子プロセス終了時に親プロセスは子プロセスのメモリの内容を親プロセスに反映させるわけだが，その際に複数の子プロセスが存在すると，子プロセス終了後に変更を親プロセスに反映させる.bss，.data セグメントのデータや子プロセスで割り当て・解放した共有メモリのデータの整合性が保てなくなる．

この問題の解決策としては，子プロセスを終了させずにそのまま残し，複数の子プロセスを生成させないという実装方法が考えられる．親プロセスで生成コード実行の要求があれば子プロセスへそれを伝え，必要なデータを子プロセスへ渡す．子プロセスでは親プロセスから生成コード実行の要求を受け取ったら新たにスレッドを生成し生成コードを実行する．この方法を用いれば，子プロセスは一つだけなのでデータの不整合は生じない．

また，他の解決策としては複数スレッドから同時に生成コードを実行しないことを前提条件とすることが考えられる．この場合，当然ながら複数スレッドから同時に生成コードの実行がされる問題は考える必要はなくなる．この前提条件は現実的なものである．例えば Google Chrome(または Chromium ブラウザ) では HTML パーサー，JavaScriptVM，DOM などハイリスクなコンポーネントはサンドボックス化されたレンダリングエンジンに割り当てられており，そのレンダリングエンジンのプロセスはタブごとに生成される [8]．したがって，このブラウザでは一つのタブにつき一つの JIT エンジンが含まれたプロセスが存在しているため，複数のタブが開かれていたとしても一つの JIT エンジンへ複数スレッドから同時に生成コードの実行が行われることはないと考えられる．[6]

5.5.4 他の手法との組み合わせについて

3.1 節で説明した既存の対策手法は，それぞれが本質的に異なる手法であるため，組み合わせて用いることが可能である．我々の提案手法は，5.2 節で説明した三つの仮定を置く限りにおいて，現在のところフォルスネガティブが起こるようには考えられない．しかし，将来的に我々の手法に抜け道が発見されたり，またはそもそも仮定に反して JIT エンジンに脆弱性があったりアドレス情報の漏洩などにより ASLR を回避して return-into-libc 攻撃が可能であった場合でも，我々の手法とさらに別の手法を組み合わせて用いていれば JIT Spraying 攻撃を防ぐことができるかもしれない．

例えば，JIT コンパイラに脆弱性がないという仮定が崩れ，細工された JavaScript コードを入力されることにより子プロセス上で JIT コンパイラの脆弱性が突かれることにより本章で提案した対策手法が突破されたとしても，JIT shellcode detection がその JIT シェルコードのパターンを知っていればプログラムを停止することができるし，INSeRT を適用していれば生成コードが JIT シェルコードとして動作しないため攻撃を防ぐことが可能であるし，JITSec を適用していれば JIT シェルコードからシステムコールが呼ばれたときにプログラムを停止することができる．また，JITDefender については，攻撃によりプログラムカウンタを JIT シェルコードの位置に書き換えられたのが生成コードの実行中でなかった場合には JIT シェルコードの格納領域に実行属性が付加されていないため攻撃を防ぐことができる．しかし，運悪く生成コードの実行中にプログラムカウンタの書き換えが起こると JIT シェルコードが動作してしまう．

次に，親プロセスで DEP が有効であるという仮定が崩れた場合，親プロセスに格納されている JIT シェルコードが実行されてしまうわけであるが，これについても同様に JIT shellcode detection，INSeRT，JITSec によって JIT シェルコードの実行を防止できると考えられる．しかし，JITDefender は DEP が有効であってこその対策手法であり，DEP が有効であるという仮定が崩れると実行属性を取り除くことができなくなってしまうため我々の手法と共に回避されてしまうであろう．

最後に，ASLR により return-into-libc 攻撃を確実に防げるという仮定が崩れたとき

[6] JavaScript には言語仕様上スレッドが一つしかない．そのため一つのタブで閉じられている場合，生成コードの実行関数は一度に一つのスレッドからしか呼ばれないはずである．

のことを考える．この場合には，プロセスを分離して実行しようにも攻撃により親プロセスで実行属性を付加する関数を呼ばれてしまうため，親プロセスに格納されているJIT シェルコードが実行されてしまう．この際にも，JIT shellcode detection，INSeRT，JITSec のいずれかが適用されていれば JIT シェルコードの実行は防ぐことができる．JITDefender についてはDEP が有効であるという仮定が崩れた場合と同様である．ただし，そもそも return-into-libc 攻撃が可能ならば，わざわざ JIT シェルコードを利用しなくとも任意の関数を呼んで攻撃が可能である．しかし，それはもはや JIT Spraying 攻撃の範疇ではないためここでは議論しない．

このようないくつかの対策手法を組み合わせるということは現実にもよく行われることである．バッファオーバーフローなどのメモリ破壊を利用した攻撃を防ぐための機能であれば，例えば，Ubuntu においては実際に Stack Protector，Heap Protector，Pointer Obfuscation，ASLR，Non-Executable Memory(DEP) などいくつかのセキュリティ機構を同時に用いている [2].

5.5.5　ウェブブラウザに用いられるプロセス分離との組み合わせ

よくウェブブラウザなどで用いられるプロセスの分離はバグ混入のリスクの高いプロセスを分離し，親となるプロセスの安全を確保するという目的で取り入れられる．一方，本提案手法の目的は JIT シェルコードを実行させないことであり，バグ混入のリスクの高いプロセスでは生成コード領域に実行属性を付加させずに安全なプロセスを生成してそちらで実行属性をつけて実行を行う．そのため，この二つのプロセス分離は競合せず，今後ウェブブラウザではブラウザカーネルからバグ混入リスクの高いコンポーネントを分離し，さらにその分離されたコンポーネントから JIT コンパイルされたコードの実行プロセスを分離するという二段階のプロセス分離が用いられるようになれば，JIT Spraying 攻撃を視野に入れた強固なセキュリティが確保できると考えられる．

5.6　まとめ

本章では，JIT コンパイルにより生成されたコードの実行直前に子プロセスを生成し，その上で生成コードを実行する手法を提案した．これにより，親プロセスでは実行属性を付ける必要がなくなり，親プロセスのメモリ上にある JIT シェルコードは動作不能となる．JIT エンジンに脆弱性がないという仮定をおけば，子プロセス上でも JIT エンジン以外の全てのスレッドを排除することにより脆弱性攻撃を行うことができなくなり安全である．親プロセスでは JIT エンジン以外の脆弱なコードを動作させるスレッドが動いているかもしれないが，プロセス間のメモリ空間は独立であるため，親プロセスを攻撃することによって子プロセスのメモリ上にある実行属性のついた JIT シェルコードを実行することはできない．

我々は提案手法を V8 JavaScript エンジン上へ実装し，実際に JIT シェルコードが動作しなくなることを確かめた．V8 ベンチマークによるパフォーマンス評価により，

十分に実用的な速度で動作することも確かめられた．

また，3.3 節で述べた提案手法への要件を全て達成することができた．

Chapter 6　結論

6.1　本論文のまとめ

本論文では，JIT Spraying 攻撃を防ぐための手法を二つ提案した．

一つ目に，外部のユーザレベルプログラムを用いて実行監視を行う手法を提案した．この手法はOSの修正やJITエンジンの修正が不要なことが利点となっている．この手法をWindows XP上で実装し，JIT Spraying攻撃から守るターゲットはInternet Explorer8上で動くFlash Playerとした．メモリ領域のアクセス保護属性を変更するVirtualProtect関数を捉えてその引数からJITコンパイラが実行されることを意図していた正当な命令アドレスを見つけ出し，実行属性が与えられたメモリ領域にはブレークポイントを仕掛けて実行されたときにどこのアドレスが実行されたかわかるようにした．ブレークポイントでプログラムが停止したときに正当な命令アドレスであるか判定し，正当な命令アドレスでなかった場合には不正な実行としてプログラムを強制終了させる．しかし，実行属性の与えられた全領域にブレークポイントを仕掛けると非常に大きなオーバーヘッドがかかった．そのため，実行速度の向上のためにブレークポイントの設置位置にインターバルを持たせることができるようにした．数十バイトのインターバルをとるだけでも実行速度は大きく向上したが，それでも十分な実行速度が得られているとは判断できなかった．さらに，インターバルを設けると安全性が低下するという欠点もあった．

二つ目に，生成コード実行用のプロセスを分離する手法を提案した．この手法はJITエンジンの修正が必要となる点が欠点であるが，ブレークポイントを使う必要がないため高速な動作が期待できる．親プロセスが生成コード実行用の専用プロセスを子プロセスとして生成するが，プロセス間のメモリ領域は独立であるため，もし親プロセスが攻撃されても子プロセスに被害は及ばない．また，生成コードは子プロセスで実行させるため親プロセスでは生成コード格納領域に実行属性を付加する必要はない．子プロセスでは生成コードに実行属性を付加する必要があるが，無駄なスレッドを排除することにより安全性を確保する．このような手法を実際にV8 JavaScriptエンジンへ実装した．プロセスの分離はfork関数を用いて行い，さらに子プロセス終了後にも親プロセスが正常動作するよう共有メモリを用いてメモリの.heap，.bss，.dataセグメントの共有を行った．提案手法が実際に機能することの確認として，実際にJITシェルコードが生成されるようなJavaScriptコードを入力として与え，提案手法を実装したV8では生成コードの実行終了後，実行中共にJITシェルコードが機能しないことを確認した．オーバーヘッドについては，V8ベンチマークのスコアや実行時間の計測から通常利用時にはまったく気にならない程度であることが確認でき，我々の提案手法が十分に実用的であることがわかった．

6.2　展望

JIT Spraying 攻撃に関する研究はまだ始まったばかりである．JIT Spraying 攻撃対策技術はまだ生成コード領域を暗号化したり[1]，ソースコードに出現する定数が生成されるネイティブコードにそのままの値で出てこないようにする[2]などの対策が各々の JIT エンジンに少しずつ組み込まれるようになり始めただけで，普通の OS に JIT Spraying 攻撃を対象としたセキュリティ機構はまず組み込まれていないし，アンチウィルスソフトのような形で一般ユーザが使えるような対策ソフトウェアもない．もしそのようなソフトウェアがあったとしても，現在のところ JIT Spraying 攻撃は一般ではあまり知られていないので，ユーザが意図して JIT Spraying 攻撃対策をするということはほぼあり得ないであろう．また，そもそも JIT Spraying 攻撃が実際に行われているというのは一般ユーザにはなかなか伝わらないものであろう．例えば，JIT Spraying を利用してバッファオーバーフロー攻撃が可能な脆弱性があったとしよう．しかし，その場合に報告されるのはほとんどの場合 “バッファオーバーフローの脆弱性があった” ということだけであり，実際のところその脆弱性が Heap Spraying を利用して攻撃されるのか JIT Spraying を利用して攻撃されるのかというのはなかなか表に出てくる情報ではない．実際にどのような手法でその脆弱性が攻撃されているかがわかるのは，結局のところマルウェアの解析をしている者であったり悪意のあるウェブサイトを解析している者などセキュリティの専門家だけだと考えられる．そういったことを踏まえると，JIT Spraying 攻撃などの対策技術はユーザに透過的な形で提供されることが望ましい．それはつまり，ハードウェアや OS にデフォルトで対策技術が組み込まれていたり，JIT エンジンが勝手に対策をしてくれた方が良いということであり，ユーザが意識して対策ソフトウェアをインストールするといったことは望ましくない．そういう意味では，現在 JIT Spraying 対策技術が少しずつ JIT エンジンに組み込まれるようになってきていることは良い傾向であると言える．

しかし，JIT エンジン開発者にとってはどうであろうか．JIT エンジンは様々な言語にそれぞれ独自のものが使われる．また，一つの言語のためにいくつも JIT エンジンが開発されていることも珍しくない．さらに，JIT Spraying 攻撃の可能性はいわゆる言語処理系だけでなく，動的コード生成を行う全てのアプリケーションで潜在的に含まれる．それらの開発者全てが適切に JIT Spraying 攻撃対策をプログラムに組み込むことができるだろうか．現状では完全に不可能である．そもそも JIT Spraying 攻撃に利用できないようにどのような設計・実装を行えば良いかといった標準的な指針が存在していない．そのため，我々は今後どのような実装を行うのが JIT Spraying 攻撃を防ぐのに最良であるかをよく議論し，開発者達へ標準的な指針を示していくべきかもしれない．どのような人がアプリケーションを開発するかはわからないため，標準的な指針にはなるべくシンプルで技術的に高度でない方法が取り入れられるのが望ましい．本論文の提案手法 2 のようなプロセス分離を行う設計・実装はそういった条件を満たすものであり，他の既存手法と比較しても効果が高く，他の既存手法と組み合

[1]文献 [31] によると Flash 10.1 は生成コードページを暗号化するようであるが，文献 [26] ではその状態でも JIT Spraying 攻撃が可能であることを示唆している．

[2]V8 JavaScript engine 3.4.13 にて確認した．

わせて用いることができるなど柔軟性もある．もし標準的な指針が完成したら，どのような開発者も安全な JIT エンジンを作成できるような，自動的に JIT Spraying 攻撃対策技術を組み込める JIT エンジン開発のためのフレームワークの研究を行うのも有意義であろう．

また，もし JIT エンジン開発者が確実に対策技術を組み込むことが難しいのならば，ユーザだけでなく JIT エンジン開発者にとっても透過となる対策技術が組み込まれるのが最も望ましいであろう．したがって，最終的にはハードウェアや OS，または JIT エンジンをコンパイルする C/C++コンパイラなどに対策技術が標準的に組み込まれるようになるべきなのではないかと筆者は考える．そのような場面では本論文の提案手法 1 のような不正な命令アドレスを認識するといった思想が生きてくるかもしれない．

謝辞

本論文の作成にあたり，指導教員として研究の指導をしてくださった東京大学生産技術研究所 松浦幹太准教授に深く感謝する．本大学院へ入学したとき，筆者はまだ論文といえば研究室の一つ上の先輩の卒業論文ぐらいしか読んだことがないような状態であり，研究のいろはもわかっていなかった．そのような筆者に松浦先生は論文の探し方から丁寧に指導してくださった．また，松浦先生には様々な学会発表の機会も与えていただき，どれも非常に良い経験となった．

研究室のメンバーとしてお世話になった Jacob Schuldt さん，松田隆宏さん，施屹さん，千葉大輝さん，村上隆夫さん，Bongkot Jenjarrussakul さん，大畑幸矢さん，技術職員の細井琢朗さんにも感謝の意を表する．彼らには研究に関する議論だけでなく大学院における研究活動に関しても様々な助言をいただいた．

松浦研定期ミーティングで研究内容に関してや細かなスライドの修正案などについてまで議論してくださった警察庁の岡田智明さん，北条孝佳さん，中央大学の北川隆さん，笠松宏平さん，飛鋪亮太さんにも感謝する．筆者のミーティング発表時の彼らの質問や意見により気付かされることは多々あり，筆者の研究に少なからず影響を与えた．

研究室の運営に不可欠な研究室秘書として尽力してくださった小倉華代子さん，仲野小絵さんにも改めて感謝する．

東京大学大学院の学生の皆様にも感謝したい．彼らは非常に優秀であり，筆者は彼らから常に刺激を受け，有意義な学生生活を送るとともに研究のモチベーションを保つことができた．

参考文献

[1] CVE-2006-3459. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3459.

[2] Security/Features - Ubuntu Wiki. https://wiki.ubuntu.com/Security/Features.

[3] The Computer Language Benchmarks Game. http://shootout.alioth.debian.org/.

[4] Adobe. Flex 3 - using the debugger version of flash player. http://livedocs.adobe.com/flex/3/html/logging_03.html.

[5] Adobe. rich internet applications — adobe flash player. http://www.adobe.com/products/flashplayer/.

[6] P. Amini. paimei - project hosting on google code. http://code.google.com/p/paimei/.

[7] P. Bania. JIT spraying and mitigations. 2010. http://arxiv.org/abs/1009.1038v1.

[8] A. Barth, C. Jackson, C. Reis, and Google Chrome Team. The security architecture of the chromium browser. 2008.

[9] D. Blazakis. Interpreter Exploitation: Pointer Inference and JIT Spraying. *Black Hat DC*, 2010.

[10] P. Chen, Y. Fang, B. Mao, and L. Xie. JITDefender: A Defense against JIT Spraying Attacks. In *Future Challenges in Security and Privacy for Academia and Industry*, volume 354 of *IFIP Advances in Information and Communication Technology*, pages 142–153. Springer Boston, 2011.

[11] 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.

[12] T. Durden. Bypassing PaX ASLR protection. *Phrack*, 59, July 2002.

[13] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '09, pages 88–106, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] R. S. Engelschall. OSSP mm. `http://www.ossp.org/pkg/lib/mm/`.

[15] F. Gadaleta, Y. Younan, and W. Joosen. Bubble: A javascript engine level countermeasure against heap-spraying attacks. *LNCS*, 5965:1–17, 2010.

[16] Google Inc. v8 - V8 JavaScript Engine - Google Project Hosting. `http://code.google.com/p/v8/`.

[17] Google Inc. V8 Benchmark Suite - version 5. `http://v8.googlecode.com/svn/data/benchmarks/v5/run.html`.

[18] E. Grevstad. CPU-Based Security: The NX Bit. May 2004. `http://www.hardwarecentral.com/reviews/article.php/12095_3358421_/CPU-Based-Security-The-NX-Bit.htm`.

[19] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 402–416, Washington, DC, USA, 2008. IEEE Computer Society.

[20] M. Howard, M. Miller, J. Lambert, and M. Thomlinson. Windows isv software security defenses. 2010. `http://msdn.microsoft.com/en-us/library/bb430720.aspx`.

[21] F.-H. Hsu, C.-H. Huang, C.-H. Hsu, C.-W. Ou, L.-H. Chen, and P.-C. Chiu. HSP: A solution against heap sprays. *Journal of Systems and Software*, 83:2227–2236, November 2010.

[22] KASPERSKY lab. Gumblar について. 2009. `http://www.kaspersky.co.jp/support/mm/bn/1002_malware.html`.

[23] Microsoft. Vertualprotect 関数. `http://msdn.microsoft.com/ja-jp/library/cc430214.aspx`.

[24] D. Moore, C. Shannon, and k. claffy. Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, IMW '02, pages 273–284, New York, NY, USA, 2002. ACM.

[25] PacketStorm. 25byte-execve. 2009. `http://packetstormsecurity.org/shellcode/25bytes-execve.txt`.

[26] M.-C. Pan and S.-T. Tsai. The Flash JIT Spraying is Back. Jun 2011. http://exploitspace.blogspot.com/2011/06/flash-jit-spraying-is-back_04.html.

[27] V. R. Pandya. iPhone Security Analysis. *Master's thesis, San Jose State University*, 2008. http://www.cs.sjsu.edu/faculty/stamp/students/pandya_vaibhav.pdf.

[28] Randy Kath. The Debugging Application Programming Interface. 1992. http://msdn.microsoft.com/en-us/library/ms809754.aspx.

[29] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: a defense against heap-spraying code injection attacks. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 169–186, Berkeley, CA, USA, 2009. USENIX Association.

[30] C. Rohlf and Y. Ivnitskiy. Attacking Clientside JIT Compilers. *Black Hat USA*, 2011.

[31] A. Sintsov. Twitter / @asintsov: No JIT-SPRAY in Flash 10.1 ... Jun 2010. https://twitter.com/#!/asintsov/status/15947640318.

[32] A. Sintsov. Writing jit shellcode for fun and profit. 2010. http://dsecrg.com/files/pub/pdf/Writing%20JIT-Spray%20Shellcode%20for%20fun%20and%20profit.pdf.

[33] E. H. Spafford. The Internet Worm Program: An Analysis. *Computer Communication Review*, January 1989.

[34] M. Torres. PyPy's Speed Center. http://speed.pypy.org/.

[35] T. Wei, T. Wang, L. Duan, and J. Luo. Secure dynamic code generation against spraying. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 738–740, New York, NY, USA, 2010. ACM.

[36] 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*, pages 323 –328, march 2011.

[37] 大澤 文孝. IE8の16倍高速に - Internet Explorer 9の実力:ITpro. http://itpro.nikkeibp.co.jp/article/COLUMN/20110708/362205/.

発表文献

ジャーナル

i 市川顕, 松浦幹太. “実行プロセス分離による JIT シェルコード実行防止”, 情報処理学会論文誌, (推薦論文として投稿済み・審査中).

国際会議

ii Ken Ichikawa, Kanta Matsuura. “Preventing execution of JIT shellcode by isolating running process”, **Poster session**, IWSEC 2011. 東京, 11 月・2011 年.

iii Ken Ichikawa, Kanta Matsuura. “Preventing execution of JIT shellcode by isolating running process”, **Works-in-Progress session**, ACSAC 2011. Florida, U.S.A., 12 月・2011 年.

国内会議

iv 市川顕, 松浦幹太. “実行プロセス分離による JIT シェルコード実行防止”, Computer Security Symposium 2011 (CSS 2011). 新潟, 10 月・2011 年.

v 市川顕, 松浦幹太. “実行プロセス分離による JIT シェルコード実行防止とその実装・評価”, 2012 年 暗号と情報セキュリティシンポジウム (SCIS 2012) 予稿集 CDROM, 4E2-4. 石川, 1-2 月・2012 年.

研究会

vi 市川顕, 松浦幹太. “実行監視による JIT Spraying 攻撃検知”, 情報処理学会研究報告, 2011-CSEC-52, pp.1-7. 大阪, 3 月・2011 年.

受賞リスト

1. “研究奨励賞” 受賞, 2010 年度 ISS スクエアシンポジウム. 3 月・2011 年. (ポスター発表 “実行監視による JIT Spraying 攻撃検知” に対して)
2. “Best Poster Award” 受賞, IWSEC 2011. 11 月・2011 年. (発表文献 ii に対して)