情報処理学会論文誌 Vol.53 No.9 2302–2312 (Sep. 2012)

推薦論文

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[1,a)] 松浦 幹太[1]

受付日 2012年1月13日, 採録日 2012年6月1日

概要：JIT コンパイラは従来のインタプリタよりもかなり高速に動作するため，近年多くのアプリケーションに採用されるようになっている．しかし，JIT コンパイラを悪用した JIT Spraying という手法が公表されており問題となっている．JIT Spraying 攻撃を利用すると，従来バッファオーバフローなどの脆弱性攻撃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．本稿では，JIT Spraying 攻撃を防止するための手段としてプロセス分離を用いた手法を提案する．この手法では，JIT コンパイルされたコードの実行直前に新たなプロセスを生成し，そのプロセス上で JIT コンパイルされたコードを走らせる．この手法は JIT エンジンの修正を必要とするが，他の JIT Spraying 攻撃対策研究と比較して，タイミングにより脆弱になることがなく，生成されるコードの変更も必要としない．また，実行時間のオーバヘッドも実用上問題とならない程度である．

キーワード：脆弱性攻撃，メモリ破壊攻撃，JIT コンパイラ

Preventing Execution of JIT Shellcode by Isolating Running Process

Ken Ichikawa[1,a)] Kanta Matsuura[1]

Received: January 13, 2012, Accepted: June 1, 2012

Abstract: Recently, many applications use JIT compilers because it is faster than traditional interpreters. However, JIT Spraying Attacks that abuse JIT compilers were made public. If attackers use JIT Spraying Attacks, they can bypass security structures such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) simultaneously. In this paper, we propose a process isolating method for preventing JIT Spraying Attacks. This method generates a new process just before execution of JIT compiled code, and JIT compiled code runs on the process. This method needs modification of JIT engine but compared with other JIT Spraying Attacks countermeasures, it is not vulnerable depending on timing and it does not need to change generated code. In addition, its running time overhead is marginal in practical use.

Keywords: vulnerability attacks, memory corruption attacks, JIT compiler

1. はじめに

バッファオーバ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，それは DEP（Data Execution Prevention）などと呼ばれる [1], [2]．また，アドレス空間配置のランダム化は ASLR（Address Space Layout Randomization）と呼ばれ，これも OS で実装される [2]．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

1 東京大学生産技術研究所
Institute of Industrial Science, the University of Tokyo, Meguro, Tokyo 153–8505, Japan
a) 1kawaken@gmail.com

本稿の内容は 2011 年 10 月のコンピュータセキュリティシンポジウム 2011（CSS2011）にて報告され，同プログラム委員長により情報処理学会論文誌ジャーナルへの掲載が推薦された論文である．

© 2012 Information Processing Society of Japan

が，DEP と ASLR を組み合わせることで，それらの攻撃は困難になる．

ところで，近年，プログラミング言語を処理するエンジンに JIT（Just-In-Time）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．それによって，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 Blazakis により JIT コンパイラを悪用する攻撃手法である JIT Spraying 攻撃が示された [3]．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フローなどの脆弱性を利用した攻撃が可能となる．

本稿ではそのような JIT Spraying 攻撃を防ぐための 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．本手法では，生成コード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は JIT シェルコードとして使われる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．

我々は提案手法を V8 JavaScript エンジン [4] 上に実装し，性能評価を行った．実行時間のオーバヘッドは実利用においてほぼ気にならない程度のものとなった．

本稿の構成であるが，2 章で JIT Spraying 攻撃について解説を行う．3 章では JIT Spraying 攻撃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を紹介する．5 章では提案手法の実装について紹介する．6 章では我々の実装についての評価を行う．そして 7 章で結論を述べる．

2.　JIT Spraying 攻撃

JIT Spraying 攻撃は JIT コンパイラを悪用した攻撃である．

2.1　JIT コンパイラ

JIT コンパイラは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，コンパイルしてネイティブコードを実行することによる高速な動作とコンパイルの手間やプラットフォーム間の違いを吸収する従来のインタプリタの利点をあわせ持つ．JIT コンパイラは現在，よくウェブブラウザなどで使われる Java，.net，JavaScript，Flash などの処理系に搭載されていることが多い．これは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に対応するためにより高速な処理系が必要とされているためと考えられる．例として，Internet Explorer で初めて JIT コンパイラが採用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 [5]．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 [6]．

2.2　JIT Spraying 攻撃の一例

ウェブブラウザを介した JIT Spraying 攻撃の一例を図 **1** 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 JavaScript や Flash などを実行する際に JIT エンジンに処理を委託する．JIT エンジンは JIT コンパイラを用いてプログラム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た JavaScript などのプログラムには JIT コンパイル後にメモリの広大な領域に生成されたネイティブコードがばらまかれるようなプログラムが記述されており，JIT コンパイラはそのとおりにネイティブコードを生成してヒープメモリ上にばらまく．そのうえで，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フローなどが利用されることとなる．それによりウェブブラウザの実行位置がヒープメモリ上に移される．ヒープメモリ上には 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おりそれが実行されるが，そのネイティブコードがシェルコードとなりうることがあり，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．これは，近年流行している，ユーザの気づかぬ間にマルウェアをダウンロードしてインストールし実行する drive-by download 攻撃に利用することも可能である．

本稿では JIT コンパイラによって生成されるシェルコー

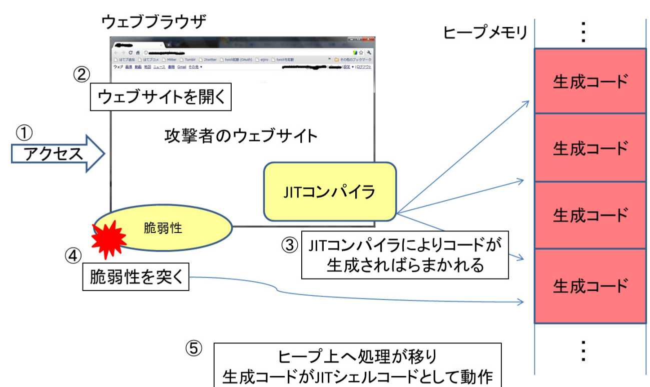

図 1　JIT Spraying 攻撃の一例
Fig. 1　An example of JIT Sprayng Attacks.

© 2012 Information Processing Society of Japan

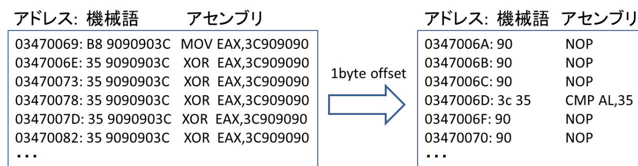

図 2　JIT コンパイラにより生成されたコードが JIT シェルコードとなる仕組み

Fig. 2　A mechanism of JIT shellcode generation from JIT compiled code.

ドを JIT シェルコードと呼ぶ．図 2 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．図 2 の左のコードは JIT コンパイルにより生成されたコードであるが，一見したところただ単に 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 byte だけずらして解釈すると図 2 の右のコードになる．すると，本来オペランドであった機械語の “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” がとられ，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．

2.3　DEP と ASLR の回避

DEP が有効であると，本来実行する必要のないメモリセグメントである.stack，.heap，.bss，.data から実行可能属性を取り除き，実行できないようにすることができる．ASLR は各メモリセグメントの基底のアドレスをランダム化するなどして攻撃者が詳細なメモリアドレスの推測をするのを困難にする．これらのセキュリティ機構により多くのメモリ破壊を利用する脆弱性攻撃を防ぐことができるようになったが，JIT Spraying 攻撃では DEP と ASLR を回避して再び脆弱性攻撃を行うことを可能にしてしまう．

JIT コンパイラは実行時にネイティブコードを生成する仕様上，メモリのヒープ領域へ実行属性を付加せざるをえない．シェルコードは実行属性が付けられたヒープ領域に格納されている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．また，JIT コンパイラに JIT シェルコードとなるようなネイティブコードを大量に吐かせれば，ヒープ上の広大な領域が JIT シェルコードで埋められ，攻撃者は JIT シェルコードのあるアドレスを見定めてプログラムの実行位置をジャンプさせるときにある程度いい加減にジャンプさせても高い確率で JIT シェルコードが実行できるという状況が生まれる．そのため，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 を同時に回避されてしまう．

2.4　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，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 が機能していれば防御が可能である[*1]．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．

3.　関連研究

すでにいくつかある 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 6.3 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．

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 [8]．このアルゴリズムは 32 bit 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 32 bit 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捜査する．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，すべての 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 シェルコードも存在する．

De Groef らによる JITSec [9] は GNU/Linux kernel を修正し，システムコールを呼び出すインタフェー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.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，.heap や.stack 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ため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

*1　アドレス情報が漏洩しないと仮定する．もし何らかのアドレス情報が漏洩する場合，ランダム化したアドレスを突き止められてしまい ASLR を回避され，DEP を無効にする関数を呼ばれてしまう可能性がある [7].

© 2012 Information Processing Society of Japan

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正常に動作できないことが予想される．

Wei らは INSeRT [10] 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，生成コードの即値やレジスタをランダム化する．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，JavaScript コードの実行時間に数%のオーバヘッドが生じてしまうことが欠点である．

Chen らによる JITDefender [11] 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メモリの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．

4. 提案手法

我々の提案は JIT Spraying 攻撃防御のためのプロセス分離である．プロセスの分離自体は，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．もしすべての要素を 1 つのプロセス上で動かすと，どれか 1 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，たとえば各タブごとに 1 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．これにより，1 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 [12] などがある．しかし，これらのプロセス分離手法はブラウザカーネルを守ることに主眼が置かれており，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さら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ものではない．たとえば，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 [13]．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．IE8 についても保護モードと呼ばれるサンドボックスのような環境で動作するが，同様の問題をかかえている．

そこで，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，最初から立ち上がっている親プロセスは自身のメモリのデータ格納用の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．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データ用領域に実行属性を付加する必要があるが，そのコード実行という目的以外の不要なスレッドをすべて排除すれば（JIT エンジンに脆弱性がないことを前提にすると）JIT シェルコードのあるアドレスへ処理を移されることはないため子プロセスでの攻撃は成功しない．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，親プロセスへの攻撃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．

IE8 や Google Chrome のプロセス分離と提案手法におけるプロセス分離を比較した図を**図 3** に示す．IE8 や Google Chrome のプロセス分離では脆弱性混入リスクの高い分離先のプロセスで JIT シェルコードが実行属性を付加されたままとなるため，JIT シェルコードが動作し，JIT Spraying 攻撃が行われてしまう．分離元のプロセスは比較的安全性が高いが，このプロセスではブラウザカーネルを動作させ JIT エンジンは動作させない．したがって JIT シェルコードは置かれない．一方，我々の提案手法におけるプロセス分離では，脆弱性混入リスクがある JIT エンジンを利用する側のプログラムを動作させる分離元のプロセスでは JIT シェルコードに実行属性を付加させないため，JIT シェルコードがあったとしても動作せず，JIT Spraying 攻撃を行うことはできない．また，分離先プロセスでは JIT シェルコードに実行属性を付加するが，こちらのプロセスでは不要なスレッドをすべて排除し，信頼できる JIT エンジン側のコードの一部だけを動作させることにより比較的安全な環境を作り，脆弱性がないことを仮定する．そのため，分離先のプロセスでも JIT Spraying 攻撃は行われない．

© 2012 Information Processing Society of Japan

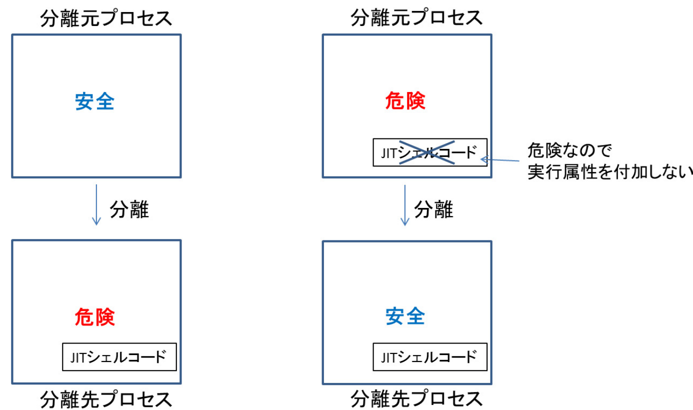

図 3 IE8 や Google Chrome のプロセス分離と提案手法のプロセス分離の比較
Fig. 3 Comparison of IE8 and Google Chrome's process isolation with proposed process isolation.

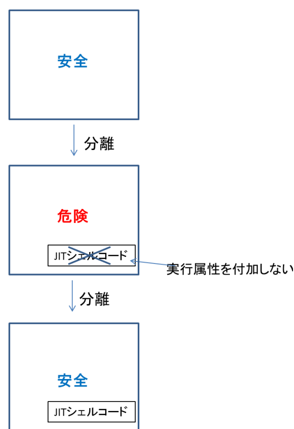

図 4 IE8 や Google Chrome のプロセス分離と提案手法のプロセス分離を同時に用いた場合
Fig. 4 A case of using both IE8 and Google Chrome's process isolation and proposed process isolation.

また，両者のプロセス分離は目的も異なり，共存も可能と考えられる．その場合には図 **4** のように 2 段構えのプロセス分離となり，ウェブブラウザ側でまず最初にブラウザカーネルから JIT エンジンとその他の脆弱性混入リスクの高いコンポーネントを分離する．JIT エンジンではさらに JIT コンパイラにより生成されたコードを実行させるための安全なプロセスを分離させる．これにより，ブラウザカーネルを守ると同時に JIT Spraying 攻撃を防ぐことが可能であると考えられる．

5. 実装

我々は提案手法を V8 JavaScript エンジン（バージョン 2.1.10）に実装した．実行環境は Ubuntu11.04 (32-bit) である．V8 は Google 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，その他の 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は，たとえばサーバサイドの JavaScript 環境である Node.js で利用されている．

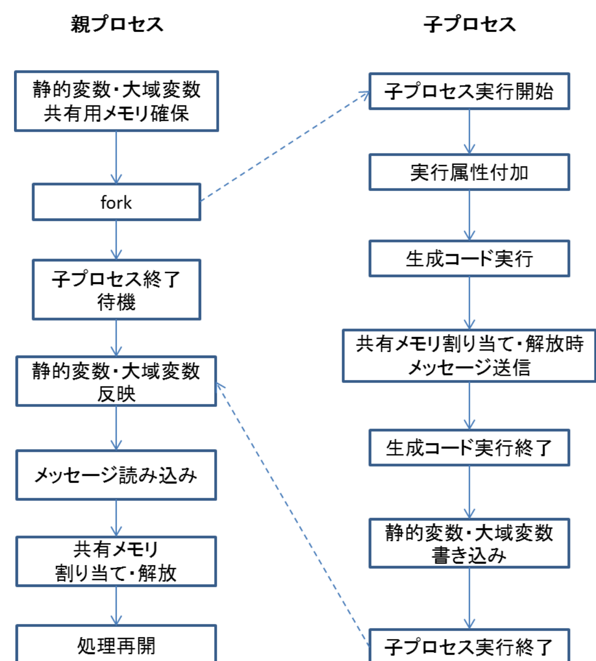

図 5 プロセス分離に必要な処理の流れ
Fig. 5 A flow of processing for process isolation.

提案したプロセス分離に必要な処理を図 **5** にまとめる．

5.1 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，プロセス間のメモリ空間は独立である．また，fork 関数で生

© 2012 Information Processing Society of Japan

成された子プロセスは開始時点では1つの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となる．子プロセスは実行開始とともに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たりすることがある．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当て・解放処理などを行い処理を再開する．

5.2　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．子プロセスの終了後にも親プロセスの実行は継続され，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うる．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する必要がある．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よい．

ヒープメモリの共有については，共有メモリを用いた．ページ単位でメモリを割当て・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は最初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[*2]．

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込む．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　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．

JITシェルコード生成のために，**図6**のようなJavaScriptコードを作成した．*jit_shellcode_str1* は何もしない関数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* は関数の引数にJITコンパイル後シェルコードとなるような値を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値である．次のforループで指定回数（この例では1,000回），NOPスレッドとなる値をシェルコードとなる値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．これにより関数呼び出し時に引数をスタックにpushする命令列がNOPスレッド+シェルコードとなりうるネイティブコードがコンパイル後に生成される．当然ながら実際にそのようなネイティブコード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JITシェルコードが生成される．今回作成したコード

[*2] 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32 MB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．

© 2012 Information Processing Society of Japan

表 1　JIT シェルコードの動作可否
Table 1　JIT shellcode work or not.

	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

```
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 6　JIT シェルコードが生成される JavaScript コード
Fig. 6　JavaScript code that can generate JIT shellcode.

では，シェルコードが実行されると/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 1 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[*3]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 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となる．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 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 1** にまとめる．

6.　評価

6.1　実験環境

Ubuntu 11.04 (32-bit) 上で実験を行った．詳細な実験環境を**表 2** 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．

[*3] 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．

© 2012 Information Processing Society of Japan

表 4 V8 ベンチマークの実行時間
Table 4 Running time of V8 benchmarks.

Benchmarks	Original V8	Our V8	Overhead (s)	Overhead (%)
Richards	1.023 s	1.034 s	0.011 s	1.08%
DeltaBlue	1.024 s	1.036 s	0.012 s	1.17%
Crypto	2.039 s	2.053 s	0.014 s	0.69%
RayTrace	1.028 s	1.041 s	0.013 s	1.26%
EarleyBoyer	2.041 s	2.060 s	0.019 s	0.93%
RegExp	1.045 s	1.054 s	0.009 s	0.86%
Splay	1.428 s	1.469 s	0.041 s	2.87%
Average	1.375 s	1.392 s	0.017 s	1.23%

表 2 実験に使用したマシンの環境
Table 2 Experiment environment.

OS	Ubuntu 11.04 (natty) 32-bit
Kernel	Linux 2.6.38-13-generic
プロセッサ	Intel (R) Core (TM) i7 CPU M640 2.80 GHz
メモリ	3.4 GiB 認識

表 3 V8 ベンチマークのスコア
Table 3 Scores of V8 benchmark.

Benchmarks	Original V8	Our V8
Richards	5501.76±67.74	5477.19±72.64
DeltaBlue	6927.83±78.72	6740.34±68.51
Crypto	4933.58±32.10	4923.95±53.25
RayTrace	10112.78±239.79	10170.80±73.44
EarleyBoyer	21459.93±319.90	21587.61±139.89
RegExp	3089.20±25.30	3081.28±29.72
Splay	13743.03±139.42	13567.16±148.92
TotalScore	65768.12±453.60	65548.33±246.85

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．

6.2 性能評価

6.2.1 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 [15] 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 3** にその結果を示す．表 3 の値は各ベンチマークを 1,000 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 0.34%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ヘッドがほぼないためと考えられる．わずかなオーバ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れる．

また，個々のベンチマークを見ると RayTrace と Earley-Boyer 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．

6.2.2 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 4** に示す．表 4 の値はベンチマークを 1,000 回実行したときの実時間[*4]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 0.009〜0.041 s，平均して 0.017 s のオーバヘッドがあり，割合にすると 0.69〜2.87%，平均では 1.23%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，オーバヘッドは生成コード実行中にかかるオーバヘッド，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，割当て・解放された共有メモリの反映など）がオーバヘッドとしてかかるためである．

6.2.3 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

*4 bash の組み込みの time コマンドで表示される real の値

© 2012 Information Processing Society of Japan

情報処理学会論文誌 Vol.53 No.9 2302–2312 (Sep. 2012)

表 6 既存研究との比較
Table 6 Comparison with related works.

	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	脆弱な時間
Our V8	必要	不要	なし	なし	なし	なし
JIT shellcode detection [8]	不要	不要	なし	あり	あり	なし
JITSec [9]	不要	必要	なし	あり	なし	なし
INSeRT [10]	必要	不要	あり	なし	あり	なし
JITDefender [11]	必要	不要	なし	なし	なし	あり

表 5 共有メモリ使用量
Table 5 Usage of shared memory.

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 60 MiB
gnome-system-monitor	18.1 MiB
gedit	13.6 MiB
nautilus	12.7 MiB

最大で約 60 MiB*5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 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ともに示す．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．

6.3 既存研究との比較

表 6 に既存研究との比較を示す．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．

まず JIT shellcode detection についてであるが，この手法は文献 [8] 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うるため，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うるため，フォルスネガティブも存在する．

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からのシステムコール呼び出しがブロックされる．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うるかもしれないと考えると，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 [10] 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，まったく無防備な状態となるため脆弱な時間が存在する．

これらと比較した我々の提案手法の利点は，まず OS の修正が必要でないことがあげられる．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ため，そもそも攻撃は行われない．そのため，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．最後に，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 に対するアドバンテージとなる．

*5 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．

© 2012 Information Processing Society of Japan

6.4 現在のプロセス分離手法実装の問題点

6.4.1 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．たとえば 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かか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フェー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．この方法をとれば，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．

6.4.2 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 1 つだけなのでデータの不整合は生じない．

7. 結論

本稿では，JIT Spraying 攻撃を防ぐための手法として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．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．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 領域の共有を行った．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，実行中ともに JIT シェルコードが機能しないことを確認した．オーバヘッドについては，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分かった．

参考文献

[1] Grevstad, E.: CPU-Based Security: The NX Bit (online), available from ⟨http://www.hardwarecentral.com/reviews/article.php/12095_3358421_/CPU-Based-Security-The-NX-Bit.htm⟩ (accessed 2012-01-04).

[2] Howard, M., Miller, M., Lambert, J. and Thomlinson, M.: Windows ISV Software Security Defenses (online), available from ⟨http://msdn.microsoft.com/en-us/library/bb430720.aspx⟩ (accessed 2012-01-04).

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

[4] Google Inc.: v8 – V8 JavaScript Engine – Google Project Hosting (online), available from ⟨http://code.google.com/p/v8/⟩ (accessed 2011-12-23).

[5] 大澤文孝：IE8 の 16 倍高速に—Internet Explorer 9 の実力：ITpro（オンライン），入手先 ⟨http://itpro.nikkeibp.co.jp/article/COLUMN/20110708/362205/⟩（参照 2011-12-23）．

[6] Torres, M.: PyPy's Speed Center (online), available from ⟨http://speed.pypy.org/⟩ (accessed 2011-12-23).

[7] Durden, T.: Bypassing PaX ASLR protection, *Phrack*, Vol.59 (2002).

[8] Bania, P.: JIT spraying and mitigations (2010) (online), available from ⟨http://arxiv.org/abs/1009.1038v1⟩.

[9] De Groef, W., Nikiforakis, N., Younan, Y. and Piessens, F.: JITSec: Just-in-time security for code injection attacks, *Benelux Workshop on Information and System Security* (*WISSEC 2010*) (2010).

[10] Wei, T., Wang, T., Duan, L. and Luo, J.: INSeRT: Protect Dynamic Code Generation against spraying, *2011 International Conference on Information Science and Technology* (*ICIST*), pp.323–328 (online), DOI:

© 2012 Information Processing Society of Japan

10.1109/ICIST.2011.5765261 (2011).
[11] Chen, P., Fang, Y., Mao, B. and Xie, L.: JITDefender: A Defense against JIT Spraying Attacks, *Future Challenges in Security and Privacy for Academia and Industry*, IFIP Advances in Information and Communication Technology, Vol.354, pp.142–153, Springer Boston (2011).
[12] Grier, C., Tang, S. and King, S.T.: Secure Web Browsing with the OP Web Browser, *Proc. 2008 IEEE Symposium on Security and Privacy*, Washington, DC, USA, IEEE Computer Society, pp.402–416 (online), DOI: 10.1109/SP.2008.19 (2008).
[13] Barth, A., Jackson, C., Reis, C. and Google Chrome Team: The Security Architecture of the Chromium Browser (2008) (online), available from ⟨http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf⟩.
[14] Engelschall, R.S.: OSSP mm (online), available from ⟨http://www.ossp.org/pkg/lib/mm/⟩ (accessed 2011-12-23).
[15] Google Inc.: V8 Benchmark Suite - version 5 (online), available from ⟨http://v8.googlecode.com/svn/data/benchmarks/v5/run.html⟩ (accessed 2011-12-23).

推薦文

近年，Blazakis により，JIT（Just-In-Time）コンパイラを悪用した JIT Spraying という攻撃手法が示され，攻撃者はこの攻撃手法を利用して DEP（Data Execution Prevention）や ASLR（Address Space Layout Ramdomization）を回避し，バッファオーバフロー攻撃などの脆弱性を利用した攻撃ができる．本稿は，JIT Spraying 攻撃を防止するための手段としてプロセス分離を用いた手法を提案しており，提案手法では，JIT コンパイルされたコードの実行直前に新たなプロセスを生成し，そのプロセス上で JIT コンパイルされたコードを実行するという方法がとられている．また，提案手法は，V8 JavaScript エンジン上で実装されており，実行時間のオーバヘッドは，実用上，問題にならない程度であると報告されている．本稿は，JIT Spraying 攻撃対策に貢献する論文と考え，推薦論文として推薦する．

（コンピュータセキュリティシンポジウム 2011
プログラム委員長　四方順司）

市川 顕

平成 24 年 3 月東京大学大学院情報理工学系研究科電子情報学専攻修士課程修了．同年 4 月富士通株式会社入社．ソフトウェア開発に従事．

松浦 幹太　（正会員）

平成 4 年 3 月東京大学工学部電気工学科卒業．平成 9 年 3 月東京大学大学院工学系研究科電子工学専攻博士課程修了．博士（工学）．平成 9 年 4 月東京大学生産技術研究所助手等を経て，平成 14 年 4 月同助教授（法令改正により平成 19 年 4 月より准教授）．平成 12 年 3 月から平成 13 年 3 月までケンブリッジ大学客員研究員．情報セキュリティの研究に従事．ACM シニア会員（2009 年），電子情報通信学会シニア会員（2009 年），IEEE シニア会員（2011 年）．IACR，社会・経済システム学会会員．日本セキュリティマネジメント学会理事．情報処理学会コンピュータセキュリティ研究会主査．著書に情報セキュリティ概論（昭晃堂），セキュリティマネジメント学—理論と事例（共立出版）等．

© 2012 Information Processing Society of Japan