x86 JIT コンパイラ上で任意コードを実行する方法

竹 迫 良 範†

近年 Web ブラウザは JavaScript の実行速度を向上させるために x86 JIT エンジンを搭載しているが，これらの JIT エンジンには任意の x86 コードを実行されてしまう脆弱性が存在する．JIT エンジンは自分自身で生成したコードを実行するため，書き込み可能なメモリ領域を確保し，実行フラグを立てる．最近の Windows OS では，DEP（データ実行防止）機能があり，スタックやヒープ領域で x86 コードが実行されることを防いでいる．さらに ASLR（アドレス空間配置のランダム化）では，攻撃対象のアドレスを推測困難にすることによって，攻撃が成功することを防止している．しかし，DEP や ASLR を突破する新しい攻撃として，Return-Oriented Programming や JIT-Spraying と呼ばれる手法が知られるようになった．本発表では実際に JavaScript の JIT エンジンを使って任意の x86 コードを実行する方法を解説し，このような x86 JIT コンパイラの脆弱性を保護する安全なコード生成手法について考察する．

How to execute arbitrary code on x86 JIT Compiler

Yoshinori Takesako†

The modern Web browsers have own x86 JIT engine, in order to make high-speed JavaScript possible. However, they have some arbitrary code execution vulnerabilities. Because the JIT engine allocates sections of memory and marks them as executable. The modern Windows operating system has security features. DEP (Data Execution Prevention) prevents the stack and heap memory areas from being executable. ASLR (Address Space Layout Randomization) helps to prevent certain exploits from succeeding by making it more difficult for an attacker to predict target addresses. But the Return-Oriented Programming and the JIT-Spraying are known as few method of breaking through DEP and ASLR. I explain how to generate arbitrary x86 codes from JavaScript by controlling a JIT engine. I would like to discuss together how to protect vulnerability issues with x86 JIT compiler.

1. はじめに

最近の Web ブラウザ（Firefox, Chrome, Opera, IE）は JavaScript の高速化に力を入れており，x86 JIT エンジンを搭載していることが当前となっている．

2. バッファオーバーフロー脆弱性

バッファの溢れによりメモリ領域が破壊され，プログラムが意図しない動作をしてしまう脆弱性のことをバッファオーバーフローと言う．データ境界を正しくチェックせずにスタック変数上で文字列のコピーを行なった場合，関数呼び出し時にスタックに積んでいたリターンアドレスが別の値に上書きされ，次の命令が別の場所にジャンプしてしまう．このとき，攻撃者がスタック上にある変数のアドレスを推測できてしまうと，文字列コピー時に悪意のあるシェルコードを配置して，リターンアドレスをうまくシェルコードの開始アドレスに書き換えればバッファオーバーランが成立してしまう（**図 1**）．このような脆弱性が Web ブラウザに存在しているとセキュリティ上非常に問題である．

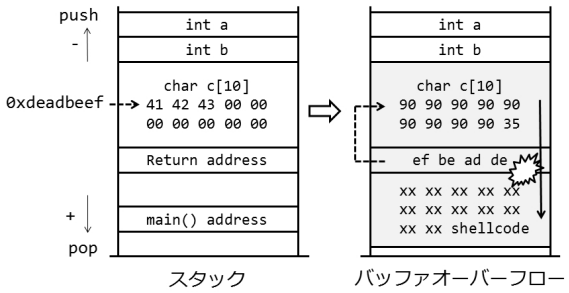

図 1 バッファオーバーフローによる x86 任意コード実行の仕組み

† サイボウズ・ラボ株式会社
Cybozu Labs, Inc.

```
#include <stdio.h>

unsigned char x86[] = {
  0x8b, 0x44, 0x24, 0x04, // mov eax,[esp+4]
  0x03, 0x44, 0x24, 0x08, // add eax,[esp+8]
  0xc3 // ret
};

int main()
{
    int (*add)(int,int) = (void *)x86;
    printf("add(1, 2): %d\n", add(1, 2));
}
```

図 2 x86 命令で足し算する実行コードを埋め込んだ C プログラム

3. DEP による防御

セキュリティ対策強化のため，Windows XP SP2 以降からデータ実行防止機能（DEP）が搭載されるようになった．DEP により，許可されていないヒープ領域やスタック上でコード実行ができなくなる．一昔前の環境では**図 2** の C プログラムをコンパイルすればそのまま x86 コードを実行できたが，DEP が有効になっている環境だとエラーとなり実行できない．

JIT コンパイラ (Just-In-Time compiler) は自分自身で生成した x86 コードを実行するため，書き込み可能なデータ領域を確保し，OS の API を呼び出して実行フラグを立てるようにしている．

Windows 環境では malloc ではなく VirtualAlloc API を使って PAGE_EXECUTE フラグ付きのヒープ領域を確保する（**図 3**）．Linux 環境では malloc で確保したデータ領域に対して mprotect システムコールを呼び出して実行ビットを有効にする．通常は OS が管理するページ単位 (4KB 等) で範囲を指定する．

4. ASLR による防御

ASLR(Address Space Layout Randomization) とは，実行プロセスの開始アドレスやスタックの位置，共有ライブラリのロード位置などをランダムに配置し，攻撃者から絶対アドレスを予測しにくくするセキュリティ防御技術である．

Windows Vista，7，Windows Serever 2008 以降で ASLR がデフォルトで有効となった．これにより攻撃者がプログラムのバッファオーバーフロー脆弱性を発見したとしても，OS 側の DEP と ASLR の保護機能により，リモートからの任意コード実行は大幅に難しくなった．Linux はカーネル 2.6.12 以降から ASLR の機能が有効になっている．

```
#include <stdio.h>
#include <string.h>
#include <windows.h>

unsigned char x86[] = {
  0x8b, 0x44, 0x24, 0x04, // mov eax,[esp+4]
  0x03, 0x44, 0x24, 0x08, // add eax,[esp+8]
  0xc3 // ret
};

int main() {
  void *p = VirtualAlloc(NULL, 4096 * 1,
        MEM_COMMIT, PAGE_EXECUTE_READWRITE);
  if (p && memcpy(p, x86, sizeof(x86))) {
    int (*add)(int,int) = (void *)p;
    printf("add(1, 2): %d\n", add(1, 2));
    VirtualFree(p, 0, MEM_RELEASE);
  }
}
```

図 3 実行フラグ付きのヒープ領域を確保して x86 命令を呼び出す

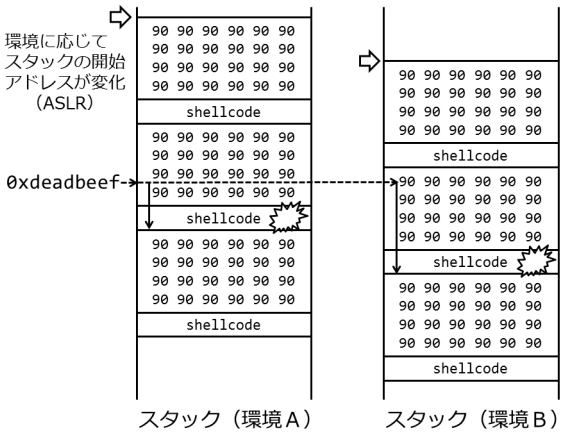

図 4 Heap-Spraying では，正確な番地を指定する必要がなくなる

4.1 Heap-Spraying の登場

現在では Heap-Spraying と呼ばれる新しい攻撃手法によって ASLR が攻略されることとなった．攻撃者は実行フラグの立ったデータ領域に大量の NOP 命令（0x90）とシェルコードを散りばめる．ASLR でアドレスの開始位置が推測できない場合でも，スプレーを吹き付けるように大量のメモリに対して上記 NOP 命令とシェルコードを塗り潰しておけば，厳密なアドレスを指定しなくても大体の範囲があっていれば攻撃が成立してしまう（**図 4**）．半分は運に任せた手法なので 100%攻撃が成功するとは限らないが，ASLR のエントロピーが低い場合は脆弱性となり得る．

4.2 シェルコードの実行防止効果

Windows XP SP3 で電卓（cacl.exe）を起動する

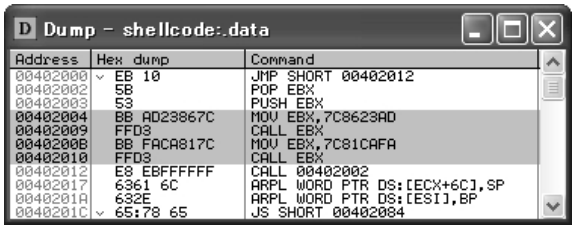

図 5 電卓（cacl.exe）を起動するシェルコードの例

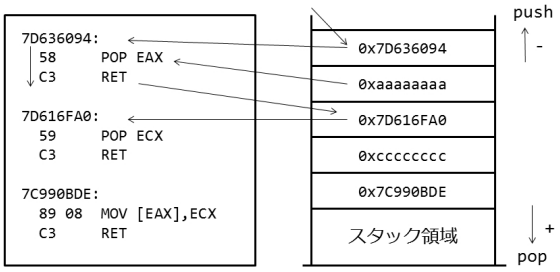

図 6 コード断片を利用して MOV [EAX],ECX を実行する

シェルコードを**図 5** に示す☆．0x7c8623ad 番地は kernel.dll の WinExec 関数，0x7c81cafa 番地は ExitProcess 関数がある場所であるが，ASLR が有効になっている Windows Vista，Windows 7 の環境では起動時に毎回 DLL のアドレスが変化するようになっている．

5. Return-Oriented Programming

Return-Oriented Programming(ROP) とは，実行可能メモリ領域にある 2〜3 バイト程度のコード断片をたくさん利用して目的のシェルコードを実行する攻撃テクニックである（**図 6**）．

5.1 ROP による攻撃手法

まず，攻撃者は実行可能フラグの立っているメモリ領域の中で 0xC3（RET 命令）が出現する箇所を検索し，その直前で POP 命令や XCHG 命令が実行されるようなアドレスを調べる．たとえば，Windows XP SP3 の環境で shell32.dll のコード実行領域は 0x7D5B1000〜0x7D7AEFFF 番地にマップされる（**図 7**）．

このメモリ領域の中で 0x7D636094 番地にジャンプすると 0x58（POP EAX 命令）が存在し，その次に 0xC3（RET 命令）がある（**図 8**）．

レジスタを操作する POP 命令の次に RET 命令が存在する場所や，MOV 命令の次に RET 命令がある場所のアドレス番地を利用すると図 6 のように特定の値をスタックに積んでおくだけで，攻撃者は書き込みたいメモリ番地に任意の値を書き込むことができる．

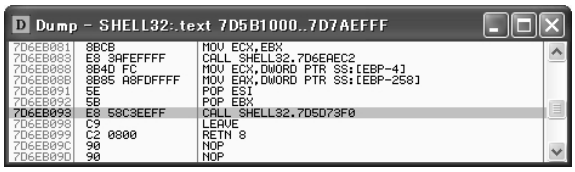

図 7 実行可能なメモリ領域の中で C3 が出現する箇所を検索する

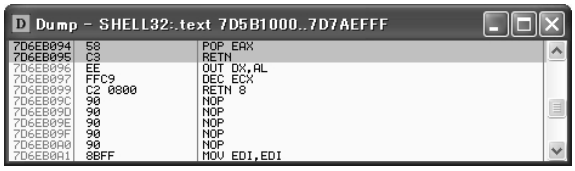

図 8 0x7D636094 に POP と RET の組み合わせが存在する

図 6 で実行される命令は以下と等価である．

```
MOV  EAX, 0xaaaaaaaa
MOV  ECX, 0xcccccccc
MOV [EAX], ECX
```

このように Return-Oriented Programming では，スタックに積まれたアドレスへの RET 命令をたくさん繰り返しながら目的のプログラムを実行する．

5.2 DEP の回避

OS 側のデータ実行防止 (DEP) によりスタック上のコード実行が禁止されている場合でも，スタック溢れによるバッファオーバーフロー脆弱性に ROP の手法を適用すると，絶対アドレスに散りばめられたコード断片を DEP を回避しながら実行することができる．

5.3 Mac OS X の脆弱性

Mac OS X では ASLR と同等の機能はないが，スタック領域でのコード実行は禁止されており，共有ライブラリがロードされるアドレスはインストール時に変化するようになっている．しかし，ダイナミックリンカの dyld ライブラリは特定のアドレスにロードされるようになっており，`__IMPORT` セクションは読み書きも実行も可能な領域であるため，ROP による攻撃が可能である[1)]．

5.4 Windows XP での攻撃事例

Adobe Reader 9.0 で使用される bib.dll が毎回特定のアドレスにロードされることが悪用され，実際に Windows XP SP3 上で DEP を回避するゼロデイ攻撃として ROP のテクニックが利用された[2)]．

☆ Windows Xp Pro SP3 (calc.exe) Shellcode 31 Bytes
http://www.shell-storm.org/shellcode/files/shellcode-612.php

6. JavaScript における数値の扱い

JavaScript の言語仕様は，ECMAScript(ECMA-262)[3] として定義されている．JavaScript の数値は，すべて IEEE 754 の倍精度浮動小数点数であると仕様で定められている（**図 9**）．

6.1 浮動小数点数から 0x90 を生成する方法

JavaScript の変数 x に 0x90909090 をそのまま代入したとしても，格納されるメモリイメージは 32bit の 0x90909090 とはならず，倍精度 64bit 浮動小数点数のビット表現「0x41e2121212000000」となる（**図 10**）．

格納されるメモリイメージに 0x90（NOP 命令）を列挙したい場合は，JavaScript の変数にそのようなメモリイメージに変換される浮動小数点数を代入する．

JavaScript x86 JIT コンパイラは，**図 11** のプログラムを JavaScript から x86 機械語に変換する（図 10）．

具体的に以下の JavaScript プログラム

```
var x = 0x90909090;
```

は，次の x86 機械語命令に置き換わる．

```
0150FF53:  MOV [EDI+0x638],0x12000000
0150FF5D:  MOV [EDI+0x63C],0x41E21212
```

多くの x86 JIT コンパイラでは，浮動小数点数のビット表現を 2 つの 32bit 整数に分割して代入する．

6.2 浮動小数点数のビット表現

機械語の命令列の中に 0x90 を列挙したい場合は，そのような値になる浮動小数点を指定して代入する．

具体的には，以下の JavaScript プログラム

```
var y = -6.828527034422786e-229;
```

を JIT で実行すると，次の機械語が生成される．

```
0150FF67:  MOV [EDI+0x640],0x90909090
0150FF71:  MOV [EDI+0x644],0x90909090
```

```
var x = 0x90909090; // NG
var y = -6.828527034422786e-229; // OK
var z = -6.828527034422786e-229; // OK
```

図 11 メモリイメージに 0x90 を列挙したい JavaScript コード

このように浮動小数点数のビット表現を知っていれば，JIT コンパイラが生成する命令列の中に攻撃者が欲しいバイト列を埋め込むことができる．

6.3 XOR ビット演算

JavaScript の数値は基本的に浮動小数点数であるが，XOR や AND，OR のビット演算を行うときのみ 32bit 整数に一時的に変換されることが ECMAScript の仕様に明記されている[3]．これを利用して近年の JavaScript JIT コンパイラでは，数値のビット演算が行われるタイミングで 32bit 整数演算の機械語命令を生成するようになっている．

```
var a = 0x12345678 ^ 0x12345678
      ^ 0x12345678 ^ 0x12345678
      // ...
```

上記 JavaScript コードは JIT コンパイラによって以下の x86 機械語命令に変換されても仕様上問題ない．

```
+00: B8 78563412   MOV EAX,0x12345678
+05: 35 78563412   XOR EAX,0x12345678
+0A: 35 78563412   XOR EAX,0x12345678
+0F: 35 78563412   XOR EAX,0x12345678
   :
```

このような JIT コンパイラの性質を利用して，攻撃コードをメモリ上に展開するのが JIT-Spraying と呼ばれる手法である．JavaScript だけではなく，Flash Player に搭載されている ActionScript VM も JIT-Spraying の研究対象であり，実際の攻撃事例も多い[4]．

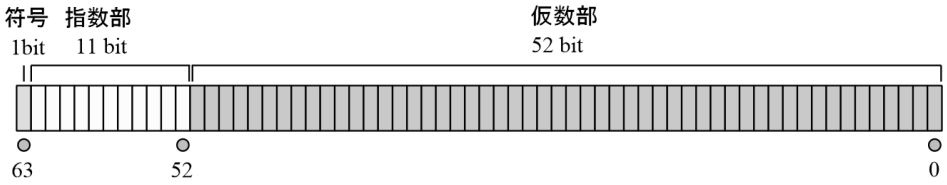

図 9 IEEE 754 浮動小数点数のビット表現（倍精度 64 ビット）

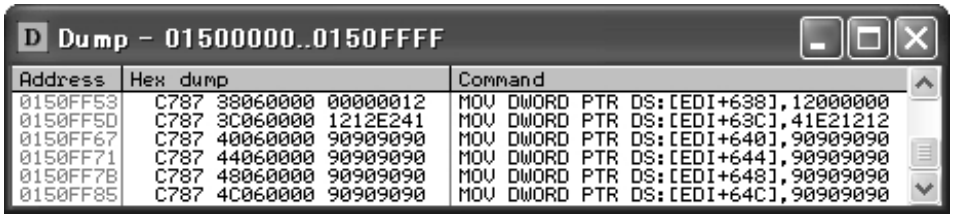

図 10 浮動小数点数の代入は 2 回の MOV 命令に JIT される

7. JIT-Spraying

JIT-Spraying は既存の OS のセキュリティ機能である DEP と ASLR を回避する新しい攻撃手法である．

バッファオーバーフローが発生するスタック領域や通常のヒープ領域では DEP によりコード実行が禁止されている上，ダイナミックリンクライブラリ (DLL) がロードされる開始アドレスの位置は ASLR によって起動時に毎回値が異なっているため，最近の Windows OS ではシェルコードの作成が非常に困難になっている．しかし，スクリプト言語を高速に実行するため，最近の JavaScript や ActionScript などの処理系では x86 JIT コンパイラの搭載が当然となっている．

JIT-Spraying は JIT コンパイラが機械語のコード生成と実行に利用しているメモリ領域に対して攻撃コードを送り込む．具体的には，攻撃者は JIT コンパイラによって実行が許可されたヒープ領域に対して，図 4 のような大量の NOP 命令（0x90）とシェルコードを散りばめる．

7.1 JIT コンパイラで 0x90 を生成する方法

JIT コンパイラ上で大量の 0x90(NOP 命令) を生成するには，たくさんの 32bit 整数に対して XOR ビット演算を適用するコードを書けば良いことが知られている[4]．ただし実行が成立する確率は 100%ではない．

具体的には 32bit の整数 0x3c909090 を利用する．

```
x ^= 0x3c909090 ^ 0x3c909090 ^ 0x3c909090
   ^ 0x3c909090 ^ 0x3c909090 ^ 0x3c909090
   ^ 0x3c909090 ^ 0x3c909090 ^ 0x3c909090
   // ...
```

JavaScript で大量の 0x3c909090 と XOR ビット演算するコードを書くと，JIT コンパイラにより以下の x86 機械語が生成される（**図 12**）．

```
+00: 35 909090903C    XOR EAX,0x3C909090
+05: 35 909090903C    XOR EAX,0x3C909090
+0A: 35 909090903C    XOR EAX,0x3C909090
+0F: 35 909090903C    XOR EAX,0x3C909090
```

ここで，もしも実行開始アドレスが+00 ではなく命令途中の+04 にずれた場合，x86 命令の解釈が変わり，以下の機械語が実行されることになる（**図 13**）．

```
+04: 3C 35            CMP AL,0x35
+06: 90               NOP
+07: 90               NOP
+08: 90               NOP
+09: 3C 35            CMP AL,0x35
+0B: 90               NOP
+0C: 90               NOP
+0D: 90               NOP
+0E: 3C 35            CMP AL,0x35
```

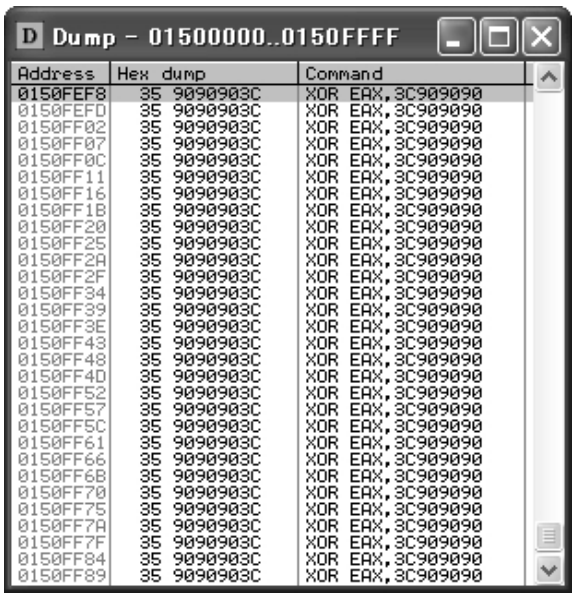

図 12 大量の XOR EAX,0x3C909090 命令を列挙する

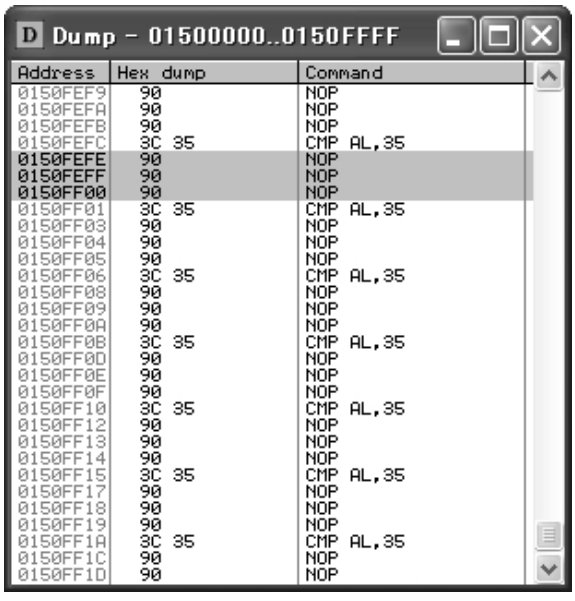

図 13 開始位置が 1 バイトずれると NOP 命令に変化する

7.2 JIT-Spraying で任意コードを実行する

JIT コンパイラの生成する 5 バイトの XOR 命令を利用すると，境界の 2 バイトを CMP AL,0x35 とみなせば，その次の 3 バイトで NOP 以外の自由な命令を書き込める．たとえば，次の XOR 命令を 4 回実行する x86 コードを見てみよう．

```
+00: 35 9090B87F    XOR EAX,0x7FB89090
+05: 35 BBAA903C    XOR EAX,0x3C90AABB
+0A: 35 B4CC903C    XOR EAX,0x3C90CCB4
+0F: 35 B0DD503C    XOR EAX,0x3C50DDB0
```

開始アドレスが+00 ではなく+01 になった場合，以下の x86 命令が実行される．

```
+01: 90             NOP
+02: 90             NOP
+03: B8 7F35BBAA    MOV EAX,0xAABB357F
+08: 90             NOP
+09: 3C 35          CMP AL,0x35
+0B: B4 CC          MOV AH,0xCC
+0D: 90             NOP
+0E: 3C 35          CMP AL,0x35
+10: B0 DD          MOV AL,0xDD
+12: 50             PUSH EAX
```

これは任意の 32bit 整数 0xAABBCCDD を EAX に代入してスタックに PUSH するコードと等価である．次に EBX も任意の値に書き換えることができて，

```
+00: 35 9090BB7F    XOR EAX,0x7FBB9090
+05: 35 2211903C    XOR EAX,0x3C901122
+0A: 35 B733903C    XOR EAX,0x3C9033B7
+0F: 35 B344903C    XOR EAX,0x3C9044B3
+14: 35 5889033C    XOR EAX,0x3C038958
```

これらの 3 バイトの空き領域を利用すれば任意のアドレスに対して任意のコードを書き込むことができる．

```
+03: BB 7F352211    MOV EBX,0x1122357F
+08: 90             NOP
+09: 3C 35          CMP AL,0x35
+0B: B7 33          MOV BH,0x33
+0D: 90             NOP
+0E: 3C 35          CMP AL,0x35
+10: B3 44          MOV BL,0x44
+12: 90             NOP
+13: 3C 35          CMP AL,0x35
+15: 58             POP EAX
+16: 89 03          MOV [EBX],EAX
```

JIT-Spraying による攻撃では，これらの手法を組み合わせて実行フラグの立ったヒープ領域に対して大量の NOP 命令（0x90）とシェルコードを散りばめる（**図 14**）．実際には 1/5 の確率で通常の XOR 命令の先頭にジャンプすることがあるため攻撃が成立する確率は約 80%以下である．

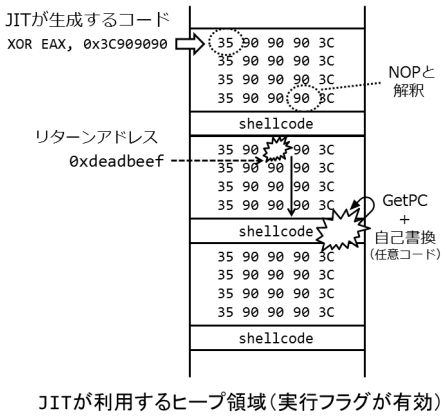

図 14 JIT-Spraying で DEP と ASLR を攻略する方法

シェルコード中では現在実行中の EIP レジスタの値を取得し，書き込みフラグと実行フラグが両方有効になっているアドレスに対して任意の値を書き込んで，そこにジャンプすれば任意コードが実行できる．

FSTENV 命令を利用して☆EIP を取得する GetPC コードを XOR 命令の中で書くと**図 15** となる[5]．
3 バイトの隙間では CALL 命令は使用できない．

```
+00: D9 D0          FNOP
+02: 54             PUSH ESP
+03: 3C 35          CMP AL,0x35
+05: 58             POP EAX
+06: 90             NOP
+07: 90             NOP
+08: 3C 35          CMP AL,0x35
+0A: 6A F4          PUSH -0x0C
+0C: 59             POP ECX
+0D: 3C 35          CMP AL,0x35
+0F: 01 C8          ADD EAX,ECX
+11: 90             NOP
+12: 3C 35          CMP AL,0x35
+14: D9 30          FSTENV DS:[EAX]
```

図 15 FSTENV 命令を利用して EIP を取得する GetPC コード

☆ Re:GetPC code(was:Shellcode from ASCII) Jun 27 2003 08:22
http://www.securityfocus.com/archive/82/327100/2009-02-24/1

8. 評　　価

Mozilla Firefox に搭載されている JavaScript JIT エンジン JaegerMonkey（JavaScript-C 1.8.5+ 2011-04-16 版）を利用して XOR ビット演算を 1000 万回ループする JavaScript プログラム（**図 16**）を実行した（表 1）.

```
for (var i = 0; i < 10000000; i++) {
  a ^= 0x11111111 ^ 0x22222222 ^ 0x33333333;
  b ^= 0x44444444 ^ 0x55555555 ^ 0x66666666;
  c ^= 0x77777777 ^ 0x88888888 ^ 0x99999999;
}
```

図 16　JavaScript で xor ビット演算を 1000 万回ループ

JIT なしの通常の JavaScript インタプリタで実行にかかった時間は 3,645 ミリ秒であったが，method JIT を適用すると約 1/30 の時間の 115 ミリ秒で同じ処理を終え，trace JIT を適用すると，さらに約 1/3 の時間の 38 ミリ秒で済んだ（表 1）.

8.1 JaegerMonkey method JIT

JaegerMonkey の method JIT で動的に生成される x86 コードを見ると，EAX ではなく EBX レジスタに対して XOR 演算が行われている（図 17）. XOR EBX 命令は 5 バイトではなく 6 バイトの命令長のため，残念ながら，さきほどの JIT-Spraying のテクニックをそのまま適用することができない.

8.2 JaegerMonkey trace JIT

JaegerMonkey の trace JIT で動的に生成される x86 コードはさらに最適化が進み，途中の XOR 演算で EAX レジスタを使用している（図 18）. 評価に使用した JavaScript プログラム（図 16）の `b ^= 0x44444444 ^ 0x55555555 ^ 0x66666666` の箇所を大量の NOP 命令とシェルコードを生成するようなプログラムに置き換える. もしもブラウザのバグでリターンアドレスを書き換えられる脆弱性が存在すれば，該当領域にジャンプさせることで任意コードの実行が可能であり，危険である[6].

9. 考　　察

JIT コンパイラが生成する x86 命令は，EAX レジスタを使用する場合において最短形式を使うものが多いが[7]，以下のようにあえて長い表現を利用することで攻撃の成功率を下げることができるのではないか.

```
+00: 35    12345678   XOR EAX,0x78563412
+05: 81 F0 12345678   XOR EAX,0x78563412
```

今回の JIT コンパイラの問題は，生成する x86 機械語の命令の中で 32bit の即値 (imm32) を攻撃者が自由に指定できてしまうのが問題である. これは文字列を扱うときと同様に，実行フラグが禁止されている別のヒープ領域に 32bit の整数値を格納し，アドレス参照の形で XOR 命令を実行すれば解決できる.

```
+00: 33 87 44040000   XOR EAX,[EDI+0x444]
+06: 33 9F 88040000   XOR EBX,[EDI+0x488]
```

他に 32bit の数値が機械語の中に出現するのは，メモリ参照時のポインタであるが，これらのアドレスを攻撃者がコントロールするのは難しいため，現状では対策しなくてもよいと考える.

セキュリティ上の理由により，そろそろ現代のコンピュータプログラムも古来のハーバードアーキテクチャのように，実行プログラムのあるコード領域とデータ領域の境界を明確に分ける必要があるのではないだろうか. 現在の x86 JIT コンパイラの実装の多くは，まだその境界が曖昧である.

参　考　文　献

1) Dino A. Dai Zovi : *Mac OS X x86 Return-Oriented Exploitation*, Trail of Bits (2010) http://trailofbits.files.wordpress.com/2010/07/mac-os-x_roe.pdf.
2) VU#486225:*Adobe Flash ActionScript AVM2 newfunction vulnerability*, US-CERT (2010) http://www.kb.cert.org/vuls/id/486225 .
3) Standard ECMA-262 ECMAScript Language Specification Edition 5.1, (2011) http://www.ecma-international.org/publications/standards/Ecma-262.htm.
4) Alexey Sintsov : *Writing JIT Shellcode for fun and profit*, DSecRG (2010) http://dsecrg.com/files/pub/pdf/Writing%20JIT-Spray%20Shellcode%20for%20fun%20and%20profit.pdf.
5) Dion Blazakis : *Interpreter Exploitation: Pointer Inference and JIT Spraying*, (2010) http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf.
6) Alexey Sintsov: *JIT-Spray Attacks and Advanced Shellcode*, HITB Amsterdam (2010) http://dsecrg.com/pages/pub/show.php?id=26
7) Chris Rohlf, Yan Ivnitskiy: *Attacking Clientside JIT Compilers*, Matasano Security (BlackHat 2011) http://www.matasano.com/research/jit/

表 1 JavaScript-C 1.8.5+ 2011-04-16 版での実行結果

JaegerMonkey	実行コマンド	x86 生成コード	時間 [ミリ秒]
JavaScript JIT なし	`js.exe xor.js`	インタプリタ	3,645
method JIT あり	`js.exe -m xor.js`	**図 17** 参照	115
trace JIT あり	`js.exe -j xor.js`	**図 18** 参照	38

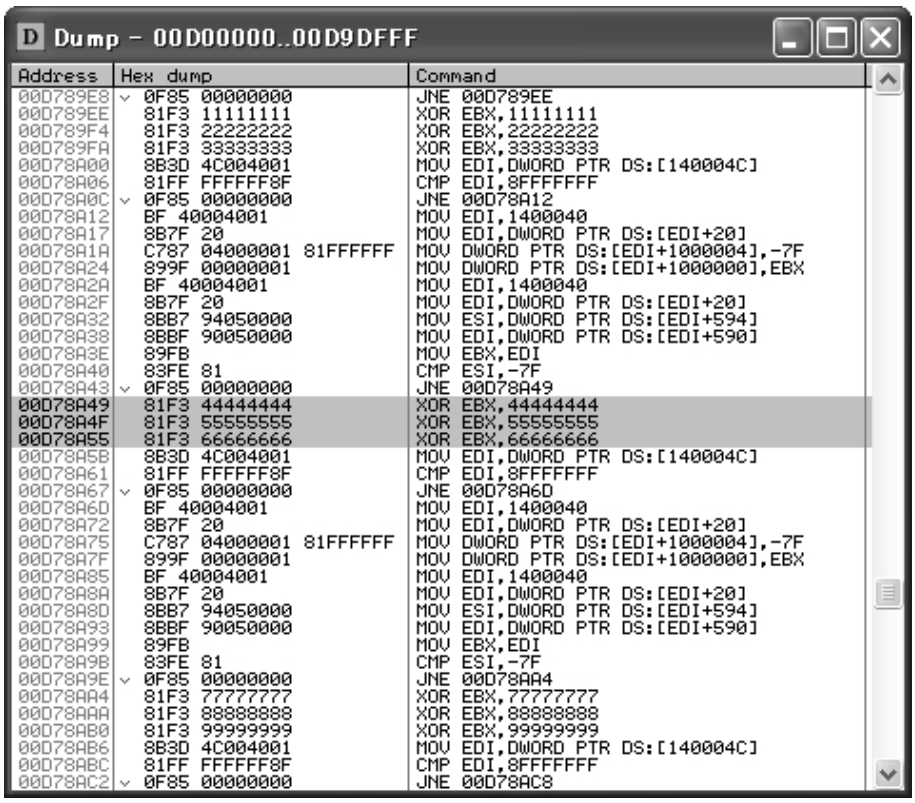

図 17 JaegerMonkey method JIT が生成する x86 コード

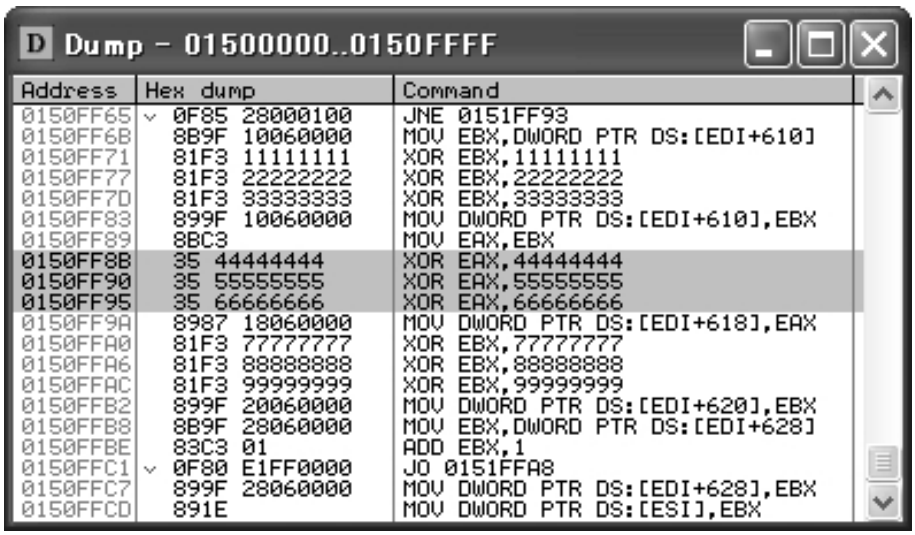

図 18 JaegerMonkey trace JIT が生成する x86 コード