

Guide du Développeur Metasploit 3.0

Guide du Développeur Metasploit 3.0

The Metasploit Staff – <https://www.metasploit.com>
msfdev(@)metasploit.com

Dernière modification (Version française): 08/06/2006
Jérôme ATHIAS – <https://www.securinfos.info>
webmaster(@)securinfos.info

Contenu

1 Introduction

- 1.1 Pourquoi Ruby?
- 1.2 Conception et Architecture

2 Rex

- 2.1 Assembleur
 - 2.1.1 Packing d'entier
 - 2.1.2 Ajustement de pointeur de Pile (Stack pointer)
 - 2.1.3 Génération d'opcode spécifique à l'Architecture
- 2.2 Encodage
- 2.3 Exploitation
 - 2.3.1 Egghunter
 - 2.3.2 Génération d'enregistrement SEH
- 2.4 Jobs
- 2.5 Logging
 - 2.5.1 LEV0 -Défaut (Default)
 - 2.5.2 LEV1 -Extra
 - 2.5.3 LEV2 -Verbeux (Verbose)
 - 2.5.4 LEV3 -Folie (Insanity)
- 2.6 Base de données d'Opcodes
- 2.7 Post-exploitation
- 2.8 Protocoles
 - 2.8.1 DCERC
 - 2.8.2 HTTP
 - 2.8.3 SMB
- 2.9 Services
- 2.10 Sockets
 - 2.10.1 Classes Comm
 - 2.10.2 Sockets TCP

- 2.10.3 Sockets SSL
- 2.10.4 Switch board routing table
- 2.10.5 Subnet walking

2.11 Synchronisation

- 2.11.1 Notification d'événements
- 2.11.2 Reader/Writer locks
- 2.11.3 Comptage par référence
- 2.11.4 Thread-safe operations

2.12 Ui

- 2.12.1 Text

3 Framework Core

3.1 DataStore

3.2 Notifications d'événements

- 3.2.1 Événements d'Exploit
- 3.2.2 Événements du Framework général
- 3.2.3 Événements de reconnaissance
- 3.2.4 Événements de Session

3.3 Gestionnaires du Framework

- 3.3.1 Gestion de Module
- 3.3.2 Gestion de Plugin
- 3.3.3 Gestion de Reconnaissance
- 3.3.4 Gestion de Session
- 3.3.5 Gestion de Job

3.4 Classes d'Utilitaires

- 3.4.1 Driver d'Exploit
- 3.4.2 Payload encodé

4 Framework Base

4.1 Configuration

4.2 Logging

4.3 Sérialisation

4.4 Sessions

- 4.4.1 CommandShell
- 4.4.2 Meterpreter

4.5 Framework Simplifié

- 4.5.1 Exploit
- 4.5.2 NOP
- 4.5.3 Payload
- 4.5.4 Recon

5 Framework Ui

6 Framework Modules

6.1 Encoder

- 6.1.1 encode
- 6.1.2 do_encode
- 6.1.3 Méthodes d'aide

6.2 Exploit

- 6.2.1 Stances
- 6.2.2 Types
- 6.2.3 Interface
- 6.2.4 Accesseurs et Attributs
- 6.2.5 Mixins

6.3 Nop

- 6.3.1 generate_sled
- 6.3.2 nop_repeat_threshold

- 6.4 Payload
 - 6.4.1 Interface
 - 6.4.2 Types
 - 6.4.3 Handlers
- 6.5 Recon

7 Framework Plugins

8 Framework Sessions

- 8.1 CommandShell
- 8.2 Meterpreter

9 Méthodologies

A Exemples

- A.1 Framework
 - A.1.1 Obtenir des informations sur un module
 - A.1.2 Encodé le contenu d'un fichier
 - A.1.3 Enumérer les modules
 - A.1.4 Lancer un exploit en utilisant la framework base
 - A.1.5 Lancer un exploit en utilisant le framework core
- A.2 Module de Framework
 - A.2.1 Encoder
 - A.2.2 Exploit
 - A.2.3 Nop
 - A.2.4 Payload
 - A.2.5 Recon
- A.3 Plugin de Framework
 - A.3.1 Plugin d'interface utilisateur console

Chapitre 1

Introduction

Le framework Metasploit est un framework open-source conçu pour fournir ou chercheurs en sécurité et aux pen-testers un modèle uniforme pour le développement rapide d'exploits, payloads, encodeurs, générateurs de NOP, et outils de reconnaissance.

Le framework fournit aux écrivains d'exploits ;) la possibilité de réutiliser une grande partie des bouts de code qui auraient autrement dus être recopiés ou réimplémentés pour chaque exploit.

Pour aider la cause, l'équipe Metasploit est fière de présenter la prochaine évolution majeure du framework d'exploitation: la version 3.0.

La version 3.0 du framework est une refonte de la branche 2.x écrite entièrement en Ruby.

L'objectif principal de la branche 3.0 est de rendre le framework facile à utiliser et l'étendre du côté programmation.

Cet objectif n'englobe pas seulement le développement de modules de framework, comme les exploits, mais également le développement d'outils tiers, et de plugins qui peuvent être utilisés pour accroître les fonctionnalités de la suite complète.

En développant un framework facile à utiliser au niveau programmation, il en résulte que les exploits et autres extensions devraient être plus faciles à comprendre et à implémenter que ceux des versions précédentes.

Ce document fournit des explications sur les objectifs de conception, les méthodologies, et des détails d'implémentation de la version 3.0 du framework.

1.1 Pourquoi Ruby?

Pendant le développement du framework, la question récurrente posée à l'équipe Metasploit fut pourquoi Ruby fut choisi comme langage de développement.

Le langage de programmation Ruby fut choisi parmi d'autres options, comme python, perl, et C++ pour plusieurs raisons. La première (et principale) raison est que Ruby est un langage avec lequel l'équipe MetaSploit à du plaisir à programmer.

Après avoir passé du temps à analyser d'autres langages et factoriser les expériences préalables, Ruby s'avéra offrir à la fois une approche simple et puissante d'un langage interprété.

Le degré d'introspection et les aspects orientés objet fournis par Ruby coïncident parfaitement avec les besoins du framework. La nécessité de construction automatique de classes pour la réutilisation de code pour le framework fut un élément clé dans la décision, et ce fut une chose à laquelle Perl ne correspondait pas parfaitement.

En plus de cela, la syntaxe est incroyablement simple et fournit le même niveau de fonctionnalités que les autres langages, comme Perl.

La seconde raison pour laquelle Ruby fut retenu, est son support indépendant de la plateforme pour le threading.

Alors que bon nombre de limitations furent rencontrées pendant le développement du framework sous ce modèle, l'équipe Metasploit a observé une amélioration notable de la performance et utilisabilité par rapport à la branche 2.x.

Les prochaines versions de Ruby (les séries 1.9) vont ramener les API de threading actuelles à des threads natifs pour le système d'exploitation sur lequel est compilé l'interpréteur ce qui va solutionner bon nombre de problèmes existants avec l'implémentation courante (comme permettre l'utilisation des opérations de blocage).

En même temps, le modèle de threading existant s'est révélé être de loin supérieur vis à vis du modèle de forking conventionnel, spécialement sur les plateformes sans implémentation native de fork, comme Windows.

Une autre raison de choisir Ruby est l'existence d'un interpréteur natif pour la plateforme Windows.

Alors que Perl a une version de cygwin et une version ActiveState, les deux sont infestés par des problèmes d'utilisation.

Le fait que l'interpréteur Ruby peut être compilé et exécuté nativement sur Windows améliore considérablement les performances.

De plus, l'interpréteur est également très petit et peut facilement être modifié dans l'événement où il y a un bogue.

Le langage de programmation Python était également un candidat. Il ne fut pas préféré à Ruby pour d'autres raisons. La première raison est liée aux contraintes syntaxiques imposées par Python, comme l'indentation.

...

1.2 Conception et Architecture

Le framework a été conçu pour être le plus modulaire possible, pour encourager la réutilisation de code à travers différents projets.

La pièce fondamentale de l'architecture est la librairie Rex qui est le diminutif de Ruby Extension Library¹.

Certains des composants fournis par Rex incluent un sous-système d'emballage de socket, des implémentations de clients et serveurs de protocoles, un sous-système de logging, des classes utilitaires d'exploitation, et un grand nombre d'autres classes utiles.

Rex elle-même est conçue pour ne pas avoir de dépendances en dehors de ce qui est installé par défaut avec Ruby.

Dans le cas où une classe Rex dépende de quelque chose qui n'est pas incluse dans l'installation par défaut de Ruby, l'échec de trouver une telle dépendance ne doit pas amener à une impossibilité d'utiliser Rex.

Le framework lui-même se décompose en plusieurs parties.

Le niveau le plus bas est le "framework core".

Le framework core est responsable de l'implémentation de toutes les interfaces requises qui permettent d'interagir avec les modules, sessions et plugins d'exploits.

Cette librairie principale est étendue par la librairie de base du framework qui est conçue pour fournir des routines d'emballage plus simples pour s'occuper du framework core et également fournit des classes d'utilitaires pour s'occuper des différents aspects du framework, comme l'état de sérialisation d'un module en différents formats de sortie.

Finalement, la librairie de base est étendue par le framework UI qui implémente le support pour les différents types d'interfaces utilisateur pour le framework lui-même, comme la console de commandes et l'interface web.

Séparés du framework lui-même, se trouvent les modules et plugins qu'il va supporter.

Un module framework est défini comme étant un des exploits, payload, encodeur, générateur de NOP ou outil RECON.

Ces modules ont une structure et interface prédéfinie pour être chargés dans le framework.

Un plugin du framework est très lâchement défini comme quelque chose qui étend les fonctionnalités du framework ou améliore une fonctionnalité existante pour la faire agir d'une manière différente.

Les plugins peuvent ajouter de nouvelles commandes aux interfaces utilisateur, enregistrer tout le trafic réseau ou réaliser n'importe quelle autre action utile.

La Figure 1.1 illustre les dépendances inter-paquetage du framework.

Les sections suivantes vont s'attarder sur chaque paquetage décrits ci-dessus et les différents sous-systèmes importants qui se trouvent dans chaque paquetage.

La documentation complète sur les classes et APIs mentionnées dans ce document peut être trouvée sur le site Metasploit.com

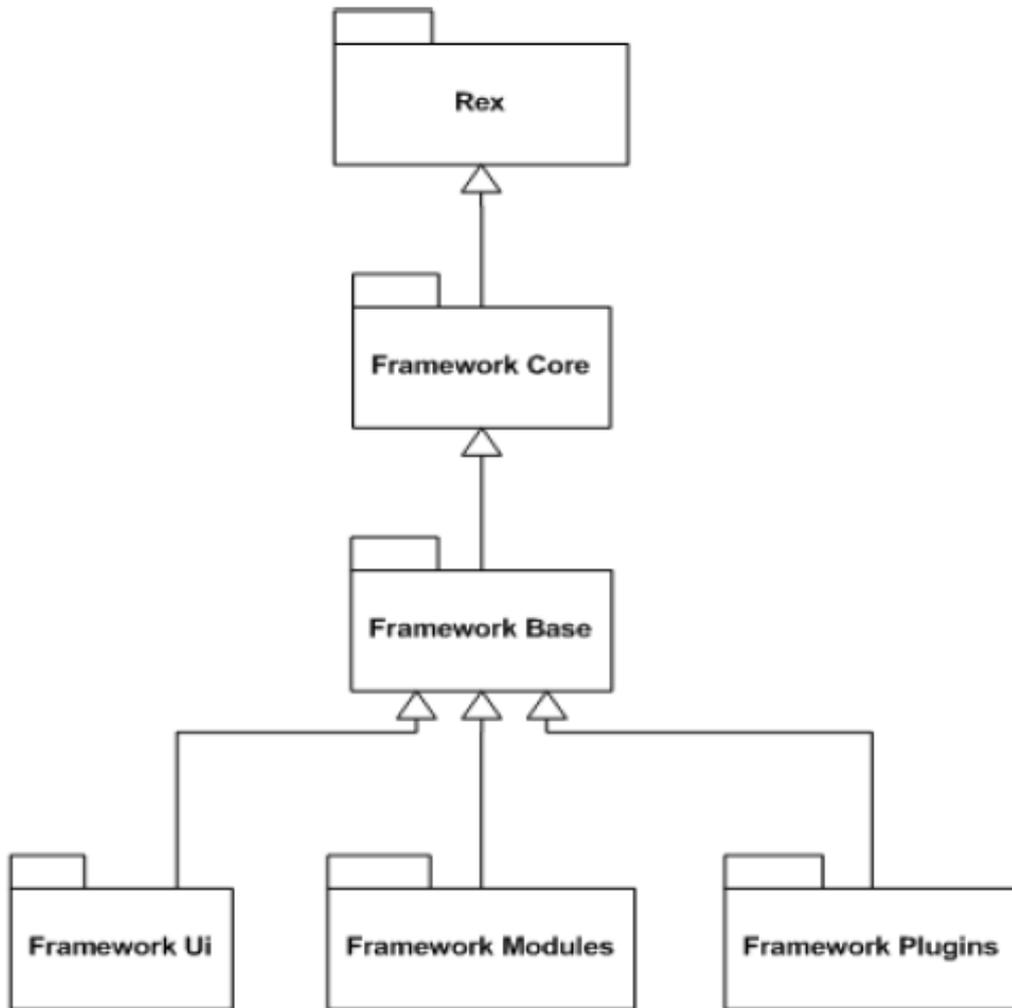


Figure 1.1: Dépendances des paquets du Framework 3.0

Chapitre 2

Rex

La librairie Rex est une collection de classes et modules qui peuvent être utiles à plus d'un projet. Les classes les plus utiles fournies par la librairie sont documentées dans les sections suivantes. Pour utiliser Rex; un script Ruby doit requérir Rex.

2.1 Assembleur

En écrivant des exploits, il est souvent nécessaire de générer des instructions assembleur à la volée avec des opérandes variables, comme des valeurs immédiates, es registres, etc. Pour supporter ce besoin, la librairie Rex fournit des classes de type « Rex::Arch » qui implémentent des routines de génération d'opcodes dépendants de l'architecture, ainsi que d'autres méthodes spécifiques à l'architecture, comme le packaging d'entier.

2.1.1 Packaging d'entier

Packer un entier dépend de l'ordre des octets de l'architecture cible (big endian ou little endian). La méthode « Rex::Arch.pack adr » supporte le packaging d'entier en utilisant le type d'architecture spécifié (ARCH XXX).

2.1.2 Ajustement de pointeur de pile (Stack pointer adjustment)

Certains exploits nécessitent un ajustement du pointeur de pile préalable à l'exécution d'un payload qui modifie la pile afin de prévenir de la corruption du payload lui-même. Pour supporter ceci; la méthode « Rex::Arch.adjust stack pointer » fournit une manière de générer les opcodes qui correspondent à l'ajustement du pointeur de pile d'une certaine architecture par le valeur d'ajustement spécifiée. La valeur d'ajustement peut être positive ou négative.

2.1.3 Génération d'opcodes spécifiques à l'architecture

Chaque architecture qui supporte la génération dynamique d'opcodes possède une classe de type « Rex::Arch », comme « Rex::Arch::X86 ». La classe X86 supporte la génération d'instructions jmp, call, push, mov, add, et sub.

2.2 Encodage

Encoder des buffers en utilisant des algorithmes comme XOR peut parfois être utile du contexte d'un exploit. Pour cette raison, la librairie Rex fournit un ensemble de classes basiques qui implémentent différents types d'encodeurs XOR, comme des encodeurs XOR de longueur de clé variable et encodeurs de retour additionnel. Ces classes sont utilisées par le framework pour implémenter différents types d'encodeurs basiques qui peuvent être utilisés par les modules d'encodage. Les classes pour l'encodage des buffers sont nommées de type « Rex::Encoding ».

2.3 Exploitation

Bien souvent, les vulnérabilités vont partager un vecteur d'attaque commun ou vont nécessiter un ordre spécifique des opérations pour achever le but final d'exécution du code.

Pour assister dans cette voie, la librairie Rex possède un ensemble de classes qui implémentent certaines des nécessités communes dont un exploit peut avoir besoin.

2.3.1 Egghunter

Dans certains cas; l'exploitation d'une vulnérabilité peut être limitée par la quantité d'espace de payload qui existe dans la zone du débordement.

Cela peut parfois empêcher les méthodes normales d'exploitation du fait de l'impossibilité de placer un payload standard dans l'espace disponible.

Pour résoudre ce problème, un écrivain d'exploit :) peut utiliser un payload egghunt qui recherche l'espace d'adressage du processus cible pour un egg qui est préfixé pour un payload plus grand.

Cela nécessite qu'un attaquant ait la possibilité de coller le payload le plus grand ailleurs en mémoire avant l'exploitation.

Dans le cas où un egghunter est nécessaire, la classe « `Rex::Exploitation::Egghunter` » peut être utilisée.

2.3.2 Génération d'enregistrement SEH

Un vecteur d'attaque qui est particulièrement courant sur la plateforme Windows est la sur-écriture de SEH.

Lorsque ceci arrive, un enregistrement de SEH est sur-écrit sur la pile avec des données contrôlées par l'utilisateur.

Pour LEVERAGE ceci, l'adresse du gestionnaire de l'enregistrement est pointé vers une adresse qui va directement ou indirectement amener au contrôle du flot d'exécution.

Pour faire fonctionner ceci, la plupart des attaquants vont faire pointer l'adresse du gestionnaire vers un emplacement d'une instruction `pop/pop/ret` définie quelque part dans l'espace d'adressage.

Cette action retourne 4 octets avant l'emplacement de l'adresse du gestionnaire sur la pile.

Dans la plupart des cas, les attaquants vont définir 2 des 4 octets comme équivalents à une instruction `short jump` qui va sauter par dessus l'adresse du gestionnaire et atterrir dans le payload contrôlé par l'attaquant.

Alors que cette approche courante fonctionne bien, il existe beaucoup de possibilité d'amélioration.

La classe « `Rex::Exploitation::Seh` » permet la génération de l'enregistrement SEH normal (statique) via la méthode "generate static seh".

Malgré tout, elle permet également la génération d'un enregistrement dynamique qui possède une longueur de `short jump` aléatoire et un padding aléatoire entre la fin de l'enregistrement et le payload. Cela peut être utilisé pour rendre l'exploit plus difficile à détecter par un IDS.

La génération d'un enregistrement dynamique est fournit par "generate dynamic seh record".

Les deux méthodes sont englobées par la méthode "generate seh record" qui décide laquelle utiliser en fonction du niveau d'évasion.

2.4 Jobs

Dans certains cas, il est utile de découper certaines tâches en différents travaux (jobs).

Les jobs sont simplement définis comme des éléments de travail finis qui ont tâche spécifique.

En utilisant cette définition, la librairie Rex fournit une classe nommée « `Rex::JobContainer` » qui propose une interface pour coordonner différentes tâches finies d'une manière centralisée.

De nouveaux jobs peuvent être ajoutés au conteneur de jobs en appelant la méthode "add job".

Une fois ajouté, un job peut être démarré en effectuant un appel à la méthode "start job".

A n'importe quel moment, un job peut être interrompu en appelant "stop job" qui va également supprimer le job via la méthode "remove job".

Pour plus d'informations sur l'utilisation de ces routines API, veuillez vous référer à la documentation auto-générée sur le site Metasploit.

2.5 Logging

La librairie Rex fournit un support pour une journalisation basique des chaînes vers des LOG SINKS arbitraires, comme un gros fichier ou une base de données.

L'interface de logging est exposée aux programmeurs, à travers un ensemble de méthodes définies globalement: `dlog`, `ilog`, `wlog`, `elog`, et `rlog`.

Ces méthodes représentent debug logging, information logging, warning logging, error logging, et raw logging

A chaque méthode peut être passé un message de log, une source de log (le nom du composant ou paquetage d'où provient le message), et un niveau de log qui est un nombre entre 0 et 3.

Les sources de log peuvent être enregistrées à la volée via "register log source" et le niveau défini via "set log level".

Les niveaux de log permettent de masquer les messages de log verbeux quand ils ne sont pas nécessaires. L'utilisation des 3 niveaux de log est définie ci-dessous:

2.5.1 LEV 0 -Défaut

Ce niveau de log est le niveau par défaut si aucun n'est spécifié. Il doit être employé quand un message de log doit toujours être affiché quand le logging est activé.

Très peu de messages devraient apparaître à ce niveau de log en dehors des informations nécessaires et erreurs/warnings.

Le logging de débogage en mode 0 n'est pas conseillé.

2.5.2 LEV 1 -Extra

Ce niveau de log devrait être utilisé quand plus d'informations sont nécessaires pour comprendre la cause d'une erreur ou d'un warning ou pour obtenir des informations de débogage qui pourraient donner des explications à pourquoi quelque chose se passe.

2.5.3 LEV 2 -Verbeux

Ce niveau de log devrait être utilisé quand des informations verbeuses sont nécessaires pour analyser le fonctionnement du framework.

Ce devrait être le niveau de débogage par défaut pour toutes les informations non présentes en niveau 0 ou 1.

Il est recommandé d'utiliser ce niveau de log par défaut si vous n'êtes pas sûr.

2.5.4 LEV 3 -Insanity

Ce niveau de log présente des informations très verbeuses sur le fonctionnement du framework, comme par exemple des informations détaillées sur l'état des variables durant certaines phases comme les itérations de boucle, les appels de fonctions, ...

Il devrait rarement être utilisé mais quand il l'est; les informations fournies devraient rendre facile l'analyse de n'importe quel problème.

2.6 Base de données d'Opcodes

La librairie Rex fournit une classe qui rend possible d'interagir avec la base de données d'opcodes Metasploit d'une manière programmatique.

La classe qui fournit cette fonctionnalité se trouve dans "Rex::Exploitation::OpcodeDb::Client". Pour plus d'informations sur comment interagir avec la BDD d'opcodes, veuillez vous référer à la documentation auto-générée sur le site Metasploit.

2.7 Post-exploitation

La librairie Rex fournit des implémentations côté client pour des outils avancés post-exploitation comme DispatchNinja et Meterpreter.

Ces deux interfaces post-exploitation sont conçues pour être utilisables en dehors du contexte d'un exploit. Un ensemble de classes de type Rex::Post sont disponibles dans le but d'agir comme une interface généralisée aux systèmes distants via les clients post-exploitation, s'ils sont supportés.

Ces classes permettent aux programmeurs d'écrire des outils automatisés qui peuvent opérer sur les machines distantes d'une manière indépendante de la plateforme.

Tant qu'il est vrai que ces plateformes peuvent avoir des ensembles de fonctionnalités différents pour certaines actions, la plupart des actions courantes auront un équivalent.

2.8 Protocoles

La librairie Rex inclut le support des protocoles les plus courants comme HTTP et SMB pour aider le développement d'exploits spécifiques à un protocole et permettre la facilité d'utilisation dans d'autres projets.

Chaque implémentation de protocole existe sous "Rex::Proto".

2.8.1 DCERC

La librairie Rex supporte une implémentation assez robuste d'une portion de l'ensemble de fonctionnalités DCERPC et inclut le support pour effectuer des actions évasives comme la fragmentation de paquets et le bind multi-contexte.

Les classes qui supportent l'interface client DCERPC peuvent être trouvées sous "Rex::Proto::DCERPC".

2.8.2 HTTP

La librairie Rex supporte un client et serveur HTTP minimum.

Alors que des implémentations de classe de protocole sont fournies à la fois en webrick et dans d'autres parties de l'ensemble de bibliothèques standards de ruby par défaut, Il fut considéré que les implémentations actuelles ne convenaient pas parfaitement pour un but d'utilisation générale du fait de l'existence d'un blocage de requête et d'autres choses. La bibliothèque HTTP rex fournit également des classes pour filtrer les requêtes et les réponses. Les classes du protocole HTTP se trouvent dans l'espace de nommage "Rex::Proto::Http".

2.8.3 SMB

Les classes de type "Rex::Proto::SMB" offrent un support robuste du protocole SMB. Ces classes supportent la connexion à des serveurs SMB et l'établissement d'authentifications, tout comme les autres actions SMB.

2.9 Services

Une des limitations identifiées dans la branche 2.x du framework était qu'il était impossible de partager les listeners sur une machine locale en tentant deux exploits différents qui nécessitaient tous les deux d'écouter sur le même port. Pour résoudre le problème, la version 3.0 du framework fournit le concept de services qui sont des listeners enregistrés qui sont initialisés une seule fois, puis partagés par les requêtes suivantes pour allouer le même service. Cela rend possible de faire des choses comme avoir deux exploits attendant pour une requête HTTP sur le port 80 sans avoir de conflit. C'est particulièrement utile pour ne pas avoir à se soucier des restrictions de pare-feu sur les ports de sortie et tester plusieurs exploits côté client contre un hôte avec toutes les instances d'exploits différents écoutant sur le même port pour des requêtes. Le sous-système permet également la redirection de port TCP via la classe "Rex::Services::LocalRelay".

2.10 Sockets

Une des fonctionnalités les plus importantes de la bibliothèque Rex est l'ensemble de classes qui englobent les sockets. Le sous-système de socket fournit une interface pour la création de sockets pour un protocole donné en utilisant ce qui est appelé une classe "Comm factory". Le but de la classe "Comm factory" est de faire utiliser le transport et classes fondamentales pour établir la connexion pour un socket opaque donné. Cela rend possible pour les connexions socket d'être établies en utilisant les facilités de socket locales, comme en utilisant une sorte de système de proxification socket tunnelée, comme c'est le cas pour le pivotement de connexion Meterpreter. Les sockets sont créés en utilisant la classe "socket Parameter" qui est initialisée soit directement, soit via l'utilisation d'un hash. L'initialisation du hash de la classe de paramètres ressemble à celui de l'initialisation de socket de Perl. Les attributs du hash supportés par la classe Parameter sont documentés dans le constructeur de la classe. Il existe différentes façons de créer des sockets. La première est de simplement appelé Rex::Socket.create avec un hash qui sera utilisé pour créer un socket du type approprié en utilisant la classe spécifiée ou la "Comm factory" par défaut. Une seconde approche qui peut être utilisée est d'appeler la méthode "Rex::Socket::create param" qui prend une instance de Parameter initialisée comme argument.

Les autres méthodes impliquent l'utilisation de méthodes factory spécifiques au protocole, comme "create tcp", "create tcp server", et "create udp".

Toutes les trois prennent un hash comme paramètre qui est traduit en une instance de Parameter et passé pour la création actuelle.

Tous les sockets ont 5 attributs majeurs qui sont partagés en commun, si bien que certains peuvent ne pas être initialisés à chaque fois.

Les premiers attributs fournissent des informations sur l'hôte distant et le port et sont respectivement "peerhost" et "peerport".

Les autres attributs fournissent des informations sur l'hôte local et le port local et sont respectivement "localhost" et "localport".

Enfin, chaque socket a un hash d'informations contextuelles qui fut utilisé lors de sa création; c'est l'attribut "context".

Bien que la plupart des exploits aient un hash vide, certains exploits peuvent avoir un hash qui contient des informations d'état qui peuvent être utilisées pour traquer l'origine du socket.

Le framework utilise cette fonctionnalité pour associer les sockets avec les instances du framework, exploit et payload.

2.10.1 Classes Comm

L'interface Comm utilisée dans la librairie possède une simple méthode "create" qui prend en argument une instance Parameter.

Le but de cette approche factory est de fournir une méthode indépendante de localisation et transport pour créer des instances d'objets socket compatible en utilisant une méthode factory généralisée.

Pour les connexions établies directement depuis le poste local; la classe Rex::Socket::Comm::Local est utilisée.

Pour les connexions établies via une autre machine; une classe Comm factory intermédiaire spécifique est utilisée, comme la classe Comm Meterpreter.

L'interface Comm supporte également les gestionnaires de notification d'événements enregistrés pour quand certaines choses surviennent, comme avant et après la création d'un socket.

Cela peut être utilisé par des projets externes pour augmenter l'ensemble de fonctionnalités d'un socket ou pour changer sa comportement par défaut.

2.10.2 Sockets TCP

Les sockets TCP dans la librairie Rex sont implémentés comme un mixin, Rex::Socket::Tcp, qui étend la classe Socket de base intégrée dans Ruby quand la Comm factory locale est utilisée.

Ce mixin inclut également les mixins Rex::IO::Stream et Rex::Socket

Pour les serveurs TCP; la classe Rex::Socket::TcpServer devrait être utilisée.

2.10.3 Sockets SSL

Les sockets SSL sont implémentés au dessus du mixin normal Rex TCP socket et utilisent le support Ruby d'OpenSSL.

Le module utilisé pour les sockets TCP SSL est Rex::Socket::SslTcp

2.10.4 Switch board routing table

Une des améliorations dans la version 3.0 du framework est le concept de table locale de routage qui contrôle qu'elle Comm factory est utilisée pour une route particulière.

Ceci est utile pour les scénarios où une machine est compromise et écartée du réseau interne qui ne peut pas être directement atteint.

En ajustant la table de routage du panneau de commutateur (switch board) pour pointer le sous-réseau (subnet) local à travers un Comm Meterpreter tournant sur l'hôte qui est écarté du réseau, il est possible de forcer la librairie socket d'utiliser automatiquement la Comm factory Meterpreter quand quelque chose essaie de communiquer avec les hôtes du sous-réseau local.

Ce support est implémenté via la classe `Rex::Socket::SwitchBoard`

2.10.5 Subnet walking

La classe `Rex::Socket::SubnetWalker` fournit une manière d'énumérer toutes les adresses IP dans un sous-réseau comme une adresse et un masque de sous-réseau.

2.11 Synchronisation

Du fait de l'utilisation du multi-threading, la librairie Rex fournit des classes extra qui n'existent pas par défaut dans la librairie standard Ruby.

Ces classes fournissent des primitives de synchronisation supplémentaires.

2.11.1 Notification d'événements

Alors que Ruby possède le concept de `ConditionVariable`, il manque le concept complet d'événements de notification.

Les événements de notification sont largement utilisés sur les plateformes comme Windows.

Ces événements peuvent être attendus ou signalés, soit temporairement soit de manière permanente.

Veuillez vous référer à la documentation Microsoft pour plus d'informations.

Ce support est fourni par la classe "`Rex::Sync::Event`".

2.11.2 Blocages Lecture/Ecriture (Reader/Writer locks)

Une primitive de threading courante est le blocage de lecture/écriture (reader/writer lock).

Les blocages de lecture/écriture sont utilisés pour permettre d'avoir plusieurs threads lisant une ressource en même temps tout en autorisant seulement l'accès exclusif à un seul thread quand des opérations d'écriture sont nécessaires.

Cette primitive est particulièrement utile pour les ressources qui ne sont pas mises à jour très souvent car cela peut drastiquement réduire les controverses de blocage.

Bien que cela puisse être exagéré d'avoir une telle primitive de synchronisation dans la librairie, c'est quand même cool.

L'implémentation du blocage de lecture/écriture est fournie par la classe `Rex::ReadWriteLock`

Pour bloquer une ressource en lecture; la méthode "lock read" est utilisée.

Pour bloquer une ressource en écriture; la méthode "lock write" est utilisée.

2.11.3 Reference counting

Dans certains cas il est nécessaire de compter par référence une instance d'une manière synchronisée pour qu'elle ne soit pas vidée ou détruite avant que la dernière référence ne soit partie. Pour ce faire, la classe "Rex::Ref" peut être utilisée avec la méthode "refinit" pour initialiser les références à 1, et les méthodes "ref" et "deref" qui font ce que leur nom indique. Quand le nombre de référence passe à zéro; la méthode "cleanup" est appelée sur l'instance de l'objet pour lui donner une chance de restaurer les choses à la normale d'une manière similaire à un destructeur.

2.11.4 Thread-safe operations

Certaines des fonctions intégrées dans Ruby ne sont pas sauvées vis-à-vis des threads, dans le sens où elles peuvent empêcher les autres threads Ruby d'être programmés dans certaines conditions.

Pour résoudre ce problème, les fonctions qui ont des problèmes ont été englobées avec des implémentations qui s'assurent que tous les threads Ruby ne seront pas bloqués.

Les méthodes spécifiques qui nécessitaient une modification sont "select" et "sleep".

2.12 Ui

La librairie Rex fournit un ensemble de classes d'aide qui peuvent être utiles à certains médiums d'interface utilisateur.

Ces classes ne sont pas incluses par défaut en requérant Rex, un programmeur doit s'assurer de requérir rex/ui pour obtenir les classes décrites dans cette section.

Actuellement, le seul médium d'interface utilisateur qui possède des classes concrètes définies est le text, qui est synonyme du médium d'interface utilisateur console.

2.12.1 Text

Le médium d'interface utilisateur text fournit des classes qui permettent à un programmeur d'interagir avec des handles d'entrée/sortie d'un terminal.

Il fournit également des classes pour simuler un pseudo shell de commandes de la manière la plus robuste possible.

Input

La classe Rex::Ui::Text::Input agit comme une classe de base pour des médiums d'entrée utilisateur plus spécifiques.

L'interface de la classe de base fournit un ensemble basique de méthodes pour lire l'entrée utilisateur (gets), vérifier si l'entrée standard a été fermée (eof?), et d'autres.

Il y a actuellement deux classes qui étendent la classe de base.

La première est "Rex::Ui::Text::Input::Stdio".

Cette classe utilise simplement la variable globale \$stdin de Ruby.

C'est la manière la plus basique d'acquérir l'entrée utilisateur.

La seconde classe est "Rex::Ui::Text::Input::Readline" qui interagit avec l'utilisateur via la librairie readline.

Si readline n'est pas installée, la classe ne fonctionnera pas.

Ces deux classes peuvent être utilisées par les classes shell décrites plus tard dans cette section.

Output

La classe `Rex::Ui::Text::Output` implémente l'interface abstraite plus généralisée `Rex::Ui::Output`

Le but de cette classe est de définir un ensemble de fonctions qui peuvent être utilisées pour fournir des sorties utilisateur.

Il y a actuellement deux classes qui implémentent l'interface de sortie textuelle.

La première est `Rex::Ui::Text::Output::Buffer`

Ce meidum de sortie sérialise le texte affiché dans un buffer qui peut être récupéré l'attribut de l'instance `buf`.

La seconde classe est `Rex::Ui::Text::Output::Stdio`

Cette classe est le complément de la classe d'entrée `stdio` et utilise simplement la variable globale `$stdout` pour fournir une sortie au terminal utilisateur.

Shell

La classe `Rex::Ui::Text::Shell` fournit une classe de base pour un pseudo-shell simple qui peut être utilisé pour implémenter un shell interactif avec l'utilisateur.

La classe est instanciée en passant une chaîne prompt et un caractère prompt (qui par défaut est ">") au constructeur.

Par défaut, les instances de classes entrée/sortie du shell sont initialisées respectivement à des instances de `Rex::Ui::Text::Input::Stdio` et `Rex::Ui::Text::Output::Stdio`

Pour changer les instances de classes entrée/sortie; un appel peut être effectué à la méthode "init ui".

Pour utiliser le shell; un appel doit être fait à la méthode "run" de l'instance shell.

Cette méthode accepte soit un contexte de bloc, à qui seront passées des chaînes d'entrée basées sur une ligne, ou agira dans un mode callback quand un appel à la méthode "run single" est effectué sur l'instance shell.

Si la seconde méthode est utilisée; la classe sera surdéfinie avec une implémentation arrangée de la méthode "run single".

Dispatcher Shell

La classe `Rex::Ui::Text::DispatcherShell` étend la classe "`Rex::Ui::Text::Shell`" en introduisant le concept d'une interface de dispatcher de commandes généralisée.

Le dispatcher shell fonctionne en surdéfinissant la méthode "run".

Contrairement à la classe shell de base, le dispatcher shell fournit un mécanisme par lequel les dispatchers de commandes peuvent être enregistrés pour traiter le texte entré d'une manière normalisée.

Tous les dispatchers de commandes doivent inclure le mixin

`"Rex::Ui::Text::DispatcherShell::CommandDispatcher"` qui fournit un ensemble de méthodes d'aide, s'occupant principalement sans englober le texte de sortie.

L'enregistrement d'un dispatcher de commandes est accompli en appelant soit "enstack dispatcher" ou "append dispatcher".

La méthode "enstack dispatcher" insert l'instance de dispatcher de commandes fournie, lui donnant ainsi l'opportunité de traiter en premier les commandes.

La méthode "append dispatcher" insert l'instance de dispatcher de commandes fournie à la fin de l liste.

Pour enlever des dispatchers de commandes, les méthodes complémentaires "destack dispatcher" et "remove dispatcher" peuvent être utilisées.

Quand une ligne d'entrée arrive, la classe shell de base appelle la méthode "run" surdéfinie qui découpe la chaîne en entrée en un tableau d'arguments délimités par des caractères shell normaux.

Le premier argument dans la chaîne est ensuite évalué en relation avec tous les autres dispatchers de commandes enregistrés en vérifiant pour voir si l'un d'entre eux implémente une méthode appelée "cmd <arg 0>". Si c'est le cas, le shell dispatcher appelle la méthode et lui passe le tableau d'argument. Pour rendre possible de générer automatiquement un menu d'aide pour tous les dispatchers de commandes enregistrés; chaque dispatcher de commandes doit implémenter une méthode nommée "commands" qui doit retourner un hash qui associe les commandes avec une description de l'opération qu'elles effectuent.

Table

La classe `Rex::Ui::Text::Table` peut être utilisée pour formater des données sous la forme d'une table avec un entête, des colonnes et des lignes.

Pour plus d'informations sur l'utilisation de la classe table; merci de vous reporter à la documentation générée automatiquement sur le site Metasploit.

Souscripteurs (Subscribers)

La librairie Rex supporte la création de classes conçues pour s'enregistrer aux interfaces d'entrée/sortie via l'interface "Rex::Ui::Subscriber".

Ce mixin fournit une méthode appelée "init ui" à qui on peut passer une instance de classe d'entrée et de sortie.

Ces instances doivent implémenter les interfaces "Rex::Ui::Text::Input" et "Rex::Ui::Output", respectivement.

Une fois que init ui a été appelée, les appels suivants aux méthodes comme "print line" seront passés à l'instance de la classe de sortie initialisée.

Si aucune instance de classe n'a été définie, l'appel sera ignoré.

Cela permet de fournir une manière par laquelle les classes peuvent interagir avec l'interface utilisateur seulement quand désiré.

Pour désactiver l'interaction avec l'interface utilisateur; il faut appeler "reset ui" qui va désactiver les classes d'entrée et de sortie futures pour la classe.

Chapitre 3

Framework Core

Le framework core implémente l'ensemble de classes qui fournissent une interface aux modules et plugins du framework.

La partie core du framework est conçue en utilisant une approche basée sur des instances. Cela signifie que l'état complet du framework peut être contenu dans une seule instance de classe, permettant aux programmeurs d'avoir plusieurs instances concurrentes et séparées du framework en même temps plutôt que d'avoir à tout partager dans une seule instance singleton.

La version majeure courante du framework core peut être accédée via "Msf::Framework::Major" et la version mineure via "Msf::Framework::Minor".

Une version combinée de ces deux versions peut être accédée via "Msf::Framework::Version" ou "framework.version" au niveau instance.

La révision courante de l'interface framework core peut être accédée via "Msf::Framework::Revision".

Le framework core est accédé via une instance de la classe "Msf::Framework".

La création d'une instance de framework est illustrée en figure 3.1.

```
framework = Msf::Framework.new
```

Figure 3.1: Création d'une instance de framework

L'instance de framework elle-même n'est rien d'autre qu'une manière de connecter les différents sous-systèmes critiques du framework core, comme la gestion des modules, la gestion des sessions, des événements, etc.

La manière d'utiliser ces sous-systèmes sera décrite dans les sections suivantes.

Pour utiliser la librairie framework core, un script Ruby requière msf/core.

3.1 DataStore

Chaque instance du framework possède une instance de la classe Msf::DataStore qui peut être accédée via framework.datastore

Le but de la datastore dans la version 3.0 du framework est d'agir comme un remplaçant au concept d'environnement dans la branche 2.X.

La datastore est simplement un hashé de valeurs qui peut être utilisé soit par les modules soit par le framework lui-même pour référencer les valeurs du programmeur ou contrôlées par l'utilisateur.

L'interaction avec la datastore est illustré en figure 3.2.

```
framework.datastore['foo'] = 'bar'  
if (framework.datastore['foo'] == 'bar')  
  puts "'foo' is 'bar'"  
end
```

Figure 3.2: Création d'une instance de framework

Les modules vont hérités des valeurs depuis la datastore globale du framework si elles ne sont pas trouvées dans la datastore du module.

Cet aspect sera discuté plus en détails dans le chapitre 6.

3.2 Notifications d'événements

Un des objectifs principal dans la version 3.0 du framework fut de fournir aux développeurs un système de notification d'événements utile qui permette d'effectuer des actions arbitraires quand certains événements du framework surviennent.

Pour supporter ceci; chaque instance du framework peut avoir des gestionnaires d'événements enregistrés via l'attribut `framework.events` qui est une instance de la classe `Msf::EventDispatcher`

La classe `EventDispatcher` supporte l'enregistrement de gestionnaires d'événements pour quelques catégories basiques différentes.

Ces catégories vont être abordées individuellement.

Un des bons aspects du framework VENT-DRIVEN est que les modules peuvent automatiquement indiquer leur intérêt à être enregistré pour les notifications d'événements simplement en implémentant les MIXINS souscripteur d'événements suivants.

Quand un module est chargé dans le framework; il détectera automatiquement qu'il inclut une ou plusieurs interfaces de souscripteur et enregistrera automatiquement le module avec les notificateurs d'événements appropriés.

Cela permet aux modules d'effectuer certaines actions quand certains événements surviennent.

3.2.1 Événements d'Exploit

Les souscripteurs d'événement peuvent être enregistrés pour être notifiés quand des événements liés à l'exploitation sont reçus.

Pour enregistrer un souscripteur d'événement d'exploit; un appel au souscripteur d'exploit "`framework.events.register`" doit être réalisé.

Cette méthode doit se voir passé une instance d'un objet qui inclue le mixin "`Msf::ExploitEvent`".

Le type d'événement notifié à ce souscripteur sera quand un exploit réussi.

Si un exploit réussi; la méthode "`on exploit success`" du souscripteur sera appelée avec l'instance de l'exploit qui a réussi et l'instance de la session qu'il a créée.

Pour supprimer un souscripteur d'événement; il faut appeler `framework.events.remove` en passant l'instance de l'objet qui fut utilisée pour ajouter le souscripteur en premier lieu.

3.2.2 Événements généraux du framework

Pour recevoir des notifications d'événements sur les événements internes au framework, un souscripteur d'événement général peut être enregistré via la méthode de souscripteur général "`framework.events.register`".

Cette méthode prend comme argument une instance d'un objet qui inclut le mixin "`Msf::GeneralEventSubscriber`".

Quand un module est chargé dans l'instance du framework, la procédure "`on module load`" sera appelée si elle est non-nulle et on lui passera le nom de référence et la classe associés avec le nouveau module chargé.

Quand une instance de module est créée, la procédure "`on module created`" sera appelée si non-nulle et on lui passera l'instance du nouveau module créé.

Pour enlever un souscripteur d'événement; on doit appeler "`framework.events.remove general subscriber`" en passant l'instance de l'objet qui a été utilisée pour ajouter le souscripteur en premier.

3.2.3 Événements de Reconnaissance

Pour recevoir des notifications sur les événements de reconnaissance, comme quand un nouvel hôte ou service est détecté; un souscripteur d'événements de reconnaissance peut être enregistré via la méthode "framework.events.add recon subscriber".

Cette méthode prend une instance d'un objet qui implémente un ou plusieurs des mixins "Msf::ReconEvent::HostSubscriber" ou "Msf::ReconEvent::ServiceSubscriber".

Quand un nouvel hôte est détecté ou un attribut d'un hôte a changé; un appel sera effectué à la méthode "on host changed" du souscripteur d'événements, en considérant qu'elle implémente le mixin "Msf::ReconEvent::HostSubscriber".

Quand un nouveau service est détecté ou un attribut d'un service a changé; un appel sera effectué à la méthode "on service changed" du souscripteur d'événements, en considérant qu'elle implémente le mixin "Msf::ReconEvent::ServiceSubscriber".

Pour retirer un souscripteur d'événements; un appel à "framework.events.remove recon subscriber" devra être effectué, en lui passant l'instance de l'objet qui a été utilisé pour ajouter le souscripteur en premier.

3.2.4 Événements de Session

Pour recevoir des notifications sur les événements de sessions, un souscripteur d'événements de session peut être enregistré via la méthode "framework.events.add session subscriber".

Cette méthode prend une instance d'un objet qui implémente le mixin "Msf::SessionEvent".

Quand une nouvelle session est ouverte; le framework va appeler la méthode "on session open" du souscripteur avec l'instance de la session qui vient d'être ouverte comme premier argument.

Quand une session se termine; le framework va appeler la méthode "on session close" avec l'instance de la session qui s'est fermée.

Pour retirer un souscripteur d'événement; il faut appeler "framework.events.remove session subscriber" en lui passant l'instance de l'objet qui a été utilisée pour ajouter le souscripteur en premier.

3.3 Gestionnaires du Framework

Le framework core lui-même est composé de différents gestionnaires qui sont responsables de certains des aspects basiques du framework, comme la gestion des modules et plugins.

3.3.1 Gestion de Module

La gestion de module est l'une des parties les plus importantes du framework.

La classe Msf::ModuleManager est responsable de la fourniture de l'interface pour charger les modules et pour agir comme une factory pour la création d'instance de module.

Le gestionnaire de module lui-même peut être accédé via l'attribut framework.modules

Le chargement des modules est accompli en ajoutant un chemin de recherche au gestionnaire de module en appelant à la méthode "add module path".

Cette méthode va automatiquement charger tous les modules trouvés dans le répertoire spécifié.

Les modules sont symboliquement identifiés par un nom de référence.

Le nom de référence prend une forme similaire à un chemin et est partiellement contrôlé par le chemin du système de fichiers depuis lequel le module est chargé.

Un exemple de nom de référence serait un exploit labellisé "windows/ftp/wsftpd".

Le chemin du module doit être conforme au standard de répertoire de module, avec la structure du répertoire de base étant la même au sous-répertoire de modules dans la distribution du framework.

Cela signifie que l'exploit sera chargé depuis "exploits/windows/ftp/wsftpd.rb".

Il est important de noter que les exploits doivent conserver une hiérarchie d'espace de nommage qui reflète le chemin dans lequel ils sont localisés.

Par exemple, l'exemple décrit précédemment aurait la classe déclarée comme "Msf::Exploits::Windows::Ftp::Wsftpd".

Il est nécessaire que le gestionnaire de module du framework sache quel nom utiliser pour voir quelle classe a été ajoutée après le chargement du fichier.

Le nom de référence d'un module peut être accédé via l'attribut "refname" sur à la fois la classe du module et ses instances.

Pour aider à résoudre le problème potentiel d'ambiguïté pour le nom de module entre les différents types de modules, les modules peuvent également être référencés par un nom de référence complet. Ce nom est le même que le nom de référence du module mais est préfixé par le type du module.

Par exemple, l'exploit "windows/ftp/wsftpd" deviendrait "exploit/windows/ftp/wsftpd".

Le nom de référence complet peut être accédé via l'attribut "fullname" sur à la fois la classe du module et ses instances.

Pour rendre facile à utiliser le gestionnaire de module, chaque type de module différent est décomposé en une classe plus basique appelée "module set" qui est implémentée par la classe Msf::ModuleSet. Le but d'un module set est d'agir comme une factory localisée pour chaque type de module différent (exploit, encodeur, nop, ...).

Chaque module set spécifique à un type peut être accédé soit par "framework.type" soit par "framework.modules.type".

Par exemple, si l'on veut énumérer les modules d'exploit, on utilisera la méthode "framework.exploits" pour accéder au module set des exploits.

Les ensembles de module sont implémentés sous la forme d'un hash qui associe les noms de référence des modules avec leurs classes respectives.

Pour créer une instance d'un module, il faut appeler la méthode "create" d'un module set en passant le nom de référence du module qui doit être instancié.

Par exemple, pour créer une instance d'un exploit nommé "windows/ftp/wsftpd", l'appel suivant doit être fait:

```
framework.exploits.create('windows/ftp/wsftpd')
```

Figure 3.3: Création d'une instance d'un module du framework

La table montrée en figure 3.4 montre la relation entre les types de module et les accesseurs de module set du framework.

Pour recharger les contenus d'un module, il faut appeler "reload module" en passant l'instance de module qui doit être rechargé.

Cela va entraîner le framework à relier les contenus du chemin lié au module et automatiquement créer une nouvelle instance du module.

Type de Module	Accesneur
MODULE ENCODER	framework.encoders
MODULE EXPLOIT	framework.exploits
MODULE NOP	framework.nops
MODULE RECON	framework.recon
MODULE PAYLOAD	framework.payloads

Figure 3.4: Types de modules et leurs accesseurs framework

3.3.2 Gestion de Plugins

Une des nouvelles fonctionnalités dans la version 3.0 du framework est le concept de plugins. Contrairement aux modules, les plugins de framework sont là pour ajouter des fonctionnalités au framework ou pour changer le comportement des aspects existants du framework.

Les plugins ont une définition très LOOSE en termes de SCOPE dans lequel ils peuvent opérer.

Par exemple, un plugin pourrait ajouter un type de module complètement nouveau pour l'utiliser avec le framework.

Un plugin peut également ajouter des commandes aux interfaces utilisateur existantes qui interagissent avec le framework.

Un plugin peut également enregistrer des souscripteurs d'événements personnalisés pour réaliser des choses comme entrainer le Meterpreter à lister automatiquement le contenu du disque C d'un ordinateur quand une nouvelle session Meterpreter est créée.

Les possibilités sont infinies.

Le gestionnaire de plugin peut être accédé via l'accesneur "framework.plugins" qui est une instance de la classe Msf::PluginManager

Pour charger un plugin, il faut appeler "framework.plugins.load" avec le chemin du plugin à charger.

Un second paramètre optionnel peut être passé à la méthode load qui est un hash de paramètres qui peuvent être utiles au plugin, comme les handles "LocalInput" et "LocalOutput" pour les utiliser avec des chaînes vers l'écran indépendamment du medium utilisé actuellement.

La table présentée en figure 3.5 montre les éléments de hash prédéfinis qui peuvent être passés dans le hash d'option.

Elément du Hash	Description
LocalInput	L'instance de classe d'entrée locale qui implémente l'interface Rex::Ui::Text::Input
LocalOutput	L'instance de classe de sortie locale qui implémente l'interface Rex::Ui::Output
ConsoleDriver	L'instance du driver console de Msf::Ui::Console::Driver.
WebDriver	L'instance du driver console de Msf::Ui::Web::Driver.

Figure 3.5: Eléments du hash du constructeur optionnel de Plugin

Tous les plugins sont comptabilisés par référence. Cela permet d'implémenter des plugins singleton qui peuvent être chargés plus d'une fois mais n'auront qu'une seule instance sous-jacente.

Le nombre de références à une instance d'un plugin est automatiquement incrémenté à chaque fois que "load" est appelé sur lui.

Pour décharger un plugin du framework; on utilise "framework.plugins.unload" en lui passant l'instance du plugin précédemment chargé comme premier argument.

Du fait que chaque plugin est compté par référence, un plugin ne sera déchargé que lorsque son compteur de références atteindra zéro.

Pour plus de détails sur l'implémentation des plugins de framework, voir le chapitre 7.

3.3.3 Gestion de Reconnaissance

Le gestionnaire de reconnaissance (recon manager) est utilisé pour fournir une interface pour rapporter des informations sur les hôtes, services, et d'autres entités de reconnaissance.

Ces rapports sont traqués en interne par le gestionnaire de reconnaissance qui est implémenté par la classe `Msf::ReconManager`

Le gestionnaire de reconnaissance peut être accédé via l'accessor "framework.reconmgr".

Cette partie du framework est actuellement en cours de refonte de conception et ne possède donc pas de documentation actuellement.

3.3.4 Gestion de Session

Le gestionnaire de session est utilisé pour traquer les sessions créées dans une instance de framework après qu'un exploit réussisse.

Le but des sessions est de présenter des fonctionnalités à un programmeur qui lui permettent d'interagir avec.

Par exemple, une session de shell de commandes permet aux programmeurs d'envoyer des commandes et de lire les réponses via une API prédéfinie.

Pour plus d'informations, reportez-vous au chapitre 8.

Le gestionnaire de session lui-même peut être accédé via l'accessor "framework.sessions" et est une instance de la classe `Msf::SessionManager`

L'objectif principal du gestionnaire de session est de fournir une interface pour enregistrer de nouvelles sessions et leurs assigner un identificateur de session unique, ainsi que permettre aux sessions d'être désenregistrées quand elles sont détruites.

L'enregistrement d'une session avec le gestionnaire de session du framework se fait en appelant la méthode "framework.sessions.register" qui prend une instance de session comme argument.

Cette méthode va assigner à la session un identificateur unique de session et l'ajouter au hash de sessions géré.

Les sessions peuvent être énumérées en appelant "framework.sessions.each sorted" ou en appelant n'importe quelle méthode d'énumération compatible avec un hash.

Pour obtenir l'instance de la session associée avec un identificateur de session particulier, la méthode "framework.sessions.get" peut être utilisée.

Quand une session est détruite, un appel à la méthode "framework.sessions.deregister" doit être fait avec l'instance de la session à détruire comme premier argument.

3.3.5 Gestion de Job

Chaque instance de framework supporte l'exécution de différentes tâches dans le contexte de threads jusqu'au concept de jobs.

L'interface de job peut être accédée via l'accessor "framework.jobs" qui est une instance de la classe "Rex::JobContainer".

Pour plus d'informations sur les jobs, se reporter à l'explication de job dans la documentation de Rex à la section 2.4.

3.4 Classes d'utilitaires

Certaines classes dans le framework core permettent de réaliser certaines tâches plus facilement sans être en dehors des aspects du core du framework.

Ces classes sont décrites ci-dessous.

3.4.1 Driver Exploit

La classe `Msf::ExploitDriver` encapsule la tâche d'exécution d'un module d'exploit en terme de validation des options de module requises, validation de la sélection de la cible (`target`), génération d'un payload spécifié, et exécution de l'initialisation et libération de l'exploit et du payload.

Ces opérations sont celles qui doivent être déclenchées en tentant d'exécuter un exploit.

Une instance d'un driver d'exploit est initialisée en figure 3.7.

```
driver = Msf::ExploitDriver.new(framework)  
driver.payload = payload_instance  
driver.exploit = exploit_instance  
driver.target_idx = 0  
session = driver.run
```

Figure 3.6: Utilisation de la classe `ExploitDriver`

Quand la méthode "run" est appelée, la première étape est de valider les options requises par le payload et l'exploit qui ont été choisis.

Ceci est fait en appelant la méthode publique "validate" sur l'instance du driver d'exploit.

Dans le cas où l'option échoue à la validation ou qu'un index de cible n'a pas été correctement choisi, une exception sera retournée à l'appelant.

Après la fin de la validation, l'élément datastore `TARGET` est défini avec l'index de cible sélectionné.

Puis une version encodée du payload est générée en appelant "generate payload" sur l'instance de l'exploit.

Ensuite, l'exploit est construit en appelant "setup" sur l'instance du module d'exploit et finalement, le code de l'exploit courant est déclenché en appelant "exploit" sur l'instance du module d'exploit.

Une fois que l'exploitation s'est terminée, le driver d'exploit appelle la méthode "stop handler" sur l'instance du module d'exploit, puis appelle la méthode "cleanup" sur l'instance du module d'exploit

L'on peut aussi indiquer au driver d'exploit de lancer l'exploit dans le contexte d'un job.

Quand ceci est fait, l'opération d'exploitation est faite dans le contexte d'un thread d'un job en appelant "framework.jobs.start bj job".

L'on peut dire au driver d'exploit d'utiliser un job en définissant l'attribut "use job" à true.

3.4.2 Payload Encodé

Le but de la classe `Msf::EncodedPayload` est d'encapsuler l'opération d'encodage d'un payload avec un ensemble arbitraire de besoins.

Pour générer un payload encodé, une instance de la classe "Msf::EncodedPayload" doit être créée en passant à son constructeur une instance d'un payload et un hash optionnel de besoins qui sera utilisé pendant la phase de génération.

Ceci se fait en appelant la méthode "create" de la classe, comme illustré en figure ??.

```

encoded = Msf::EncodedPayload.create(payload_instance,
  'BadChars' => "\x0a\x0d",
  'Space' => 400,
  'Prepend' => "\x41\x41",
  'Append' => "\xcc\xcc",
  'SaveRegisters' => "edi",
  'MinNops' => 16)

```

Figure 3.7: Création d'une instance d'un EncodedPayload

Une fois qu'une instance de payload encodé est créée, l'étape suivante est d'effectuer un appel à la méthode "generate" de la classe qui va retourner la version encodée du payload.

Après la génération, les attributs suivants peuvent être accédés sur l'instance du payload encodés pour obtenir des informations sur le payload encodé.

La Figure 3.8 présente les attributs et leurs buts.

Pour contrôler le comportement de la classe encoded payload; un hash optionnel peut être passé au constructeur.

La table en figure 3.9 décrit les options qui peuvent être spécifiées et l'effet qu'elles ont sur le comportement.

Attribut	Description
raw	Le buffer de payload brut non-encodé.
encoded	Le buffer de payload encodé qui doit être égal au brut si aucun n'encodeur n'a été sélectionné.
nop_sled_size	La taille du traineau de NOP ajouté au début du payload encodé. Zéro si aucun NOP n'a été généré.
nop_sled	La portion du NOP sled du payload encodé, s'il y en a une.
encoder	L'instance du module encodeur qui a été utilisée pour encoder le payload.
nop	L'instance du module encodeur qui a été utilisée pour générer le NOP sled, s'il y en a une.

Figure 3.8: Attributs d'une instance de Msf::EncodedPayload

Elément du Hash	Description
BadChars	Une chaîne de caractères interdits à éviter lors de l'encodage.
Encoder	Le nom de l'encodeur préféré à utiliser.
MinNops	Le nombre minimum de NOPs à générer.
MaxNops	Le nombre maximum de NOPs à générer.
Space	L'espace disponible pour le payload. Si cette valeur n'est pas spécifiée, le padding de NOP ne sera pas réalisé et il n'y aura aucune restriction sur la taille du payload.
SaveRegisters	Une liste séparée par un espace des registres à sauver lors de la génération du NOP sled.
Prepend	Instructions brutes ou texte à ajouter au début du payload encodé.
Append	Instructions brutes ou texte à ajouter à la fin du payload encodé.

Figure 3.9: Options du constructeur de Msf::EncodedPayload

Chapitre 4

Framework Base

La framework base est une couche de librairie construite au dessus du framework core qui ajoute des classes qui rende plus facile d'agir sur le framework.

Elle fournit également un ensemble de classes qui peuvent être utiles au développement d'outils tiers qui ne correspondent pas nécessairement à l'objectif du framework core lui-même.

Les classes qui composent la framework base sont décrites dans les sections suivantes.

Pour utiliser la librairie de base du framework, un script Ruby doit requérir "msf/base".

4.1 Configuration

Un aspect important d'une installation de framework gérée est le concept de configuration persistante et méthodes pour obtenir des informations sur la structure d'une installation, comme le répertoire de base (root directory) de l'installation et d'autres types d'attributs.

Pour faciliter ceci; la librairie framework base fournit la classe "Msf::Config" qui possède des méthodes pour obtenir différents répertoires d'installation.

Elle supporte également la sérialisation des fichiers de configuration.

La table présentée en figure 4.1 décrit les différentes méthodes qui peuvent être utilisées pour obtenir des informations de configuration.

4.2 Logging

La librairie framework base fournit une classe d'emballage qui peut être utilisée pour contrôler les logs de débogage à un niveau administratif en fournissant des méthodes pour activer les sources de log et pour contrôler les logs qui sont appliqués aux sessions créées depuis une instance de framework.

Pour initialiser le logging; on doit appeler "Msf::Logging.init" qui va enregistrer les sources "rex", "core" et "base" comme étant dirigées vers "framework.log" dans le répertoire "Msf::Config.log".

Méthode	Description
install_root	Le répertoire racine de l'installation (root directory).
config_directory	Le répertoire de configuration (configuration directory) (~/.msf3).
module_directory	Répertoire racine de l'installation + '/modules'.
plugin_directory	Répertoire racine de l'installation + '/plugins'.
log_directory	Répertoire de configuration + '/logs'.
session_log_directory	Répertoire de configuration + '/logs/sessions'.
user_module_directory	Répertoire de configuration + '/modules'.
data_directory	Répertoire racine de l'installation + '/data'.
config_file	Répertoire de configuration + '/config'.
load	Charge le contenu d'un fichier de configuration et retourne une instance d'un objet "Rex::Parser::Ini".
save	Sauvegarde le hash d'option fourni dans le fichier de configuration spécifié comme 'ConfigFile' dans le hash d'options hash ou le fichier "config" par défaut.

Figure 4.1: Méthodes de l'accessor Msf::Config

Des sources de log individuelles peuvent être successivement activée ou désactivée en appelant "Msf::Logging.enable log source" et "Msf::Logging.disable log source", respectivement. Quand le logging de session est activé; des appels peuvent être effectués à "start session log" et "stop session log" qui agissent sur une instance de session spécifiée pour démarrer et interrompre le logging dans un fichier de log spécifique à la session dans le répertoire de log "Msf::Config.session log directory".

4.3 Sérialisation

Pour faciliter la vie aux développeurs du framework; la librairie framework base fournit une classe qui peut être utilisée pour sérialiser des informations sur les modules, comme leur description, options, et d'autres informations dans un format uniforme et humainement lisible.

Cette classe est la classe "Msf::Serializer::ReadableText".

Pour plus d'informations, veuillez-vous reporter à la documentation autogénérée de l'API sur le site Metasploit.

4.4 Sessions

Alors que le framework core possède un concept abstrait de sessions, comme décrit à travers du module de base "Msf::Session", la framework base fournit actuellement certaines implémentations concrètes. Cette séparation fut établie pour éliminer des implémentations de session spécifique à chaque module du framework core comme le core ne doit pas avoir de dépendances conceptuelles sur les modules qui l'utilisent.

La librairie de base, de l'autre côté, est plus une couche (layer) de facilitation pour les souscripteurs du framework.

Les deux sessions actuellement implémentées dans la librairie de base sont les sessions "CommandShell" et "Meterpreter".

4.4.1 CommandShell

La session de commandes shell implémente l'interface "Msf::Session::Provider::SingleCommandShell" face à un flux connecté, comme une connexion TCP par exemple.

Pour plus d'informations sur ce mixin; se référer au chapitre 8.

4.4.2 Meterpreter

La session Meterpreter implémente les mixins "Msf::Session::Interactive" et "Msf::Session::Comm". Cela lui permet d'opérer via un shell utilisateur interactif et aussi indiquer au framework que le trafic internet peut être routé (pivoté) via la session en utilisant un factory socket Comm.

La session elle-même est MERELY une extension de la classe "Rex::Post::Meterpreter" qui opère face à un flux connecté, comme une connexion TCP.

4.5 Framework Simplifié

Le Framework simplifié fournit des méthodes qui rendent le framework et les différents types de modules plus faciles à utiliser en fournissant des méthodes d'emballage qui gèrent la plupart des actions qui seraient communes à un souscripteur du framework.

Pour créer une instance de framework simplifié; il faut appeler la méthode "Msf::Simple::Framework.create" avec un hash optionnel.

La valeur de retour est une instance de la classe "Msf::Framework" qui a été étendue par le mixin "Msf::Simple::Framework".

Les instances de framework existantes peuvent également être simplifiées en appelant la méthode "Msf::Simple::Framework.simplify" avec l'instance de framework existante comme premier argument. Toutes les instances de module d'une instance de framework simplifié seront automatiquement simplifiées par les mixins spécifiques au type de module.

La création d'une instance de framework simplifié entraîne automatiquement l'initialisation des classes "Msf::Config" et "Msf::Logging".

N'importe quel fichier de configuration existant est également chargé automatiquement.

Le répertoire de module global par défaut (répertoire Msf::Config::module) et le répertoire de module spécifique à l'utilisateur (Msf::Config::user) sont ajoutés comme chemins de recherche à l'instance de framework qui entraîne le chargement de tous les modules contenus dans ces deux répertoires.

Finalement, un souscripteur d'événements général est enregistré dans l'instance de framework qui sera appelé à chaque fois qu'une instance de module est créée dans le framework.

Cela permet au framework simplifié de simplifier chaque instance de module créée.

Chaque type de module possède un mixin de module de framework simplifié qui est automatiquement utilisé pour étendre les instances de module créées via le souscripteur d'événements général décrit plus haut.

Par exemple; quand une instance d'un module d'exploit est créée, l'instance est étendue par le mixin "Msf::Simple::Exploit".

Chaque mixin de module différent fournit une méthode d'aide (helper) ou des méthodes pour piloter l'action ou les actions primaires de ce type de module.

De plus, chaque instance de module possède des méthodes qui peuvent être employées pour enregistrer et restaurer des éléments de configuration spécifique à un module via les méthodes "save config" et "load config".

Chaque mixin spécifique à un module est décrit individuellement ci-dessous.

4.5.1 Exploit

Le mixin exploit simplifié fourni dans "Msf::Simple::Exploit" étend chaque instance de module d'exploit avec une méthode appelée "exploit single".

Cette méthode prend en argument un paramètre hash qui est utilisé pour contrôler l'exploitation de quelque chose en créant une instance d'une classe "Msf::ExploitDriver" et en effectuant toutes les initialisations et configuration requises du module avant d'effectuer l'appel à la méthode "run" du driver d'exploit.

Si l'exploitation réussie; la valeur de retour est une instance de session.

Sinon; une exception est déclenchée ou rien ne sera retourné.

Pour plus d'informations sur les éléments de hash qui peuvent être passés; veuillez vous référer à la documentation autogénérée de l'API sur le site Metasploit.

4.5.2 NOP

Le mixin NOP simplifié fourni dans "Msf::Simple::Nop" étend chaque instance de module nop avec une méthode appelée "generate simple".

Cette méthode prend en arguments: la longueur du traineau (sled) à générer et le hash d'options qui seront utilisés pour la génération.

En cas de succès; la valeur de retour est un buffer qui est encodé en utilisant la classe

"Msf::Simple::Buffer" en utilisant le format spécifié dans le hash d'option comme élément "Format".

Si aucun format n'est spécifié; la version RAW du NOP sled est retournée.

4.5.3 Payload

Le mixin payload simplifié fourni dans "Msf::Simple::Payload" étend chaque instance de module payload avec une méthode "generate simple".

Cette méthode prend en argument un hash d'options qui sont utilisées pour générer un buffer de payload.

Les éléments qui peuvent être utilisés dans le hash d'option peuvent être trouvés dans la documentation autogénérée sur le site Metasploit.

Si l'opération réussie; le buffer de payload encodé sera sérialisé au format fourni dans l'élément "Format" du hash.

Si le format n'est pas RAW; n'importe quel payload étagé sera aussi adjoint au buffer sérialisé.

4.5.4 Recon

L'interface de reconnaissance est en cours de conception.

Chapitre 5

Framework Ui

La librairie Interface Utilisateur du framework est utilisée pour encapsuler le code commun à différents médiums d'interface utilisateur pour permettre le développement d'outils tiers et l'extension d'interfaces utilisateur personnalisées séparées de celles fournies avec le framework lui-même.

Chaque médium d'interface utilisateur différent est encapsulé dans une classe de driver abstraite "Msf::Ui::Driver" qui est conçue pour avoir une interface actuelle qui est spécifique au médium d'interface utilisateur sous-jacent qui est utilisé.

La classe driver de base héritée définit seulement trois méthodes qui seront communes à toutes les interfaces utilisateur : "run", "stop" et "cleanup"

Leurs noms sont explicites.

Chacun des médiums d'interface utilisateur actuellement défini sera abordé individuellement dans les sections suivantes.

Pour utiliser la librairie ui; un script Ruby doit requérir "msf/ui".

Chapitre 6

Modules Framework

Le but principal du framework Metasploit est de faciliter le développement de modules qui peuvent se plugger dans le framework core et être partagés avec d'autres modules existants.

Par exemple, un module d'encodage avancé peut être pluggé dans le framework et sera automatiquement appliqué aux payloads d'une architecture et plateforme compatible.

Ainsi il n'y a aucune modification de code requise du fait que tous les modules sont conformes à une interface prédéfinie à travers laquelle ils peuvent être accédés via le framework.

Un autre exemple: de nouveaux payloads peuvent être développés et sont immédiatement utilisables par tous les exploits sans modification.

Cela élimine la nécessité d'avoir à copier des morceaux de payload statiques dans les exploits comme c'est très courant pour les exploits Proof Of Concept.

Ce chapitre est dédié aux interfaces que chaque type de module expose pour bien comprendre ce qui est nécessaire pour implémenter chaque type de module.

A un certain niveau, tous les modules héritent de la classe module base dans `Msf::Module`.

Cette classe implémente toutes les choses communes aux modules du framework Metasploit, comme les accesseurs et les attributs.

Quand un module est chargé dans le framework, une copie de la classe qui est ajoutée est faite, qui est ce qui sera utilisé pour les prochaines instanciations du module.

La classe copiée possède certains de ses attributs définis pour permettre au framework de retrouver certaines informations sur le module d'un seul coup sans avoir à créer une instance de celui-ci.

Cette information peut être accédée via un ensemble de méthodes de classe et d'attributs qui sont décrits en figure 6.1.

Pour supporter l'initialisation générique; chaque module définit son propre hash d'information personnalisé qui est éventuellement passé au constructeur de `Msf::Module`.

Cette classe d'information est ensuite assignée à l'attribut `named module info` de l'instance, puis est traitée.

Les parties communes a tous les modules sont individualisées et transformées en types uniformes qui peuvent être accédés via des méthodes de l'instance.

Les mêmes méthodes qui sont accessibles via la classe module peuvent aussi être utilisées via la classe de l'instance (comme illustré en figure 6.1).

Méthode	Description
framework	L'instance framework à laquelle le module est associée.
type	Le type symbolique du module. Parmi: <code>MODULE ENCODER</code> , <code>MODULE EXPLOIT</code> , <code>MODULE NOP</code> , <code>MODULE PAYLOAD</code> , ou <code>MODULE RECON</code>
fullname	Le nom symbolique complet du module incluant son type chaîne. Par exemple: <code>exploit/windows/ms03 026</code>
rank	La note entière du module indiquant sa qualité. La note est utilisée par le framework lors du choix parmi les encodeurs, payloads, et générateurs de NOP à utiliser.
rank_to_s	Retourne la représentation sous forme d'une chaîne de la note du module.
refname	Le nom de la référence symbolique du module. Par exemple: <code>windows/ms03 026</code>
orig_cls	La classe originelle, non-dupliquée qui a été chargée pour le module.
file_path	Le chemin d'où a été chargé le module.

Figure 6.1: Méthodes de la classe `Msf::Module`

La table en figure 6.2 illustre comment les éléments communs du hash d'information du module sont séparés dans leurs types de données respectifs et les méthodes qui peuvent être utilisées pour les accéder. Certains des accesseurs au hash d'information possèdent également des méthodes d'aide qui rendent plus facile d'interagir avec eux.

Par exemple, le tableau d'éléments de hash "Arch" contenu dans l'attribut "arch" peut être sérialisé en une chaîne séparée par des virgules en appelant "arch to s".

Les architectures peuvent également être énumérées en appelant "each arch" en lui passant un bloc qui accepte l'architecture comme paramètre.

Il est également possible de vérifier si un module supporte une architecture en appelant la méthode "arch?" en lui passant l'architecture comme paramètre.

Tout comme les architectures, les plateformes peuvent être sérialisées en une chaîne en appelant "platform to s".

L'élément "Author" du hash peut aussi être converti en une chaîne séparée par des virgules d'auteurs en appelant "author to s".

Le tableau des instances de "Msf::Author" contenu dans l'attribut tableau "author" peut être énuméré en appelant "each author" en lui passant un bloc qui prend une instance d'auteur comme premier argument.

La classe "Msf::Module" possède également certaines méthodes d'aide communes qui permettent aux utilisateurs de vérifier rapidement si un module est d'un type spécifique en appelant l'ensemble de méthodes "<type>?".

Par exemple, si un appelant veut voir si une instance de module est un exploit; il peut appeler "mod.exploit?".

Elément du Hash	Accesseur	Type	Description
Name	name	String	Le nom court du module.
Alias	alias	String	Une chaîne alias pour le "refname" du module.
Description	description	String	Une description plus longue du module.
Version	version	String	La révision courante du module dérivé.
License	license module.	String	La licence sous laquelle est distribué le
Author	author	Array	Un tableau d'instances de Msf::Author
Arch	arch	Array	Un tableau d'architectures (comme ARCH X86).
Platform	platform	PlatformList	Une instance de Msf::PlatformList
References	references	Array	Un tableau d'instances de Msf::Reference
Options	options	OptionContainer	Les Options transportées dans le hash sont ajoutées au conteneur option du module.
AdvancedOptions	options	OptionContainer	Les Options transportées dans le hash sont ajoutées au conteneur option du module comme options avancées.
DefaultOptions	options	OptionContainer	Les options enregistrées précédemment ont leurs valeurs par défaut modifiées.
Privileged	privileged	Bool	Si le module nécessite ou attribue un accès privilégié.
Compat	compat	Hash	Un hash de drapeaux de compatibilité.

Figure 6.2: Accesseurs du hash d'information Msf::Module

Du fait que la librairie Rex introduit le concept d'usines (factories) de communication socket (à travers la classe Comm), chaque module possède un attribut qui peut retourner l'instance Comm qui a été utilisée ou préférée. Par défaut, tous les modules retournent "Rex::Socket::Comm::Local".

Chaque module a son propre datastore basée sur une instance qui est une instance de la classe "Msf::ModuleDataStore" et peut être accédée via l'accessoire datastore.

Cela réplique la fonctionnalité fournie par le datastore global du framework dans le sens où cela fournit une variable localisée à l'association de valeur pour l'utilisée pour satisfaire aux options requises.

Par exemple, si un module requière que l'option RHOST soit définie à une valeur, le datastore du module doit posséder une entrée de hash RHOST.

Eventuellement, les modules sont conçus pour être capables de retomber dans le datastore global du framework si leur datastore localisé ne possède pas de valeur pour une variable étant vérifiée.

Cela fournit un niveau basique d'héritage variable/valeur.

Dans certains cas, les modules pourraient vouloir partager leurs copies localisées du datastore avec les autres modules sans avoir à TAINTE le datastore global.

Cela peut être réalisé en appelant la méthode "share datastore" sur une instance de module en lui passant une instance de datastore comme premier argument.

Au final, les modules du framework sont conçus pour être capables d'indiquer leurs compatibilités relatives avec les autres modules.

Par exemple, un exploit peut vouloir indiquer qu'il est compatible avec une classe spécifique de médiums de connexion de payload.

Ceci est accompli via l'élément du hash "Compat".

Après que la couche de compatibilité a été initialisée, des appels peuvent être effectués à une méthode "compatible?" d'un module en passant une autre instance de module comme argument.

Si l'instance de module fournie est compatible avec l'instance avec laquelle elle est vérifiée, alors "true" est retourné.

Cette interface basique fournit une vue généralisée du comportement et des exceptions des modules du framework.

Néanmoins, tous les types de modules possèdent des interfaces bien définies pour s'occuper des actions qu'ils sont censés entreprendre.

Ces interfaces spécifiques vont être décrites dans les sections suivantes.

6.1 Encodeur

Les modules encodeurs sont utilisés pour générer les versions transformées des payloads bruts d'une manière qui leur permet d'être restaurés sous leur forme originelle au moment de l'exécution, puis être exécutés.

Pour accomplir ceci, la plupart des encodeurs vont prendre la forme brute du payload et le passer dans une sorte d'algorithme d'encodage, comme un XOR bit à bit.

Après que la version encodée soit générée, un moignon (stub) de décodage est préfixé à la version encodée du payload.

Ce moignon est responsable d'effectuer l'opération inverse sur le buffer attaché au décodeur quand il s'exécute.

Une fois que le décodeur restaure le payload sous sa forme originelle, il va transférer l'exécution au début du payload maintenant normalisé.

Pour supporter ce modèle d'encodage; le framework Metasploit fournit la classe "Msf::Encoder" qui hérite de la classe de base "Msf::Module".

Tous les encodeurs doivent hériter de la classe "Msf::Encoder" à un certain niveau pour s'assurer que les méthodes spécifiques à un encodeur sont incluses dans la classe dérivée.

A l'instar du hash d'information de module, les encodeurs possèdent quelques éléments de hash d'informations spécialisés qui décrivent des informations sur l'encodeur étant utilisé.

Les informations que les modules encodeurs ont besoin de décrire sont les attributs de l'encodeur qui sont CONVEYED via l'élément du hash d'information "Decoder".

L'élément "Decoder" du hash d'information référence un autre hash qui contient les propriétés spécifiques au décodeur.

Elles sont décrites dans la table présentée en figure 6.3 avec leurs types et accesseurs d'instance de module.

Elément du Hash	Accesseur	Type	Description
Stub decoder	stub	String	Le moignon brut à préfixer aux payloads encodés.
KeyOffset decoder	key offset	Fixnum	L'offset de la clé dans le moignon de décodage.
KeySize decoder	key size	Fixnum	La taille de la clé de décodage en octets.
BlockSize decoder	block size	Fixnum	La taille de chaque bloc encodé en octets.
KeyPack decoder	key pack	String	L'ordre des octets à utiliser en paquetant la clé 'V' par défaut.

Figure 6.3: Accesseurs du hash d'information Decoder de Msf::Encoder

Chacune des méthodes décrites en figure 6.3 est conçue pour pouvoir être surdéfinie, ainsi les classes encodeur dérivées peuvent dynamiquement choisir les valeurs retournées par elles plutôt que d'être forcées à s'initialiser dans un élément de hash statique.

Le hash du décodeur lui-même peut être accédé via la méthode hash decoder au cas où un module encodeur veuille CONVEY des informations non-standards dans le hash pour une référence plus tard.

Il est peut-être plus important que le vecteur d'initialisation du décodeur comment le processus d'encodage est exposé.

La classe de base "Msf::Encoder" implémente une méthode d'instance nommée "encode" qui prend un buffer comme premier argument, une chaîne de caractères interdits (ou nul) comme second argument, et un état d'encodeur optionnel comme troisième argument.

La méthode "encode" englobe le processus d'encodage en termes de sélection d'une clé de décodage, initialisation de l'état de l'encodeur, puis déclenchement réel de l'opération d'encodage.

Une fois complétée; le buffer encodé est retourné à l'appelant.

C'est la première méthode que le framework utilise en interagissant avec les modules encodeurs du framework.

6.1.1 encode

A un niveau plus détaillé, la méthode "encode" crée en premier lieu une instance de la classe "Msf::EncoderState" si aucune n'a été spécifiée comme troisième argument.

Le rôle de l'état d'encodeur est de contenir les informations sur une opération d'encodage spécifique d'une manière non-globale.

Après avoir créé l'instance de l'état d'encodeur, encode ajoute au début de chaque donnée spécifique à un encodeur au payload brut qui peut être nécessaire via la méthode d'instance "prepend buf" sur le module encodeur.

Cette méthode est censée être surdéfinie et utilisée comme nécessaire.

Par défaut, une chaîne vide est retournée, laissant effectivement le buffer dans le même état où il était quand il lui est passé.

Après avoir ajouté en début du buffer brut comme nécessaire, la méthode encode choisit ensuite une clé de décodeur si la méthode "key size method" du décodeur retourne une valeur non-nulle et l'état du décodeur a actuellement une clé nulle.

Cela se fait en appelant la méthode "find key" sur le module encodeur qui a une implémentation par défaut qui est censée fonctionner sur tous les modules encodeurs.

Une fois qu'une clé a été sélectionnée, la méthode "init key" est appelée sur l'objet état du décodeur pour définir les attributs de clé "state.key" et "state.orig".

Si aucune clé n'est trouvée, une exception "Msf::NoKeyError" est déclenchée.

L'étape suivante est d'initialiser certains des attributs spécifiques de l'état d'encodeur en appelant la méthode "init state" sur l'instance du module encodeur qui stocke simplement l'offset de clé, taille et pack du décodeur actuellement définis comme attributs de l'état d'encodeur comme transportés via les méthodes de l'accessor sur l'instance du module encodeur lui-même.

L'état d'encodeur se voit ensuite défini comme attributs les caractères interdits et le buffer brut, ainsi ils peuvent être contextuellement référencés dans tout le processus d'encodage.

Avec l'état d'encodeur finalement défini, l'étape suivante est de commencer le processus d'encodage en appelant la méthode "encode begin" sur l'instance du module encodeur.

Cette méthode ne fait rien par défaut, mais elle est conçue pour permettre aux modules encodeur dérivés d'altérer les attributs de l'état d'encodeur avant de réellement débiter le processus d'encodage.

Une fois que "encode begin" se termine, la méthode encode appelle la méthode "do encode" en lui passant le buffer, les caractères interdits et l'état d'encodeur initialisé.

C'est la méthode qui fait réellement le travail d'encodage et peut potentiellement être surdéfinie si l'implémentation par défaut n'est pas appropriée pour un encodeur donné.

Une fois que "do encode" se termine, la méthode encode appelle "encode end" et lui passe l'état d'encodeur comme argument.

L'implémentation par défaut de cette méthode ne fait rien, mais est fournie de telle sorte qu'un encodeur peut hooker dans la finalisation du processus d'encodage pour altérer les résultats qui seront retournés à l'appelant.

6.1.2 do encode

La méthode "do encode" est actuellement le cheval de labour du processus d'encodage.

Elle commence par faire une copie du moignon de décodeur en appelant la méthode "decoder stub" de l'instance du module encodeur en lui passant l'état d'encodeur comme argument.

La méthode "decoder stub" est la seule qui prend un état d'encodeur comme argument comme certains encodeurs peuvent générer des moignons de décodeur dynamiques en fonction de l'état.

Après avoir obtenu le stub de décodeur, l'étape suivante est de substituer la version packée de la clé du décodeur à l'offset qui a été transporté dans le hash d'information du décodeur via le "KeyOffset" et "KeySize" tout comme le "KeyPack".

Ces attributs sont obtenus via les attributs de l'état du décodeur du fait qu'il est possible qu'un encodeur dérivé puisse vouloir altérer leurs valeurs pour être non-statique entre les itérations du processus d'encodage.

Finalement, l'encodage par bloc actuel survient simplement en parcourant le buffer brut dans les morceaux (chunks) de taille d'un bloc en appelant la méthode "encode block" sur chaque chunk.

Cette méthode se voit passé l'état d'encodeur et le chunk à encoder.

Par défaut, la méthode "encode block" retourne simplement le bloc qu'on lui passe, mais tous les encodeurs sont censés surdéfinir cette méthode pour retourner la valeur encodée du bloc basée sur l'état actuel de l'encodeur.

Après que tous les blocs aient été encodés, l'attribut "encoded" de l'état d'encodeur va contenir la version d'encodage de chaque bloc.

La méthode "do encode" ajoute ensuite le stub de décodeur au début du buffer encodé, puis vérifie pour voir si le buffer complet (STUB + buffer encodé) contient un caractère interdit.

Si un caractère interdit est trouvé, une exception "Msf::BadcharError" est déclenchée à l'appelant, indiquant quel caractère et sa position dans le buffer encodé.

Si tout se passe bien; la méthode "do encode" retourne "true".

6.1.3 Méthodes d'aide

En interne, la classe du module encodeur possède quelques instances de méthodes d'aide qui peuvent être utilisées par les classes dérivées pour rendre les choses plus faciles.

Par exemple, la classe de base du module encodeur possède une méthode nommée "badchars?" qui peut être utilisée pour vérifier si le buffer fourni contient un des caractères interdits spécifiés.

Si tel est le cas, l'index du premier caractère interdit est retourné. Sinon, on retourne zéro.

6.2 Exploit

Les modules d'exploit sont utilisés pour accroître les vulnérabilités d'une manière qui permet au framework d'exécuter du code arbitraire.

Cette large définition englobe des choses comme l'exécution de commandes et l'exécution de code qui sont définis en termes de payloads dans la nomenclature du framework.

Le support des modules exploit est fournit par la classe de base "Msf::Exploit".

Tous les modules d'exploit doivent dérivés de la classe de base "Msf::Exploit" à un certain niveau.

L'interface primaire exposée par les modules exploit au framework est des méthodes qui peuvent être utilisées pour vérifier si une cible est vulnérable et lancer l'exploit.

Ces méthodes sont présentées plus loin dans cette section.

A m'instar du hash d'information de module, les modules d'exploits possèdent quelques éléments de hash d'information spécifiques au module exploit qui sont utilisés pour contrôler la manière dont le framework interagit avec le module exploit et pour contrôler le module exploit lui-même.

Ces éléments de hash d'information spécifiques au module exploit sont présentés dans la table en figure 6.4:

Elément du Hash	Accesseur	Type	Description
Stance	stance	Exploit::Stance	Exploit::Stance::Aggressive ou Exploit::Stance::Passive
Targets	targets	Array	Un tableau d'instances de Msf::Target
DefaultTarget	default_target	Fixnum	L'index de la cible par défaut a utilisé s'il y en a.
Payload	payload_info	Hash	Un hash d'éléments qui contrôle l'interaction de l'exploit avec les payloads.

Figure 6.4: Eléments du hash d'information de Msf::Exploit

Les sous-sections suivantes vont décrire les différences entre les différents types et positions (stances) des modules d'exploit, ainsi que les interfaces qui peuvent être employées pour travailler avec.

6.2.1 Stances - Positions

Dans la version 3.0 du framework, les modules exploit sont conçus pour prendre une position (stance) qui décrit comment ils vont procéder à l'exploitation d'une vulnérabilité d'un niveau général.

Du fait qu'il y a un débat sur la manière avec laquelle ce découpage doit se passer, le framework les classe en deux catégories de base appelées "stances".

La première stance qu'un exploit peut prendre est une stance agressive.

Dans ce mode, un exploit déclenche activement un exploit.

La deuxième stance qu'un exploit peut prendre est une stance passive.

Dans ce mode, un exploit attend que quelque chose se passe, comme un client qui se connecte à un serveur, avant de déclencher l'exploit.

Les stances ne sont pas conçues pour prendre en compte la localité.

Elles découpent seulement la façon dont un exploit va opérer.

Le framework utilise la stance d'un exploit pour montrer comment cela doit se passer pour exécuter la méthode exploit.

Par exemple, les exploits passifs vont implicitement prendre plus de temps du fait qu'il attende qu'un événement déclenche l'exploitation.

Pour cette raison, il est meilleur pour le framework de lancer des exploits passifs dans le contexte d'un job plutôt que de bloquer sur leur routine exploit.

De plus, les exploits passifs peuvent être capables d'exploiter plus d'une cible avant qu'ils soient complétés.

Pour indiquer une stance passive, un module doit initialiser l'élément "Stance" du hash d'information à "Msf::Exploit::Stance::Passive".

Si un module souhaite prendre une stance agressive, ce qui est la valeur par défaut, il doit initialiser l'élément "Stance" du hash d'information à "Msf::Exploit::Stance::Aggressive."

6.2.2 Types

Pour catégoriser davantage les exploits, chaque exploit est caractérisé par un type d'exploit.

Le but du type d'exploit est d'indiquer s'il s'agit d'un exploit pour une machine distante, une application locale ou les deux.

Le type exploit distant (remote exploit), indiqué par "Msf::Exploit::Type::Remote", est utilisé pour spécifier au framework que l'exploit est conçu pour opérer contre une machine différente de la machine locale.

Cela ne limite pas explicitement l'exploit à l'utilisation d'une communication réseau, bien que c'est majoritairement le cas.

Les modules exploit peuvent indiquer qu'ils sont des exploits distants en héritant de "Msf::Exploit::Remote", qui hérite de "Msf::Exploit".

Le type exploit local (local exploit), indiqué par "Msf::Exploit::Type::Local", est utilisé pour spécifier au framework que l'exploit est conçu pour opérer contre une application ou un service tournant sur la machine locale.

Cette définition le limite donc à une exploitation sur la machine locale autre que via une communication réseau.

Les modules exploit peuvent indiquer qu'ils sont des exploits locaux en héritant de "Msf::Exploit::Local", qui hérite de "Msf::Exploit".

Le troisième type d'exploit "Msf::Exploit::Type::Omni" est utilisé pour indiquer au framework que le module exploit est capable d'opérer à la fois localement et à distance. Les modules exploit qui correspondent à ce critère doivent hériter directement de la classe "Msf::Exploit".

6.2.3 Interface

Pour interagir avec les modules exploit, le framework utilise une interface prédéfinie qui est exposée par la classe de base "module exploit". Ses méthodes sont décrites dans les sous-sections suivantes.

check

La méthode "check" d'un module exploit est utilisée pour indiquer si oui ou non une machine distante semble vulnérable.

L'implémentation par défaut de la méthode "check" retourne simplement, par défaut, qu'elle n'est pas supportée par le module exploit.

Néanmoins, un ensemble complet de codes peuvent être retournés. Ils sont exposés en figure 6.5

Check Code	Description
Exploit::CheckCode::Safe	La cible n'est pas exploitable.
Exploit::CheckCode::Detected	Le service cible tourne, mais ne peut pas être validé.
Exploit::CheckCode::Appears	La cible semble être vulnérable.
Exploit::CheckCode::Vulnerable	La cible est vulnérable.
Exploit::CheckCode::Unsupported	L'exploit ne supporte pas la vérification.

Figure 6.5: Codes retournés par les appels à exploit.check

exploit

La méthode "exploit" du module exploit est le point d'entrée utilisé pour déclencher le processus d'exploitation.

Avant d'appeler cette méthode, le framework va s'assurer que toutes les options requises ont été définies et qu'un payload a été généré pour être utilisé par l'exploit.

Ensuite, cela réalise l'action nécessaire pour déclencher la vulnérabilité en question.

setup

Si une instance de payload a été créée et assignée à l'exploit, la méthode "setup" va initialiser le gestionnaire de payload en appelant "setup handler" sur lui et va démarrer le gestionnaire en appelant "start handler".

La méthode "setup" est appelée par le framework avant d'appeler la méthode "exploit".

cleanup

La méthode "cleanup" offre à un module exploit une chance de retirer n'importe quelle ressource qui a été créée pendant l'appel à l'exploit, et offre aussi à la classe de base du module exploit une chance d'appeler "cleanup handler" sur l'instance du payload associé avec l'exploit, s'il y en a un.

generate payload

Cette méthode est utilisée par le framework pour générer un payload en utilisant soit une instance de payload passée comme argument, soit en utilisant l'attribut "payload instance" de l'instance du module exploit.

La valeur de retour est une instance d'un "EncodedPayload" qui prend en compte certains des facteurs limitants de payload décrits dans le hash d'information du payload.

Elle prend également en compte des facteurs limitants de payload liés à la spécificité d'une cible.

Le payload encodé en résultant est assigné à l'attribut payload du module d'exploit.

generate single payload

Cette méthode génère un payload encodé en utilisant soit l'instance de payload spécifiée, ou l'instance de payload assignée à l'exploit, et le retourne à l'appelant sous la forme d'une instance d'"EncodedPayload".

Le payload encodé n'est pas assigné comme attribut d'instance.

regenerate payload

Cette méthode englobe simplement "generate single payload" en considérant l'instance du payload de l'exploit comme premier argument.

6.2.4 Accesseurs et Attributs

Les modules exploit possèdent bon nombre d'accesseurs et d'attributs qui peuvent être utilisés par des modules exploit dérivés pour rendre leur vie plus facile.

Ces accesseurs et attributs sont décrits ci-dessous.

compatible payloads

Cette méthode retourne un tableau de payloads qui sont compatibles avec la cible actuellement sélectionnée, ou avec les cibles si aucune n'a été sélectionnée.

Le tableau retourné est composé d'un tableau à deux éléments qui sont le nom de référence du payload compatible et la classe associée avec le payload.

Cette méthode prend en compte les restrictions d'architecture et de plateformes spécifiées par la cible actuellement sélectionnée, s'il y en a une.

handler

La méthode "handler" est utilisée par les exploits pour passer des informations au payload associé qui peuvent être requises ou utiles pour détecter si une session a été créée.

Par exemple, tous les payloads FIND-STYLE nécessitent la connexion originelle qui a été utilisée pour déclencher la vulnérabilité.

En appelant la méthode "handler" avec le socket qui a été utilisé, le payload peut vérifier et voir si une session a été créée.

make nops

Dans certains cas, un exploit peut avoir besoin de générer un NOP sled en dehors du contexte de la génération d'un payload encodé normal.

Pour faire ceci, un appel peut être effectué à la méthode d'instance "make nops" en spécifiant la longueur du sled qui doit être généré.

nop generator

Cette méthode retourne une instance du premier générateur de nop compatible.

nop save registers

Cette méthode retourne les informations du "NOP save register" de la cible choisie si l'attribut "target" est non-nul et l'attribut de registre "target.save" est non-nul.

Sinon, la valeur de l'élément "SaveRegisters" du hash d'information du module est retournée.

payload

Cet attribut est une instance d'un "Msf::EncodedPayload" après qu'un appel à "generate payload" ait été effectué.

payload append

Cette méthode retourne les informations jointes au payload de la cible choisie si l'attribut "target" est non-nul et que l'attribut "target.payload append" est non-nul.

Sinon, la valeur de l'élément "Append" du hash d'information du payload est retournée.

payload badchars

Cette méthode retourne la valeur de l'élément "BadChars" du hash d'information du payload.

payload info

Cette méthode retourne la valeur de l'élément "Payload" du hash d'information du module qui est utilisé pour transmettre les restrictions de payload spécifiques au module.

payload instance

Cet attribut est défini avec l'instance du payload qui a été utilisée pour générer le payload encodé transmis dans l'attribut "payload".

payload max nops

Cette méthode retourne la longueur maximum du NOP sled du payload de la cible choisie si l'attribut "target" est non-nul et que l'attribut "target.payload max nops" est non-nul.
Sinon, la valeur de l'élément de hash "MaxNops" dans le hash d'info du payload est retournée.

payload min nops

Cette méthode retourne la longueur minimum du NOP sled du payload de la cible sélectionnée si l'attribut cible ("target") est non-nul et que l'attribut "target.payload min nops" est non-nul.
Sinon, la valeur de l'élément de hash "MinNops" dans le hash d'info du payload est retournée.

payload prepend

Cette méthode retourne l'information "append" du payload de la cible sélectionnée si l'attribut cible ("target") est non-nul et que l'attribut "target.payload append" est non-nul.
Sinon, la valeur de l'élément de hash "Append" dans le hash d'info du payload est retournée.

payload prepend encoder

Cette méthode retourne l'information "prepend encoder" du payload de la cible sélectionnée si l'attribut cible ("target") est non-nul et que l'attribut "target.payload prepend encoder" est non-nul.
Sinon, la valeur de l'élément de hash "PrependEncoder" dans le hash d'info du payload est retournée.

payload space

Cette méthode retourne l'espace maximum pour le payload du payload de la cible sélectionnée si l'attribut cible ("target") est non-nul et que l'attribut "target.payload space" est non-nul.
Sinon, la valeur de l'élément de hash "Space" dans le hash d'info du payload est retournée.

stack adjustment

Cette méthode retourne les instructions associées avec l'ajustement du pointeur de pile (stack pointer) par une quantité fixée d'une manière indépendante de l'architecture.

En premier lieu, la méthode regarde si un ajustement de pile spécifique à la cible a été défini, et si oui l'utilise.

Sinon, la méthode utilise l'ajustement de pile spécifié en tant que valeur de l'élément de hash "StackAdjustment" dans le hash d'info du payload.

A partir de là, la méthode tente de générer les instructions associées avec l'architecture spécifique de la cible ou du module.

target

Cet attribut retourne l'instance de "Msf::Target" associée avec l'index de la cible qui a été défini dans le datastore du module via la valeur de l'option "TARGET".

Si l'index est invalide ou nul; on retourne zéro.

Cet attribut est généralement utilisé par les exploits pour obtenir des informations d'adressage spécifiques à la cible.

6.2.5 Mixins

L'une des modifications majeures de l'architecture dans la version 3.0 du framework est l'introduction des mixins d'exploit.

Le but des mixins d'exploit est de réduire, et dans la plupart des cas éliminer, le code dupliqué qui est souvent partagé par les modules d'exploits qui tente d'exploiter des vulnérabilités trouvées dans certaines implémentations de protocoles spécifiques.

Les mixins fournissent également une manière de partager du code qui est souvent utilisé, indépendamment du protocole, comme la génération d'un enregistrement SEH durant l'exploitation d'une sur-écriture du SEH.

En plaçant ce code dans des mixins, le framework peut augmenter le support en niveaux de partage et introduire des choses comme l'évasion normalisée sans avoir à modifier chaque exploit existant.

L'encapsulation est très puissante.

Ces mixins sont là pour être inclus dans les exploits qui en ont besoin.

On peut inclure plus d'un mixin dans le même exploit.

Au fur et à mesure que le framework grossit, le nombre de mixins d'exploit utilisables va grossir également.

Ce document va présenter certains des mixins existants.

Msf::Exploit::Brute

Le mixin brute force fournit une implémentation flexible qui peut être utilisée d'une manière générique par les exploits qui souhaitent supporter l'attaque par force brute.

Ce mixin implémente la méthode "exploit" et détecte si la cible actuellement sélectionnée est une cible de force brute.

Si tel est le cas, le mixin réalise tout le parcours d'adresses requis, basé sur les adresses de début et de fin spécifiées pour la cible.

Pendant chaque itération, le mixin appelle la méthode "brute exploit" avec l'état courant de l'adresse qui doit être implémenté par la classe dérivée.

Si la méthode "exploit" est appelée avec une cible qui n'est pas concernée par l'attaque de type force brute; le mixin appelle la simple méthode "exploit".

Msf::Exploit::Egghunter

Le but du mixin egghunter est d'encapsuler la génération d'un egghunter spécifique à une architecture et une plateforme comme fourni par la classe "Rex::Exploitation::Egghunter".

Cette fonctionnalité est fournie par la méthode "generate egghunter" du mixin qui prend en compte l'architecture et plateforme de la cible actuellement choisie.

Msf::Exploit::Remote::DCERPC

Le mixin DCERPC fournit des méthodes utiles aux exploits qui tentent de LEVERAGE les vulnérabilités dans des applications DCERPC.

Il fournit également une interface d'évasion unifiée qui permet à n'importe quel exploit qui utilise le mixin d'utiliser l'évasion BIND multi-contexte et la fragmentation de paquets.
Ce mixin définit automatiquement les options "RHOST" et "RPORT".
Il définit aussi deux options avancées: "DCEFragSize" et "DCEMultiBind".

Msf::Exploit::Remote::Ftp

Le mixin FTP fournit un ensemble de méthodes utiles lors de l'interaction avec un serveur FTP, comme l'identification et l'envoi de commandes.
Ce mixin inclue le mixin "Msf::Exploit::Remote::Tcp".

Ce mixin définit automatiquement les options "RHOST", "RPORT", "USER", et "PASS".

Msf::Exploit::Remote::HttpClient

Le mixin client HTTP enveloppe certaines méthodes pour créer une instance de "Rex::Proto::Http::Client" pour que les exploits dérivés puissent simplement appeler "connect" sur leur instance de module pour établir une connexion HTTP vers un serveur distant.
Ce mixin définit aussi automatiquement les options "RHOST", "RPORT", et "VHOST".

Msf::Exploit::Remote::HttpServer

Le mixin serveur HTTP enveloppe la création ou réutilisation d'un serveur HTTP local qui est utilisé dans l'exploitation de clients HTTP, comme les navigateurs.
Ce mixin inclue le mixin "Msf::Exploit::Remote::TcpServer".

Msf::Exploit::Remote::SMB

Le mixin SMB implémente des méthodes utiles lors de l'exploitation de vulnérabilités à travers le protocole SMB.
Il fournit des méthodes pour se connecter et s'authentifier à un serveur SMB, ainsi que d'autres méthodes d'aide pour opérer sur la connexion SMB une fois établie.
Ce mixin inclue le mixin "Msf::Exploit::Remote::Tcp".

Ce mixin enregistre automatiquement les options "RPORT", "SMBDirect", "SMBUSER", "SMBPASS", "SMBDOM", et "SMBNAME".
Il définit également les options avancées "SMBPipeWriteMinSize", "SMBPipeWriteMaxSize", "SMBPipeReadMinSize", et "SMBPipeReadMaxSize".

Msf::Exploit::Remote::Tcp

Le mixin TCP implémente une interface client TCP basique qui peut être utilisées d'une manière générique pour se connecter ou communiquer avec des applications qui dialoguent via TCP.
Ce mixin définit automatiquement les options "RPORT", "RHOST", et "SSL".

Msf::Exploit::Remote::TcpServer

Le mixin serveur TCP implémente un serveur TCP basique qui peut être utilisé pour exploiter des vulnérabilités dans des clients qui dialoguent via TCP.

Ce mixin définit automatiquement les options "SRVHOST" et "SRVPORT".

Msf::Exploit::Remote::Udp

Le mixin UDP implémente une interface client UDP basique qui peut être utilisées d'une manière générique pour se connecter ou communiquer avec des applications qui dialoguent via UDP.

Ce mixin définit automatiquement les options "RPORT", "RHOST", et "SSL".

Msf::Exploit::Seh

Le mixin SEH implémente quelques méthodes qui peuvent être utilisées par les exploits qui exploitent le vecteur d'exploitation de sur-écriture du SEH.

Le but de ce mixin est d'englober la génération d'éléments d'enregistrement SEH d'une manière rendant possible de prendre en compte des niveaux d'évasion supérieurs.

Ceci est accompli en utilisant la classe "Rex::Exploitation::Seh".

Ce mixin définit automatiquement l'option avancée "DynamicSehRecord".

6.3 Nop

Les modules de génération de NOP sont utilisés pour créer une chaîne d'instructions qui non pas d'effet réel, lorsqu'elles sont exécutées sur une machine, autre que d'altérer l'état des registres ou inverser les flags processeur.

Tous les modules NOP doivent hériter de la classe de base "Msf::Nop" à un certain niveau.

Les modules NOP sont assez simples comparés aux autres types de modules du framework. Il y a seulement deux méthodes utilisées par le framework pour agir avec les modules NOP.

6.3.1 generate sled

La méthode "generate sled" effectue l'action que son nom indique.

Il prend la longueur du NOP sled à générer comme premier argument et un hash de paramètres optionnels comme second argument.

Le hash contrôle certains des comportements du générateur de NOP.

La table présentée en figure 6.6 montre les éléments qui peuvent être passés par le framework pour générer le sled (traîneau).

Une fois la génération du sled terminée, le buffer du traîneau de NOP est retourné si la méthode réussie.

Elément du Hash	Type	Description
Random	Booléen	Indique que la génération de NOP aléatoire doit être utilisée.
SaveRegisters	Tableau	Un tableau des registres spécifiques à l'architecture qui ne doivent pas être touchés par les instructions générées dans le NOP sled.
BadChars	Chaîne	La chaîne de caractères interdits, s'il y en a, qui doivent être évités par le NOP sled.

Figure 6.6: Arguments optionnels du hash de "Msf::Nop generate sled"

6.3.2 nop repeat threshold

Cette méthode retourne simplement le nombre de fois par défaut pour tenter de trouver un octet NOP valide en générant le NOP sled.

La valeur par défaut est 10000.

Ceci est utilisé principalement comme référence par les modules NOP pendant la génération d'un sled.

6.4 Payload

Les modules payload fournissent au framework du code qui peut être exécuté après qu'un exploit réussisse pour prendre le contrôle du flot d'exécution.

Les payloads peuvent être des chaînes de commandes ou des instructions brutes, mais au final ils finissent en code qui sera exécuté sur la machine cible.

Pour fournir cette ensemble de fonctionnalités; le framework offre la classe de base "Msf::Payload" qui implémente des routines communes à tous les payloads, ainsi que des attributs utiles.

L'une des différences majeures entre les modules payload et les autres types de modules du framework, est qu'ils sont une composition de quelques mixins différents qui résultent en un ensemble de fonctionnalité payload complet.

Les payloads sont à la base une implémentation de la classe "Msf::Payload".

Néanmoins, ils incluent également le support nécessaire pour gérer la partie client de n'importe quelle connexion que le payload pourrait faire à travers les gestionnaires.

Les gestionnaires (handlers) seront traités plus en détails plus tard dans cette section.

Hormis les handlers, les payloads sont aussi décomposés en trois types de payload distincts: "singles", "stagers", et "stages".

Ces types de payload seront détaillés plus tard dans ce chapitre.

De plus, contrairement aux autres modules du framework, les modules payload ne vont pas nécessairement correspondre un à un avec les noms de module qui peuvent être utilisés dans le framework. Cela tient au fait que le framework va automatiquement générer des permutations entre les différents types de module, ils pourront ainsi être utilisés dans différentes combinaisons sans avoir à être liés ensemble statiquement.

Ceci est particulièrement utile pour les payloads de type "staged" (étagés) car il est possible pour les "stagers" et "stages" d'être automatiquement fusionnés ensemble au moment du chargement plutôt que d'avoir à créer statiquement une association dans les fichiers de module.

C'est une amélioration majeure vis-à-vis de la version 2.x du framework.

Pour aider à mieux visualiser la hiérarchie de payload, le diagramme de la figure 6.7 montre la hiérarchie de classe d'un type particulier de payload connu comme un payload "staged".

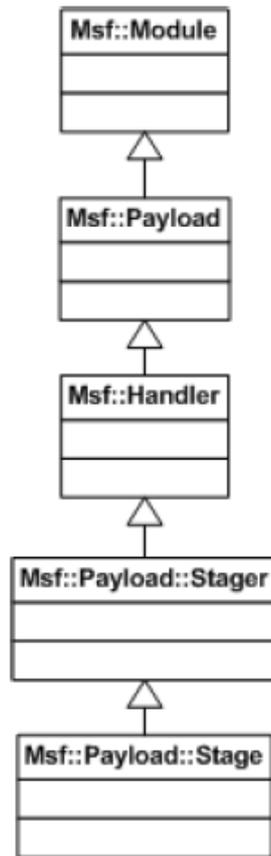


Figure 6.7: Hiérarchie de classe d'un payload staged

6.4.1 Interface

Le framework utilise une interface uniforme et bien définie pour travailler avec les modules payload. Comme les autres modules, les modules payload possèdent des éléments de hash d'information spécifique au module.

La table présentée en figure 6.8 montre les éléments spécifiques au hash d'information d'un module payload et les accesseurs qui peuvent être utilisés pour les accéder.

En utilisant les informations spécifiques au payload, le framework pilote la classe payload en employant un ensemble de méthodes spécifique.

Ces méthodes sont décrites en détails plus ci-dessous.

Elément du Hash	Accesseur	Type	Description
BadChars	badchars	String	La chaîne des caractères interdits pour ce payload, s'il y en a.
SaveRegisters	save_registers	Array	Un tableau des registres spécifiques à l'architecture qui doivent être sauvegardés quand on utilise ce payload.
Payload	module_info['Payload']	Hash	Un hash des informations spécifiques à ce payload.
Convention	convention	String	La convention d'étageage (staging) utilisée par ce payload, s'il y en a.
SymbolLookup	symbol_lookup	String	La méthode utilisée pour résoudre les symboles par le payload, s'il y en a.

Handler	handler_klass	Msf::Handler::Xxx	La classe handler qui doit être utilisée avec ce payload, ou Msf::Handler::None.
Session	session	Msf::Session::Xxx	La classe de session à créer quand le payload réussit.

Figure 6.8: Accesseurs du hash d'information de Msf::Payload

compatible convention?

Cette méthode vérifie si la convention de staging fournie est compatible avec la convention de staging du module payload courant.

Si la convention de staging du module payload courant est non définie (ce qui est le cas pour un payload non-étagé) ou que la convention coïncide; alors on retourne vrai.

Si le type du payload courant est un des étagés, et que la convention spécifiée est non définie; alors on retourne vrai également.

Dans tous les autres cas, on retourne faux.

compatible encoders

Cette méthode retourne un tableau des encodeurs compatibles où chaque élément du tableau est un tableau de 2 éléments qui contiennent le nom de référence de l'encodeur et la classe du module encodeur.

compatible nops

Cette méthode retourne un tableau des générateurs NOP compatibles où chaque élément du tableau est un tableau de 2 éléments qui contiennent le nom de référence du générateur NOP et la classe du module NOP.

connection type

Cette méthode retourne le type de connexion utilisée pour ce payload comme dérivé du gestionnaire de payload.

generate

Cette méthode entraîne le payload sous-jacent à être généré.

Cette méthode fonctionne en appelant la méthode "payload" sur l'instance du module payload et en créant une copie réplique de celle-ci.

A partir de là, n'importe quelle variable définies est substituée comme transportée via l'attribut "offsets". Le buffer substitué résultant est ensuite retourné à l'appelant.

payload type

Cette méthode retourne le type du payload qui est implémenté par la classe dérivée.

Soit: "Msf::Payload::Type::Single", "Msf::Payload::Type::Stager", ou "Msf::Payload::Type::Stage".

size

Cette méthode retourne la taille du payload comme retournée par un appel à "generate".

staged?

Cette méthode retourne true si le type du payload est soit "Stager" soit "Stage".

substitute vars

Cette méthode substitue les variables en utilisant le hash d'offsets comme guide.

Elle appelle également "replace var" avant de réaliser la substitution qui confèrent aux classes dérivées une chance de réaliser une substitution de variables personnalisée avant d'utiliser les équipements intégrés.

validate

Cette méthode englobe l'appel à la méthode "validate" du conteneur d'options du payload.

6.4.2 Types

Les payloads du framework sont divisés en trois types distincts.

Le premier type de payload qui peut être implémenté est nommé "single payload".

Les payloads simples n'ont qu'une étape et ne subissent pas un processus de staging.

Un exemple d'un "single payload" typique est un qui se connecte en arrière à un attaquant et lui fournit un shell sans aucune étape intermédiaire.

Le second type de payload est "stager".

Les étapes sont responsables de la connexion inverse à l'attaquant d'une certaine manière et procède à une deuxième étape de payload.

Le troisième type de payload est "stage" et représente ce qui est exécuté par un payload "stager".

Ces trois types de payload permettent au framework de générer dynamiquement diverses combinaisons de payloads.

Single

Comme décrit ci-dessus, les payloads simples sont autosuffisant, ce sont des payloads à une seule étape qui réalisent une tâche logique sans nécessiter un deuxième code.

Les payloads single sont les plus simples parmi les trois types.

Pour les payloads single, l'élément "Payload" du hash du hash d'informations du module va contenir un sous-hash avec quelques éléments clés.

La table présentée en figure 6.9 décrit les éléments du hash qui sont utilisés par le framework et les accesseurs utilisés pour les obtenir.

Elément du Hash	Accesseur	Type	Description
Payload	payload	String	Le payload brut associé avec ce module payload.
Offsets	offsets	Hash	Un tableau de variables qui doivent être substituées avec des offsets spécifiques basés sur le datastore du module.

Figure 6.9: Accesseurs du sous-hash d'informations Payload

Pour les payloads single, le hash Payload contient typiquement un élément de sous-hash Payload qui contient le payload brut.

Ceci est illustré ci-dessous:

```
{
'Payload' =>
{
'Payload' => "\xcc\xcc\xcc",
'Offsets' => ...
}
}
```

Stage

Un payload Stage est une implémentation d'une tâche indépendante de la connexion comme lancer un shell de commandes ou exécuter une commande arbitraire.

Les payloads stage sont combinés avec divers étageurs (stagers) du framework pour produire un ensemble de payloads orientés connexion multi-étapes.

Ceci est fait automatiquement par le framework en associant les payloads stage avec les stagers qui ont une convention de staging compatible.

La convention de staging décrit la manière dont laquelle les informations de connexion sont passées du stager à l'étage en termes de quel registre doit contenir un descripteur de fichier par exemple.

Les stages et stagers sont aussi comparés par leur convention de consultation de symbole si nécessaire, ainsi les étapes peuvent admettre que certains emplacements en mémoire contiennent des routines qui peuvent être utiles.

Les payloads stage transportent leurs contenus de payload bruts en relation avec l'élément de hash d'informations du module Stage.

Les éléments du sous-hash sont similaires aux payloads single en ce sens qu'ils ont un élément "Payload" et "Offsets".

Les payloads stage sont MEANINGLESS à moins qu'il n'y ait un stager compatible.

Stager

Un payload stager est une implémentation d'un payload qui établit un certain canal de communication avec l'attaquant pour lire dans ou autrement obtenir un second payload stage à exécuter.

Par exemple, un stager peut se connecter en arrière à un attaquant sur un port défini et lire le code à exécuter.

Les stagers transportent le contenu de leur payload brut en relation avec l'élément du hash d'information du module Stager.

Les éléments du sous-hash sont similaires aux payloads single dans le sens où ils ont un élément "Payload" et "Offsets".

De plus, les payloads étagés possèdent des méthodes d'accès supplémentaires aux payloads single. Par exemple, le payload et offsets d'un stager peuvent être obtenus via les accesseurs "payload" et "offsets".

Le payload et offsets d'une étape peuvent être obtenus via les accesseurs "payload" et "offsets".

Le code suivant montre comment ces éléments du hash seront définis:

```
{
  'Stager' =>
  {
    'Payload' => "\xcc\xcc\xcc",
    'Offsets' => ...
  },
  'Stage' =>
  {
    'Payload' => "\xcc\xcc\xcc",
    'Offsets' => ...
  }
}
```

6.4.3 Handlers

Les gestionnaires (handlers) constituent l'un des composants clé d'un payload.

Ils sont responsables de la gestion côté attaquant de l'établissement de la connexion qui doit être créée par le payload étant transmis par l'exploit.

Les différents handlers seront détaillés plus tard dans cette sous-section.

Les handlers eux-mêmes agissent comme les mixins qui sont fusionnés dans une classe de module payload.

Le framework interagit avec les handlers via une interface bien définie.

Avant d'initialiser un exploit, le framework va appeler les méthodes "setup handler" et "start handler" dans un handler de payload, qui entraîneront l'initialisation du handler en préparation pour une connexion payload.

Quand une connexion arrive, le handler appelle la méthode "handle connection" sur l'instance du payload. Cette méthode peut être surdéfinie par le payload quand nécessaire afin de réaliser des tâches personnalisées.

Par exemple, les payloads étagés vont initialiser le transfert de la seconde étape à travers la connexion établie, puis appeler l'implémentation par défaut qui entraîne la création d'une session de connexion.

Quand un exploit a terminé, le framework va appeler les méthodes "stop handler" et "cleanup handler" dans les handlers de payload pour l'arrêter d'écouter pour des connexions futures.

Bind TCP

Le gestionnaire bind TCP est fourni par la classe "Msf::Handler::BindTcp".

Il va tenter d'établir une connexion à une machine cible sur un port donné (spécifié dans LPORT).
Si une connexion est établie; un appel est réalisé à "handle connection" en passant par le socket associé avec la connexion.

Find port

Le handler find port est fourni par la classe "Msf::Handler::FindPort".

Quand un exploit appelle la méthode "handler" avec une connexion socket, le gestionnaire find port va tenter de voir si le socket a maintenant été RE-PURPOSED pour être utilisé par le payload.

Le handler find port est conçu pour être utilisé pour les payloads qui cherchent un socket en comparant les noms de port qui vont de paire avec la machine cible.

Find tag

Le gestionnaire de recherche de port (find port handler) est fourni par la classe "Msf::Handler::FindTag".

Quand un exploit appelle la méthode du handler avec une connexion socket, le gestionnaire de recherche de port va tenter de voir si le socket a été RE-PURPOSED pour utilisation par

le payload. Le gestionnaire "find tag" est à utiliser pour trouver des payloads du style socket qui recherche un socket basé sur la présence d'un tag sur le fil.

None

Si un payload n'établit pas de connexion d'aucune sorte, le gestionnaire "Msf::Handler::None" est utilisé.

Reverse TCP

Le gestionnaire "reverse TCP" est fourni par la classe "Msf::Handler::ReverseTcp".

Il va écouter sur un port pour les connexions entrantes et va faire un appel à "handle connection" quand elles entrent.

6.5 Recon

Le module de reconnaissance est toujours en cours de reconception et n'est actuellement pas documenté.

Chapitre 7

Plugins du Framework

La version 3.0 du framework offre un nouveau type de concept framework qui est celui du plugin de framework.

A contrario des modules; les plugins de framework sont conçus pour altérer ou augmenter le framework lui-même.

L'objectif est volontairement étendu pour encourager la créativité!

Tous les plugins doivent se trouver dans l'espace de nommage "Msf::Plugin" et ils doivent hériter de la classe de base Msf::Plugin.

Les plugins sont chargés dans le framework en appelant "framework.plugins.load" avec un chemin de fichier contenant le plugin.

Le framework va alors se charger de charger le plugin et de créer une instance de la classe trouvée dans le fichier spécifié, en admettant que la classe a été ajoutée à l'espace de nommage "Msf::Plugin".

Quand le framework crée une instance d'un plugin; il appelle le constructeur du plugin et lui passe l'instance de framework depuis lequel il est créé.

Il passe également un hash de paramètres arbitraires, dont certains ont une fonctionnalité prédéfinie comme décrit dans le chapitre sur le gestionnaire de plugins dans la documentation du framework core.

Alternativement, un plugin peut se voir passé des paramètres d'initialisation personnalisés via le hash options.

Pour comprendre le type de choses dont un plugin framework est capable; différents exemples théoriques seront énumérés dans ce chapitre.

Premier exemple; un plugin qui ajoute simplement une nouvelle commande à l'interface console.

Le plugin d'exemple inclus dans la distribution par défaut du framework illustre comment cela peut être réalisé.

Un plugin plus avancé pourrait automatiser certaines actions quand une session Meterpreter est créée, comme automatiser le téléchargement des hashés de mots de passe et les passer à un casseur de mots de passe.

Un autre exemple serait d'introduire un nouveau type de module dans le framework. Cela serait accompli en étendant l'instance du framework existante pour supporter des accesseurs pour gérer le nouveau type de module.

Chapitre 8

Sessions de Framework

L'objectif final pour un exploit est de fournir à l'attaquant une sorte de session qui lui permet d'exécuter des commandes et réaliser d'autres actions sur une machine cible.

Dans la plupart des cas, cette session est un interpréteur de commande classique dont les entrées/sorties sont véhiculées via une connexion socket.

Néanmoins, un shell de commande n'est pas particulièrement automatisable sans qu'il soit encapsulé dans une classe qui permet un accès au shell au niveau d'un script de commandes.

C'est pourquoi la version 3.0 du framework intègre des classes de session généralisées qui peuvent être utilisées par le framework, les plugins et des scripts externes pour automatiser le processus de contrôle d'une session créée après une exploitation réussie.

Pour fournir un niveau extensible de contrôle d'automatisation, les sessions framework peuvent implémenter un ou plusieurs des mixins fournisseurs d'accès contenus dans l'espace de nommage "Msf::Session::Provider".

La distribution actuelle du framework fournit 4 interfaces basiques qui peuvent être implémentées par des sessions spécifiques.

1. MultiCommandExecution

Cette interface fournit des méthodes qui peuvent être utilisées pour exécuter des commandes multiples simultanément sur la machine cible.

Cette interface est un sur-ensemble de l'interface "SingleCommandExecution".

2. MultiCommandShell

Cette interface fournit des méthodes pour exécuter plusieurs shells de commandes simultanément sur la machine cible.

Cette interface est un sur-ensemble de l'interface "SingleCommandShell".

3. SingleCommandExecution

Cette interface fournit des méthodes pour exécuter une seule commande sur la machine cible.

4. SingleCommandShell

Cette interface fournit des méthodes pour exécuter un unique shell de commande sur la machine cible.

En implémentant une ou plusieurs de ces méthodes; les sessions peuvent être programmiquement rendues automatisables au niveau le plus basique.

Hormis depuis les interfaces standards, les sessions peuvent également optionnellement implémenter le mixin "Msf::Session::Comm" qui est supposé être utilisé pour canaliser le trafic réseau via une machine distante.

Les sessions qui implémentent le mixin Msf::Session::Comm peuvent être utilisées en conjonction avec le SWITCH BOARD ROUTING TABLE présent dans la librairie Rex.

A l'heure où nous écrivons ces lignes, il existe deux implémentations de session basiques qui se trouvent dans la librairie de base du framework.

Ces deux sessions sont décrites dans les sections suivantes.

8.1 Shell de Commandes

La session shell de commandes fournie par "Msf::Sessions::CommandShell " implémente l'interface "Msf::Session::Provider::SingleCommandShell".

Les méthodes utilisées pour interagir avec le shell sont simplement tunnelées à travers le flux associé avec le côté distant de la connexion.

Chaque payload qui rend un shell de commandes doit retourner une instance de cette session.

8.2 Meterpreter

La session Meterpreter fournie par "Msf::Sessions::Meterpreter" implémente l'interface "Msf::Session::Comm" et est également capable d'implémenter certaines des autres interfaces automatisées.

En implémentant l'interface Comm, toutes les sessions Meterpreter peuvent être utilisées pour diriger le trafic réseau.

Chapitre 9

Méthodologies

Une des choses primordiales à comprendre avant de commencer à écrire un module pour le framework est: les méthodologies qui doivent être appliquées.

Le but de la version 3.0 du framework est de rendre les modules plus faciles à implémenter et en même temps de les rendre plus robustes.

Avec cet objectif en tête, tous les programmeurs souhaitant écrire des modules pour le framework devraient prêter attention à ce chapitre.

En premier lieu, les modules doivent être simples.

Dans le cas où un module devient compliqué ou grand; il serait bon de regarder plus prêt si une partie du code pourrait être mieux généralisée dans un mixin qui pourrait plus tard être partagé avec d'autres modules.

C'est particulièrement vrai dans le cas où un exploit manipule un protocole qui pourrait être utile plus tard à d'autres exploits.

Cela peut également être le cas quand un exploit tente de déclencher une vulnérabilité qui a une approche générale qui peut être appliquée à d'autres modules d'exploit.

Deuxièmement, les modules doivent être propres.

Un des facteurs clé lorsque l'on fait n'importe quel type de développement est de s'assurer de l'uniformité de la conception et de l'implémentation.

Cela ne s'applique pas seulement aux nommages mais aussi aux choses comme l'indentation.

Si un module possède une indentation et/ou des règles de nommages inconsistantes; sa lisibilité sera énormément réduite.

Chaque programmeur a droit à son style de programmation, mais il s'assurer le conserver tout au long du développement d'une unité donnée.

Finalement, l'encapsulation est reine.

Si un module a besoin de réaliser une action qui pourra peut-être être modifiée en un algorithme différent, plus tard, encapsuler l'opération dans une interface généralisée est une bonne méthode pour s'assurer que le code n'aura pas à être réécrit ou ne sera pas altéré dans le futur.

Appendice A

Exemples

Ce chapitre contient différents exemples qui illustrent comment on peut interagir avec le framework et les autres bibliothèques pour réaliser différentes tâches.

Le code source de ces exemples se trouve dans le répertoire "documentation" incluse avec toutes les versions de la version 3.0 du framework.

A.1 Framework

Cette section contient des exemples spécifiques à l'interaction avec le framework lui-même.

A.1.1 Récupération d'infos sur un module

Cet exemple montre comment des informations sur un module peuvent être facilement sérialisées en un format lisible.

```
#!/usr/bin/ruby
$.unshift(File.join(File.dirname(__FILE__), '..', '..', '..', 'lib'))
require 'msf/base'
if (ARGV.empty?)
  puts "Utilisation: #{File.basename(__FILE__)} module_name"
  exit
end

framework = Msf::Simple::Framework.create

begin

  # Création de l'instance du module.
  mod = framework.modules.create(ARGV.shift)

  # Capture les informations sur le module dans un format texte lisible.
  puts Msf::Serializer::ReadableText.dump_module(mod)
rescue
  puts "Erreur: #{$!}\n\n#{@.join("\n")}"
end
```

A.1.2 Encoder le contenu d'un fichier

Cet exemple montre comment encoder un fichier en utilisant un encodeur framework.

```
#!/usr/bin/ruby
$.unshift(File.join(File.dirname(__FILE__), '..', '..', '..', 'lib'))
require 'msf/base'
if (ARGV.empty?)
  puts "Utilisation: #{File.basename(__FILE__)} encoder_name file_name format"
```

```

    exit
end
framework = Msf::Simple::Framework.create
begin

    # Création de l'instance de l'encodeur.
    mod = framework.encoders.create(ARGV.shift)
    puts(Msf::Simple::Buffer.transform(
        mod.encode(IO.readlines(ARGV.shift).join), ARGV.shift || 'ruby'))
rescue
    puts "Erreur: #{ $! }\n\n#{ $@.join("\n") }"
end

```

A.1.3 Enumérer les modules

Cet exemple montre comment énumérer tous les modules du framework et afficher leurs types et noms de référence.

```

#!/usr/bin/ruby
$.unshift(File.join(File.dirname(__FILE__), '..', '..', '..', 'lib'))
require 'msf/base'
framework = Msf::Simple::Framework.create

# Enumérer tous les modules du framework.
framework.modules.each_module { |name, mod| puts "#{mod.type}: #{name}"
}

```

A.1.4 Exécution d'un exploit en utilisant la framework base

Cet exemple montre comment utiliser le framework core directement pour lancer un exploit. Il utilise la méthode d'emballage (wrapper) d'exploit simplifié fournie par le mixin "Msf::Simple::Exploit".

```

#!/usr/bin/ruby
$.unshift(File.join(File.dirname(__FILE__), '..', '..', '..', 'lib'))
require 'msf/base'
if (ARGV.length == 0)
    puts "Utilisation: #{File.basename(__FILE__)} exploit_name payload_name OPTIONS"
    exit
end
framework = Msf::Simple::Framework.create
exploit_name = ARGV.shift || 'test/multi/aggressive'
payload_name = ARGV.shift || 'windows/meterpreter/reverse_tcp'
input = Rex::Ui::Text::Input::Stdio.new
output = Rex::Ui::Text::Output::Stdio.new
begin

# Initialise l'instance d'exploit
exploit = framework.exploits.create(exploit_name)

```

```

# Allume le feu.
session = exploit.exploit_simple(
  'Payload' => payload_name,
  'OptionStr' => ARGV.join(' '),
  'LocalInput' => input,
  'LocalOutput' => output)

# Si une session est ouverte; essaie d'interagir avec elle.
if (session)
  output.print_status("Session #{session.sid} créée; interaction...")
  output.print_line
  session.init_ui(input, output)
  session.interact
else
  output.print_line("Exploit complété, pas de session créée.")
end

rescue
  output.print_error("Erreur: #{!}\n\n#{@$@.join("\n")}")
end

```

A.1.5 Exécution d'un exploit en utilisant le framework core

Cet exemple montre comment utiliser le framework core directement pour lancer un exploit. Il utilise la classe Framework de base du framework, ainsi le chemin du module est défini automatiquement, mais s'appuie exclusivement sur les classes du framework core pour tout le reste.

```

#!/usr/bin/ruby
$.unshift(File.join(File.dirname(__FILE__), '..', '..', '..', 'lib'))
require 'msf/base'
if (ARGV.length == 0)
  puts "Usage: #{File.basename(__FILE__)} exploit_name payload_name OPTIONS"
  exit
end
framework = Msf::Simple::Framework.create
exploit_name = ARGV.shift || 'test/multi/aggressive'
payload_name = ARGV.shift || 'windows/meterpreter/reverse_tcp'
input = Rex::Ui::Text::Input::Stdio.new
output = Rex::Ui::Text::Output::Stdio.new
begin

# Création de l'instance du driver d'exploit.
driver = Msf::ExploitDriver.new(framework)

# Initialise les instances du driver et payload d'exploit
driver.exploit = framework.exploits.create(exploit_name)
driver.payload = framework.payloads.create(payload_name)

# Importe les options spécifiées au format VAR=VAL depuis la ligne de commande spécifiée
driver.exploit.datastore.import_options_from_s(ARGV.join(' '))

```

```

# Partage le datastore de l'exploit avec le payload.
driver.payload.share_datastore(driver.exploit.datastore)

# Initialise l'index de cible avec ce qu'il y a dans le datastore de l'exploit ou avec 0 par défaut
driver.target_idx = (driver.exploit.datastore['TARGET'] || 0).to_i

# Initialise les interfaces utilisateur exploit et payload.
driver.exploit.init_ui(input, output)
driver.payload.init_ui(input, output)

# Allume le feu.
session = driver.run

# Si une session est ouverte; essaie d'interagir avec elle.
if (session)
  output.print_status("Session #{session.sid} créée, interaction...")
  output.print_line
  session.init_ui(input, output)
  session.interact
else
  output.print_line("Exploit complété, pas de session créée.")
end

rescue
  output.print_error("Erreur: #{$!}\n\n#{@.join("\n")}")
end

```

A.2 Module de Framework

Cette section présente quelques exemples de modules framework.

A.2.1 Encodeur

Cet exemple présente un encodeur très basique qui retourne simplement le bloc qui lui est passé.

```

module Msf
  module Encoders
    class Sample < Msf::Encoder
      def initialize
        super(
          'Name' => 'Encodeur exemple',
          'Version' => '$Revision: 1.30 $',
          'Description' => %q{
            Encodeur exemple qui retourne seulement le bloc qu'on lui passe.
          },
          'Author' => 'skape',
          'Arch' => ARCH_ALL)
      end
      #
      # Retourne le buffer non modifié à l'appelant.
      #
      def encode_block(state, buf)

```

```

        buf
    end
end
end
end

```

A.2.2 Exploit

Cet exemple d'exploit montre comment un module d'exploit peut être écrit pour exploiter un bogue dans un serveur TCP arbitraire.

```

module Msf
class Exploits::Sample < Msf::Exploit::Remote
#
# Cet exploit affecte les serveurs TCP, on utilise donc le mixin TCP client.
#
include Exploit::Remote::Tcp
def initialize(info = {})
super(update_info(info,
'Name' => 'Exploit exemple',
'Description' => %q{
    Cet exemple illustre l'exploitation d'une vulnérabilité dans un serveur TCP.
},
'Author' => 'skape',
'Version' => '$Revision: 1.30 $',
'Payload' =>
{
    'Space' => 1000,
    'BadChars' => "\x00",
},
'Targets' =>
[
    # Target 0: Windows All
    [
        'Windows Universal',
        {
            'Platform' => 'win',
            'Ret' => 0x41424344
        }
    ],
],
'DefaultTarget' => 0))
end
#
# L'exploit exemple indique seulement que l'hôte distant est toujours vulnérable.
#
def check
return Exploit::CheckCode::Vulnerable
end
#
# La méthode exploit se connecte au service distant et envoie 1024 fois le caractère A suivi par la fausse
adresse de retour et enfin le payload.

```

```

#
def exploit
  connect
  print_status("Envoi du payload de #{payload.encoded.length} octets...")
  # Construction du buffer pour la transmission
  buf = "A" * 1024
  buf += [ target.ret ].pack('V')
  71

  buf += payload.encoded
  # Envoi
  sock.put(buf)
  sock.get
  handler
end
end
end

```

A.2.3 Nop

Cette classe implémente un générateur de NOP sled très basique qui retourne juste une chaîne de 0x90 de la taille spécifiée.

```

module Msf
  module Nops
    class Sample < Msf::Nop
      def initialize
        super(
          'Name' => 'Générateur NOP exemple',
          'Version' => '$Revision: 1.30 $',
          'Description' => 'Générateur Exemple de NOP avec un simple octet',
          'Author' => 'skape',
          'Arch' => ARCH_X86)
      end
      #
      # Retourne une chaîne de 0x90 de la longueur spécifiée.
      #
      def generate_sled(length, opts)
        "\x90" * length
      end
    end
  end
end
end
end

```

A.2.4 Payload

Ce payload exemple est conçu pour déclencher une exception de débogueur via int3.

```

module Msf
  module Payloads

```

```

module Singles
module Exemple
  include Msf::Payload::Single
  def initialize(info = {})
    super(update_info(info,
      'Name' => 'Debugger Trap',
      'Version' => '$Revision: 1.30 $',
      'Description' => 'Déclenche une exception de débogueur via int3',
      'Author' => 'skape',
      'Platform' => 'win',
      'Arch' => ARCH_X86,
      'Payload' =>
      {
        'Payload' => "\xcc"
      }
    ))
  end
end
end
end
end
end

```

A.2.5 Recon

Les modules de reconnaissance sont en cours de reconception : pas d'exemples disponibles actuellement.

A.3 Plugin de Framework

A.3.1 Plugin pour l'interface utilisateur console

Cette classe présente un plugin exemple. Les plugins peuvent changer le comportement du framework en ajoutant de nouvelles fonctionnalités, de nouvelles commandes aux interfaces utilisateur, etc. Ils sont conçus pour avoir une définition très LOOSE afin de les rendre aussi utile que possible.

```

module Msf
class Plugin::Sample < Msf::Plugin
  ###
  #
  # Cette classe implémente un exemple de dispatcher de console de commandes.
  #
  ###
  class ConsoleCommandDispatcher
    include Msf::Ui::Console::CommandDispatcher
    #

    # Le nom du dispatcher.
    #
    def name

```

```

        "Exemple"
    end

    #
    # Retourne le hash des commandes supportées par ce dispatcher.
    #
    def commands
        {
            "exemple" => "Une commande exemple ajoutée par le plugin Exemple"
        }
    end

    #
    # Cette méthode gère la commande exemple.
    #
    def cmd_sample(*args)
        print_line("Tu as passé cet argument: #{args.join(' ')}")
    end
end

#
# Le constructeur est appelé quand une instance du plugin est créée. L'instance du framework à
# laquelle le plugin est associé est passée dans le paramètre framework
# Les plugins doivent appeler le constructeur parent en héritant de Msf::Plugin pour s'assurer que
# l'attribut framework de leur instance est paramétré
#
def initialize(framework, opts)
    super
    # Si ce plugin est chargé dans le contexte d'une application console qui utilise le driver de
    # l'interface utilisateur console du framework:
    # enregistre les commandes du dispatcher de console
    add_console_dispatcher(ConsoleCommandDispatcher)

    print_status("Plugin Exemple chargé.")
end

#
# La routine cleanup pour les plugins leur donne une chance d'annuler n'importequelle action qu'ils
# peuvent avoir fait sur le framework.
# Par exemple, si un dispatcher de console a été ajouté, alors il peut être retiré dans la routine
# cleanup.
#
def cleanup
    # Si on a enregistré précédemment un dispatcher de console avec la console: on le
    # désenregistre maintenant.
    remove_console_dispatcher('Exemple')
end

#
# Cette méthode renvoie un nom court et explicite pour le plugin.
#
def name
    "exemple"
end

```

```
end
```

```
#
```

```
# Cette méthode renvoie une brève description du plugin. Elle ne devrait pas dépasser  
# 60 caractères.
```

```
#
```

```
def desc
```

```
    "Illustre l'utilisation de plugins du framework"
```

```
end
```

```
end
```

```
end
```