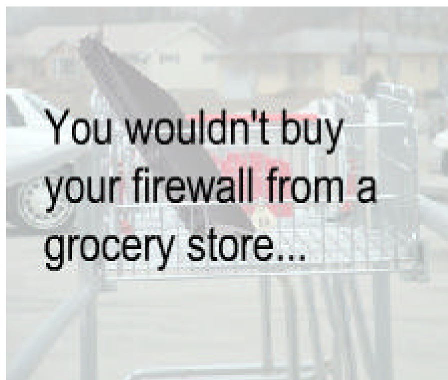
Metasploit Framework (Part One of Three)
The Prometheus Of Exploitation

by Pukhraj Singh *and* K.K. Mookhey
last updated July 12, 2004



sploit
(n.) Exploit. A defect in the game code (see bug) or design that can be used to gain unfair advantages. *(Source: Dictionary of MMORPG Terms)*

At present the exploit development community (hackers and security professionals alike) is more sentient than ever before. The timeline between the release of an advisory and the development of an exploit has shrunk to a great extent. Exploit development, which was considered more of Wiccan art, has reached large masses. The network security administrator needs to be more vigilant then ever before as the enemy is always one step ahead with the latest exploit in his hand.

Exploit development tools and automated frameworks for exploit testing and simulation is the need of the hour. Metasploit Framework (MSF) is something which fits the bill. Its latest release has the agility and muscle quite comparable to its high priced commercial counterparts and the lethality to code an exploit in the shortest possible timeframe, due to a very well defined interface for development. With a complete exploit environment, working exploits, effectual payloads and added handlers, this is one tool which the penetration testers must utilize.

This article provides an insight into the basics of exploit development frameworks, with a special focus on the Metasploit Framework and how it can be exploited to save time and resources. We describe its usage with graphical illustrations, detail the various commands available, describe features, give practical examples, and most importantly, use these skills to develop new exploits and test out new techniques. The article concludes with elucidating why MSF will influence the future of exploitation in a momentous and positive way.

1. Prologue

I would like to begin my article with reference to some relatively current happenings. The Microsoft advisory (MS04-011) discussed and fixed lot of critical security vulnerabilities in various Windows operating systems [ref 1]. Two of which that interested me were the SSL PCT and Local Security Authority overflows which could lead to remote compromise. As almost immediately, working exploits were released to the public, catching administrators and security professionals unaware and unprepared.

Putting yourself in the mindset of a security administrator in a typical IT company, the exploits added ever more to the existing security burden. Already said and done, this is a wild goose chase where the malicious attacker is ahead of the game, but with a methodical approach the security professional can turn the cards.

Security patches, IDS, firewalls, and so on should not be the only criteria of safety. Succumbing to the pressure of the situation, many nooks and corners of the network can go unprotected and unlatched, which generally become the source of compromise. This is what happens when we hear news about a big network being compromised by hackers using known vulnerabilities. And that's exactly the reason why the Sasser worm hit 250,000 computers, even two weeks after MS released the high-profile security patch.

In my opinion, the solution is the usual, "think like an attacker" approach. The penetration tester should go on a hacking spree by testing his own network, in what I call *Threat Evasion Penetration Testing*. This is where exploit frameworks come into play, which automate the art of exploitation.

2. Groundwork

Exploits still have a feeling of awe attached to them. Busy pen-testers shrug the idea of exploit development as an idle past time of a hacker. This is in a way true. Exploit development in itself is an art. It requires paramount knowledge, patience, precious time and above all the undying spirit of learning by trial-and-error.

2.1 Memory organization

The basic exploitation techniques can be methodically categorized, like any other technical issue. Before going further, however, the reader must be aware of the basic process of memory organization [ref 2]. A process running in memory has the following sub-structures:

Code is the read-only segment that contains the compiled executable code of the program.

Data and ***BSS*** are writable segments containing the static, global, initialized and un-initialized data segments and variables.

Stack is a data structure based on Last-In-First-Out ordering. Items are *push*ed and *pop*ped from the top of the stack. A Stack Pointer (*SP*) is a register which points to the top of the stack (in most cases). When data is pushed on the stack, SP points to (the top of the stack). Stack grows towards negative memory addresses. It is used for storing the context of a process. A process pushes all its local and dynamic data on to the stack. Instruction Pointer (*IP*) is a register used to point to the address of the next instruction to be executed. The processor looks at IP each time to find the next instruction to be executed. When an abrupt program redirection takes place (generally due to *jmp* or *call*) the address of the next instruction, after returning back from redirection, can be lost. In order to overcome this problem the program stores the address of the next instruction to be executed (after returning from jmp or call) on the stack, and it is called the return address (implemented through assembly instruction *RET*). This is how a normal program containing many function calls and *goto* instructions keeps track of right path of execution.

Heap is basically the rest of the memory space assigned to the process. It stores data which have a lifetime in between the global variables and local variables. The allocator and deallocator work to assign space to dynamic data and free heap memory respectively [ref 3].

This was a brief fly-over on the basics of process organization. Now I describe some techniques recurrently used to abuse the harmony of process organization.

2.2 Buffer overflows

The word gives goose bumps to any person who has dealt with them, be it a coder, an application tester or the security administrator. Some say it's the biggest security risk of the decade [ref 4]. The technique of exploitation is straightforward and lethal. The stack of the program stores the data in order whereby the parameters passed to the function are stored first, then the return address, then the previous stack pointer and subsequently the local variables. If variables (like arrays) are passed without boundary checks, they can be overflowed by shoving in large amounts of data, which corrupts the stack, leading to the overwrite of the return address and consequently a segmentation fault. If the trick is craftily done we can modify the buffers to point to any location, leading to capricious code execution [ref 5].

2.3 Heap overflows

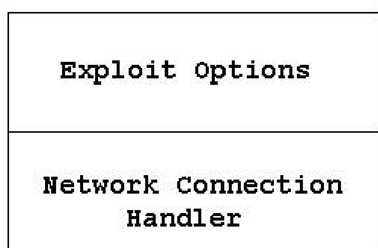
The allocated memory in a heap is organized as a doubly linked list. By performing an overflow we can modify the pointers of the linked list to point into memory. Heap overflows are hard to exploit and are more common in Windows as they contain more prominent data which can be exploited. In the case of a malloc memory allocation system, the information regarding the free and allocated memory is stored within the heap. An overflow can be triggered by exploiting this management information such that we can write to random memory areas afterwards, which can lead to code execution [ref 6].

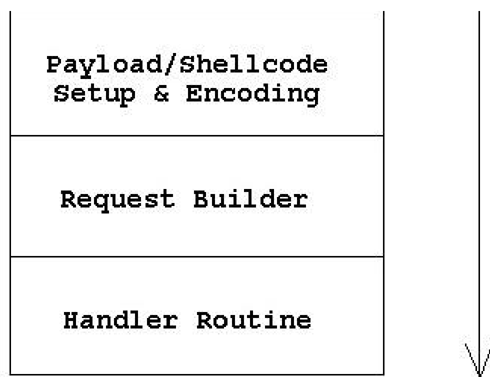
So how is the overflow triggered? There are many weapons in the stockpile like strings and string functions, format strings, null pointers, integer overflows, signed issues and race conditions which can be a help to generate exceptional conditions in a process [ref 7].

I stress the fact that this article was not meant to be a definitive guide to various exploitation techniques. We only provide a quick overview of what is important, in order to get a solid understanding of the things to come in subsequent parts of this article. They just act as pointers for further reference.

3. The Birth of an Exploit

Now comes the most exciting part, coding the exploit. The body or structure of an exploit can be divided into various components, as described in **Figure 1** [ref 8]. We describe some exploit miscellany which will help us to analyze the figure as shown.





Stages Of Exploit Development

Figure 1

3.1 Shellcode

This is the payload which is to be executed after exploitation. In most cases we redirect the path of execution so that the injected payload is executed [ref 9]. Hence the return address is made to point to this shellcode. It comprises of assembly instructions encoded as a binary string which perform operations like spawning a shell. A good piece of shellcode must be a trade-off between and size and complexity. There are a lot of verifications to be made during payload encoding like keeping a check on restricted characters. Nowadays, payloads have been customized to be very short and require less space. They can execute many complex operations from opening a listening socket to even loading a compiler on the remote computer.

3.2 Injection vector

The pointer or offset where the shellcode is placed in a process and the return address is modified to point to.

3.3 Request builder

This is the code which triggers the exploit. If it's related to string functions, then scripting languages are generally preferred.

3.4 Handler Routine

This part generally consumes the majority of the code. This is a handler for the shellcode doing operations like linking to a bindshell, or connecting console to a socket.

3.5 User options handler

It is basically a user level front-end providing the user with various control options like remote target selection, offset selection, verbosity, debugging and other options. This forms majority of the exploit code and makes the code quite bulky.

3.6 Network connection Handler

This comprises of the various routines which handle network connections like name resolution, socket establishment, error handling etc.

As we can see there is a lot of unnecessary and repetitive code which makes the exploit really bulky and error prone.

4. Some Common Problems

In the course of development, many problems are faced which hinder the exploit development process. The mad race of people trying to release the exploit first leads to a lot of bad and unnecessarily complicated code.

Some exploits need an understanding of deeper concepts and research, such as exploits based on network protocols (RPC, SMB and SSL) and obfuscated APIs. Also, not much information is revealed in the advisory so there is always the need for experimentation.

Finding target values is also one big headache which involves lot of trial and error.

Finally, most payloads are hard coded and any changes breaks the exploit.

Many firewalls and IPSes detect and block shellcode.

Time is of primary concern, and some exploits consume quite a lot of time and concentration, both of which are the precious assets of a security researcher.

All said and done, coding exploit is one hell of a messy job!

All said and done, coding exploit is one hell of a messy job!

5. Here It Comes!

Enter the Metasploit Framework (MSF)! According to the *MSF User Crash Course* [ref 10] guide,:

> *"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."*

In my words, the Metasploit Framework is a singular solution to all the above discussed problems. The framework has matured itself to quite an extent in the 2.0 release version. It's more stable, has very attractive features and a very instinctive user interface for exploit development.

The major features which give an edge to MSF over other options are:

- It is primarily written in Perl (with some parts in assembly, Python and C), which means clean efficient code and rapid plug-in development.
- Pre-packaged support for extensible tools, libraries and features like debugging, encoding, logging, timeouts and random nops and SSL.
- An intelligible, intuitive, modular and extensible exploit API and environment.
- Highly optimized muti-platform, mutli-featured payloads which are dynamically loadable.
- Enhanced handler and callback support, which really shortens the exploit code.
- Support for various networking options and protocols which can be used to develop protocol dependent code.
- Supplementary exploits are included, which help us to test out exploitation techniques and sample exploits developed.
- It is Open Source Software and has a dedicated developer community for support.
- Support for advanced features and third party tools like InlineEgg, Impurity, UploadExec and chainable proxies.

It's clear that MSF is definitely a tool the penetration-tester must get acquaintained with. It gives the art of exploitation a whole new paradigm.

6. Installation

Currently, the Metasploit Framework works efficiently on Linux and Windows. There are some minor compatibility issues, but they can be uncared for. Users can download the latest release for Windows and Linux from http://www.metasploit.com/projects/Framework/downloads.html.

The installation is very trivial and intuitive, and the download packages are in extract and run state. The simple Windows installation is shows in **Figure 2**. In the case of Linux, decompress the archive (which is in the format framework-2.x.x.tar.gz), where the framework directory contains compiled binaries which are for various utilities. While running on Linux it is advised that Term::ReadLine::Gnu (for tab completion support) and Net::SSLeay (for SSL support) modules be installed (these are found in the extras directory).

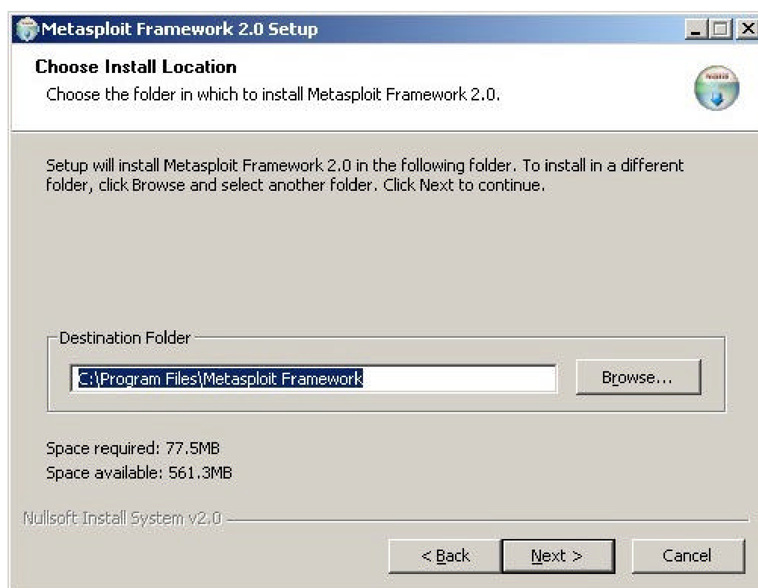


Figure 2

The Windows environment is based on a stripped down Cygwin environment, which is a wise solution as it provides a very handy console to the user. However, there were some problems

with support of Active State Perl, hence it supports Cygwin Perl only. The installation is packaged as an executable setup, which installs the Metasploit Framework in the specified directory (see figure 2) and adds shortcuts to it.

7. Concluding Part One

In the first part of this article, we did a walk-through of the various exploitation techniques which are frequently being used. We delve into the basics of exploit development and how the exploit code can be broken down into logical sub-structures. We became aware of the major problems plaguing the process of exploit development and how the Metasploit Framework acts as a solution to these problems. We also discussed some of its features and the installation procedure.

In part two we will provide an insight into the various usage and command options, running and adding new exploits, changing environment settings and other advanced features of the Metasploit Framework.

References

1. Microsoft Security Bulletin
http://www.microsoft.com/technet/security/bulletin/ms04-011.mspx

2. Stack, Pointers and Memory, Lally Singh http://www.biglal.net/Memory.html

3. A Memory Allocator, Doug Lea http://gee.cs.oswego.edu/dl/html/malloc.html

4. Buffer overflows likely to be around for another decade, Edward Hurley
http://searchsecurity.techtarget.com/originalContent/0,289142,sid14_gci860185,00.html

5. Buffer Overflows Demystified, Murat http://www.enderunix.org/docs/eng/bof-eng.txt

6. Badc0ded - How to exploit program vulnerabilities,
http://community.core-sdi.com/~juliano/bufo.html

7. Once upon a free(), Phrack 57 Article 9 by Anonymous
http://www.phrack.org/show.php?p=57

8. Presentation on Advanced Exploit Development at HITB, HD Moore (PDF)
http://conference.hackinthebox.org/materials/hd_moore/HDMOORE-SLIDES.pdf

9. Designing Shellcode Demystified, Murat http://www.enderunix.org/docs/en/sc-en.txt

10. Crash Course User Guide for Metasploit Framework, (PDF)
http://metasploit.com/projects/Framework/docs/CrashCourse-2.0.pdf

About the authors

Pukhraj Singh is a security researcher at Network Intelligence (I) Pvt. Ltd. His areas of interest include working with exploits, monitoring honeypots, intrusion analysis and penetration testing.

K. K. Mookhey is the CTO and Founder of Network Intelligence.