A Friendly Intro to Cobalt Strike's UDRL

11th Sept 2025

Obligatory `whoami`

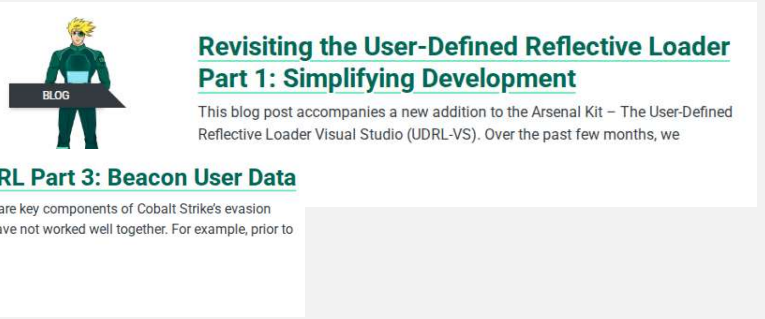
WHOAMI

- Hugo /@rwxstoned
- Adversary Emulation
- Some Blue Team background too

Why this talk ?

Why this talk

There are pieces of knowledge, but no glue.




```
SECTION( B ) NTSTATUS resolveLoaderFunctions( PAPI pApi )
{
    PPEB    Peb;
    HANDLE  hNtdll;

    Peb = NtCurrentTeb()->ProcessEnvironmentBlock;
    hNtdll = FindModule( H_LIB_NTDLL, Peb, NULL );

    if( !hNtdll )
    {
        return -1;
    };
```

https://github.com/kyleavery/AceLdr/blob/main/src/ace.c

Agenda

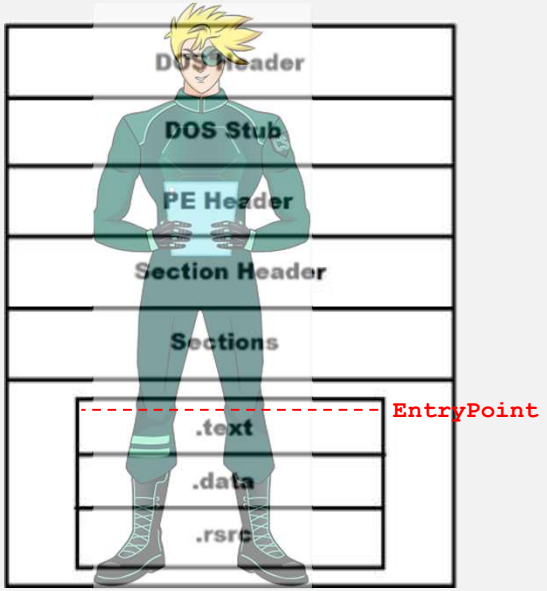
- What’s a Cobalt Strike Beacon ?
- UDRL: why ?
- Getting started with UDRL-VS
- Demo

What’s a Cobalt Strike Beacon exactly ?

Cobalt Strike Beacon

What's a PE File

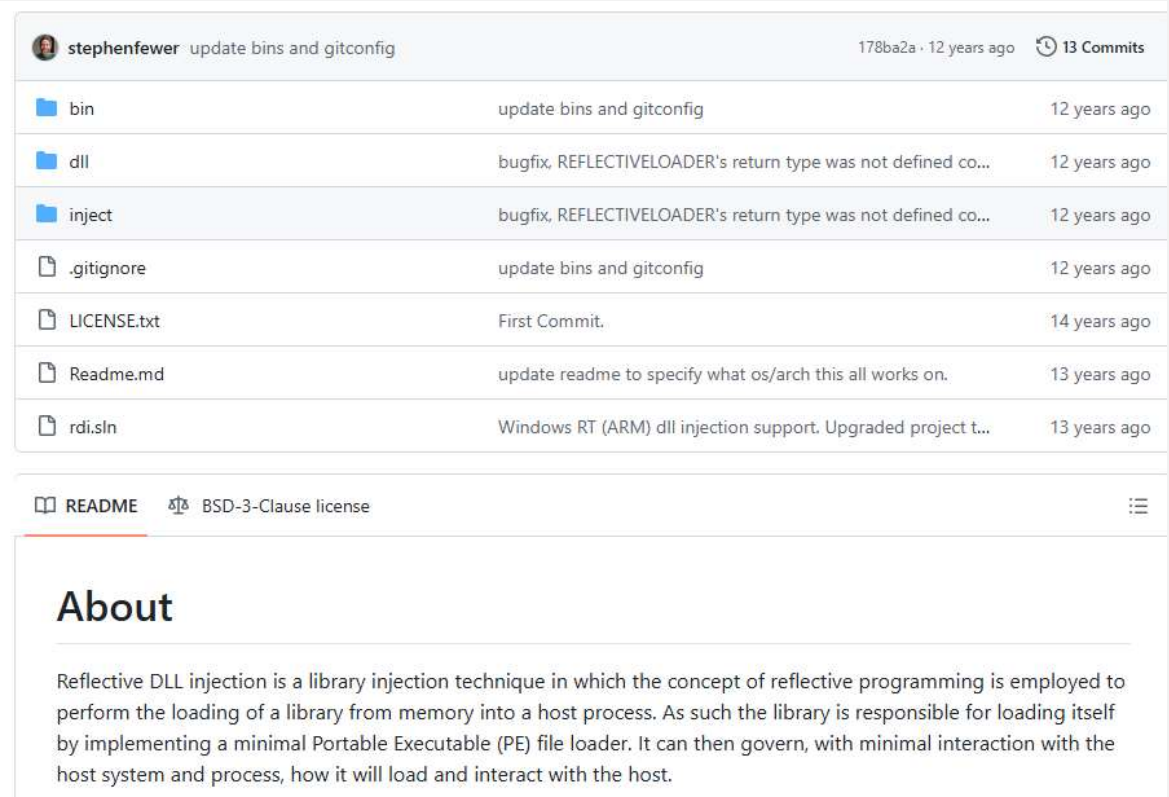
- Back to basics: a Cobalt Strike implant is a DLL.
- A PE file on disk is different to a PE in memory.
- Reading a PE from disk, and "mapping" it is done by Windows (`LoadLibrary()` for instance)
- This involves a lot of parsing, copying sections in memory, marking them properly (RX, RW, etc…), ultimately executing the `EntryPoint`.



Cobalt Strike Beacon

Reflective Loading

- Loading a PE file was intended to be done by the OS.
- Stephen Fewer showed how to do it manually, still widely used in malware 15 years later.



stephenfewer update bins and gitconfig — 178ba2a · 12 years ago — 13 Commits

bin	update bins and gitconfig	12 years ago
dll	bugfix, REFLECTIVELOADER's return type was not defined co...	12 years ago
inject	bugfix, REFLECTIVELOADER's return type was not defined co...	12 years ago
.gitignore	update bins and gitconfig	12 years ago
LICENSE.txt	First Commit.	14 years ago
Readme.md	update readme to specify what os/arch this all works on.	13 years ago
rdi.sln	Windows RT (ARM) dll injection support. Upgraded project t...	13 years ago

README BSD-3-Clause license

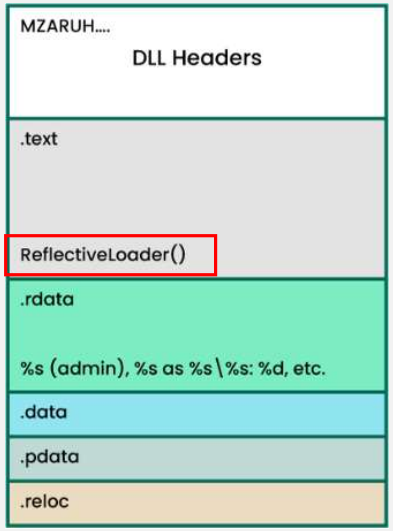
About

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. 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.

Cobalt Strike Beacon

Reflective Loading

- A Reflective Loader is a piece of code embedded within the PE file, capable of autonomously loading it in-memory.
- Historically:

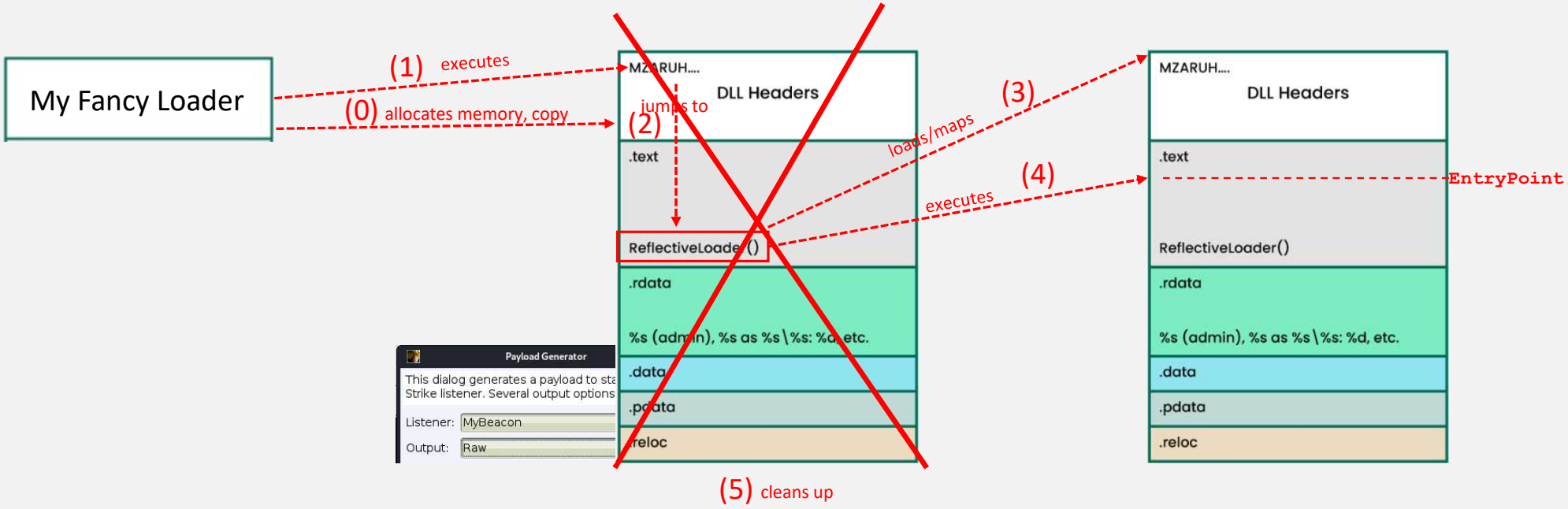


- Alternative:

Cobalt Strike Beacon

Reflective Loading

So what happens when your “loader” runs a Cobalt Strike “raw beacon” ?

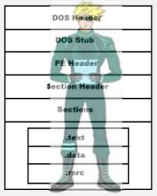


Cobalt Strike Beacon

Reflective Loading

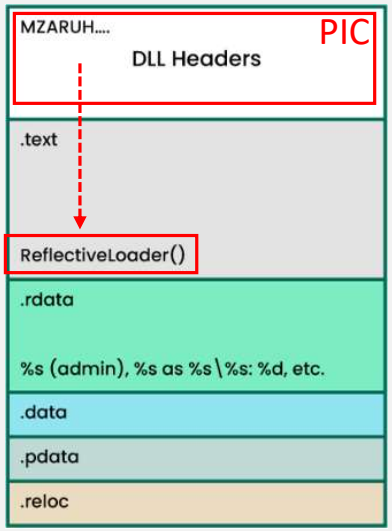
In summary, you are loading a (Reflective)loader.

The equation:

`MyFancyLoader ( Loader (`  `) )`

Cobalt Strike Beacon

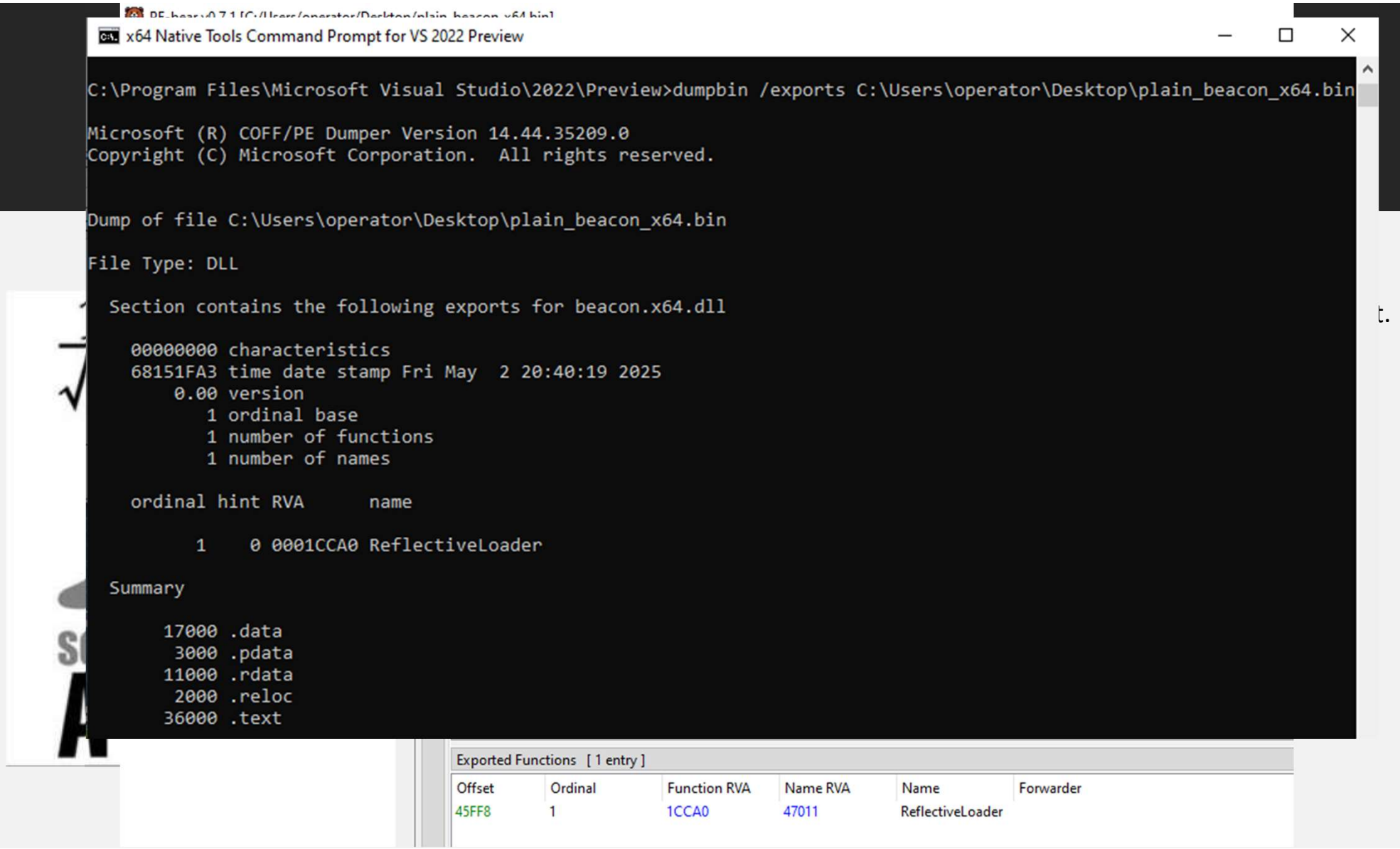
Reflective Loading

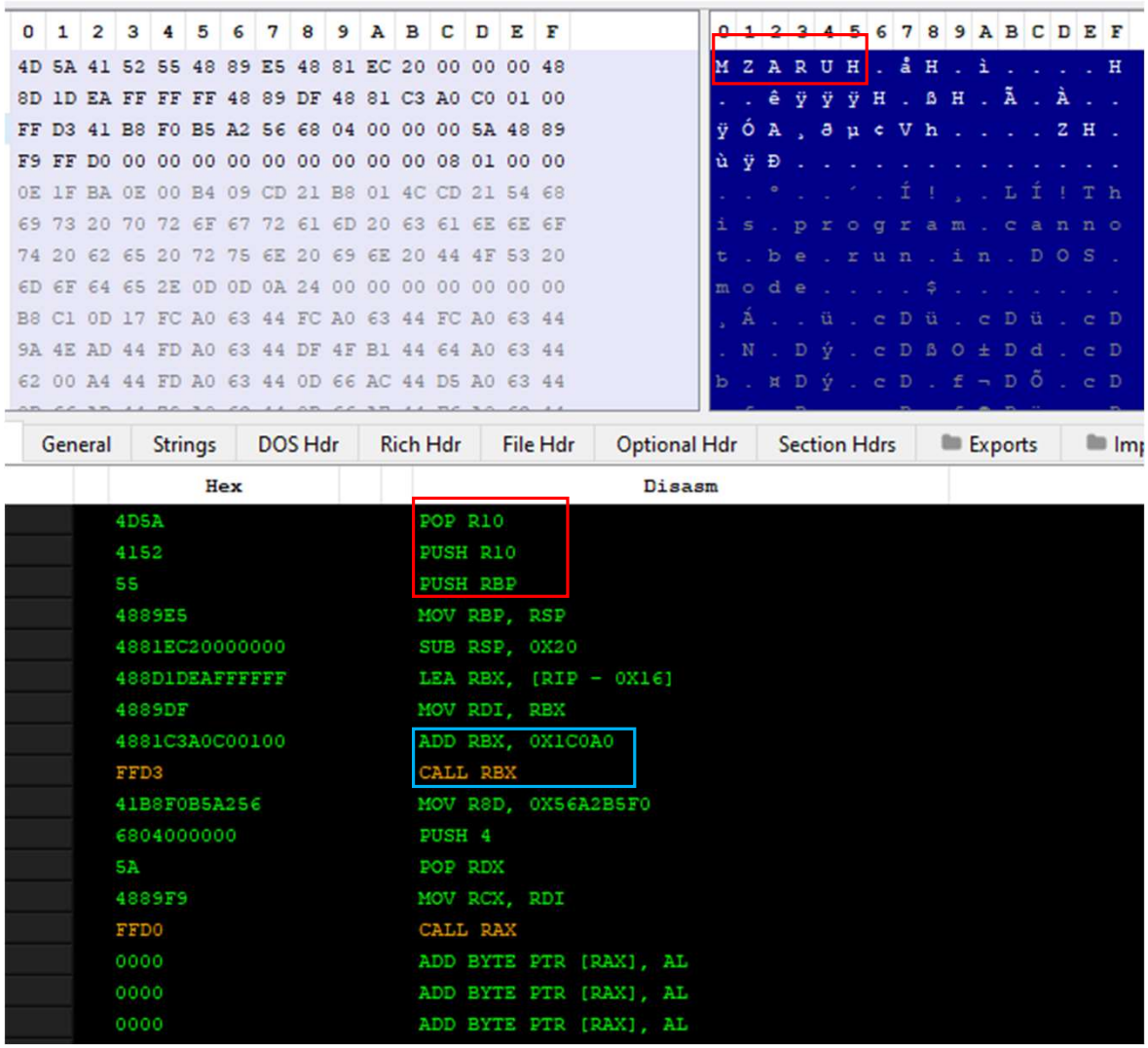


How can we execute a PE header ?!

The PE header generated for your “raw beacon” is actually also Position-Independ Code (PIC), so it is valid code.

This is possible because “MZ” as assembly is `pop r10`, It can be nullified with `push r10` which is …. “AR”


x64 Native Tools Command Prompt for VS 2022 Preview
C:\Program Files\Microsoft Visual Studio\2022\Preview>dumpbin /exports C:\Users\operator\Desktop\plain_beacon_x64.bin
Microsoft (R) COFF/PE Dumper Version 14.44.35209.0
Copyright (C) Microsoft Corporation. All rights reserved.
Dump of file C:\Users\operator\Desktop\plain_beacon_x64.bin
File Type: DLL
Section contains the following exports for beacon.x64.dll
00000000 characteristics
68151FA3 time date stamp Fri May 2 20:40:19 2025
0.00 version
1 ordinal base
1 number of functions
1 number of names
ordinal hint RVA name
1 0 0001CCA0 ReflectiveLoader
Summary
17000 .data
3000 .pdata
11000 .rdata
2000 .reloc
36000 .text
Exported Functions [1 entry]
Offset Ordinal Function RVA Name RVA Name Forwarder
45FF8 1 1CCA0 47011 ReflectiveLoader

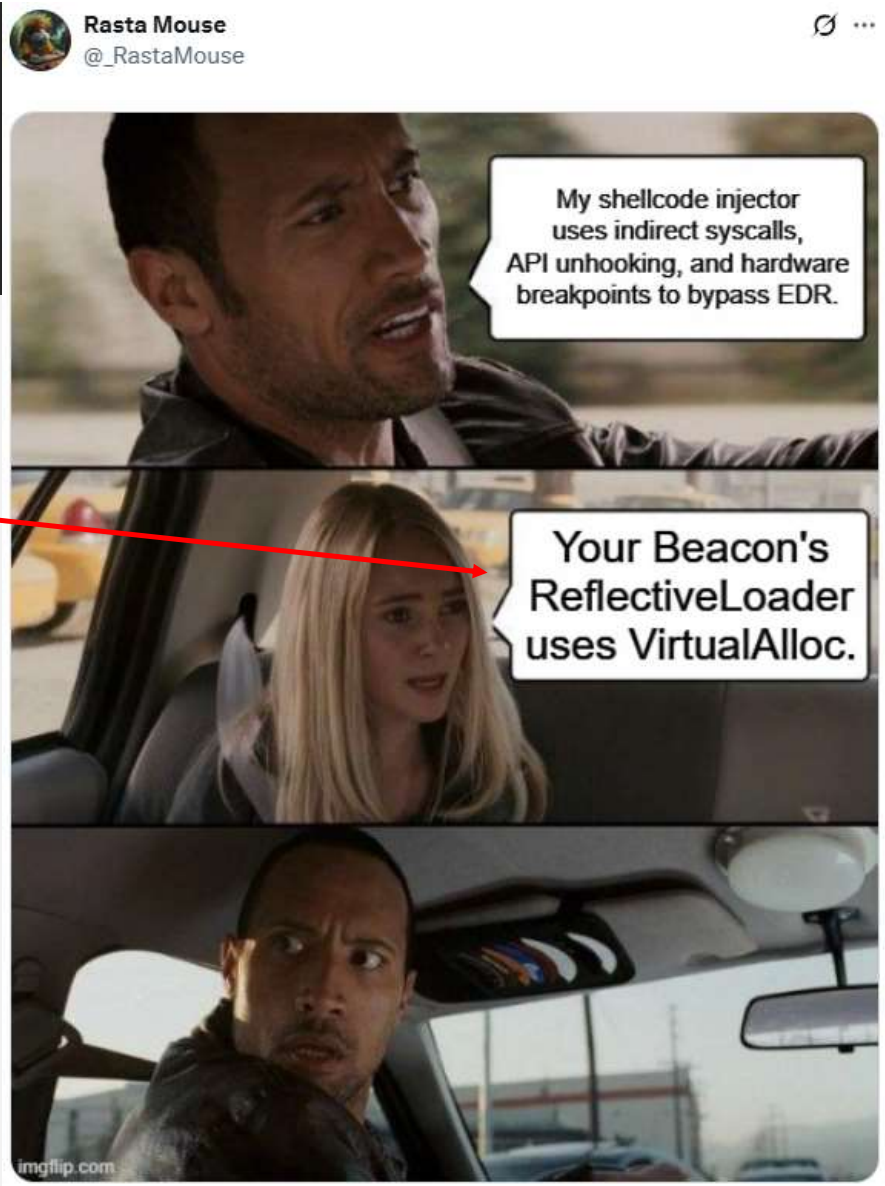

0 1 2 3 4 5 6 7 8 9 A B C D E F
4D 5A 41 52 55 48 89 E5 48 81 EC 20 00 00 00 48
8D 1D EA FF FF FF 48 89 DF 48 81 C3 A0 C0 01 00
FF D3 41 B8 F0 B5 A2 56 68 04 00 00 00 5A 48 89
F9 FF D0 00 00 00 00 00 00 00 00 00 00 08 01 00 00
0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68
69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F
74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20
6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00
B8 C1 0D 17 FC A0 63 44 FC A0 63 44 FC A0 63 44
9A 4E AD 44 FD A0 63 44 DF 4F B1 44 64 A0 63 44
62 00 A4 44 FD A0 63 44 0D 66 AC 44 D5 A0 63 44
0 1 2 3 4 5 6 7 8 9 A B C D E F
M Z A R U H
General
Strings
DOS Hdr
Rich Hdr
File Hdr
Optional Hdr
Section Hdrs
Exports
Hex
Disasm
4D5A POP R10
4152 PUSH R10
55 PUSH RBP
4889E5 MOV RBP, RSP
4881EC20000000 SUB RSP, 0X20
488D1DEAFFFFFF LEA RBX, [RIP - 0X16]
4889DF MOV RDI, RBX
4881C3A0C00100 ADD RBX, 0X1C0A0
FFD3 CALL RBX
41B8F0B5A256 MOV R8D, 0X56A2B5F0
6804000000 PUSH 4
5A POP RDX
4889F9 MOV RCX, RDI
FFD0 CALL RAX
0000 ADD BYTE PTR [RAX], AL
0000 ADD BYTE PTR [RAX], AL
0000 ADD BYTE PTR [RAX], AL

Introducing UDRL

UDRLs

Why

How to remediate this ?



UDRLs

Why

Obfuscation history with Cobalt Strike:

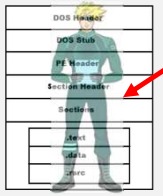
1. *Replace strings (malleable profile: `strrep`)*
2. *Tweak how the beacon is loaded in memory (malleable profile: `stage {}`)*
3. *Sleep (Sleepmask)*
4. *Customize the Sleepmask (Sleepmask)*
5. **Customize how the beacon is loaded in memory**

UDRLs

Why

Replace strings (malleable profile: `strrep`)

`MyFancyLoader ( Loader ( )  )`



UDRL: define from scratch how the beacon is loaded in memory

Tweak how the beacon is loaded in memory (malleable profile: `stage {}`)

UDRLs

Why

README MIT license

AceLdr - Avoid Memory Scanners

A position-independent reflective loader for Cobalt Strike. Zero results from Hunt-Sleeping-Beacons, BeaconHunter, BeaconEye, Patriot, Moneta, PE-sieve, or MalMemDetect.

```
SECTION( B ) PVOID copyBeaconSections( PVOID buffer, REG reg )
```

```
pApi->ntdll.NtAllocateVirtualMemory = FindFunction( hNtdll, H_API_NTALLOCATEVIRTUALMEMORY );
pApi->ntdll.NtProtectVirtualMemory  = FindFunction( hNtdll, H_API_NTPROTECTVIRTUALMEMORY );
pApi->ntdll.RtlCreateHeap           = FindFunction( hNtdll, H_API_RTLCREATEHEAP );
```

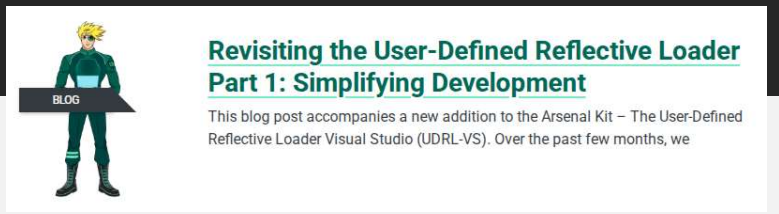
```
Api.ntdll.NtAllocateVirtualMemory( ( HANDLE )-1, &MemoryBuffer, 0, &Reg.Full, MEM_COMMIT, PAGE_READWRITE );
```

Strike Reflective Loader

A proof-of-concept User-Defined Reflective Loader (UDRL) which aims to recreate, integrate, and enhance Cobalt Strike's evasion features!

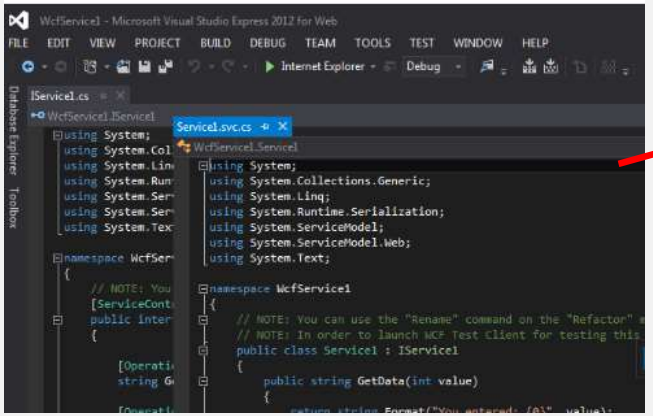
UDRLs

UDRL-VS



Cobalt Strike response was to introduce a standardized Visual Studio template building a simple loader (UDRL-VS)

Go from this:

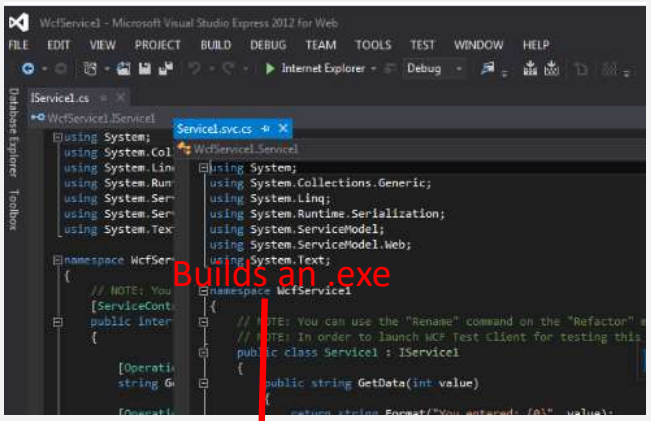


Build

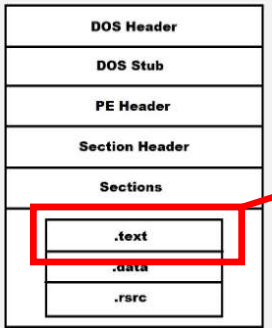
To this:

UDRLs

Extracting the PIC UDRL from an executable

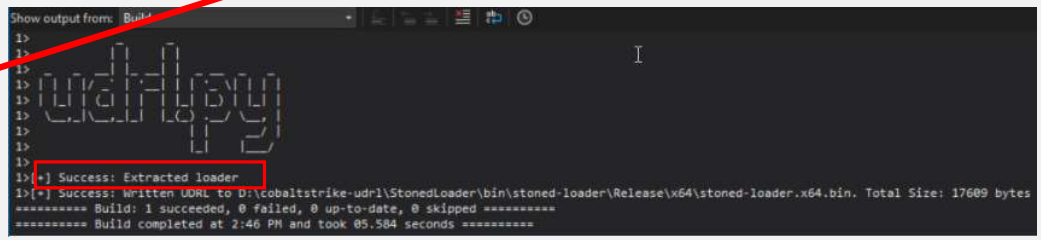


Builds an .exe



Extracts the .text section

UDRL-VS comes with a post-build Python job (`udrl.py`) which extracts the actual PIC blob from the executable



UDRLs

PIC Constraints

- Everything must be in a `.text` section (strings must be “stack strings”)
- Most functions calls must be resolved at runtime (you don’t execute `VirtualAlloc()`: you find its address, then you execute the function at that address).
- `MYSTRUCT myStruct = { 0 };` will fail as it executes `memcpy()` under the hood. Initialize it to 0x00 yourself.
- Certain Visual Studio optimizations need to be disabled to avoid these types of behavior.

Expect some frustration.

UDRLs

PIC Constraints

Normal Developer

Malware Developer

PIC Developer

UDRLs

PIC Constraints

The UDRL-VS template provides various helpers to help you address most of those challenges:

- Define strings with `PIC_STRING(myvar, "[*] Loaded at 0x%p\n");`
- Compile-time hashing readily available in the template.
- `PRINT()` macro to replace `printf()`

UDRLs

UDRL Constraints

If you want to use UDRL, with great powers come great responsibilities…

```
stage {
     set userwx "false";
     set compile_time "14 Jul 2009 8:14:00";
     set image_size_x86 "512000";
     set image_size_x64 "512000";
     set obfuscate "true";
```

YOU allocate memory and copy sections into them

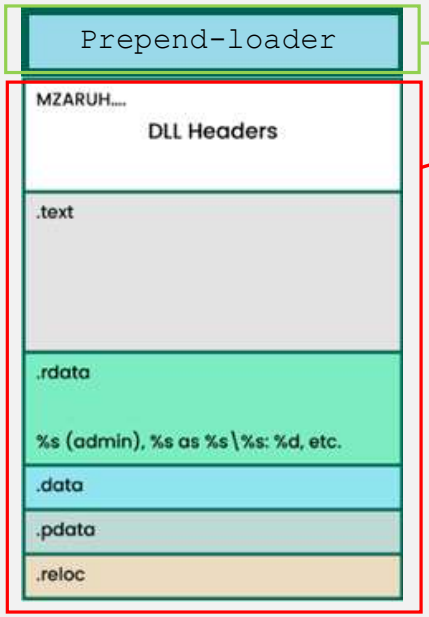
YOU put whichever values you want in the various headers (are they even needed … ?)

YOU decide what to copy in memory, for instance, headers are not needed for actual execution, cf. the **`obfuscation-loader`**:
https://www.cobaltstrike.com/blog/revisiting-the-udrl-part-2-obfuscation-masking

UDRLs

Release vs Debug mode

How to test a PIC blob of code ? (it is NOT an .exe)



PIC UDRL blob compiled by Visual Studio in Release mode and extracted.

“raw beacon”, generated by console (Aggressor scripts allow you to modify this)

Options:

1. Create a simple loader, allocate memory, run your whole implant, see what happens [realistic]
2. Build it as an .exe, artificially embed the raw beacon in a variable [convenient]

UDRLs

Release vs Debug mode

Debug mode allows you to run a mock loader as an .exe. You can use `PRINT()` to print statements to the console.

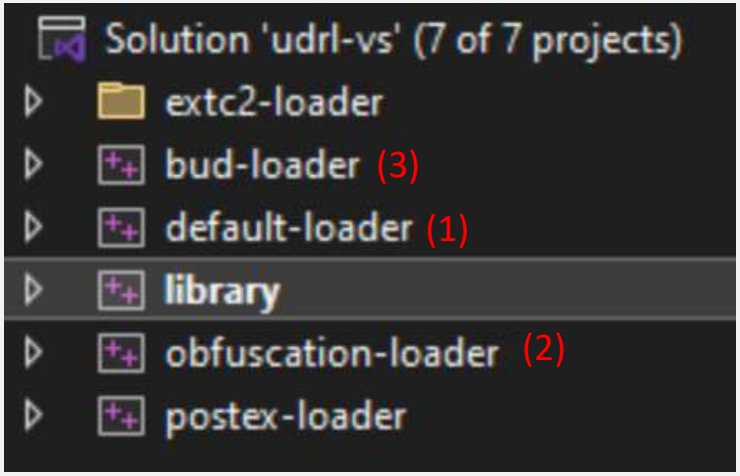
In Release mode… there is no console. A solution is to monitor the process in WinDbg.

`PRINT()` is available in Debug mode natively. For Release mode, you can use `OutputDebugSring()` to log strings and see them in WinDbg. I have implemented a Macro helper making it easier.

Demo

What Next ?

- Download the UDRL-VS (in the Artifact Kit)
- Check the 3 loaders, starting with the most basic one: `default-loader`
- `Obfuscation-loader` shows how to perform the mapping with stealthier options
- `Bud-loader` shows how to leverage a new feature of Cobalt Strike to pass arbitrary data to the Cobalt Strike beacon. This allows you to make full use of other Cobalt Strike features (notably the Sleepmask and BeaconGate)



Thank You!

References

- The blog series published by Cobalt Strike:
  https://www.cobaltstrike.com/blog/revisiting-the-udrl-part-1-simplifying-development
- A helper Macro to help print out debug statements from UDRL:
  https://rwxstoned.github.io/2025-07-06-Better-debugging-UDRL/
- Some good documentation on Cobalt Strike's key elements, including the UDRL:
  https://rastamouse.me/udrl-sleepmask-and-beacongate/