

ANALYZE ME IF YOU CAN

An introduction to malware anti-analysis techniques for dummies

Neswin Nigad

grit8086



It runs. And does nothing.

```
> main.exe  
  
# no files touched.  no network.  no payload.  
# clean exit
```

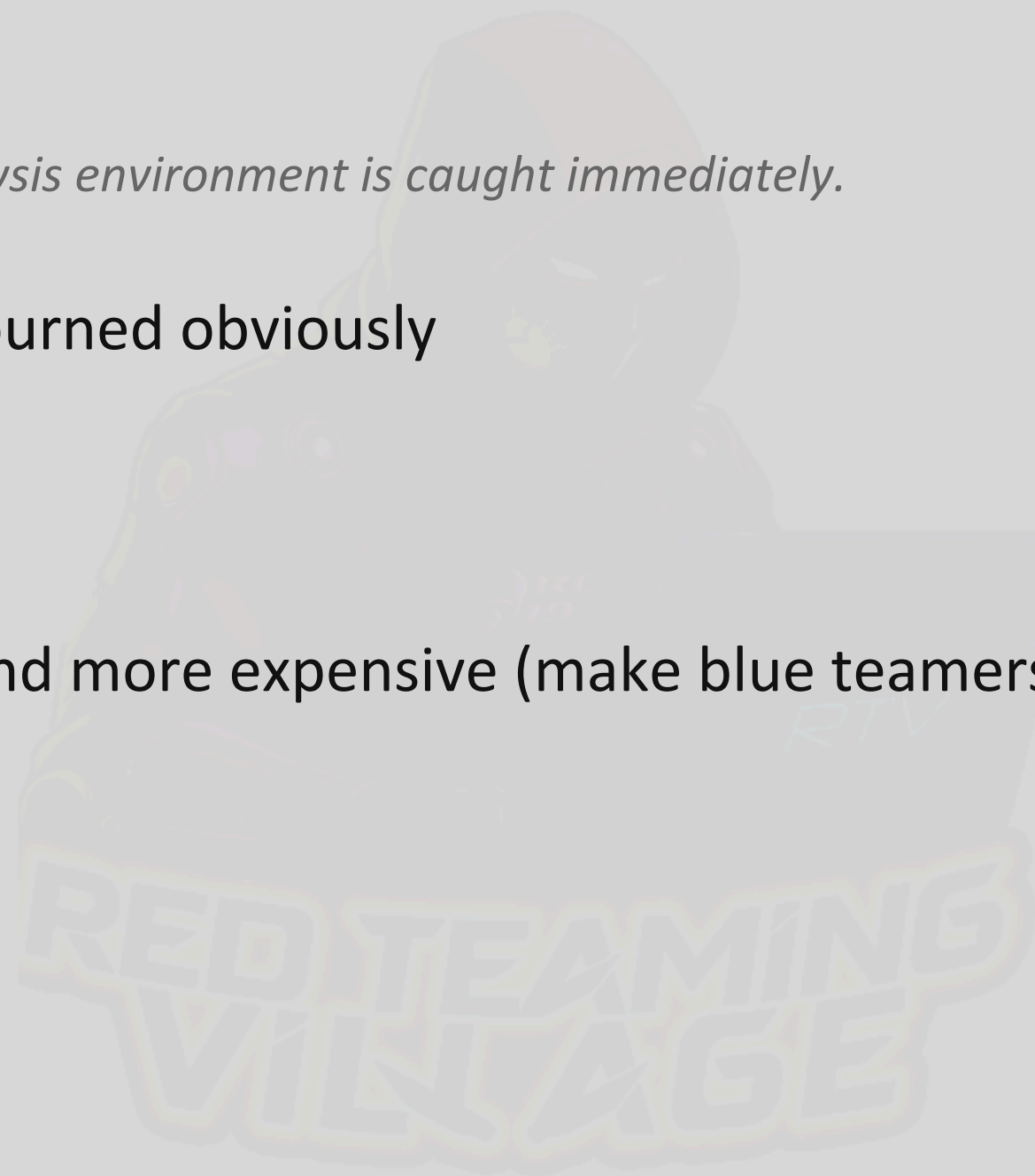
Doing nothing
is a decision.

The sample decided this environment was not worth detonating in.

Why anti-analysis?

Malware that detonates immediately in an analysis environment is caught immediately.

- We don't want our malware to get burned obviously
- The goal is not invisibility
- The goal is making analysis harder and more expensive (make blue teamers cry)



Whoami?

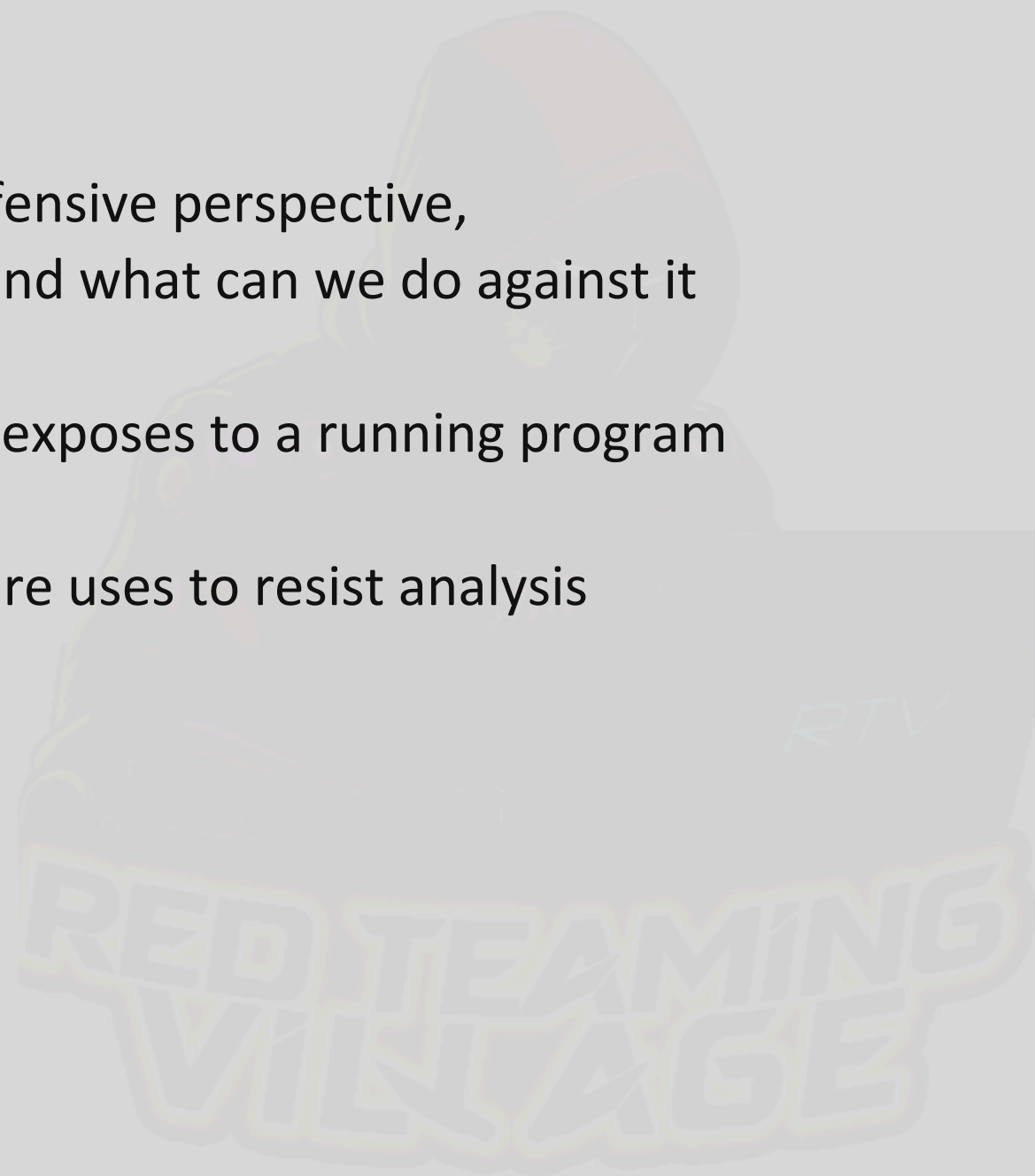
Neswin Nigad / grit8086

- Security Consultant @ **THEOS**
- Offensive Tradecraft R&D
- Fighter, poet, I create song covers, and interested in neuropsychology
- Fun fact: I like cats and I love IShowSpeed



Objectives of this talk

- Understand both the defensive and offensive perspective, what they can do and what they see, and what can we do against it
- Identify what an analysis environment exposes to a running program
- Explore the general approaches malware uses to resist analysis



What are the “threats” that we are trying to avoid?

Dynamic analysis

Running the binary and observing its behaviour in real time I.E breakpoints, hooks, runtime inspection

Static analysis

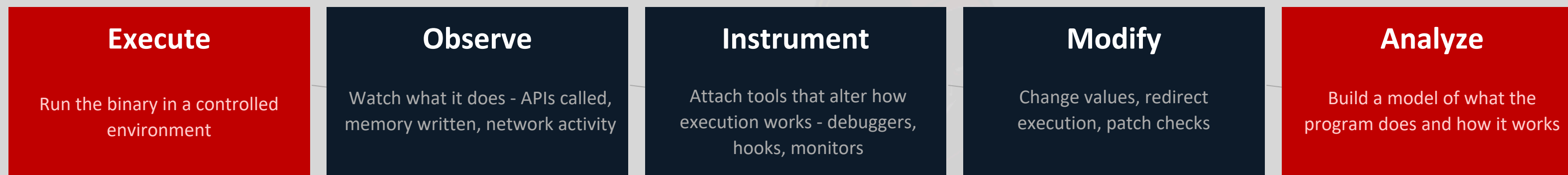
Reading the binary without executing it I.E imports, strings, control flow, recognizable patterns

Sandbox analysis

Automated detonation in a controlled environment I.E behavioral recording, short execution windows, no human

Each observer has different capabilities and different blind spots. Anti-analysis addresses all three.

What can an analyst do to our program?



This is the capability set we're designing against. Each technique in this talk addresses a specific point in this chain.

This gives us three problems to solve:

Threat 1

Runtime observation

Threat 2

Artificial environments

Threat 3

Static visibility

Threat 1: Runtime observation

If an analyst can watch the program execute, they can understand its behaviour without the program cooperating.

What this means in practice

- Debugger attached = execution is under control
- Breakpoints = analyst can pause at any instruction
- Hooks = API calls intercepted and logged
- Runtime inspection = memory, registers, callstack

What we need to determine

- Is execution currently being observed?
- Is a debugger or monitoring tool attached?
- Are there signs of runtime instrumentation?
- Is this a real execution environment?

The general approach

Detect whether execution is being observed. React when it is.

Query

Ask the operating system for characteristics associated with debugging

Compare

Evaluate the response against what a clean execution environment would return

Decide

If indicators are present = assume hostile. Alter behaviour accordingly

Concrete example - NtQueryInformationProcess

What does it allow us to determine?

```
// Query 1: "Is a user-mode debugger controlling this process?"
// Non-zero debug port = debugger detected.

pNtQIP(hProc, ProcessDebugPort, &dwPort, sizeof(DWORD_PTR), NULL );

if (dwPort != 0)
    hostile;

// Query 2: "Does this process have a debug object?"
// STATUS_PORT_NOT_SET = no debug object.
// STATUS_SUCCESS + valid handle = debugger detected.

pNtQIP(hProc, ProcessDebugObjectHandle, &hDebugObject, sizeof(HANDLE), NULL);

if (status == STATUS_SUCCESS && hDebugObject != NULL)
    hostile;
```

Why is it useful?

- Queries kernel state directly
- Not just a PEB flag
- Resolved at runtime, not visible in IAT
- Harder to trivially patch

Why not IsDebuggerPresent?

The traditional check is well-known and well-targeted.

IsDebuggerPresent() / **PEB.BeingDebugged**

```
PPEB pPeb = (PPEB)__readgsqword(0x60);  
return pPeb->BeingDebugged; // one byte
```

Reads a single flag byte in the PEB. If a debugger is attached, that byte is set to 1.

The problems

- Extremely obvious = every analyst tool knows this check
- Named directly in the IAT = visible before execution
- Commonly targeted = many bypass tools patch it automatically
- One byte flip defeats it entirely

What that looks like in practice

Traditional check = defeated easily

```
IsDebuggerPresent() = returns 1  
  
pPeb->BeingDebugged. = 0  
IsDebuggerPresent() = returns 0 ✓ bypassed
```

NtQuery = survives the byte patch

```
pPeb->BeingDebugged = 0  
NtQueryInformationProcess( ProcessDebugObjectHandle )  
  
= kernel object still present ✗ still detected
```

***"Debugger detection isn't about one magic API.
It's about identifying evidence that runtime execution is being observed."***

Threat 2: Artificial environments

Automated sandboxes and virtualized environments analyze samples automatically, without a human, in a controlled host.

What the environment exposes

- Hardware characteristics = MAC prefix, core count, RAM
- System configuration uptime, resolution, disk
- Behavioral characteristics = no user input, no documents

What can we do?

- Fingerprint the execution environment
- Read characteristics and compare against expected values

What we can read from the environment

MAC address OUI

Hardware fingerprint

```
GetAdaptersInfo = first 3 bytes of adapter MAC  
VMware: 00:0C:29   VirtualBox: 08:00:27  
Hyper-V: 00:15:5D   Parallels: 00:1C:42
```

System resources

Resource fingerprint

```
GetSystemInfo      = core count  
GlobalMemoryStatusEx = RAM  
GetDiskFreeSpaceEx = disk space
```

Screen resolution

Display fingerprint

```
EnumDisplayMonitors = callback  
Compare X/Y against known values:  
1920×1080   2560×1440   1440×900
```

None of these is definitive alone.

Why combine multiple signals?

Individual indicators produce false positives

A real machine can have low RAM. A VM can have a common resolution.

Analysts can manipulate individual characteristics

Changing the MAC prefix takes seconds. Changing all indicators simultaneously is harder.

Weak signals sum to stronger confidence

Low RAM alone means nothing. Low RAM + VMware MAC + no user files + 2-minute uptime is a pattern.

***"The goal isn't to prove 'this is a VM.'
The goal is to determine whether the environment looks sufficiently suspicious."***

Threat 3: Static visibility

Imports · API names · Strings · Constants · Control flow · Function structure · Patterns

Symbols	Search symbols	.*	Aa
Name	Address	Sect	
ApiHammering	0x140001a...	.tex	
CloseHandle	0x1400291...	.ext	
CloseHandle	0x14000b4...	.ida	
CloseHandle	0x140003b...	.tex	
DebuggerCheck	0x1400015...	.tex	
DeleteCriticalSection...	0x1400291...	.ext	
DeleteCriticalSection...	0x14000b4...	.ida	
DeleteCriticalSection...	0x140003b...	.tex	
EnterCriticalSection...	0x1400291...	.ext	
EnterCriticalSection...	0x14000b4...	.ida	
EnterCriticalSection...	0x140003a...	.tex	
FreeLibrary	0x1400291...	.ext	
FreeLibrary	0x14000b4...	.ida	
FreeLibrary	0x140003a...	.tex	
GetAdaptersInfo	0x1400291...	.ext	
GetAdaptersInfo	0x14000b4...	.ida	
GetAdaptersInfo	0x1400023...	.tex	
GetCommandLineA	0x1400291...	.ext	
GetCommandLineA	0x14000b4...	.ida	
GetCommandLineA	0x140003a...	.tex	

```
int64_t main()
{
    __main();
    puts("\n[*] nerites ");
    puts("\n[*] Phase 1: environment checks");

    if (DebuggerCheck())
    {
        fwrite("\n[!] Anti-debug triggered ", 1, 0x35, __acrt_iob_func(2));
        ApiHammering(0xc350);
        return 0xffffffff;
    }

    if (MacAddressCheck())
    {
        fwrite("\n[!] VM check triggered ", 1, 0x33, __acrt_iob_func(2));
        ApiHammering(0xc350);
        return 0xffffffff;
    }

    puts("\n[+] Phase 1 complete ");
    puts("\n[*] Phase 2: dynamic API resolution");
    HMODULE hModule = GetModuleHandleA("kernel32.dll");

    if (!hModule)
    {
        enum WIN32_ERROR rax_5 = GetLastError();
        fprintf(__acrt_iob_func(2),
            "\n[!] Failed to obtain kernel32.dll handle (GetLastError: %lu)\n",
            (uint64_t)rax_5);
        return 0xffffffff;
    }

    printf("\n[+] Phase 2: kernel32.dll handle: 0x%p\n", hModule);
    int64_t rax_9 = GetProcAddress(hModule, "VirtualAlloc");
    int64_t rax_11 = GetProcAddress(hModule, "VirtualProtect");
    int64_t rax_13 = GetProcAddress(hModule, "CreateThread");
    int64_t rax_15 = GetProcAddress(hModule, "WaitForSingleObject");
}
```

The pattern we want to prevent

Open binary:

1. "See interesting API"
2. "Understand functionality"

Concrete example (Dynamic WinAPI function resolving)

Without Dynamic API resolution

```
// Static import = three functions appear  
directly in the IAT
```

```
VirtualAlloc(0, size, MEM_COMMIT,  
PAGE_EXECUTE_READWRITE)
```

With Dynamic API resolution

```
// GetModuleHandle gets a handle to an already-loaded DLL (no disk load)  
hMod = GetModuleHandle("kernel32.dll");
```

```
// GetProcAddress walks the DLL's export table  
and returns a function pointer, no static IAT entry for the target function  
// holds the entry point address of VirtualAlloc  
pVirtualAlloc = GetProcAddress(hMod, "VirtualAlloc");
```

```
// Call through the pointer exactly like a normal function call  
pVirtualAlloc(0, size, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
```

Before & After

Before: IAT contains **VirtualAlloc, CreateThread, WaitForSingleObject**

After: IAT contains: **GetModuleHandle, GetProcAddress**
Strings contains: **VirtualAlloc, CreateThread, WaitForSingleObject, etc.**

"We're not eliminating static analysis. We're increasing the analyst's work."

Detection alone isn't enough

If the environment is hostile, the loader needs to decide what to do.

Detection -> Decision -> Response

Exit cleanly

Leave no trace. Appear to have done nothing.

Decoy

Execute harmless behaviour that resembles normal operation.

Stall

Delay execution = exhaust the analysis window without detonating.

API hammering

Generate real system activity to drain the sandbox clock harder to skip.

Stalling - why API hammering over a simple delay?

Sleep() / WaitForSingleObject()

- Simple and obvious - well-known sandbox evasion technique
- Sandboxes patch or fast-forward sleep calls routinely
- Leaves no real system activity during the wait
- Easily detected and skipped by behavioral monitors

API hammering

- Makes thousands of real, legitimate API calls
- Generates actual system activity
- Sandbox cannot skip calls without risking breaking the environment
- Much harder to distinguish from a simple sleep.

Trade-off: API hammering produces a visible spike in call volume , we can spot this by behavioral analysis. No technique is invisible.

Putting it together = the loader

Threat 1	NtQueryInformationProcess Detects kernel-level debugging indicators
Threat 2	MAC OUI + System fingerprint Identifies virtualized or controlled environments
Threat 3	Dynamic API Resolution Reduces static visibility of function names
Response	API hammering Stalls sandbox execution when hostile detected

The demo shows all four behaving together in a single binary.



Recycle Bin



Microsoft Edge



main.exe - PID: 6688 - Module: ntdll.dll - Thread: Main Thread 5658 - x64dbg

File View Debug Tracing Plugins Favourites Options Help May 27 2026 (TitanEngine)

CPU Log Notes Breakpoints Memory Map Call Stack SEH

RIP 00007FFE8CD20861 EB 00
00007FFE8CD20863 48:83C
00007FFE8CD20867 C3

RAX	0000000000000000	"l d
RBX	00007FFE8CD75910	ntd
RCX	00007FFE8CCEDA54	
RDX	0000000000000000	
RSI	0000000000000000	

M

Microsoft Edge

S

Settings System

T

Tailscale

W

Windows Accessories

Windows Administrative Tools

Windows Backup System

Windows Ease of Access

Windows PowerShell

Windows Security System

Windows System

Type here to search

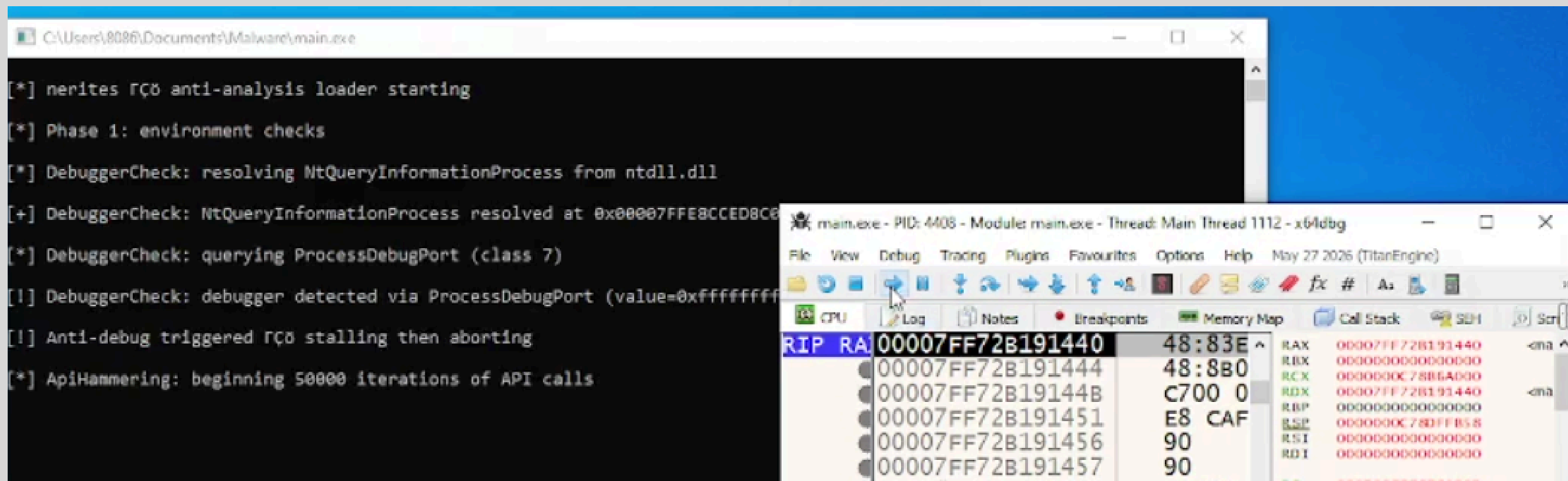


1:58 AM
9/13/2026



DEMO (Analyst environment)

An analysis environment. Detection fires. Hostile behaviour activates.



First stage check already blocked the environment from completely running our malware.

Privacy Statement

Zoom Roblox Player Left 4 Dead 2 HxD PEAK AlecaFrame

We've spent the entire talk asking:

"How can the malware avoid the analyst?"

Now ask the opposite:

"How much work will it take for the analyst to figure out the malware's behavior?"

Even with Sandbox detection, dynamic API resolution, API hammering = what does the analyst still have?

- The binary is still a binary, it can be opened in a disassembler
- There are still obvious indicators I.E visible function names, signs of usage of dynamic API resolution, strings
- The question is how much work it takes, not whether it's possible



What the analyst sees (Import Address Table)

A loader combining these techniques announces itself through its imports.

FUNCTION	MODULE	TELLS THE ANALYST
GetModuleHandleA	kernel32.dll	module handle retrieval
GetProcAddress	kernel32.dll	function resolved dynamically
GetAdaptersInfo	iphlpapi.dll	network / MAC check
Thousand API calls	etc	API hammering / stalling

GetModuleHandleA + GetProcAddress together signal that at least one function is being hidden. An analyst knows where to start looking.

What can the analyst still find in the disassembler?

```
int64_t main()
{
    __main();
    puts("\n[*] nerites ");
    puts("\n[*] Phase 1: environment checks");

    if (DebuggerCheck())
    {
        fwrite("\n[!] Anti-debug triggered ", 1, 0x35, __acrt_iob_func(2));
        ApiHammering(0xc350);
        return 0xFFFFFFFF;
    }

    if (MacAddressCheck())
    {
        fwrite("\n[!] VM check triggered ", 1, 0x33, __acrt_iob_func(2));
        ApiHammering(0xc350);
        return 0xFFFFFFFF;
    }
}
```

The control flow sequence is very predictable.

What the analyst finds in the disassembler

```
printf("\n[+] Phase 2: kernel32.dll handle: 0x%p\n", hModule);  
int64_t rax_9 = GetProcAddress(hModule, "VirtualAlloc");  
int64_t rax_11 = GetProcAddress(hModule, "VirtualProtect");  
int64_t rax_13 = GetProcAddress(hModule, "CreateThread");  
int64_t rax_15 = GetProcAddress(hModule, "WaitForSingleObject");
```

The dynamically resolved functions are also visible

Why this version is still reversible

- **GetModuleHandle + GetProcAddress in the IAT**
 - The resolution mechanism is exposed.
 - An analyst can identify that APIs are being resolved at runtime and determine how they are being resolved.
- **Function name strings still plaintext**
 - VirtualAlloc, CreateThread visible in strings output / disassembly.
 - The analyst knows which functions are being resolved even without the IAT entries.
- **Fingerprint values readable** = MAC OUI prefixes, constant values are still visible as an indicator.
- **API hammering creates behavioral signal** = Thousands of API calls in a rapid amount of time can be a potential behavioral redflag.

"Making something harder to identify isn't the same as making it impossible to reverse."

So what would we change?

The current implementation demonstrates the concepts. But the analytical weaknesses are obvious.

API hashing

Function name strings still plaintext in your loader, hashing removes them entirely.

Custom PEB walk

GetModuleHandleA + GetProcAddress still in your IAT, a PEB walk replaces them

Anti-dissassembly

Break static analysis tools. Introduce control-flow patterns that confuse disassemblers without affecting execution.

String and constant encryption

Encrypt fingerprint values at compile time. Decrypt at runtime - MAC prefixes never appear in strings.

That's the next talk.

Shoutout // Credits (proofreaders)

- **Whispergate & Hotel Zero**

- Lavender
- Hunter
- mocha
- chroma

- **Incident Response Chaos Club**

- Humpty/Tony
- BunchOfWetFrogs

- **InfinityCurve Labs**

- c5pider
- bakki

- **Bozos:**

- Lattice23
- Seifer Rija Boado
- Yensyd Francisco



Questions?



References and lab notes available on request.

Thank you for attending my talk!

Neswin Nigad

grit8086

