# Anti-Debugging Techniques

6 min read · Aug 3, 2021

David Ayman Follow



Anti-debugging techniques are ways for a program to detect if it runs under the control of a debugger. Debugging malware code enables a malware analyst to run the malware step by step, introduce changes to memory space, variable values, configurations, and more. Therefore, if debugging is done successfully, it facilitates the understanding of the malware's behavior, mechanisms, and capabilities. For obvious reasons, this is something malware authors would want to prevent. Anti-Debugging techniques are meant to ensure that a program is not running under a debugger, and in the case that it is, to change its behavior correspondingly. In most cases, the Anti-Debugging process will slow down the process of reverse engineering, but will not prevent it.

1. **Basic API Anti-Debugging**
   I. IsDebuggerPresent()
   II. CheckRemoteDebuggerPresent()
   III. OutputDebugString()
   IV. FindWindow()

2. **Advanced API Anti-Debugging**
   I. NtQueryInformationProcess
   II. NtSetInformationThread

3. **Timing**
   I. RDPMC/RDTSC

II. GetLocalTime()
III. GetSystemTime()
IV. GetTickCount()
V. ZwGetTickCount() / KiGetTickCount()
VI. QueryPerformanceCounter()
VII. timeGetTime()

## 1. Basic API Anti-Debugging

### I. IsDebuggerPresent

The simplest anti-debugging method is calling the **IsDebuggerPresent** function.

```
BOOL WINAPI IsDebuggerPresent(void);
```

IsDebuggerPresent function returns 1 if the process is being debugged, 0 otherwise. This API simply reads the PEB!BeingDebugged byte-flag (located at offset 2 in the PEB structure).

```
int main(){
 if (IsDebuggerPresent()){
  std::cout << "Debugger is present !!!" << std::endl;
  exit(-1);
 } else {
  std::cout << "Debugger is not present :)" << std::endl;
 }
 return 0;
}
```

### II. CheckRemoteDebuggerPresent

Unlike the **IsDebuggerPresent** function, CheckRemoteDebuggerPresent checks if a process is being debugged by another parallel process.

```
BOOL WINAPI CheckRemoteDebuggerPresent(__in HANDLE hProcess,
           __inout PBOOL pbDebuggerPresent);

BOOL bDebuggerPresent;
if (TRUE == CheckRemoteDebuggerPresent(GetCurrentProcess(),
&bDebuggerPresent) &&
    TRUE == bDebuggerPresent)
    ExitProcess(-1);
```

### III. OutputDebugString

This technique is deprecated as it works only for Windows versions earlier than Vista. However, this technique is too well known to not mention here.

Debuggers typically support communication with processes being debugged; such processes can send messages to the debugger with Win32 method **OutputDebugString** function.

```
void WINAPI OutputDebugString(__in_opt  LPCTSTR lpOutputString);
```

The concept is quite simple. If we call **OutputDebugString** in order to pass a string to the debugger, and a debugger is attached, then when we return back to the user code, the value in EAX will be a valid address inside the process address space.

However, if a debugger is not attached, then the value in EAX will be either 0 in Windows 7 (probably the same also in Vista) or 1 in Windows XP (tested in WinXP SP3), which are not of course valid addresses.

So in that case, if we try to read the contents of an invalid memory address, an exception will be raised **( EXCEPTION_ACCESS_VIOLATION 0xc0000005)** and we will know that a debugger is not attached.
On the other hand, if an exception does not occur then we know that a debugger is attached.

```
int main()
{
```

```
 OutputDebugStringA(“aaaaa”);

__try
 {
  __asm mov ebx, dword ptr [eax] //if not debugged it will raise
an exception cause eax will be 0 or 1
  cout << "debugger found" << endl;
 }
 __except(EXCEPTION_EXECUTE_HANDLER)
 {
  cout << "no debugger" << endl;
 }
 system ("pause");
 return 0;
}
```

### IV. FindWindow

Checking for the presence of a debugger with **FindWindow** function is probably one of the hackiest ways to check if a debugger is watching your program. The idea is that we know the names of many of the common debuggers and the Windows API provides a method to check if a window with a particular name exists. If there's a match, it means that there is most likely an instance of that debugger running. It won't however, tell you if it's attached to your process.

### Some Known Debuggers

`ollyDbg` , `x64dbg` , `x32dbg` , `IDA` , `WindDbg` , `Soft Ice`

```
int main()
{
 LPCWSTR windowName = L"x64dbg";

if (FindWindow(NULL, windowName))
 {
  MessageBoxA(NULL, "Debugger is present !!!", "Notification",
MB_OK);
  exit(-1);
 }
 return 0;
}
```

## 2. Advanced API Anti-Debugging

### I. NtQueryInformationProcess

One such poorly documented API function is the NtQueryInformationProcess function which is used to retrieve information about a target process. The function prototype looks like the following:

```
NTSTATUS WINAPI NtQueryInformationProcess(
    __in HANDLE ProcessHandle,
    __in PROCESSINFOCLASS ProcessInformationClass,
    __out PVOID ProcessInformation,
    __in ULONG ProcessInformationLength,
    __out_opt PULONG ReturnLength
);

bool IsDebugged()
{
    HWND hExplorerWnd = GetShellWindow();
    if (!hExplorerWnd)
        return false;

    DWORD dwExplorerProcessId;
    GetWindowThreadProcessId(hExplorerWnd, &dwExplorerProcessId);

    ntdll::PROCESS_BASIC_INFORMATION ProcessInfo;
    NTSTATUS status = ntdll::NtQueryInformationProcess(
        GetCurrentProcess(),
        ntdll::PROCESS_INFORMATION_CLASS::ProcessBasicInformation,
        &ProcessInfo,
        sizeof(ProcessInfo),
        NULL);

    if (!NT_SUCCESS(status))
        return false;

    return (DWORD)ProcessInfo.InheritedFromUniqueProcessId !=
dwExplorerProcessId;
}
```

### II. NtSetInformationThread

The function **ntdll!NtSetInformationThread()** can be used to hide a thread from a debugger. It is possible with a help of the undocumented value **THREAD_INFORMATION_CLASS::ThreadHideFromDebugger (0x11)**. This is intended to be used by an external process, but any thread can use it on itself. After the thread is hidden from the debugger, it will continue running but the debugger won't receive events related to this thread. This thread can perform anti-debugging checks such as code checksum, debug flags verification, etc.

However, if there is a breakpoint in the hidden thread or if we hide the main thread from the debugger, the process will crash and the debugger will be stuck.

The function prototype looks like the following:

```
NTSTATUS NTAPI NtSetInformationThread(
    __in HANDLE ThreadHandle,
    __ in THREAD_INFORMATION_CLASS ThreadInformationClass,
    __in PVOID ThreadInformation
    __in ULONG ThreadInformationLength
)
```

In the example, we hide the current thread from the debugger. This means that if we trace this code in the debugger or put a breakpoint to any instruction of this thread, the debugging will be stuck once **ntdll!NtSetInformationThread()** is called.

```
#define NtCurrentThread ((HANDLE)-2)

bool AntiDebug()
{
```

```
    NTSTATUS status = ntdll::NtSetInformationThread(
        NtCurrentThread,
        ntdll::THREAD_INFORMATION_CLASS::ThreadHideFromDebugger,
        NULL,
        0);
    return status >= 0;
}
```

## 3. Timing

### I. RDPMC/RDTSC

These instructions require the flag PCE to be set in CR4 register.

**RDPMC** instruction can be used only in Kernel Mode.

```
bool IsDebugged(DWORD64 qwNativeElapsed)
{
    ULARGE_INTEGER Start, End;
    __asm
    {
        xor  ecx, ecx
        rdpmc
        mov  Start.LowPart, eax
        mov  Start.HighPart, edx
    }
    // ... some work
    __asm
    {
        xor  ecx, ecx
        rdpmc
        mov  End.LowPart, eax
        mov  End.HighPart, edx
    }
    return (End.QuadPart - Start.QuadPart) > qwNativeElapsed;
}
```

**RDTSC** is a User Mode instruction.

```
bool IsDebugged(DWORD64 qwNativeElapsed)
{
    ULARGE_INTEGER Start, End;
    __asm
```

```
    {
        xor  ecx, ecx
        rdtsc
        mov  Start.LowPart, eax
        mov  Start.HighPart, edx
    }
    // ... some work
    __asm
    {
        xor  ecx, ecx
        rdtsc
        mov  End.LowPart, eax
        mov  End.HighPart, edx
    }
    return (End.QuadPart - Start.QuadPart) > qwNativeElapsed;
}
```

### II. GetLocalTime()

```
bool IsDebugged(DWORD64 qwNativeElapsed)
{
    SYSTEMTIME stStart, stEnd;
    FILETIME ftStart, ftEnd;
    ULARGE_INTEGER uiStart, uiEnd;

    GetLocalTime(&stStart);
    // ... some work
    GetLocalTime(&stEnd);

    if (!SystemTimeToFileTime(&stStart, &ftStart))
        return false;
    if (!SystemTimeToFileTime(&stEnd, &ftEnd))
        return false;

    uiStart.LowPart  = ftStart.dwLowDateTime;
    uiStart.HighPart = ftStart.dwHighDateTime;
    uiEnd.LowPart  = ftEnd.dwLowDateTime;
    uiEnd.HighPart = ftEnd.dwHighDateTime;
    return (uiEnd.QuadPart - uiStart.QuadPart) > qwNativeElapsed;
}
```

### III. GetSystemTime()

```
bool IsDebugged(DWORD64 qwNativeElapsed)
{
    SYSTEMTIME stStart, stEnd;
    FILETIME ftStart, ftEnd;
    ULARGE_INTEGER uiStart, uiEnd;
```

```
    GetSystemTime(&stStart);
    // ... some work
    GetSystemTime(&stEnd);

    if (!SystemTimeToFileTime(&stStart, &ftStart))
        return false;
    if (!SystemTimeToFileTime(&stEnd, &ftEnd))
        return false;

    uiStart.LowPart  = ftStart.dwLowDateTime;
    uiStart.HighPart = ftStart.dwHighDateTime;
    uiEnd.LowPart  = ftEnd.dwLowDateTime;
    uiEnd.HighPart = ftEnd.dwHighDateTime;
    return (uiEnd.QuadPart - uiStart.QuadPart) > qwNativeElapsed;
}
```

#### IV. GetTickCount()

```
bool IsDebugged(DWORD dwNativeElapsed)
{
    DWORD dwStart = GetTickCount();
    // ... some work
    return (GetTickCount() - dwStart) > dwNativeElapsed;
}
```

#### V. ZwGetTickCount() / KiGetTickCount()

Both functions are used only from Kernel Mode.

Just like User Mode GetTickCount() or GetSystemTime(), Kernel Mode ZwGetTickCount() reads from the KUSER_SHARED_DATA page. This page is mapped read-only into the user-mode range of the virtual address and read-write in the kernel range. The system clock tick updates the system time, which is stored directly in this page.

ZwGetTickCount() is used the same way as GetTickCount(). Using KiGetTickCount() is faster than calling ZwGetTickCount(), but slightly slower than reading from the KUSER_SHARED_DATA page directly.

```
bool IsDebugged(DWORD64 qwNativeElapsed)
{
    ULARGE_INTEGER Start, End;
    __asm
```

```
    {
        int  2ah
        mov  Start.LowPart, eax
        mov  Start.HighPart, edx
    }
    // ... some work
    __asm
    {
        int  2ah
        mov  End.LowPart, eax
        mov  End.HighPart, edx
    }
    return (End.QuadPart - Start.QuadPart) > qwNativeElapsed;
}
```

### VI. QueryPerformanceCounter()

```
bool IsDebugged(DWORD64 qwNativeElapsed)
{
    LARGE_INTEGER liStart, liEnd;
    QueryPerformanceCounter(&liStart);
    // ... some work
    QueryPerformanceCounter(&liEnd);
    return (liEnd.QuadPart - liStart.QuadPart) > qwNativeElapsed;
}
```

### VII. timeGetTime()

```
bool IsDebugged(DWORD dwNativeElapsed)
{
    DWORD dwStart = timeGetTime();
    // ... some work
    return (timeGetTime() - dwStart) > dwNativeElapsed;
}
```

Anti Debugger Reverse Engineering Malware Hacking

Follow

## Written by David Ayman

37 followers · 61 following

💻 Software Engineer | 🦠 Malware Analyst | ☁️ Cloud Architect. Passionate about 💎 Code Quality and 🧩 Functional Programming.

## No responses yet

N Niguelas

What are your thoughts?

## More from David Ayman

David Ayman

## Java Optimization 101: Best Practices for a Faster, More Efficient Codebase 💻🚀

Java is a powerful and versatile programming language that is widely used in the development of applications for various platforms. As the...

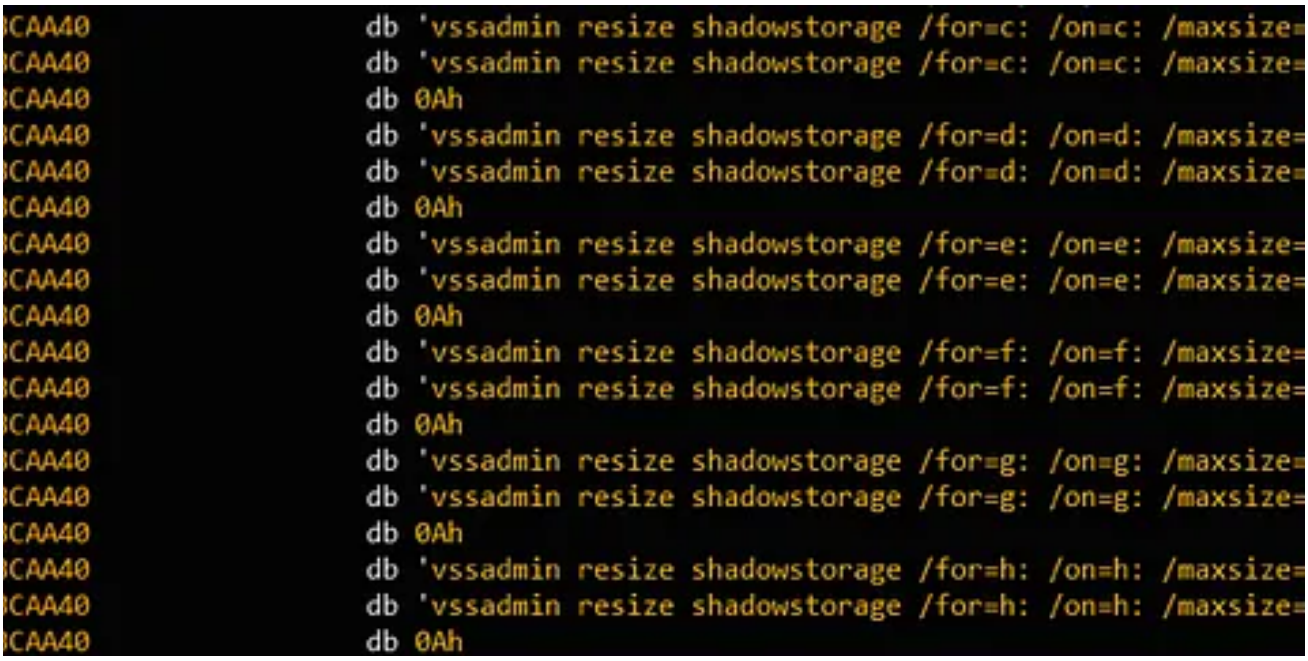
4


David Ayman

## Windows Threat Hunting: Processes

These programs are not designed to be malicious - but they do have functions that can be used for malicious purposes.

Oct 15, 2021 59



David Ayman

## Ransomware: Shadow Copies

Ransomware has been a dominant threat to organizations for several years now, causing unparalleled damage estimated in the billions of...

Oct 15, 2021 45

| | default | private | protected | public |
|---|---|---|---|---|
| same class | yes | yes | yes | yes |
| same package subclass | yes | no | yes | yes |
| same package non-subclass | yes | no | yes | yes |
| different package subclass | no | no | yes | yes |
| different package non-subclass | no | no | no | yes |

David Ayman

## Master Series: Advanced Java SE2EE

Java, as one of the most versatile programming languages, offers a wealth of features that empower developers to build robust, scalable...

Aug 27, 2024

See all from David Ayman

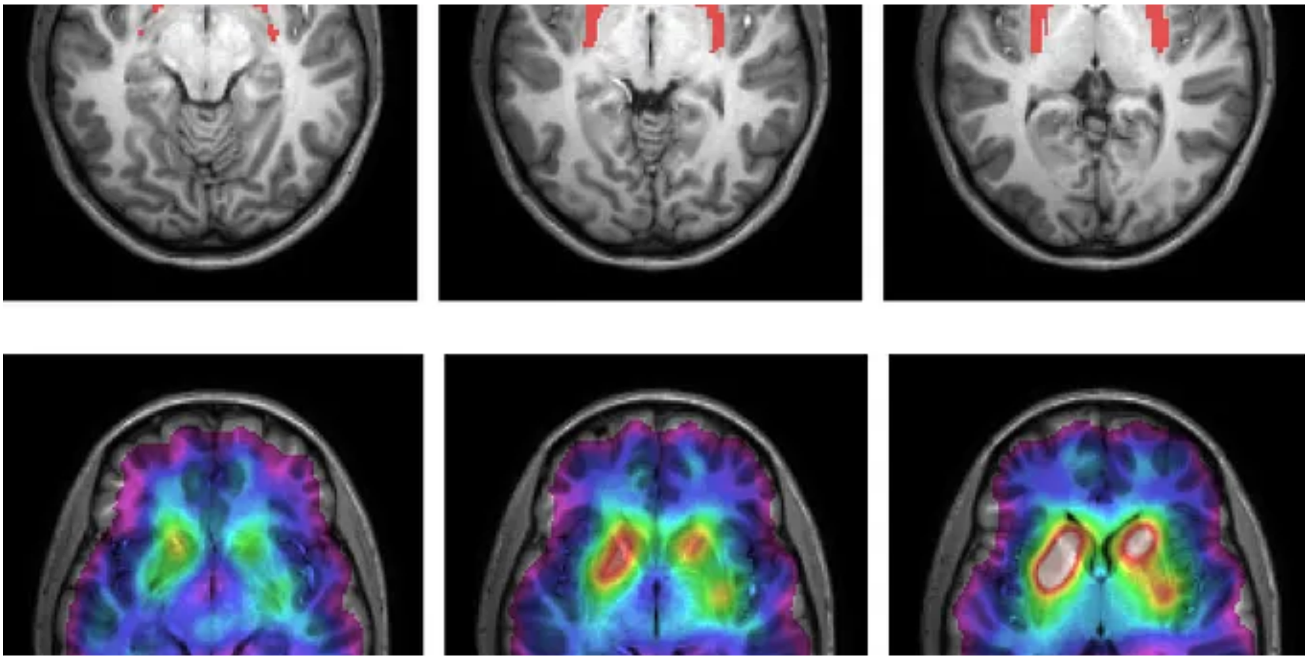
## Recommended from Medium

The CS Engineer

**Async Rust Looked Hard — Until We Compared 6 Production Outages**

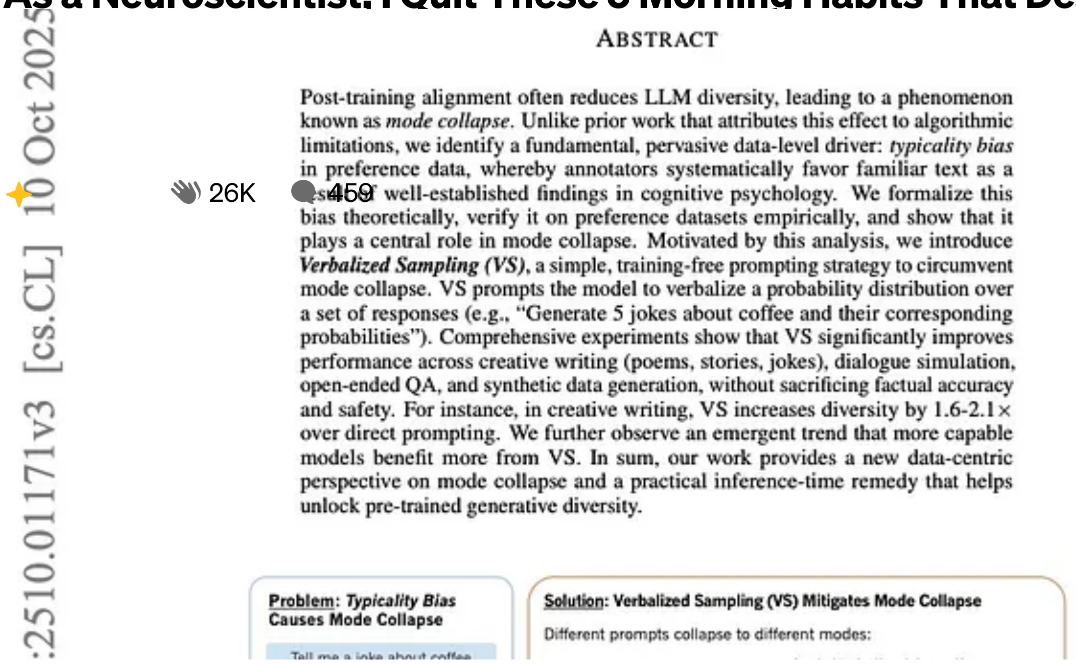
The p99 latency jumped from 42 ms to 1.8 s. :(

Feb 3 91 2



In Write A Catalyst by Dr. Patricia Schmidt

## As a Neuroscientist, I Quit These 5 Morning Habits That Destroy Your


arXiv:2510.01171v3 [cs.CL] 10 Oct 2025

Abstract

Post-training alignment often reduces LLM diversity, leading to a phenomenon known as *mode collapse*. Unlike prior work that attributes this effect to algorithmic limitations, we identify a fundamental, pervasive data-level driver: *typicality bias* in preference data, whereby annotators systematically favor familiar text as a [illegible] well-established findings in cognitive psychology. We formalize this bias theoretically, verify it on preference datasets empirically, and show that it plays a central role in mode collapse. Motivated by this analysis, we introduce ***Verbalized Sampling (VS)***, a simple, training-free prompting strategy to circumvent mode collapse. VS prompts the model to verbalize a probability distribution over a set of responses (e.g., "Generate 5 jokes about coffee and their corresponding probabilities"). Comprehensive experiments show that VS significantly improves performance across creative writing (poems, stories, jokes), dialogue simulation, open-ended QA, and synthetic data generation, without sacrificing factual accuracy and safety. For instance, in creative writing, VS increases diversity by 1.6-2.1× over direct prompting. We further observe an emergent trend that more capable models benefit more from VS. In sum, our work provides a new data-centric perspective on mode collapse and a practical inference-time remedy that helps unlock pre-trained generative diversity.



26K 459

In Generative AI by Adham Khaled

## Stanford Just Killed Prompt Engineering With 8 Words (And I Can't Believe It Worked)

ChatGPT keeps giving you the same boring response? This new technique unlocks 2× more creativity from ANY AI model — no training required...

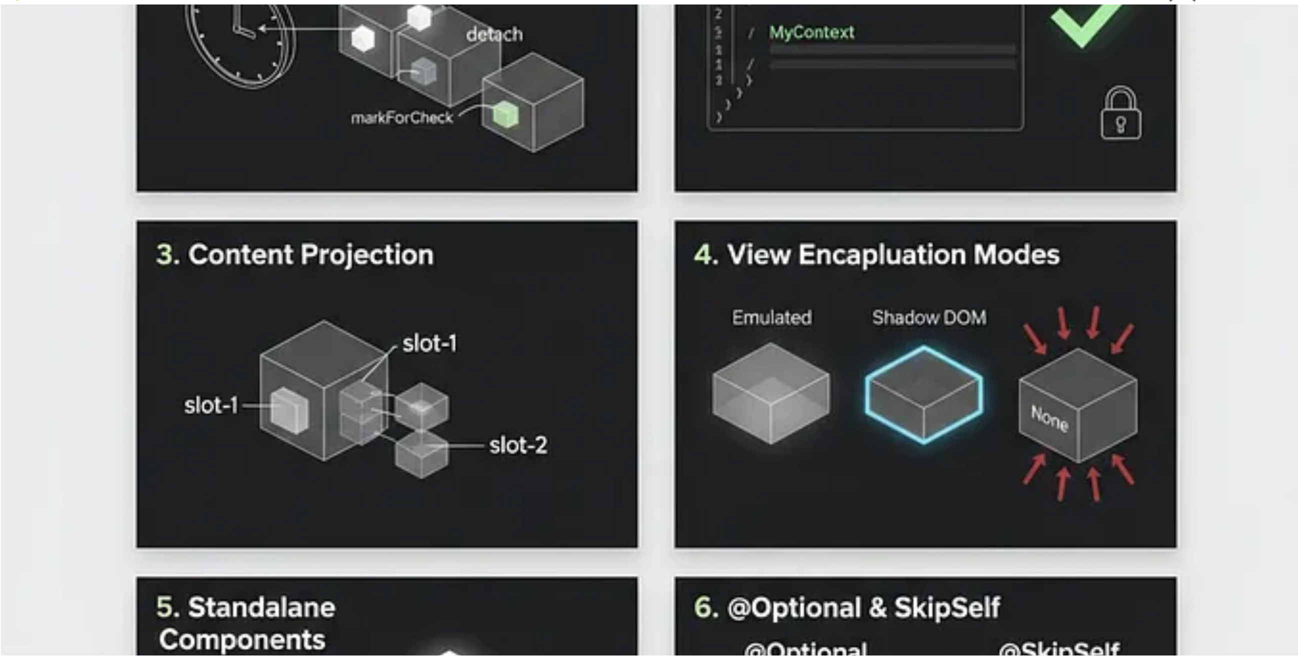
Oct 20, 2025 23K 606

Hartarto

## 15 Free OSINT Tools That Reveal Everything Online (2026 Guide)

Everything about you online leaves a trail. Emails, websites, servers, and devices continuously expose information — not because you were...

Jan 7 800 9

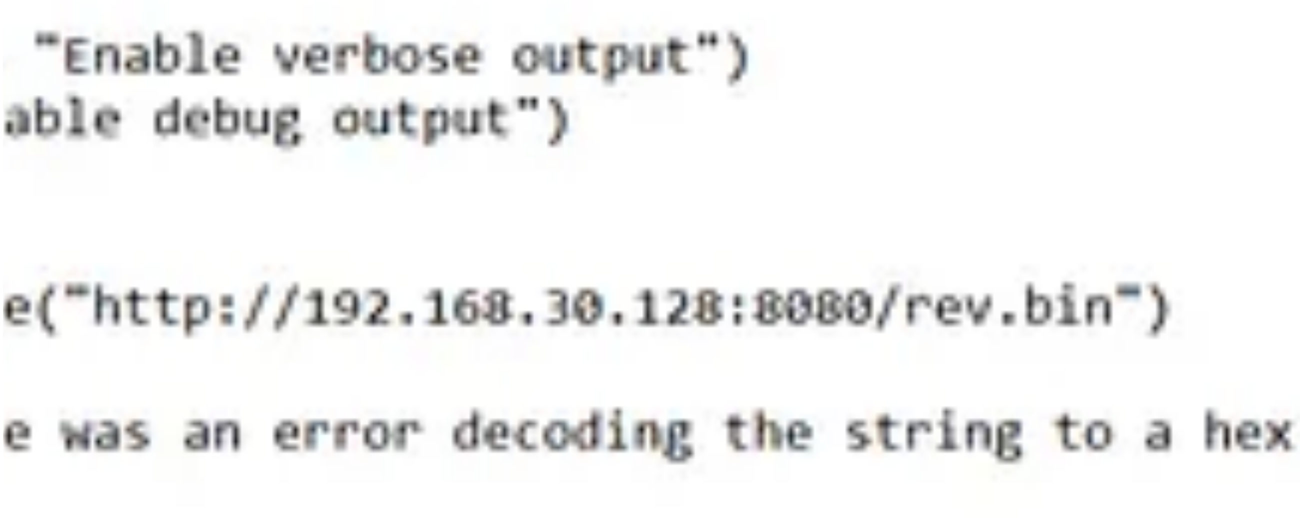



The Dialectic

## 6 Hidden Angular Features Most Devs Discover Too Late

Most Angular developers spend their early learning days fighting templates, routing, and services.

2d ago 61




lainkusanagi

**Using a Golang Shellcode Loader with Sliver C2 to Evade Antivirus**

Friend link if you aren't a member

Jan 17 46

See more recommendations