You Can See Me but You Can't Track Me: Evading Behavioral Detection with Distributed Malware and Covert Synchronization Channels

Marcus Botacin[1]

[1]Assistant Professor
Texas A&M University (TAMU), USA
botacin@tamu.edu
@MarcusBotacin
https://marcusbotacin.github.io/

Whoami

Education

- Assistant Professor @ TAMU (Since 2022)
- CS PhD @ UFPR, Brazil (2021)
- CSE/ECE BSc. + CS MSC @ UNICAMP, Brazil (2015, 2017)

Research

- **Malware** at high-level: ML-based detectors.
- **Malware** at mid-level: Sandboxes and tracers.
- **Malware** at low-level: HW-based detectors.

Current Project

- NSF SaTC: Hardware Performance Counters as the next-gen AVs.

Previously...

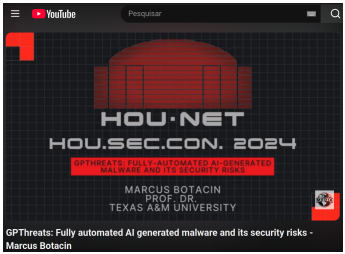


Figure: `https://www.youtube.com/watch?v=5lk_xklzcMg&t=12s`

Figure: `https://www.youtube.com/watch?v=HdWlyAzVcTI`

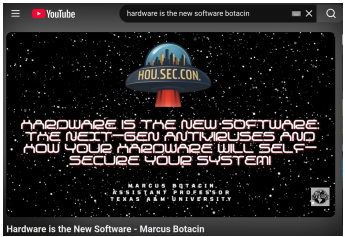
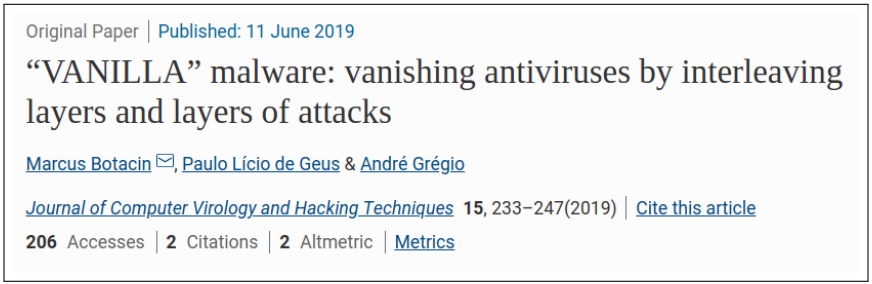


Figure: `https://www.youtube.com/watch?v=P3p--1csAws`

Today's Paper: Distributed Malware



Original Paper | Published: 11 June 2019

"VANILLA" malware: vanishing antiviruses by interleaving layers and layers of attacks

Marcus Botacin, Paulo Lício de Geus & André Grégio

Journal of Computer Virology and Hacking Techniques **15**, 233–247(2019) | Cite this article

206 Accesses | **2** Citations | **2** Altmetric | Metrics

Figure: **Source:** `https://link.springer.com/article/10.1007/s11416-019-00333-y`
Alternative Link: `https://marcusbotacin.github.io/publication/`

The Problem:

a.k.a. Academics seeing problems where they don't (really?) exist.

Motivation

- Malware increasingly targets modern multi-core systems.
- Detection solutions still largely operate in a serial manner.
- Runtime detectors commonly match sequences of API calls.
- Splitting malicious behavior across threads/processes may evade detection.
- Security solutions struggle to correlate distributed behavior.

Research Question

Can malware distribute its execution across multiple cores, threads, or processes and evade modern security solutions?

Why Multi-Core Malware Matters

- Multi-core processors are now ubiquitous.
- Malware historically adapts to platform evolution.
- Examples:
  - 32 $\rightarrow$ 64 bit malware
  - Desktop $\rightarrow$ IoT malware
  - Single-core $\rightarrow$ Multi-core malware
- Existing AVs and sandboxes are mostly serial.

Observation

Defenders still struggle with serial malware. Distributed malware introduces an additional layer of complexity.

It is Already Happening!

Sandbox Evasion.

Simple Core-Based Evasion

```
1  #define MINIMUM_CORES 2
2
3  int main() {
4
5    int cores =
6      GetMaximumProcessorCount(0);
7
8    if(cores < MINIMUM_CORES)
9        evade();
10 }
```

Implication

Simply checking available processors can reveal analysis environments.

Sandbox Fingerprinting via CPU Cores

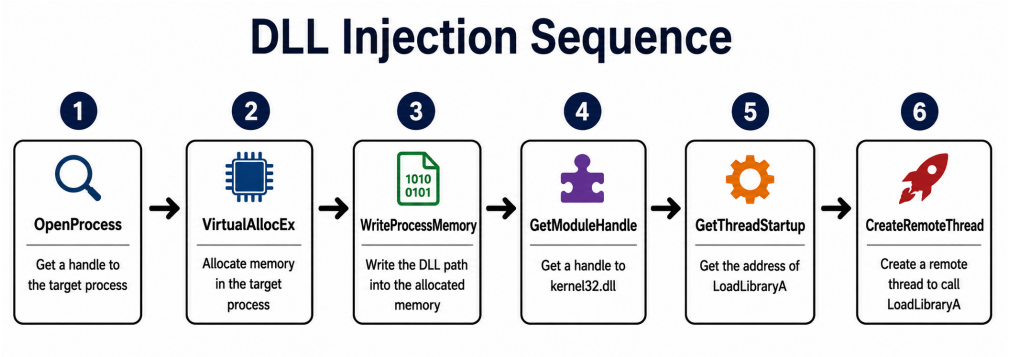
2017-06-15 02:46:53,163	**WriteConsoleA**	ConsoleHandle: `0x00000007` Buffer: `Number of processor cores: 1`

- Many malware sandboxes execute on single-core VMs.
- Malware can query the number of processors.
- Refuse execution when insufficient cores are present.
- Enables sandbox and emulator evasion.

It might get worse!

Distributed Attacks.

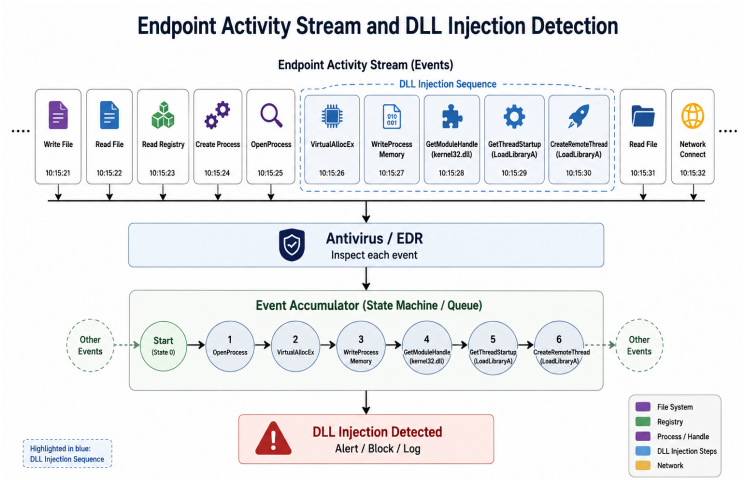
Serial DLL Injection Detection



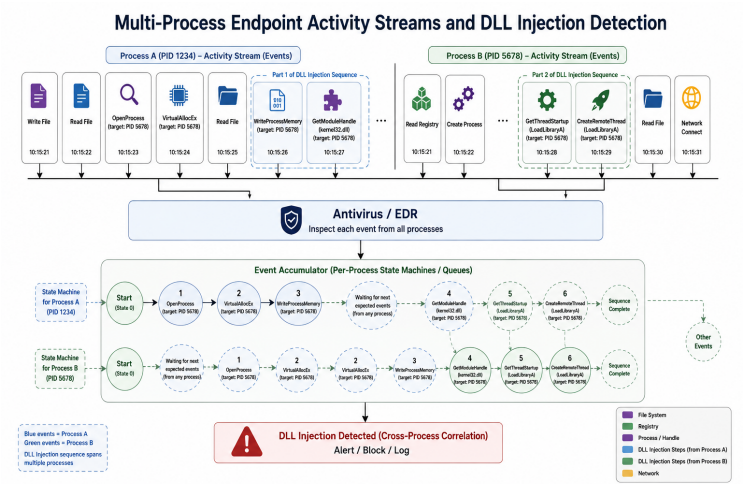
Observation

The API sequence itself becomes the signature.

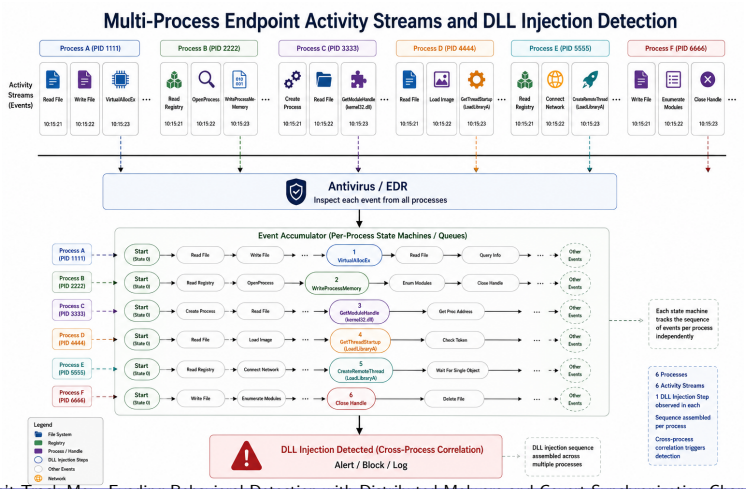
Serial DLL Injection Detection on a Single Process Stream



Serial DLL Injection Detection on a Two-Process Stream



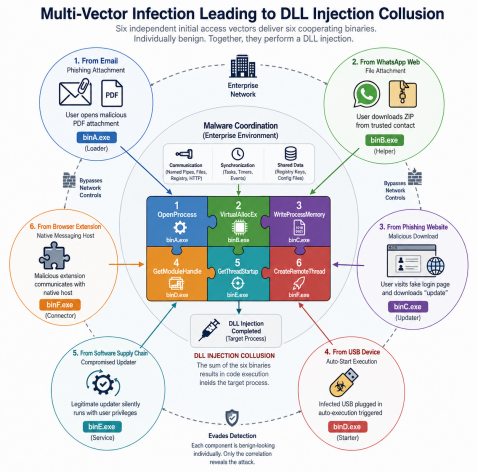
Serial DLL Injection Detection on a Three-Process Stream



How is this possible in my environment?

Multiple infection vectors.

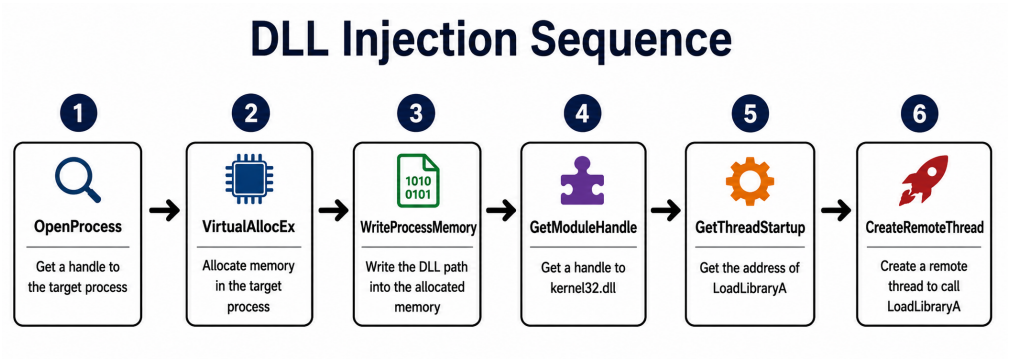
Infection Vectors



How one would implement such a malware?

Multiple options of increased stealthiness.

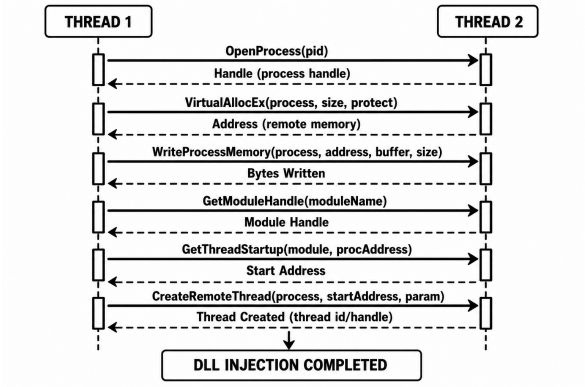
Recap: Serial DLL Injection



Distributed DLL Injection

Rationale

- Injection steps distributed across components.
- Each thread executes only a subset.
- Original malicious sequence disappears.
- Pattern matching becomes difficult.



What if we synchronize threads?

Lock-Based Distribution and Execution

- DLL injection split into multiple threads.
- Threads synchronized using:
  - Mutexes
  - Critical Sections
  - Condition Variables
- Original sequence never appears.
- Each thread executes only a fraction of the attack.

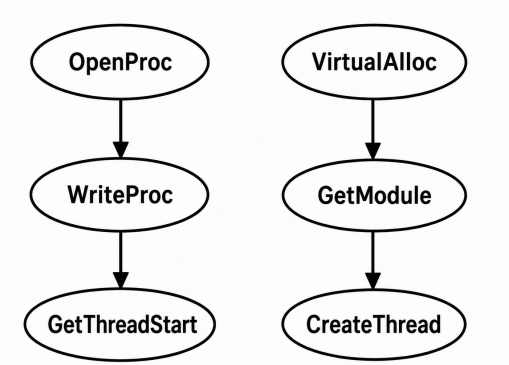


Figure: Thread 1's vs. Thread 2's Views.

Lock-Based Detection Vulnerabilities

Synchronization leaves observable artifacts.

```
1 EnterCriticalSection()
2 LeaveCriticalSection()
3
4 EnterCriticalSection()
5 LeaveCriticalSection()
```

- Thread interactions become visible.
- Lock transitions reveal execution flow.
- Correlation can reconstruct behavior.

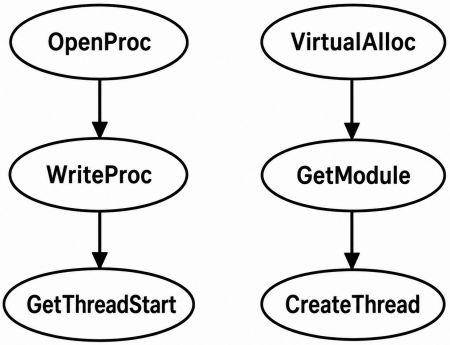


Figure: Thread 1's vs. Thread 2's Views.

What if we synchronize processes instead?

Process-Based Distribution & Detection

Distribution Strategy

- Functions distributed across processes.
- Components communicate using IPC.
- **Example Workflow:**
  - Process 1: `OpenProcess()`
  - Process 2: `VirtualAllocEx()`
  - Process 1: Remaining steps
- No single process exhibits full chain.

Detection Opportunity

- Process interaction requires sharing.

```
1 CreateFileMapping()
2 MapViewOfFile()
3 DuplicateHandle()
```

- Shared handles expose dependencies.
- Correlation enables detection.

Observation

Process separation alone is insufficient for stealth; IPC artifacts provide the missing link for behavior reconstruction.

What if we use busy waiting instead?

Busy-Wait Distribution & Memory Detection

Evasion Strategy

- Remove synchronization APIs.
- Replace locks with shared variables.
- Threads continuously poll memory.
- No mutexes or condition variables.

Goal

Eliminate observable synchronization calls entirely.

Detection (e.g., Intel PIN)

- Tracks cross-thread memory access.

```
1 TID1: reading foreign value
2 TID1: reading foreign value
3 TID1: overwrites foreign val
4 TID1: reading own value
```

- **Observed Interaction Patterns:**
  - **RFM:** Read Foreign Memory
  - **WFM:** Write Foreign Memory
  - **RWI:** Read Own Memory
  - **WOM:** Overwrite Own Memory

What if we use timers instead?

Time-Based Synchronization

Evasion Strategy

- Replace shared variables with timers.
- Threads sleep and wake up.
- No locks or shared-memory coordination.
- No explicit IPC.

Execution Flow (Thread 1)

```
1 OpenProcess(...);
2 // Wait for T2: VirtualAlloc
3 Sleep(OFFSET_1);
4 WriteProcessMemory(...);
5 // Wait for T2:
      GetModuleHandle
6 Sleep(OFFSET_2);
7 CreateRemoteThread(...);
```

Advantage

Removes both API-level and memory-level traces.

Limitation

Can fail in sandboxes that hook or accelerate `Sleep()`.

What if we use timing without a timer?

Side-Channel Synchronization & Detection

Evasion Strategy

- Synchronization through cache effects.
- No shared variables, locks, or IPC.
- No timing/sleep APIs.

Mechanism:

1. Thread performs memory action.
2. Action causes cache eviction.
3. Receiver detects cache variation.
4. Execution proceeds.

Detection (Timing Verification)

```
1 start = get_ticks();
2 tmp = *addr;
3 stop = get_ticks();
4 if ((stop-start) < THRESH){
5   is_cached = True
```

- Thread detects subtle cache misses.
- Covert channel leaves no OS footprints.

Most Stealthy Approach

Extremely difficult to correlate distributed components.

Synchronization Techniques Comparison

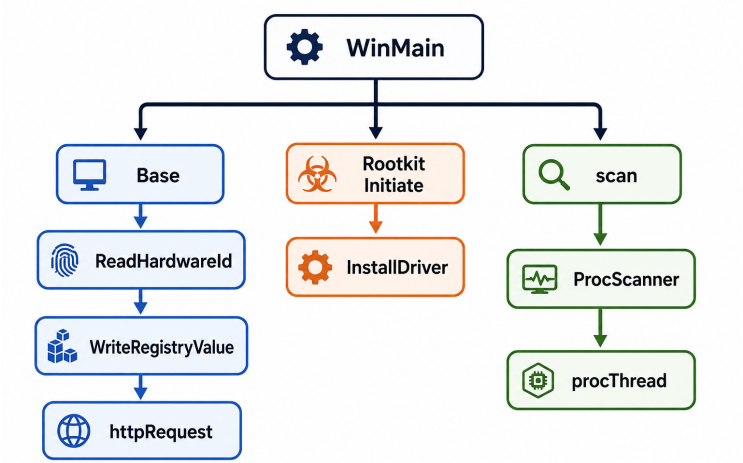
Method	API Trace	Memory Trace	Stealth	Stability
Locks	Yes	Yes	Low	High
IPC	Yes	Yes	Low	High
Busy Wait	No	Yes	Medium	Medium-High
Timers	No	No	High	Medium-Low
Side Channel	No	No	Very High	Low

Table: Security visibility decreases from top to bottom.

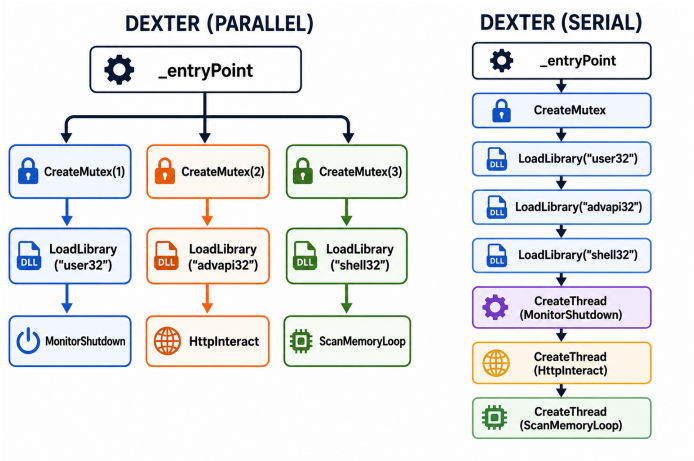
Would it apply to real malware?

Armoring 3 Real Samples.

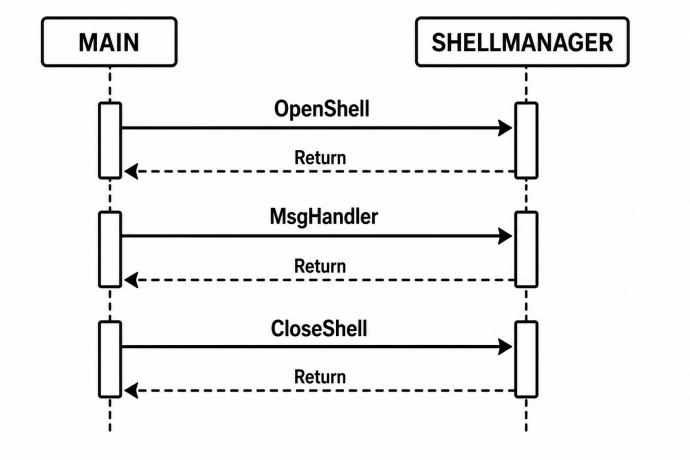
Alina Malware Distribution



Dexter Malware Distribution



Trochilus Distribution



But would it affect real AVs?

Testing on VirusTotal and more...

Static Detection Results

Finding

Simply distributing functionality significantly reduces static detection rates.

Key Observation

Code distribution alone affects AV visibility.

Malware	Serial	Threaded	Process
Alina	54%	4%	0%
Dexter	23%	5%	0%
Trochilus	2%	0%	0%

Dynamic Detection Results

Malware	Serial	Threaded	Process
Alina	100%	0%	0%
Dexter	100%	0%	0%
Trochilus	100%	0%	0%

Main Result

All evaluated AVs detected the serial malware.
None detected the distributed variants.

Complete Detection Results

Malware	Serial S	Serial D	Thread S	Thread D	Proc D
Alina	31/57	5/5	2/54	0/5	0/5
Dexter	14/61	5/5	3/61	0/5	0/5
Trochilus	1/58	5/5	0/57	0/5	0/5

S = Static Detection
D = Dynamic Detection

Root Cause

AVs monitor locally while malware behaves globally.

How Big is The Problem Right Now?

In-The-Wild Measurement.

Evidence from Real Malware

Analysis of 226,290 malware samples from MalShare.

Function	Occurrences	Samples
GetProcessAffinityMask	800	0.35%
SetThreadAffinityMask	301	0.13%
SetProcessAffinityMask	205	0.09%
GetLogicalProcessorInformation	105	0.05%
SetThreadIdealProcessor	9	0.00%

Finding

969 malware samples (0.43%) already contain thread/core-aware logic.

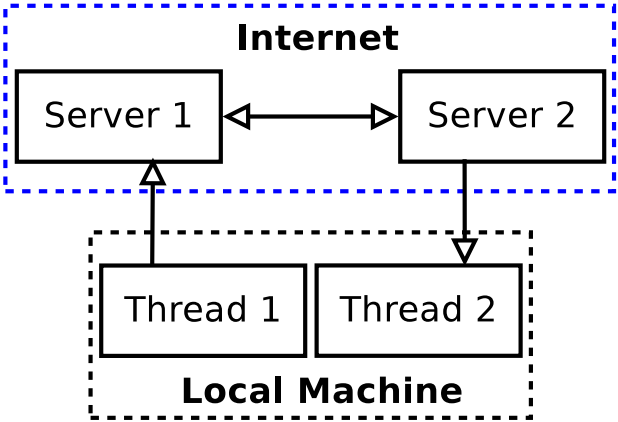
So, Why do we care?

Might become even bigger.

External Synchronization

Network Synchronization

- Components need not communicate locally.
- Independent servers can synchronize execution.
- Correlation becomes substantially harder.



Thank you!

botacin@tamu.edu
@MarcusBotacin
https://marcusbotacin.github.io/