W4N4D13

Thomas Dollé Juliette Favereau

October 27, 2026

Professor: Guillaume Bonfante

Abstract

This document presents our work on designing a malware as part of the Malware/Reverse Engineering course (Master SIRAV). The program is requiring an 8-character key in $[0\text{-}9a\text{-}zA\text{-}Z]^8$. Both the verification logic and the correct key must be obscured. We implemented a bytecode virtual machine. We focused our protections against dynamic instrumentation and disassembly-based analysis. Our primary defensive mechanism is a watchdog based on heartbeats. We acknowledge Christian Collberg and his book Surreptitious Software for the many ideas we implemented (and for those we did not have time to implement). We also thank the students from Télécom Nancy for interesting discussions on the topic.

Contents

1 Construction of the Malware **2**

2 Static Obfuscation **2**

3 MBA Generation and Permutation Polynomials **2**

4 Malware Protection **3**

1 Construction of the Malware

The core of the software contains the license verification routine. It is a slightly modified SHA-256 implementation designed to operate on four 32-bit integers, used as a 128-bit array. These constraints are tied to technical details that we discuss later.

The build/obfuscation pipeline used in the project is as follows:

Listing 1: Obfuscation pipeline used to build the final binary.

```
gcc -E in > out

./main.native in out -flatten
./main.native in out -opaque
./main.native in out -virtualize

python3 generate_vm.py in out
```

2 Static Obfuscation

A first obfuscation pass is applied afterwards: *control-flow flattening* (CFF). The idea is to completely flatten the control flow into a single `switch` statement inside an infinite loop. The program is thus transformed into a state machine.

This first step does not truly resist modern tools, which is why we enrich it with opaque predicates.

For each case of the global `switch`, we transform:

$$\texttt{next} = x \text{ into } \texttt{next} = O^x,$$

where O^x is an expression that always evaluates to x, but is difficult to simplify.

We chose *mixed Boolean-arithmetic* (MBA) constructions. A simple example is:

$$x + y = (x \oplus y) + 2 \cdot (x \wedge y),$$

which illustrates the general principle: simple arithmetic expressions can be rewritten as combinations (often linear combinations) of Boolean operations.

Modern tools can break many instances of linear MBA. Therefore, we introduce non-linearity using permutation polynomials to post-process the linear MBA expression.

To protect this part of the code as much as possible, we virtualize it by turning the main program into a bytecode interpreter. Our compiler targets a small subset of C, which explains why we could not use `struct`, arrays, or overly complex data structures.

For operations that are too complex, the program provides an oracle via the `CALLX` instruction, which allows calling host functions through the VM API.

To make decoding as difficult as possible, we apply a semantic permutation after each opcode execution through an affine pseudo-random permutation (shared with the compiler).

3 MBA Generation and Permutation Polynomials

In our approach, finding such MBA expressions is reduced to solving a linear system in the ring $\mathbb{Z}/(2^{32}\mathbb{Z})$. Because $\mathbb{Z}/(2^{32}\mathbb{Z})$ is not a field, this provides some additional difficulty when attempting to recover the original expression.

In practice, we build a basis of bitwise expressions over a small number of variables including: $a_i \oplus a_j$, $a_i \wedge a_j$, $a_i \vee a_j$ and their bitwise negations $\sim(\cdot)$.

We then solve for coefficients c_k such that a weighted sum of these basis columns equals a constant truth-table vector (constant y repeated), all modulo 2^{32}. Among the solutions, we select the most voluminous one (heuristically: the solution using the most nonzero coefficients), which tends to yield expressions with more terms.

Let R be a finite ring and let $f(x) \in R[x]$ be a polynomial. We say that f is a permutation polynomial on R if the induced function is a bijection from R to R. Over finite fields, the theory of permutation polynomials is classical. Over rings the situation is subtler but still there are useful characterizations and constructive methods.

In our implementation, we work over:

$$R = \mathbb{Z}/2^w\mathbb{Z} \text{ with } w = 32$$

A polynomial $f(x) = \sum_{k=0}^{d} a_k x^k$ is evaluated modulo 2^w, and we enforce conditions ensuring the map is a permutation of R.

For $n = 2^w$, we generate random coefficients under the following constraints:

- a_1 is odd,
- the sum of even-index coefficients $\sum_{k\geq 2,\ k \text{ even}} a_k$ is even,
- the sum of odd-index coefficients $\sum_{k\geq 3,\ k \text{ odd}} a_k$ is even.

These parity constraints completely define permutation polynomials over the ring $\mathbb{Z}/n\mathbb{Z}$.

A technical aspect of working with functions $\mathbb{Z}/2^w\mathbb{Z} \to \mathbb{Z}/2^w\mathbb{Z}$ represented by polynomials is that distinct polynomials can induce the same function. To work reliably with polynomial composition and inversion, we maintain a canonical representative.

We use a known construction based on falling factorial polynomials:

$$(x)_i = \prod_{j=0}^{i-1} (x - j)$$

$$P_i(x) = \frac{n}{\gcd(n, i!)} \cdot (x)_i$$

which vanish modulo n beyond a certain degree threshold. We compute the smallest d such that the 2-adic valuation $v_2(d!) \geq w$; then degrees $\geq d$ can be eliminated using P_d, and remaining coefficients can be reduced modulo the leading coefficient constraints derived from $\frac{n}{\gcd(n,i!)}$. This yields a canonical form in the quotient structure used by the implementation.

Given a permutation polynomial $P(x)$ modulo 2^w, we want an inverse polynomial $P^{-1}(x)$ such that:

$$P^{-1}(P(x)) = x$$

We repeatedly compute:

$$g(x) = \text{inv}(P(x)) \circ P(x)$$

check whether g is the identity, and otherwise update the candidate inverse by composing it with g. The process is iterated up to w rounds.

After solving a modular linear system to obtain a linear MBA expression, we apply a non-linear layer by composing with a permutation polynomial and its inverse :

$$\text{result} = P(P^{-1}(\text{comb_lin}))$$

4 Malware Protection

In addition to building a complex structure, we implemented different obfuscation techniques to create more obstacles and slow down the analysts' work. Specifically, the Virtual Machine protection operates across three distinct levels: anti-debugging, honeypots, and anti-PIN mechanisms. In the event of debugger detection, the program triggers a slightly altered execution path for the

VM, which intentionally omits the opcode decoding phase. Consequently, the executed bytecode loses all semantic meaning, causing the debugger to crash via a segmentation fault (*segfault*) and effectively preventing dynamic instrumentation tools like Intel PIN from continuing their analysis.

Anti-debugging: Program Self-debugging

We wanted to detect debugging to modify our program's behavior if necessary. Since a program cannot be debugged by more than one instance, we made the execution of our program dependent on a child process that debugs our parent process. If the child process is not executed, it means we are being debugged.

To achieve this, upon startup, the main process (parent) launches a copy of itself (the child). This child process uses the system API `DebugActiveProcess` to immediately attach to the parent as an exclusive debugger.

Synchronization between the two processes relies on Windows API Event Objects (Windows Events):

- The parent creates a uniquely named event (using its own PID) to avoid collisions. During its execution, it periodically waits for this event to be signaled using the `WaitForSingleObject` function with a Timeout.
- At regular intervals, the child calls the `SetEvent` API to signal the event created by the parent, proving that it is still alive and active.

We verify this works by executing the `.exe` alone; the Heartbeat signal is successfully received and the main process continues. (Figure 1).

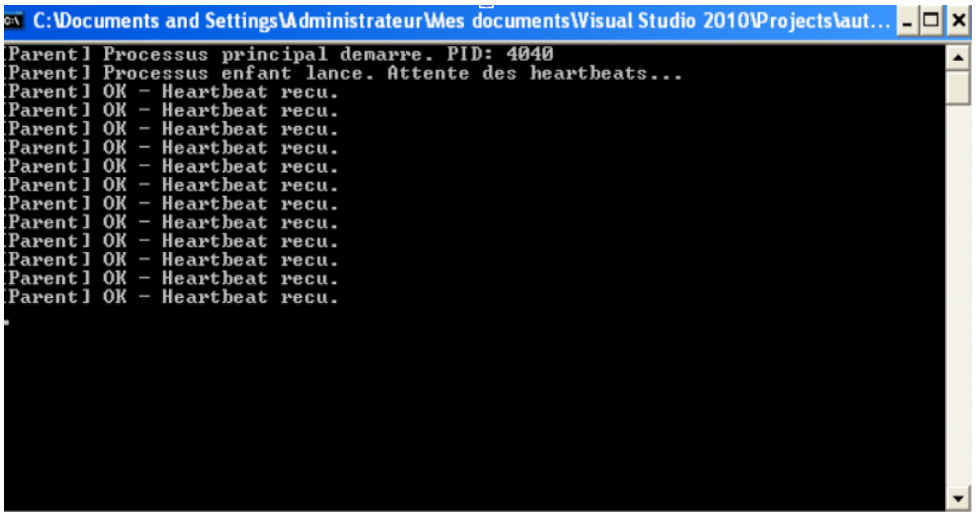


Figure 1: Normal execution showing successful heartbeat reception.

Conversely, if we open the `.exe` with IDA and launch the program in Debug mode, the child can no longer launch and the `SetEvent` calls are not emitted. The parent's wait time expires (Timeout reached), which instantly indicates that its protection has been altered. This is a sign that it is being debugged, prompting a behavior change to deceive the analyst. (Figure 2).

The problem with this technique is that it is very easy to counter in several ways: (Figure 3).

- Dynamically, the analyst can skip the conditional jump instruction.
- Dynamically, the analyst can modify the register value being compared on the fly.
- Dynamically, the analyst can simulate the `SetEvent` from our child process.
- Statically, the analyst can patch the conditional jump with a nop.

To make the analyst's work more tedious, we implemented three countermeasures to protect the heartbeat:

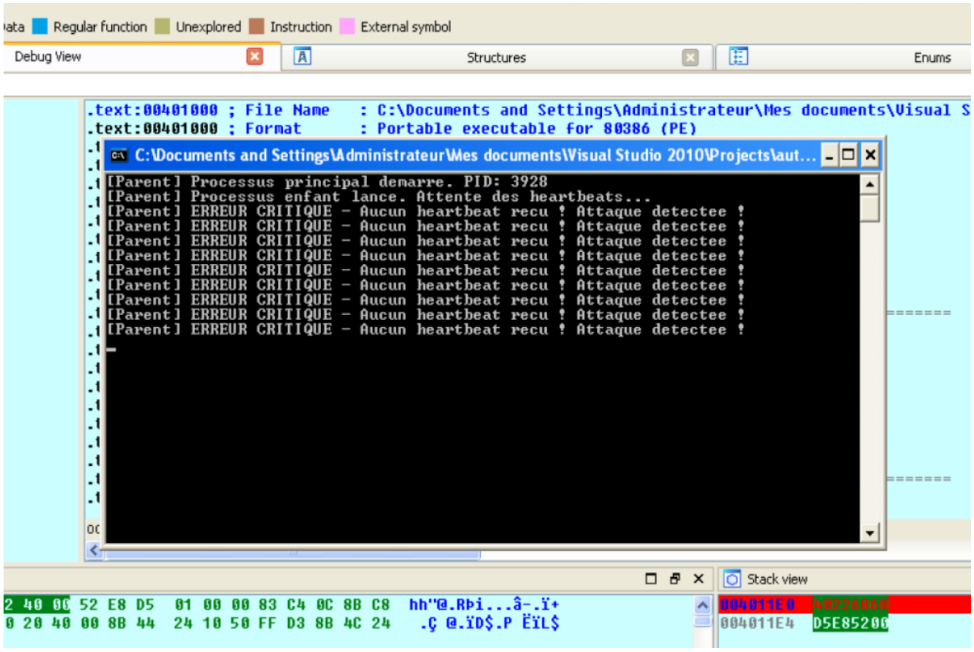


Figure 2: Execution under IDA Pro resulting in a critical error (attack detected).

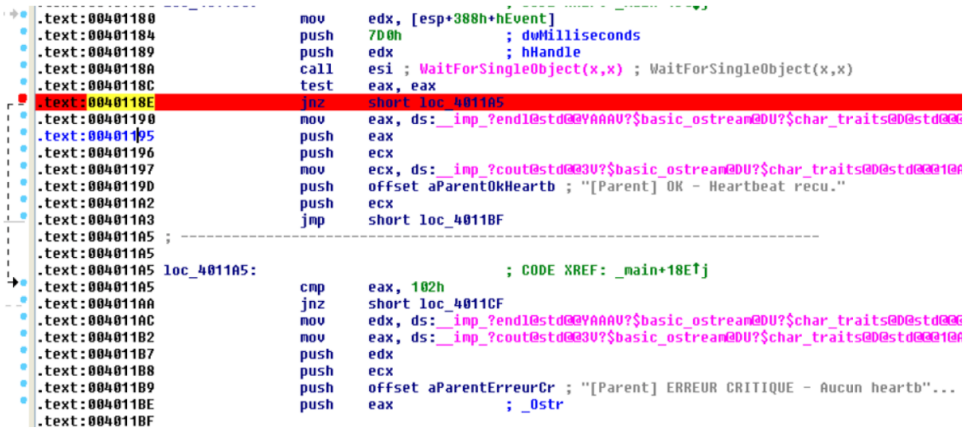


Figure 3: Disassembly view in IDA Pro showing the conditional jump associated with the heartbeat check.

1. Replace `Heartbeat_Event_<PID>` with a name derived from a random secret `DWORD` key generated by the parent and transmitted to the child as `argv[2]` in hexadecimal.
2. The parent deposits a random nonce in a named shared memory. The child reads the nonce, calculates `response = ComputeResponse(nonce, secretKey)`, writes the response to the shared memory, and then signals the event. The parent verifies the response before accepting the heartbeat.
3. Before calling `WaitForSingleObject`, the parent reads the first 5 bytes of the function in memory and compares them to the expected prologue on Windows XP x86: `8B FF 55 8B EC` (`MOV EDI,EDI / PUSH EBP / MOV EBP,ESP`). An inline hook would replace these bytes with a `JMP` (`0xE9...`).

Hiding the VM (Honeypots)

The idea is to disguise operand switch operations as AntiDebug to distract the analyst from our program. Since our program self-debugs via its child process, the AntiDebugs are fake, but we simulate credible behavior. The real protections are the challenge-response and the anti-hook, while the goal here is to add honeypots.

In the VM's opcode handlers, they simulate credible anti-debug checks but never change the VM's behavior. An analyst will see them and lose time tracing them, thinking they are the real protection mechanisms.

Implemented Honeypots:

1. `op_nop` - `NtGlobalFlag` (PEB offset `0x68`): reads the `NtGlobalFlag` field of the PEB via `__readfsdword(0x30) + 0x68`. When a debugger is present, the `0x70` bits are set.
2. `op_push` - Timing anti-single-step: a delta > 50 ms between two successive `PUSH` instructions indicates single-stepping.
3. `op_load` - Heap debug flags: On XP x86, `_HEAP.Flags` is at offset `0x0C`. During debugging, the value contains additional bits beyond `HEAP_GROWABLE` (`0x02`).
4. `op_store` - `IsDebuggerPresent`: the classic check.
5. `op_jmp` - Scan `0xCC` (`INT3`) in the function's prologue: verifies the first 16 bytes of the `op_jmp` handler itself to detect a software breakpoint set by a debugger.
6. `op_callx` - `NtQueryInformationProcess` `ProcessDebugPort` (class 7): If `ProcessDebugPort != 0`, a debugger is attached.
7. `op_ret` - `NtQueryInformationProcess` `ProcessDebugObjectHandle` (class `0x1E`): A more discreet variant that obtains the debug object's handle. It is non-null if debugged.

Blocking PinTools

The idea is that the PIN tool modifies our program's execution time. To counter it, we measure the ratio between the execution times of our function and a system function; if this ratio is modified too much, we change our program's behavior.

Intel PIN is a Dynamic Binary Instrumentation framework that JIT-compiles user-mode code by inserting analysis functions. It cannot instrument the kernel. Consequently, the user-mode code is slowed down by 10-50x, but the time spent in the kernel during a syscall remains almost unchanged.

We exploit this imbalance by measuring a ratio:

- t_{user}: `RDTSC` around a pure arithmetic loop (user-mode).
- t_{kernel}: `RDTSC` around N `CloseHandle(INVALID_HANDLE_VALUE)` calls (kernel-dominated).

Normally $t_{\text{user}}/t_{\text{kernel}} \approx 1$. Under PIN, t_{user} explodes (x20), and t_{kernel} is stable.

References

[1] Christian Collberg, Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley Professional, 2009.

[2] Justus Polzin. *Mixed Boolean Arithmetics.*