CERT-AGID

Analisi su tecnica anti-debug e anti-vm individuate in un packer VB6

2020-05-14

Table of Contents

Contesto

Tra gli analisti di malware è frequente l'uso di un packer scritto in VB6, la cui caratteristica è quella di contenere codice nativo o P-CODE senza alcun senso. Il vero codice malevolo è invece contenuto in una sezione di memoria distinta dal codice VB6 e viene richiamato tramite un'opcode di chiamata nativa (nel caso di P-CODE) o tramite l'instruzione `call` (l'unica che non ha come target funzioni censite nel progetto VB6).

La particolarità di questo packer, già osservato per AgentTesla, Trickbot e Gotkit, riguarda la serie di tecniche **anti-debug** ed **anti-vm** utilizzate, al punto che tutto il codice malevolo è dedicato all'implementazione di queste tecniche. Solo una breve parte finale eseguirà il payload.

Fin'ora le due modalità di esecuzione sono state quella del *process hollowing* ed il download di un payload da internet, successivamente decodificato[1].

Agli inizi di maggio 2020 è stata analizzato un campione, utilizzato per veicolare Ave Maria, che conteneva due tecniche interessanti:

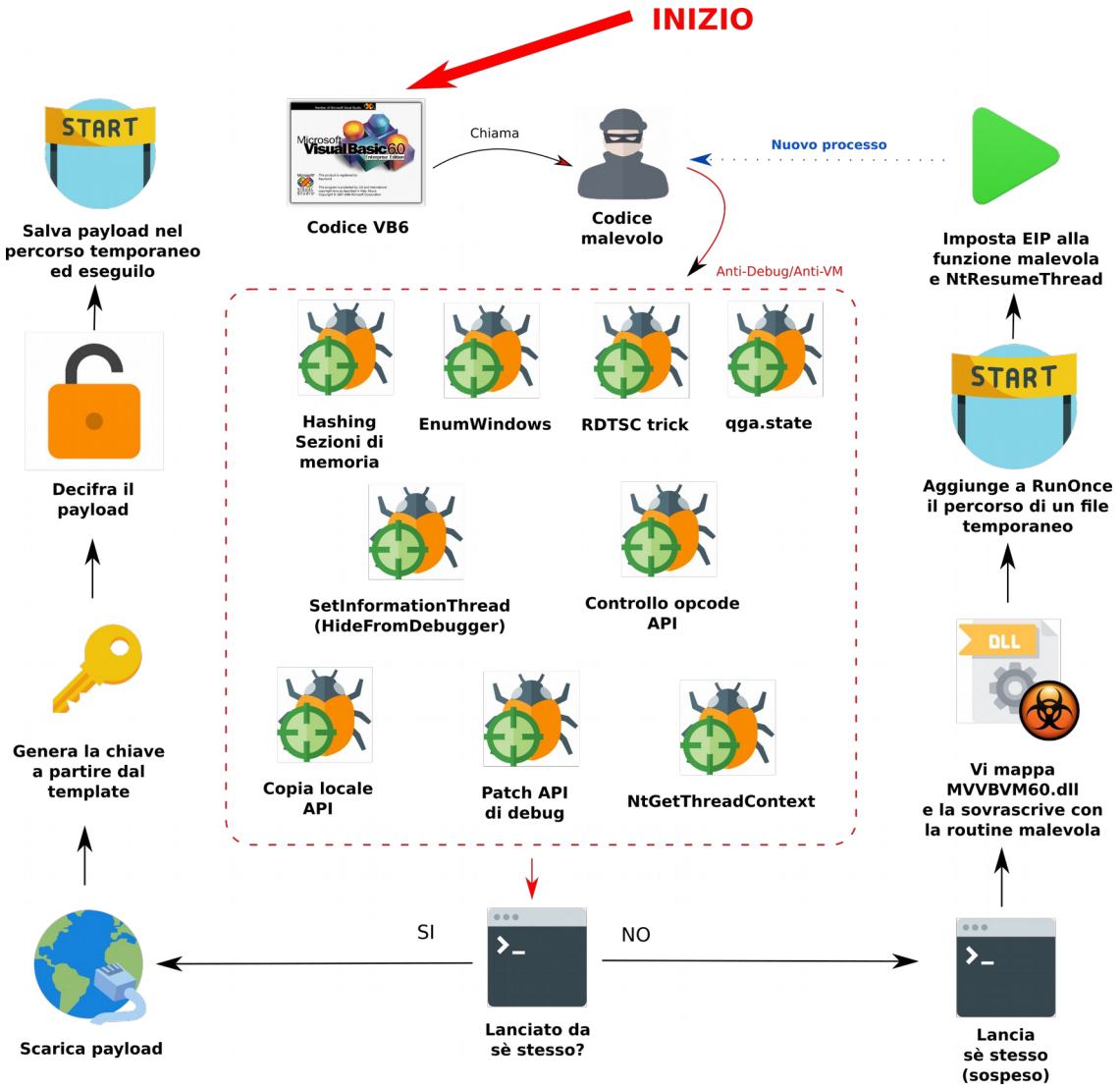
1. la prima, una tecnica **anti-debug**, era già stata osservata in una precedente variante;
2. la seconda è una tecnica **anti-vm** basata sulla profilazione di *cpuid* con *rdtsc*. Questa non è certo una tecnica nuova, l'aspetto interessante risiede nei parametri usati per il tuning dell'algoritmo di rilevazione di una VM. Rispetto alle tecniche anti-vm tradizionali, che si basano più che altro sulla presenza di artifatti visibili a livello di OS, questa si basa sulla presenza di artifatti visibili a livello architetturale[2].

1 Il primi 64 byte del payload sono scartati e la decodifica avviene tramite uno xor con una chiave di 604 bytes creata a partire da un template.

2 Indica lo stato della CPU osservabile dal software come definto dall'ISA, nello specifico dai manuali Intel e AMD.

Panoramica

Di seguito esposta una panoramica del funzionamento del packer VB6.



Rispetto alle varianti già analizzate va evidenziato:

- che il packer esegue sè stesso, mappandovi il proprio codice malevolo nel nuovo processo (camuffato da MSVBVM60.dll) e redirottando il thread principale affinchè l'esecuzione parta da suddetto codice anzichè dall'entry-point.
- sono presenti nuove funzioni anti-debug oggetto di questo documento:
  - *Hashing sezioni di memoria*
  - *RDTSC trick*

- il payload è cifrato con **xor**, I primi 64 byte sono scartati, la chiave di 604 byte generata a partire da un template di 604 byte (come la chiave) nel seguente modo:
  - un contatore a 16 bit viene inizializzato a zero ed incrementato di due fino al wrap around;
  - per ogni valore viene calcolato lo xor tra i primi due byte del payload (tolti i 64 byte iniziali): i primi due byte del template e del contatore. Se il valore ottenuto è 'MZ', allora il valore a 16 bit del contatore è usato per fare uno xor sul template. Il risultato è la chiave!

Anti-debug

Proseguiamo la panoramica con una breve descrizione delle tecniche anti-debug. Tranne quelle oggetto di questa discussione, queste non verranno dettagliate in quanto ben note in letteratura.

Hashing sezioni di memoria

Questa tecnica consiste nell'analizzare le pagine di memoria mappate nel processo e applicarvi una funzione di hash. Se l'hash corrisponde ad un valore di un insieme, l'esecuzione è redirottata verso del codice non valido. Questa tecnica è interessante perchè consente di rilevare la presenza dei plugin anti-debug (come il famoso Scylla).

EnumWindows

Usa questa chiamata API per verificare la presenza di almeno 12 finestre. I programmi eseguiti in una sandbox di solito non passano questo controllo.

RDTSC trick

Cronometra, con rdtsc, l'esecuzione di `cpuid` e verifica se rientra in un range di valori. Le VM falliscono questo test perchè `cpuid` causa un VM-exit.

Qga.state

Controlla la presenza di questo file nella cartella di QEMU.

NtSetInformationThread (HideFromDebugger)

Nasconde il thread dal debugger (rendendolo non più controllabile).

Controllo opcode API

Controlla che le API più usate (come `VirtualAlloc`) non abbiano breakpoint (opcode `0cch`), utilizza una chiamata far per passare ad un segmento di codice a 64 bit causando un crash del debugger.

Copia locale API

Per i programmi WOW64 le chiamate API usano `WOW32Reserved` nel TEB, la loro implementazione è quindi piuttosto corta e position-indipendent. Il malware può quindi facilmente

copiare il codice dell'API in un buffer temporaneo ed eseguirlo da lì (evitando breakpoint hardware).

Patch API di debug

Ripristina le prime istruzioni delle API usate dal processo di debugging (es: `DbgUiRemoteBreakin`, ZwReturnCallback) per liberare il programma dal controllo del debugger.

NtGetThreadContext

Usa quest'API per ottenere il valore dei registri DR0-7 e rilevare la presenza di breakpoint hardware.

Hashing delle sezioni di memoria

Questa tecnica anti-debug è interessante perchè efficace contro plugin come Scylla. Risulta difficile per un debugger non lasciare artefatti all'interno dell'address-space del programma debuggato, a maggior ragione per il fatto che i plugin, per nascondere il debugger, necessitano di modificare il valore ritornato da alcune API e simularne altre, per fare ciò devono necessariamente mappare del codice nel programma in esame.

L'algoritmo può essere riassunto con il seguente pseudo codice.

```
int hash(char* ptr)
{
  int a = 0x1505;
  int b;

  while (*ptr)
  {
    b = a;
    a = (a << 5) + b + *ptr++;
  }

  return a;
}

void check(int* what)
{
  for (x = 0x10000; x <= 0x7ffff000; x += 0x1000)
  {
    NtQueryVirtualMemory(0xffffffff, x, 0, &info, 0x1c, 0);
    if (not_in(info->protection, 0x10, 0x20, 0x40, 0x2, 0x4))
      continue;

    ;Parte da 0xfff e va indietro fino a trovare il primo byte nullo
    for (y = 0xfff; y >= 0 && x[y]; y--);
    if (y < 0)
      continue;

    ;ecx = z
    ;ebx = y
    ;esi = x

    ;Parte da 0 e va avanti fino a trovare il primo byte non nullo
    for (z = 0; z < y && !x[z]; z++);
    if (z >= y)
      continue;

    w = hash(x + z);

    for (int i = 0; what[i] != -1; i++)
        if (what[i] == w)
        {
            Stop
        }

  }

  return 0;
}
```

Di ogni pagina nell'address-space, da `10000h` a `7ffff000h`, viene controllata la presenza e gli attributi, se presente e di interesse, vengono definiti due puntatori: uno di inizio ed uno di fine, ed entro questo intervallo viene calcolato un hash. Se il valore rientra tra quelli presenti nella lista `what`, l'esecuzione viene redirottata verso del codice non valido. Notare che la lista `what` termina con il valore `0ffffffffh` (-1 se inteso come intero a 32 bit in complemento a due).

Efficacia

Ottima, rileva I debugger più popolari.

Contromisure

Per eludere questo controllo è sufficiente:

1. Impostare un breakpoint su `NtQueryVirtualMemory` (HW o SW, non fa differenza). Notare che essendo questo il primo controllo anti-debug del codice malevolo, questi breakpoint sono anche un modo veloce per trovare suddetto codice. Quest'API è chiamata anche dal runtime VB6 ma esaminando l'indirizzo di ritorno è facile trovare la chiamata di interesse. Una volta interrotto il processo, ritornare dall'API al codice malevolo (CTRL+F9 in x86dbg).
2. Esaminare in una finestra di memoria/dump il valore di `ESP` in modo da trovare gli argomenti della funzione. Si noterà l'indirizzo di ritorno seguito da una serie di DWORD terminate dal valore `0ffffffffh`. Il valore di questi hash è riportato sotto con il terminatore evidenziato in rosso.

```
0018F418  6C C7 9C 2D 12 8F CB DF 88 31 AA 27 20 D9 1F F2  lÇ.-..Ëß.1ª' Ù.ò
0018F428  E6 AD 17 3E 5B 18 21 7F FF FF FF FF 64 F7 18 00  æ..>[.!.ÿÿÿÿd÷..
```

3. Cambiare la prima DWORD (quella in blu) in `0ffffffffh` e riprendere l'esecuzione.

RDTSC trick

Questa tecnica è usata per determinare la presenza di una VM.

Quando in VMX non root mode (ovvero, quando si esegue codice in una VM) alcune istruzioni causano un VM-exit. Istruzioni come `rdstc` possono causarne uno condizionalmente (in base alle impostazioni nel *VMCS*), come riporta il manuale di Intel.

> RDTSC. The RDTSC instruction causes a VM exit if the "RDTSC exiting" VM-execution control is 1.

Altre istruzioni, come `cpuid`, generano un VM-exit **incodizionalmente**:

> The following instructions cause VM exits when they are executed in VMX non-root operation: CPUID, GETSEC, 1
> INVD, and XSETBV. This is also true of instructions introduced with VMX, which include: INVEPT, INVVPID,
> VMCALL, 2 VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMRESUME, VMXOFF, and VMXON.

Dato che un VM-exit è piuttosto oneroso in termini di tempo (poichè, oltre al dispatching all'handler dell'hypervisor c'è anche la gestione, da parte di questi, dello stato e dei metadati della VM), è possibile determinare la presenza di una VM dal tempo di esecuzione di `cpuid`. Il difficile è calibrare i parametri in modo che il controllo non ritorni falsi positivi o negativi.

Qualsiasi "foglia" di `cpuid` va bene, tuttavia la "foglia 1" risulta particolarmente utile perchè è usata nella paravirtualizzazione.
La foglia 1 di cpuid ritorna in ECX una serie di bit corrispondenti alle funzionalità della CPU, in particolare il bit 31 di ECX è stato marcato, da Intel e AMD, come riservato ed è quindi sempre di valore 0.
Gli hypervisor hanno stabilito la convenzione di modificare il valore ritornato da cpuid (cosa che possono fare in quanto questa genera un VM-exit) in modo che il bit 31 di ecx sia 1. Questo indica all'OS la presenza di un hypervisor con paravirtualizzazione attivata. Notare che il test del bit 31 basterebbe a determinare la presenza di una VM ma tutte gli strumenti di analisi seri hanno questa funzionalità disabilitata.

Il motivo per cui la foglia 1 è più utile di altre è perchè questa incorre necessariamente in un post processing da parte dell'hypervisor: anche nel caso in cui la paravirtualizzazione sia disabilitata, l'handler deve comunque leggere le impostazioni della VM per determinare se modificare o meno il valore. Questo fa sì che la foglia 1 sia particolarmente lenta rispetto ad altre foglie (tipo la 0) che possono passare il valore inalterato.

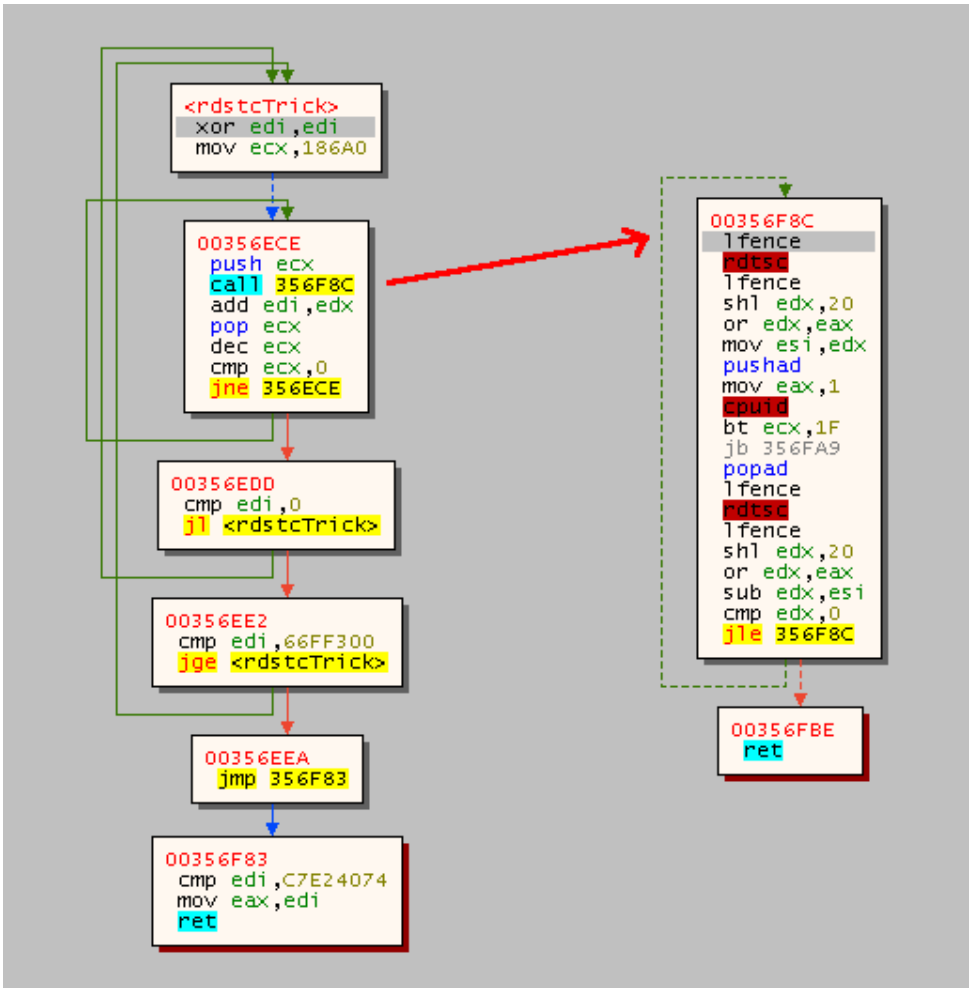
Questa tecnica consiste nel misurare il tempo di esecuzione di cpuid tramite l'istruzione `rdtsc` (che, da varie generazioni a questa parte adesso rappresenta il numero di cicli di un contatore fisso e non del core della CPU. Si veda TSC-invariant nei manuali Intel) un numero elevato di volte e sommare questi valori. Se il risultato è fuori da un range stabilito, il controllo è ripetuto, generando un ciclo infinito. Notare che, come visto prima, `rdtsc` non genera necessariamente un VM-exit, permettendo una misura accurata (molti hypervisor non hanno motivo di interferire con rdtsc, anche per via del caso di performance).

Ci sono inoltre degli accorgimenti, ben noti per chi programma in assembly per x86, da adottare quando si usa `rdtsc`, in particolare non essendo serializzante è necessario racchiuderla tra due istruzioni serializzanti (per evitare che il backend out-of-order delle CPU Intel riordino `rdtsc` con istruzioni precedenti e successive, falsando la misurazione). Ci sono pochissime istruzioni

(realmente) serializzanti che siano anche eseguibili in user-mode (CPL 3), una di queste è (casualmente) `cpuid` ma è poco utile perchè sovrascrive quattro registri GP.

Un'alternativa più utile è `lfence`, che non viene eseguita finchè le istruzioni precedenti non sono completate localmente.

Il codice del controllo è mostrato di seguito.



Notare che in questo caso specifico c'è una dependency-chain tra le prime quattro istruzioni dopo la seconda `lfence` e `rdtsc` per cui il backend non avrebbe comunque potuto riordinarle. Analogamente, essendo `cpuid` serializzante, questa non può essere riordinata. Per cui l'istruzione `lfence` dopo il primo `rdtsc` è inutile.

Questo può essere tradotto in C nel seguente modo:

```
int get_counter_delta_for_cpuid();

int loop_if_in_vm()
{
  int accumulator = 0, i;

  do
  {
```

```
    for (accumulator = 0, i = 0; i < 100000; i++)
      accumulator += get_counter_delta_for_cpuid();
  }
  while (accumulator < 0 || accumulator >= 108000000);


  return accumulator;
}
```

Dove la funzione `get_counter_delta_for_cpuid` è identica a quella mostrata in figura sopra ma con alcuni accorgimenti per renderla conforme alla convenzione di chiamata C.

```
BITS 32

GLOBAL _get_counter_delta_for_cpuid

%macro get_counter 0
  ;We need to sandwitch rdtsc between lfences to prevent OoO
  ;Technically lfence is not serializing (like cpuid is) since
  ; it will only wait for previous instruction to complete LOCALLY.
  ; But there are no stores, so it's ok
  lfence
  rdtsc
  lfence

  ;Combine the high (edx) and low (eax) part of the counter
  shl edx, 20
  or edx, eax
%endm

SECTION .text

_get_counter_delta_for_cpuid:
  push edx
  push esi

.loop:
  get_counter
  mov esi, edx


  ;Execute cpuid inside the profiling harness
  pushad
  mov eax, 1
  cpuid
  ;Test bit 31 of ecx (hypervisor present)
  bt ecx, 31
  jc .hv_present             ;But... do nothing, this can be disabled anyway
                                                ;Probably a leftover, not needed
.hv_present:
  popad

  get_counter

  sub edx, esi  ;Get the delta
  cmp edx, 0    ;delta <= 0 (i.e. edx <= esi)? This can happen
                ;as eax is ORed with edx and edx itself can overflow
                ;(especially when shifted left)
  jle .loop

  mov eax, edx  ;Make it C ABI compatible

  pop esi
  pop edx
  ret
```

Notare che è presente un test sul bit 31, ma questo non ha effetto sul codice (è di fatto, un salto che non produce nulla), è probabile che fosse usato in fase di debug dagli autori.

Il sorgente assembly sopra può essere assemblato con NASM in file COFF a 32-bit (`-fwin32`) e compilato con il seguente codice C per ottenere un eseguibile che entra in un ciclo infinito in una VM (incluso *any.run*) e mostra un messaggio in una macchina fisica.

```
#include <windows.h>

int get_counter_delta_for_cpuid();

int loop_if_in_vm()
{
  int accumulator = 0, i;
```

```
  do
  {
    for (accumulator = 0, i = 0; i < 100000; i++)
      accumulator += get_counter_delta_for_cpuid();
  }
  while (accumulator < 0 || accumulator >= 108000000);


  return accumulator;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
          (void)hInstance; (void)hPrevInstance; (void)lpCmdLine; (void)nCmdShow;


          loop_if_in_vm();
          MessageBox(NULL, "Not inside a VM", "VM",  MB_ICONINFORMATION | MB_OK);
          return 0;
}
```

Il campione è scariabile da qui.

Efficacia

Ottima, funziona su any.run, le macchine dei nostri analisti ed hybrid analysis. Notare come non sono prodotti messaggi nelle varie sandbox.

Contromisure

L’unica soluzione è quella di patchare il codice. Quando il processo entra in ciclo, interromperlo e cambiare le condizioni di uscita dai cicli.