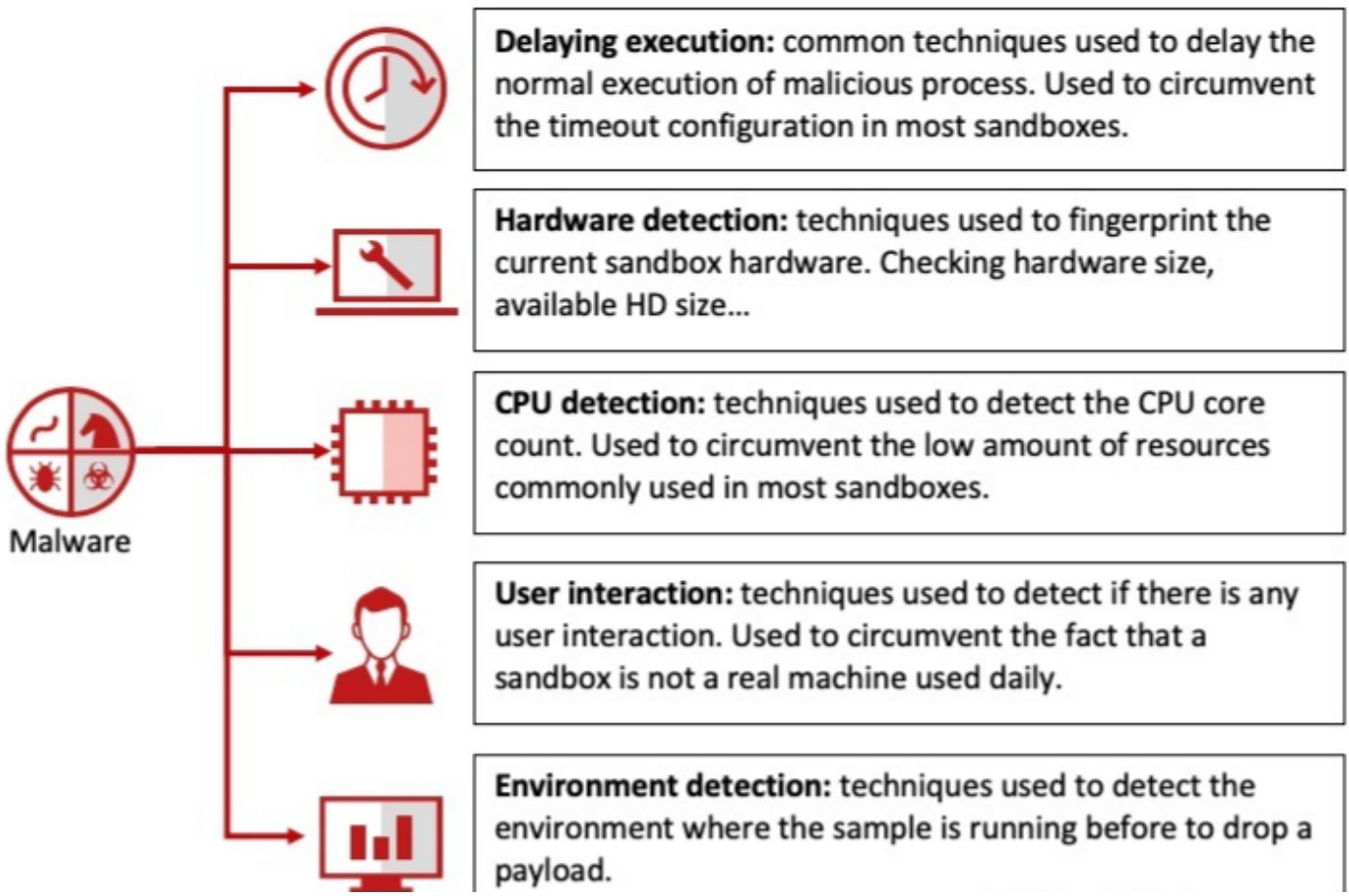



延迟执行

有许多恶意软件都使用基于时间的绕过技术，主要是利用已知的Windows API来延迟恶意代码的执行，常用的API有NtDelayExecution, CreateWaitTableTImer, SetTimer。这些技术在沙箱识别出来之前非常流行。

GetTickCount

沙箱可以识别出恶意软件并通过加速代码执行的方式来进行应对，可以使用多种方法来进行加速检查。其中一种方法就是使用Windows API GetTickCount和一行检查使用时间的代码。研究人员发现多种恶意软件家族使用的方法的变种：

```
mov     esi, ds:GetTickCount
call    esi ; GetTickCount
push    0EA60h          ; dwMilliseconds
mov     edi, eax
call    ds:Sleep
call    esi ; GetTickCount
sub     eax, edi
mov     ecx, 0E678h
```

沙箱厂商可以通过创建超过20分钟的简介使机器多次运行来轻易绕过反逃逸技术。

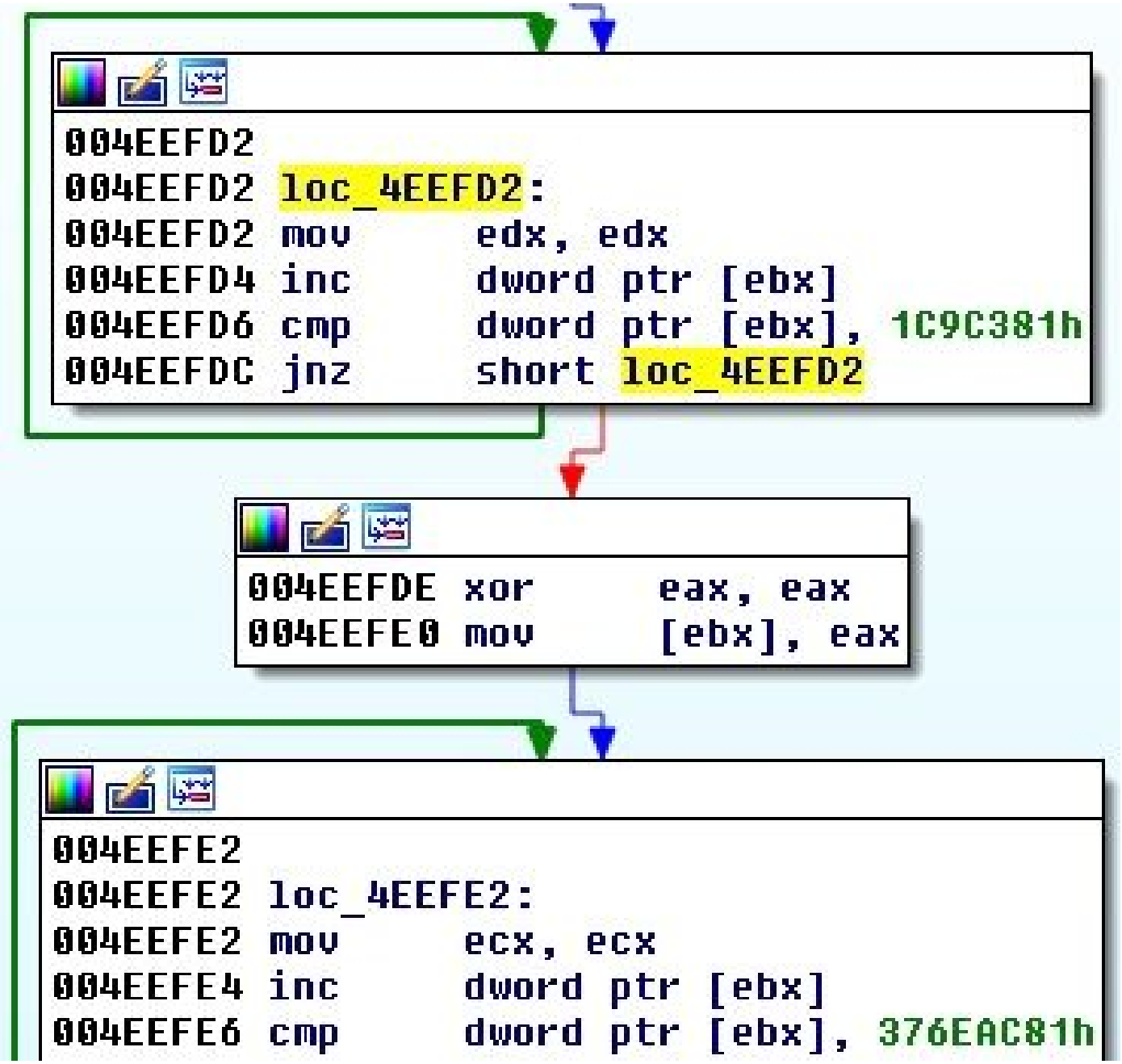
# API洪泛

## 另一种更加流行的方法是在循环中调用垃圾API来引入延迟，即所谓的API flooding。下面是恶意软件中使用该方法的代码：

```
v4 = a1 ^ dword_403070;
v16 = 0;
v15 = 20;
v14 = dword_403070;
v5 = 7814901;
do
{
  v8 = v5;
  v7 = v3;
  v6 = v4;
  GetSystemTimeAdjustment(&TimeAdjustment, &TimeIncrement, &TimeAdjustmentDisabled);
  v4 = v6;                                    // GetSystemTimeAdjustment called > 78 Lac times
  v3 = v7;
  v5 = v8 - 1;
}
```

# Inline Code

因为沙箱无法很好地处理行内代码，因此会产生DOS条件。另一方面，许多沙箱无法检测此类行为。




现在的沙箱能力更加多样化，并有代码插入和全模拟的能力，可以识别和报告暂停代码（stalling code）。因此，有了一个可以绕过大多数高级沙箱的方法。

下面是恶意软件在过去纪念使用的基于时间的绕过技术增长图：


## 硬件检测

恶意软件广泛使用的另一类绕过技术就是对硬件进行指纹操作，尤其是对物理内存的大小、可用HD大小/类型和可用CPU核数进行检查。

这些方法在Win32/Phorpiex, Win32/Comrerop, Win32/Simda和相关的恶意软件家族中非常流行。研究人员对这些变种进行追踪分析发现，Windows API DeviceIoControl()和一些控制代码被用来提取存储类型和大小的信息。

勒索软件和加密货币挖矿恶意软件也用GlobalMemoryStatusEx ()方法来检查可用的物理内存。类似的检查如下所示：

内存大小检查：

```
v2 = CreateFileA("\\\\.\\PhysicalDrive0", 0x80000000, 1u, 0, 3u, 0, 0);
Result = (BOOL)v2;
if ( v2 == (HANDLE)-1 )
{
  LODWORD(v0) = CloseHandle((HANDLE)0xFFFFFFFF);
}
else
{
  Output = (HKEY)DeviceIoControl(v2, 0x7405Cu, 0, 0, &OutBuffer, 8u, &BytesReturned, 0);
  LODWORD(v0) = CloseHandle((HANDLE)Result);
  if ( Output )
  {
    v0 = OutBuffer / 0x40000000;
    if ( OutBuffer / 0x40000000 <= 10 )
    {
      sub_402DC8();
```

下面的沙箱中实现的API拦截代码示例，可以对返回的存储大小进行操作：

```
BOOL Ret = 0;

DWORD HighPart, LowPart = 0;
GET_LENGTH_INFORMATION *LengthInfo;
Ret = Real_DeviceIoControl(hDevice, dwIoControlCode, lpInBuffer, nInBufferSize, lpOutBuffer, nOutBufferSize, lpBytesRetur

if (Ret)
    {
        if (dwIoControlCode == IOCTL_DISK_GET_LENGTH_INFO && lpOutBuffer != NULL )
        {
            LengthInfo = (GET_LENGTH_INFORMATION *)lpOutBuffer;
            if (LengthInfo->Length.QuadPart / 1073741824 <= 60)
            {
                HighPart = LengthInfo->Length.HighPart;
                LowPart = LengthInfo->Length.LowPart;
                LengthInfo->Length.HighPart = 0x000000FF;
```

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>


#define DEF_MUTEX_NAME      L"ReverseCore:DebugMe4"


void DoParentProcess();
void DoChildProcess();
```

```
void _tmain(int argc, TCHAR *argv[])
{
    HANDLE      hMutex = NULL;

    if( !(hMutex = CreateMutex(NULL, FALSE, DEF_MUTEX_NAME)) )
    {
        printf("CreateMutex() failed! [%d]\n", GetLastError());
        return;
    }

    // check mutex
    if( ERROR_ALREADY_EXISTS != GetLastError() )
        DoParentProcess();
    else
        DoChildProcess();
}


void DoChildProcess()
{
    // 8D C0 ("LEA EAX, EAX") 富档 救登绰 疙飞绢
    // 疙飞绢 辨捞 (0x17)
    __asm
    {
        nop
        nop
    }

    MessageBox(NULL, L"ChildProcess", L"DebugMe4", MB_OK);
}


void DoParentProcess()
{
    TCHAR                   szPath[MAX_PATH] = {0,};
    STARTUPINFO                 si = {sizeof(STARTUPINFO),};
    PROCESS_INFORMATION         pi = {0,};
    DEBUG_EVENT             de = {0,};
    CONTEXT                 ctx = {0,};
    BYTE                    pBuf[0x20] = {0,};
    DWORD                   dwExcpAddr = 0, dwExcpCode = 0;
    const DWORD             DECODING_SIZE = 0x14;
    const DWORD             DECODING_KEY = 0x7F;
    const DWORD             EXCP_ADDR_1 = 0x0040103F;
    const DWORD             EXCP_ADDR_2 = 0x00401048;
```

```
// create debug process
GetModuleFileName(
    GetModuleHandle(NULL),
    szPath,
    MAX_PATH);

if( !CreateProcess(
        NULL,
        szPath,
        NULL, NULL,
        FALSE,
        DEBUG_PROCESS | DEBUG_ONLY_THIS_PROCESS,
        NULL, NULL,
        &si,
        &pi) )
{
    printf("CreateProcess() failed! [%d]\n", GetLastError());
    return;
}

printf("Parent Process\n");

// debug loop
while( TRUE )
{
    ZeroMemory(&de, sizeof(DEBUG_EVENT));

    if( !WaitForDebugEvent(&de, INFINITE) )
    {
        printf("WaitForDebugEvent() failed! [%d]\n", GetLastError());
        break;
    }

    if( de.dwDebugEventCode == EXCEPTION_DEBUG_EVENT )
    {
        dwExcpAddr = (DWORD)de.u.Exception.ExceptionRecord.ExceptionAddress;
        dwExcpCode = de.u.Exception.ExceptionRecord.ExceptionCode;

        if( dwExcpCode == EXCEPTION_ILLEGAL_INSTRUCTION )
        {
            if( dwExcpAddr == EXCP_ADDR_1 )
            {
                // decoding
                ReadProcessMemory(
                    pi.hProcess,
                    (LPCVOID)(dwExcpAddr + 2),
```

```
                pBuf,
                DECODING_SIZE,
                NULL);

            for(DWORD i = 0; i < DECODING_SIZE; i++)
                pBuf[i] ^= DECODING_KEY;

            WriteProcessMemory(
                pi.hProcess,
                (LPVOID)(dwExcpAddr + 2),
                pBuf,
                DECODING_SIZE,
                NULL);

            // change EIP
            ctx.ContextFlags = CONTEXT_FULL;
            GetThreadContext(pi.hThread, &ctx);
            ctx.Eip += 2;
            SetThreadContext(pi.hThread, &ctx);
        }
        else if( dwExcpAddr == EXCP_ADDR_2 )
        {
            pBuf[0] = 0x68;
            pBuf[1] = 0x1C;
            WriteProcessMemory(
                pi.hProcess,
                (LPVOID)dwExcpAddr,
                pBuf,
                2,
                NULL);
        }
      }
    }
    else if( de.dwDebugEventCode == EXIT_PROCESS_DEBUG_EVENT )
    {
      break;
    }

    ContinueDebugEvent(de.dwProcessId, de.dwThreadId, DBG_CONTINUE);
  }
}
```