Статья Анти-отладка и Анти-ВМ и Анти-Эмуляция.

Please note, if you want to make a deal with this user, that it is blocked.

В наши дни вредоносное ПО становится всё более сложным. Аналитики кибербезопасности используют множество ПО для отладки и анализа малварей. Авторы малварей используют различные техники для автоматического обнаружения таких отладчиков и виртуальных машин. В этой статье мы разберём некоторые пользующиеся популярностью техники и способы для обхода отладчиков и песочниц.

Требуемые инструменты:

- Immunity debugger
- C/C++ компилятор (msvc или GCC)
- Виртуальная машина (Vmware от Vbox)

Отладчики. Вступление
Отладчик(Debugger) - это программа, используемая для анализа и воспроизведения исполняемых файлов. Для анализа и перехвата кода отладчики используют системные вызовы и API предоставленные операционной системой. Для перехвата одного блока кода, отладчики используют пошаговые операции, которые можно включить настройкой TRAP Flag в EFLAGS регистре. Отладчики используют различные типы точек прерывания для остановки в определённом адресе памяти. Типы прерывания, используемые отладчиками:

1. Программная остановка.
2. Аппаратная остановка.
3. Остановка памяти.
4. Условные остановки.

Программная остановка это тип прерывания, где отладчик заменяет оригинальную инструкцию на INT 0xcc инструкцию, которая вызывает программную остановку процедуры и возвращает её обратно в отладчик для обработки. В immunity debugger вы можете увидеть программную остановку нажатием ALT + b

Точки прерывания:
Адрес Модуль Активность Расшифровка
00401FF0 extracto Всегда JE SHORT extracto.00401FF7
00401FFC extracto Всегда MOV EBP,ESP
0040200A extracto Всегда CALL DWORD PTR DS:[<&KERNEL32.ExitPr

Аппаратное прерывание использует 4 регистра отладки предоставленные процессом для инициации остановки в определённой точке: DR0, DR1, DR2, DR3

Затем мы переворачиваем соответствующие биты в регистре DR7, чтобы вызвать остановку и выставитель её тип и длину.

После того, как аппаратная остановка была выставлена и достигла операционной системы, последняя вызывает INT 1 прерывание пошагового процесса.

Затем отладчики настраивают соответствующие обработчики для перехвата этих исключений.

Остановка памяти:

В остановке памяти мы используем защитные страницы для настройки обработки и если эта страница задействуется, вызывается обработчик исключений (exception handler).

Отладчики поддерживают несколько типов остановки памяти:

1. Остановка памяти при доступе к BYTE.
2. Остановка памяти при доступе к WORD.
3. Остановка памяти при доступе к DWORD.

Условные остановки:
Условные остановки контролируются отладчиком, и предоставляются пользователю только если соблюдены определённые условия.

К примеру, вы можете настроить условную останову в immunity debugger, которая имеет следующий вид:

```
CONDITION = [ESP] = 0x0077ff89
```

Эта остановка будет применена только если значение, указанное на вершине стака (stack), будет равно 0x0077ff89.

Условные остановки полезны только в случае, если вы хотите контролировать вызовы к различным API только с определёнными параметрами.

API для отладки на Windows

Windows по стандарту предоставляет API для отладки, которое используется отладчиками для отладки приложений. API предоставляемые Windows известны как windows debugging API. // Да, это так в оригинале статьи написано!

Дальше идёт образец кода для отладки приложения с использованием windows debugging API.

Code:

```
void EnterDebugLoop(const LPDEBUG_EVENT DebugEv)
{
DWORD dwContinueStatus = DBG_CONTINUE; // exception continuation
char buffer[100];
CONTEXT lcContext;
for(;;)
{
// Wait for a debugging event to occur. The second parameter indicates
// that the function does not return until a debugging event occurs.

WaitForDebugEvent(DebugEv, INFINITE);

// Process the debugging event code.

switch (DebugEv-&gt;dwDebugEventCode)
{
case EXCEPTION_DEBUG_EVENT:
// Process the exception code. When handling
// exceptions, remember to set the continuation
// status parameter (dwContinueStatus). This value
// is used by the ContinueDebugEvent function.

switch(DebugEv-&gt;u.Exception.ExceptionRecord.ExceptionCode)
{
case EXCEPTION_ACCESS_VIOLATION:
// First chance: Pass this on to the system.
// Last chance: Display an appropriate error.

break;

case EXCEPTION_BREAKPOINT:

if (!fChance)
{
dwContinueStatus = DBG_CONTINUE; // exception continuation
fChance = 1;

break;
}

lcContext.ContextFlags = CONTEXT_ALL;
GetThreadContext(pi.hThread, &amp;lcContext);

ReadProcessMemory(pi.hProcess , (LPCVOID)(lcContext.Esp ),(LPVOID)&amp;rtAddr, sizeof(void
*), NULL );

if (DebugEv-&gt;u.Exception.ExceptionRecord.ExceptionAddress == pEntryPoint)
{

printf("\n%s\n", "Entry Point Reached");
WriteProcessMemory(pi.hProcess ,DebugEv-
```

```
&gt;u.Exception.ExceptionRecord.ExceptionAddress,&amp;OrgByte, 0x01, NULL);

lcContext.ContextFlags = CONTEXT_ALL;
GetThreadContext(pi.hThread, &amp;lcContext);
lcContext.Eip--; // Move back one byte
SetThreadContext(pi.hThread, &amp;lcContext);
FlushInstructionCache(pi.hProcess,DebugEv-
&gt;u.Exception.ExceptionRecord.ExceptionAddress,1);
dwContinueStatus = DBG_CONTINUE   ; // exception continuation

putBP();
break;
}

// First chance: Display the current
// instruction and register values.
break;

case EXCEPTION_DATATYPE_MISALIGNMENT:
// First chance: Pass this on to the system.
// Last chance: Display an appropriate error.

dwContinueStatus = DBG_CONTINUE ;
break;

case EXCEPTION_SINGLE_STEP:
printf("%s", "Single stepping event ");
dwContinueStatus = DBG_CONTINUE ;
break;

case DBG_CONTROL_C:
// First chance: Pass this on to the system.
// Last chance: Display an appropriate error.
break;

default:
// Handle other exceptions.
break;
}

break;

case CREATE_THREAD_DEBUG_EVENT:

//dwContinueStatus = OnCreateThreadDebugEvent(DebugEv);
break;

case CREATE_PROCESS_DEBUG_EVENT:
printf("%s", GetFileNameFromHandle(DebugEv-&gt;u.CreateProcessInfo.hFile));
break;
```

```
case EXIT_THREAD_DEBUG_EVENT:
// Display the thread's exit code.

//dwContinueStatus = OnExitThreadDebugEvent(DebugEv);
break;

case EXIT_PROCESS_DEBUG_EVENT:

// Display the process's exit code.
return;
//dwContinueStatus = OnExitProcessDebugEvent(DebugEv);
break;

case LOAD_DLL_DEBUG_EVENT:

char *sDLLName;

sDLLName = GetFileNameFromHandle(DebugEv-&gt;u.LoadDll.hFile);

printf("\nDLl Loaded = %s Base Address 0x%p\n", sDLLName, DebugEv-&gt;u.LoadDll.lpBaseOfDll);

//dwContinueStatus = OnLoadDllDebugEvent(DebugEv);
break;

case UNLOAD_DLL_DEBUG_EVENT:
// Display a message that the DLL has been unloaded.

//dwContinueStatus = OnUnloadDllDebugEvent(DebugEv);
break;

case OUTPUT_DEBUG_STRING_EVENT:
// Display the output debugging string.

//dwContinueStatus = OnOutputDebugStringEvent(DebugEv);
break;

case RIP_EVENT:

//dwContinueStatus = OnRipEvent(DebugEv);
break;
}

// Resume executing the thread that reported the debugging event.
ContinueDebugEvent(DebugEv-&gt;dwProcessId,
DebugEv-&gt;dwThreadId,
dwContinueStatus);
}

}

int main(int argc ,char **argv)
```

```
{
DEBUG_EVENT debug_event = {0};
STARTUPINFO si;
FILE *fp = fopen(argv[1], "rb");
ZeroMemory( &amp;si, sizeof(si) );
si.cb = sizeof(si);
ZeroMemory( &amp;pi, sizeof(pi) );

CreateProcess ( argv[1], NULL, NULL, NULL, FALSE,
DEBUG_ONLY_THIS_PROCESS, NULL,NULL, &amp;si, &amp;pi );

printf("Passed Argument is %s\n", OrgName);
pEntryPoint = GetEP(fp); // GET the entry Point of the Application
fclose(fp);

ReadProcessMemory(pi.hProcess ,pEntryPoint, &amp;OrgByte, 0x01, NULL); // read the original
byte at the entry point
WriteProcessMemory(pi.hProcess ,pEntryPoint,"\xcc", 0x01, NULL); // Replace the byte at entry
point with int 0xcc

EnterDebugLoop(&amp;debug_event);  // User-defined function, not API

return 0;
}

int main(int argc ,char **argv)
{
DEBUG_EVENT debug_event = {0};
STARTUPINFO si;
FILE *fp = fopen(argv[1], "rb");
ZeroMemory( &amp;si, sizeof(si) );
si.cb = sizeof(si);
ZeroMemory( &amp;pi, sizeof(pi) );

CreateProcess ( argv[1], NULL, NULL, NULL, FALSE,
DEBUG_ONLY_THIS_PROCESS, NULL,NULL, &amp;si, &amp;pi );

printf("Passed Argument is %s\n", OrgName);
pEntryPoint = GetEP(fp); // GET the entry Point of the Application
fclose(fp);

ReadProcessMemory(pi.hProcess ,pEntryPoint, &amp;OrgByte, 0x01, NULL); // read the original
byte at the entry point
WriteProcessMemory(pi.hProcess ,pEntryPoint,"\xcc", 0x01, NULL); // Replace the byte at entry
point with int 0xcc

EnterDebugLoop(&amp;debug_event);  // User-defined function, not API

return 0;
}
```

Техники анти-отладки

Сейчас, для прерывания анализа, вредоносное ПО может обнаружить присутствие отладчика и вызывать неожиданные события. Чтобы обнаружить присутствие отладчика, вредоносная программа может либо прочитать некоторые значения, либо использовать API, чтобы определить, отлаживается вредоносная программа или нет.

Один из простых способов обнаружения отладки использует winAPI функцию, известную как KERNEL32.IsDebuggerPresent.

Code:

```
#define WIN32_LEAN_AND_MEAN
#include &lt;windows.h&gt;
#include &lt;stdio.h&gt;

int main(int argc, char **argv)
{
if (IsDebuggerPresent())
{
MessageBox(HWND_BROADCAST, "Debugger Detected", ""Debugger Detected"", MB_OK);
exit();
}
MessageBox(HWND_BROADCAST, "Debugger Not Detected", ""Debugger Not Detected"", MB_OK);
return 0;
}
```

Хотите больше информации об анти-отладке? Проверьте эту статью!

Обнаружение отладчика с использованием PEB:

Когда процесс создан с помощью CreateProcess API, и если флаг создание установлен как DEBUG_ONLY_THIS_PROCESS,
в структуре данных PEB в памяти устанавливается специальное поле.
Code:

```
#define WIN32_LEAN_AND_MEAN
#include &lt;windows.h&gt;
#include &lt;stdio.h&gt;

int __naked detectDebugger()
{
__asm
{
ASSUME FS:NOTHING

MOV EAX,DWORD PTR FS:[18]
MOV EAX,DWORD PTR DS:[EAX+30]
MOVZX EAX,BYTE PTR DS:[EAX+2]
RET

}
}

int main(int argc, char **argv)
{
if (detectDebugger())
{
MessageBox(HWND_BROADCAST, "Debugger Detected", ""Debugger Detected"", MB_OK);
exit();
}
MessageBox(HWND_BROADCAST, "Debugger Not Detected", ""Debugger Not Detected"", MB_OK);
return 0;
}
```

Обнаружение с использованием HEAP флагов:

Когда программа запущена в отладчике и создана с использованием флагов создания процесса отладки, heap флаги изменяются. Эти флаги выходят в другую локацию, в зависимости от версии операционной системы.

В основанных на Windows NT системах, эти флаги смещены на 0x0c от heap базы.

В основанных на Windows Vita системах и более поздних, они смещены на 0x40 от heap базы.

Эти флаги инициализированы как "Force flags" и "flags"

ProcessHeap Точки отчёта к структуре _Heap определены как:

Оригинал: http://www.nirsoft.net/kernel_struct/vista/HEAP.html

Code:

```
typedef struct _HEAP
{
     HEAP_ENTRY Entry;
     ULONG SegmentSignature;
     ULONG SegmentFlags;
     LIST_ENTRY SegmentListEntry;
     PHEAP Heap;
     PVOID BaseAddress;
     ULONG NumberOfPages;
     PHEAP_ENTRY FirstEntry;
     PHEAP_ENTRY LastValidEntry;
     ULONG NumberOfUnCommittedPages;
     ULONG NumberOfUnCommittedRanges;
     WORD SegmentAllocatorBackTraceIndex;
     WORD Reserved;
     LIST_ENTRY UCRSegmentList;
     ULONG Flags;
     ULONG ForceFlags;
     ULONG CompatibilityFlags;
     ULONG EncodeFlagMask;
     HEAP_ENTRY Encoding;
     ULONG PointerKey;
     ULONG Interceptor;
     ULONG VirtualMemoryThreshold;
     ULONG Signature;
     ULONG SegmentReserve;
     ULONG SegmentCommit;
     ULONG DeCommitFreeBlockThreshold;
     ULONG DeCommitTotalFreeThreshold;
     ULONG TotalFreeSize;
     ULONG MaximumAllocationSize;
     WORD ProcessHeapsListIndex;
     WORD HeaderValidateLength;
     PVOID HeaderValidateCopy;
     WORD NextAvailableTagIndex;
     WORD MaximumTagIndex;
     PHEAP_TAG_ENTRY TagEntries;
     LIST_ENTRY UCRList;
     ULONG AlignRound;
     ULONG AlignMask;
     LIST_ENTRY VirtualAllocdBlocks;
     LIST_ENTRY SegmentList;
     WORD AllocatorBackTraceIndex;
     ULONG NonDedicatedListLength;
     PVOID BlocksIndex;
     PVOID UCRIndex;
     PHEAP_PSEUDO_TAG_ENTRY PseudoTagEntries;
     LIST_ENTRY FreeLists;
     PHEAP_LOCK LockVariable;
     LONG * CommitRoutine;
     PVOID FrontEndHeap;
```

```
     WORD FrontHeapLockCount;
     UCHAR FrontEndHeapType;
     HEAP_COUNTERS Counters;
     HEAP_TUNING_PARAMETERS TuningParameters;
} HEAP, *PHEAP;
```

Программа написанная на C может быть использована для обнаружения присутствия отладчика с помощью heap флагов.

Code:

```
int main(int argc, char* argv[])
{
    unsigned int var;
    __asm
    {
        MOV EAX, FS:[0x30];
        MOV EAX, [EAX + 0x18];
        MOV EAX, [EAX + 0x0c];
        MOV var,EAX
    }

    if(var != 2)
    {
        printf("Debugger Detected");
    }
    return 0;
}
```

Обнаружение виртуальной машины или эмуляции

Образцы вредоносного ПО постоянно анализируются аналитиками в изолированных обстановках, таких как Виртуальные Машины. Для того, чтобы помешать анализу программы внутри виртуальной машины, вредоносные программы имеют защиту от виртуальных машин или просто выходят, когда вредоносное ПО запускается в изолированной среде.

Следующие техники могут быть использованы для обнаружения запущен ли малварь внутри виртуальной машины.

1. Обраружение на основе временных промежутков.
2. Обнаружение на основе "артефактов".

Обнаружение на основе временных промежутков (timing based)

The Time Stamp Counter (TSC) (Счётчик отметок времени) это 64-разрядный регистр, присутствующий на всех процессорах x86 начиная с Pentium. Он подсчитывает количество циклов с момента сброса”. (Википедия)

Если код эмулируется, то будут изменения в моментах времени.

Результат хранится EDX:EAX формате

Разница между реальной машиной обычно будет меньше 100, но если код эмулируется, то разница будет огромной.

Code:

```
int main(int argc, char* argv[])
{
    unsigned int time1 = 0;
    unsigned int time2 = 0;
    __asm
    {
        RDTSC
        MOV time1,EAX
        RDTSC
        MOV time2, EAX

    }
    if ((time2 - time1) &gt; 100)
    {
        printf("%s", "VM Detected");
        return 0;
    }
    printf("%s", "VM not present");
    return 0;
}
```

Программа выше использует инструкции временных марок для обнаружение присутствия Виртуальной Машины

Обнаружение на основе артефактов

От артефактов конфигурации виртуальной машины, сети или устройств зависит поведение малвари. Вредоносные программы обычно проверяют наличие этих артефактов, чтобы обнаружить наличие отладчика или виртуальной среды

Лучшим случаем являются артефакты регистра, Vmare создаёт ключи реестра для Virtual Disk Controller(Контроллера виртуального диска), который может быть расположен в реесте по следующему пути.

HKLM\SYSTEM\CurrentControlSet\Services\Disk\Enum\0
as “SCSI\Disk&Ven_VMware_&Prod_VMware_Virtual_S&Rev_1.0\4&XXX&XXX”

Code:

```
int main(int argc, char **argv)
{

char lszValue[100];
   HKEY hKey;
   int i=0;
    RegOpenKeyEx (HKEY_LOCAL_MACHINE, "SYSTEM\\CurrentControlSet\\Services\\Disk\\Enum", 0L,
KEY_READ , &amp;hKey);

    RegQueryValue(hKey,"0",lszValue,sizeof(lszValue));

    printf("%s", lszValue);
    if (strstr(lszValue, "VMware"))
    {
        printf("Vmware Detected");
    }

     RegCloseKey(hKey);
    return 0;

}
```

Автор: D12dOx34X
Оригинал: тыц