Anti-Debug: Debug Flags


anti-debug.checkpoint.com/techniques/debug-flags.html

Contents

Debug Flags

Special flags in system tables, which dwell in process memory and which an operation system sets, can be used to indicate that the process is being debugged. The states of these flags can be verified either by using specific API functions or examining the system tables in memory.

These techniques are the most commonly used by malware.

1. Using Win32 API

The following techniques use existing API functions (WinAPI or NativeAPI) that check system structures in the process memory for particular flags that indicate the process is being debugged right now.

1.1. IsDebuggerPresent()

The function `kernel32!IsDebuggerPresent()` determines whether the current process is being debugged by a user-mode debugger such as OllyDbg or x64dbg. Generally, the function only checks the `BeingDebugged` flag of the Process Environment Block (PEB).

The following code can be used to terminate process if it is being debugged:

Assembly Code

```
    call IsDebuggerPresent
    test al, al
    jne  being_debugged
    ...
being_debugged:
    push 1
    call ExitProcess
```

C/C++ Code

```
if (IsDebuggerPresent())
    ExitProcess(-1);
```

1.2. CheckRemoteDebuggerPresent()

The function `kernel32!CheckRemoteDebuggerPresent()` checks if a debugger (in a different process on the same machine) is attached to the current process.

C/C++ Code

```
BOOL bDebuggerPresent;
if (TRUE == CheckRemoteDebuggerPresent(GetCurrentProcess(), &bDebuggerPresent) &&
    TRUE == bDebuggerPresent)
    ExitProcess(-1);
```

x86 Assembly

```
    lea eax, bDebuggerPresent]
    push eax
    push -1  ; GetCurrentProcess()
    call CheckRemoteDebuggerPresent
    cmp [bDebuggerPresent], 1
    jz being_debugged
    ...
being_debugged:
    push -1
    call ExitProcess
```

x86-64 Assembly

```
    lea rdx, [bDebuggerPresent]
    mov rcx, -1 ; GetCurrentProcess()
    call CheckRemoteDebuggerPresent
    cmp [bDebuggerPresent], 1
    jz being_debugged
    ...
being_debugged:
    mov ecx, -1
    call ExitProcess
```

1.3. NtQueryInformationProcess()

The function `ntdll!NtQueryInformationProcess()` can retrieve a different kind of information from a process. It accepts a `ProcessInformationClass` parameter which specifies the information you want to get and defines the output type of the

`ProcessInformation` parameter.

1.3.1. ProcessDebugPort

It is possible to retrieve the port number of the debugger for the process using the `ntdll!NtQueryInformationProcess()`. There is a documented class ProcessDebugPort, which retrieves a `DWORD` value equal to `0xFFFFFFFF` (decimal `-1`) if the process is being debugged.

C/C++ Code

```
typedef NTSTATUS (NTAPI *TNtQueryInformationProcess)(
    IN HANDLE           ProcessHandle,
    IN PROCESSINFOCLASS ProcessInformationClass,
    OUT PVOID           ProcessInformation,
    IN ULONG            ProcessInformationLength,
    OUT PULONG          ReturnLength
    );

HMODULE hNtdll = LoadLibraryA("ntdll.dll");
if (hNtdll)
{
    auto pfnNtQueryInformationProcess = (TNtQueryInformationProcess)GetProcAddress(
        hNtdll, "NtQueryInformationProcess");

    if (pfnNtQueryInformationProcess)
    {
        DWORD dwProcessDebugPort, dwReturned;
        NTSTATUS status = pfnNtQueryInformationProcess(
            GetCurrentProcess(),
            ProcessDebugPort,
            &dwProcessDebugPort,
            sizeof(DWORD),
            &dwReturned);

        if (NT_SUCCESS(status) && (-1 == dwProcessDebugPort))
            ExitProcess(-1);
    }
}
```

x86 Assembly

```
    lea eax, [dwReturned]
    push eax ; ReturnLength
    push 4   ; ProcessInformationLength
    lea ecx, [dwProcessDebugPort]
    push ecx ; ProcessInformation
    push 7   ; ProcessInformationClass
    push -1  ; ProcessHandle
    call NtQueryInformationProcess
    inc dword ptr [dwProcessDebugPort]
    jz being_debugged
    ...
being_debugged:
    push -1
    call ExitProcess
```

x86-64 Assembly

```
    lea rcx, [dwReturned]
    push rcx    ; ReturnLength
    mov r9d, 4  ; ProcessInformationLength
    lea r8, [dwProcessDebugPort]
                ; ProcessInformation
    mov edx, 7  ; ProcessInformationClass
    mov rcx, -1 ; ProcessHandle
    call NtQueryInformationProcess
    cmp dword ptr [dwProcessDebugPort], -1
    jz being_debugged
    ...
being_debugged:
    mov ecx, -1
    call ExitProcess
```

1.3.2. ProcessDebugFlags

A kernel structure called EPROCESS, which represents a process object, contains the field `NoDebugInherit`. The inverse value of this field can be retrieved using an undocumented class `ProcessDebugFlags` (`0x1f`). Therefore, if the return value is `0`, a debugger is present.

C/C++ Code

```
typedef NTSTATUS(NTAPI *TNtQueryInformationProcess)(
    IN HANDLE           ProcessHandle,
    IN DWORD            ProcessInformationClass,
    OUT PVOID           ProcessInformation,
    IN ULONG            ProcessInformationLength,
    OUT PULONG          ReturnLength
    );

HMODULE hNtdll = LoadLibraryA("ntdll.dll");
if (hNtdll)
{
    auto pfnNtQueryInformationProcess = (TNtQueryInformationProcess)GetProcAddress(
        hNtdll, "NtQueryInformationProcess");

    if (pfnNtQueryInformationProcess)
    {
        DWORD dwProcessDebugFlags, dwReturned;
        const DWORD ProcessDebugFlags = 0x1f;
        NTSTATUS status = pfnNtQueryInformationProcess(
            GetCurrentProcess(),
            ProcessDebugFlags,
            &dwProcessDebugFlags,
            sizeof(DWORD),
            &dwReturned);

        if (NT_SUCCESS(status) && (0 == dwProcessDebugFlags))
            ExitProcess(-1);
    }
}
```

x86 Assembly

```
    lea eax, [dwReturned]
    push eax ; ReturnLength
    push 4   ; ProcessInformationLength
    lea ecx, [dwProcessDebugPort]
    push ecx ; ProcessInformation
    push 1Fh ; ProcessInformationClass
    push -1  ; ProcessHandle
    call NtQueryInformationProcess
    cmp dword ptr [dwProcessDebugPort], 0
    jz being_debugged
    ...
being_debugged:
    push -1
    call ExitProcess
```

x86-64 Assembly

```
    lea rcx, [dwReturned]
    push rcx     ; ReturnLength
    mov r9d, 4   ; ProcessInformationLength
    lea r8, [dwProcessDebugPort]
                 ; ProcessInformation
    mov edx, 1Fh ; ProcessInformationClass
    mov rcx, -1  ; ProcessHandle
    call NtQueryInformationProcess
    cmp dword ptr [dwProcessDebugPort], 0
    jz being_debugged
    ...
being_debugged:
    mov ecx, -1
    call ExitProcess
```

1.3.3. ProcessDebugObjectHandle

When debugging begins, a kernel object called “debug object” is created. It is possible to query for the value of this handle by using the undocumented `ProcessDebugObjectHandle` (`0x1e`) class.

C/C++ Code

```
typedef NTSTATUS(NTAPI * TNtQueryInformationProcess)(
    IN HANDLE           ProcessHandle,
    IN DWORD            ProcessInformationClass,
    OUT PVOID           ProcessInformation,
    IN ULONG            ProcessInformationLength,
    OUT PULONG          ReturnLength
    );

HMODULE hNtdll = LoadLibraryA("ntdll.dll");
if (hNtdll)
{
    auto pfnNtQueryInformationProcess = (TNtQueryInformationProcess)GetProcAddress(
        hNtdll, "NtQueryInformationProcess");

    if (pfnNtQueryInformationProcess)
    {
        DWORD dwReturned;
        HANDLE hProcessDebugObject = 0;
        const DWORD ProcessDebugObjectHandle = 0x1e;
        NTSTATUS status = pfnNtQueryInformationProcess(
            GetCurrentProcess(),
            ProcessDebugObjectHandle,
            &hProcessDebugObject,
            sizeof(HANDLE),
            &dwReturned);

        if (NT_SUCCESS(status) && (0 != hProcessDebugObject))
            ExitProcess(-1);
    }
}
```

x86 Assembly

```
    lea eax, [dwReturned]
    push eax ; ReturnLength
    push 4   ; ProcessInformationLength
    lea ecx, [hProcessDebugObject]
    push ecx ; ProcessInformation
    push 1Eh ; ProcessInformationClass
    push -1  ; ProcessHandle
    call NtQueryInformationProcess
    cmp dword ptr [hProcessDebugObject], 0
    jnz being_debugged
    ...
being_debugged:
    push -1
    call ExitProcess
```

x86-64 Assembly

```
    lea rcx, [dwReturned]
    push rcx     ; ReturnLength
    mov r9d, 4   ; ProcessInformationLength
    lea r8, [hProcessDebugObject]
                 ; ProcessInformation
    mov edx, 1Fh ; ProcessInformationClass
    mov rcx, -1  ; ProcessHandle
    call NtQueryInformationProcess
    cmp dword ptr [hProcessDebugObject], 0
    jnz being_debugged
    ...
being_debugged:
    mov ecx, -1
    call ExitProcess
```

1.4. RtlQueryProcessHeapInformation()

The `ntdll!RtlQueryProcessHeapInformation()` function can be used to read the heap flags from the process memory of the current process.

C/C++ Code

```
bool Check()
{
    ntdll::PDEBUG_BUFFER pDebugBuffer = ntdll::RtlCreateQueryDebugBuffer(0, FALSE);
    if
(!SUCCEEDED(ntdll::RtlQueryProcessHeapInformation((ntdll::PRTL_DEBUG_INFORMATION)pDebu

        return false;

    ULONG dwFlags = ((ntdll::PRTL_PROCESS_HEAPS)pDebugBuffer->HeapInformation)-
>Heaps[0].Flags;
    return dwFlags & ~HEAP_GROWABLE;
}
```

1.5. RtlQueryProcessDebugInformation()

The `ntdll!RtlQueryProcessDebugInformation()` function can be used to read certain fields from the process memory of the requested process, including the heap flags.

C/C++ Code

```
bool Check()
{
    ntdll::PDEBUG_BUFFER pDebugBuffer = ntdll::RtlCreateQueryDebugBuffer(0, FALSE);
    if (!SUCCEEDED(ntdll::RtlQueryProcessDebugInformation(GetCurrentProcessId(),
ntdll::PDI_HEAPS | ntdll::PDI_HEAP_BLOCKS, pDebugBuffer)))
        return false;

    ULONG dwFlags = ((ntdll::PRTL_PROCESS_HEAPS)pDebugBuffer->HeapInformation)-
>Heaps[0].Flags;
    return dwFlags & ~HEAP_GROWABLE;
}
```

1.6. NtQuerySystemInformation()

The `ntdll!NtQuerySystemInformation()` function accepts a parameter which is the class of information to query. Most of the classes are not documented. This includes the SystemKernelDebuggerInformation (0x23) class, which has existed since Windows NT. The SystemKernelDebuggerInformation class returns the value of two flags: KdDebuggerEnabled in al, and KdDebuggerNotPresent in ah. Therefore, the return value in ah is zero if a kernel debugger is present.

C/C++ Code

```
enum { SystemKernelDebuggerInformation = 0x23 };

typedef struct _SYSTEM_KERNEL_DEBUGGER_INFORMATION {
    BOOLEAN DebuggerEnabled;
    BOOLEAN DebuggerNotPresent;
} SYSTEM_KERNEL_DEBUGGER_INFORMATION, *PSYSTEM_KERNEL_DEBUGGER_INFORMATION;

bool Check()
{
    NTSTATUS status;
    SYSTEM_KERNEL_DEBUGGER_INFORMATION SystemInfo;

    status = NtQuerySystemInformation(
        (SYSTEM_INFORMATION_CLASS)SystemKernelDebuggerInformation,
        &SystemInfo,
        sizeof(SystemInfo),
        NULL);

    return SUCCEEDED(status)
        ? (SystemInfo.DebuggerEnabled && !SystemInfo.DebuggerNotPresent)
        : false;
}
```

Mitigations

- For `IsDebuggerPresent()`: Set the `BeingDebugged` flag of the Process Environment Block (PEB) to 0. See BeingDebugged Flag Mitigation for further information.
- For `CheckRemoteDebuggerPresent()` and `NtQueryInformationProcess()`: As `CheckRemoteDebuggerPresent()` calls `NtQueryInformationProcess()`, the only way is to hook the `NtQueryInformationProcess()` and set the following values in return buffers:
  - 0 (or any value except -1) in case of a `ProcessDebugPort` query.
  - Non-zero value in case of a `ProcessDebugFlags` query.
  - 0 in case of a `ProcessDebugObjectHandle` query.
- The only way to mitigate these checks with `RtlQueryProcessHeapInformation()`, `RtlQueryProcessDebugInformation()` and `NtQuerySystemInformation()` functions is to hook them and modify the returned values:
  - `RTL_PROCESS_HEAPS::HeapInformation::Heaps[0]::Flags` to `HEAP_GROWABLE` for `RtlQueryProcessHeapInformation()` and `RtlQueryProcessDebugInformation()`.
  - `SYSTEM_KERNEL_DEBUGGER_INFORMATION::DebuggerEnabled` to 0 and `SYSTEM_KERNEL_DEBUGGER_INFORMATION::DebuggerNotPresent` to 1 for the `NtQuerySystemInformation()` function in case of a `SystemKernelDebuggerInformation` query.

2. Manual checks

The following approaches are used to validate debugging flags in system structures. They examine the process memory manually without using special debug API functions.

2.1. PEB!BeingDebugged Flag

This method is just another way to check `BeingDebugged` flag of PEB without calling `IsDebuggerPresent()`.

32Bit Process

```
mov eax, fs:[30h]
cmp byte ptr [eax+2], 0
jne being_debugged
```

64Bit Process

```
mov rax, gs:[60h]
cmp byte ptr [rax+2], 0
jne being_debugged
```

WOW64 Process

```
mov eax, fs:[30h]
cmp byte ptr [eax+1002h], 0
```

C/C++ Code

```
#ifndef _WIN64
PPEB pPeb = (PPEB)__readfsdword(0x30);
#else
PPEB pPeb = (PPEB)__readgsqword(0x60);
#endif // _WIN64

if (pPeb->BeingDebugged)
    goto being_debugged;
```

2.2. NtGlobalFlag

The `NtGlobalFlag` field of the Process Environment Block (`0x68` offset on 32-Bit and `0xBC` on 64-bit Windows) is 0 by default. Attaching a debugger doesn't change the value of `NtGlobalFlag`. However, if the process was created by a debugger, the following flags will be set:

- `FLG_HEAP_ENABLE_TAIL_CHECK` (0x10)
- `FLG_HEAP_ENABLE_FREE_CHECK` (0x20)
- `FLG_HEAP_VALIDATE_PARAMETERS` (0x40)

The presence of a debugger can be detected by checking a combination of those flags.

32Bit Process

```
mov eax, fs:[30h]
mov al, [eax+68h]
and al, 70h
cmp al, 70h
jz  being_debugged
```

64Bit Process

```
mov rax, gs:[60h]
mov al, [rax+BCh]
and al, 70h
cmp al, 70h
jz  being_debugged
```

WOW64 Process

```
mov eax, fs:[30h]
mov al, [eax+10BCh]
and al, 70h
cmp al, 70h
jz  being_debugged
```

C/C++ Code

```
#define FLG_HEAP_ENABLE_TAIL_CHECK   0x10
#define FLG_HEAP_ENABLE_FREE_CHECK   0x20
#define FLG_HEAP_VALIDATE_PARAMETERS 0x40
#define NT_GLOBAL_FLAG_DEBUGGED (FLG_HEAP_ENABLE_TAIL_CHECK |
FLG_HEAP_ENABLE_FREE_CHECK | FLG_HEAP_VALIDATE_PARAMETERS)

#ifndef _WIN64
PPEB pPeb = (PPEB)__readfsdword(0x30);
DWORD dwNtGlobalFlag = *(PDWORD)((PBYTE)pPeb + 0x68);
#else
PPEB pPeb = (PPEB)__readgsqword(0x60);
DWORD dwNtGlobalFlag = *(PDWORD)((PBYTE)pPeb + 0xBC);
#endif // _WIN64

if (dwNtGlobalFlag & NT_GLOBAL_FLAG_DEBUGGED)
    goto being_debugged;
```

2.3. Heap Flags

The heap contains two fields which are affected by the presence of a debugger. Exactly how they are affected depends on the Windows version. These fields are `Flags` and `ForceFlags`.

The values of `Flags` and `ForceFlags` are normally set to `HEAP_GROWABLE` and `0`, respectively.

When a debugger is present, the `Flags` field is set to a combination of these flags on Windows NT, Windows 2000, and 32-bit Windows XP:

- `HEAP_GROWABLE` (2)
- `HEAP_TAIL_CHECKING_ENABLED` (0x20)
- `HEAP_FREE_CHECKING_ENABLED` (0x40)
- `HEAP_SKIP_VALIDATION_CHECKS` (0x10000000)

- `HEAP_VALIDATE_PARAMETERS_ENABLED` (0x40000000)

On 64-bit Windows XP, and Windows Vista and higher, if a debugger is present, the `Flags` field is set to a combination of these flags:

- `HEAP_GROWABLE` (2)
- `HEAP_TAIL_CHECKING_ENABLED` (0x20)
- `HEAP_FREE_CHECKING_ENABLED` (0x40)
- `HEAP_VALIDATE_PARAMETERS_ENABLED` (0x40000000)

When a debugger is present, the `ForceFlags` field is set to a combination of these flags:

- `HEAP_TAIL_CHECKING_ENABLED` (0x20)
- `HEAP_FREE_CHECKING_ENABLED` (0x40)
- `HEAP_VALIDATE_PARAMETERS_ENABLED` (0x40000000)

C/C++ Code

```
bool Check()
{
#ifndef _WIN64
    PPEB pPeb = (PPEB)__readfsdword(0x30);
    PVOID pHeapBase = !m_bIsWow64
        ? (PVOID)(*(PDWORD_PTR)((PBYTE)pPeb + 0x18))
        : (PVOID)(*(PDWORD_PTR)((PBYTE)pPeb + 0x1030));
    DWORD dwHeapFlagsOffset = IsWindowsVistaOrGreater()
        ? 0x40
        : 0x0C;
    DWORD dwHeapForceFlagsOffset = IsWindowsVistaOrGreater()
        ? 0x44
        : 0x10;
#else
    PPEB pPeb = (PPEB)__readgsqword(0x60);
    PVOID pHeapBase = (PVOID)(*(PDWORD_PTR)((PBYTE)pPeb + 0x30));
    DWORD dwHeapFlagsOffset = IsWindowsVistaOrGreater()
        ? 0x70
        : 0x14;
    DWORD dwHeapForceFlagsOffset = IsWindowsVistaOrGreater()
        ? 0x74
        : 0x18;
#endif // _WIN64

    PDWORD pdwHeapFlags = (PDWORD)((PBYTE)pHeapBase + dwHeapFlagsOffset);
    PDWORD pdwHeapForceFlags = (PDWORD)((PBYTE)pHeapBase + dwHeapForceFlagsOffset);
    return (*pdwHeapFlags & ~HEAP_GROWABLE) || (*pdwHeapForceFlags != 0);
}
```

2.3. Heap Protection

If the `HEAP_TAIL_CHECKING_ENABLED` flag is set in `NtGlobalFlag`, the sequence `0xABABABAB` will be appended (twice in 32-Bit and 4 times in 64-Bit Windows) at the end of the allocated heap block.

If the `HEAP_FREE_CHECKING_ENABLED` flag is set in `NtGlobalFlag`, the sequence `0xFEEEFEEE` will be appended if additional bytes are required to fill in the empty space until the next memory block.

C/C++ Code

```
bool Check()
{
    PROCESS_HEAP_ENTRY HeapEntry = { 0 };
    do
    {
        if (!HeapWalk(GetProcessHeap(), &HeapEntry))
            return false;
    } while (HeapEntry.wFlags != PROCESS_HEAP_ENTRY_BUSY);

    PVOID pOverlapped = (PBYTE)HeapEntry.lpData + HeapEntry.cbData;
    return ((DWORD)(*(PDWORD)pOverlapped) == 0xABABABAB);
}
```

Mitigations

For PEB!BeingDebugged Flag:

Set the `BeingDebugged` flag to 0. This can be done by DLL injection. If you use OllyDbg or x32/64dbg as a debugger, you can choose various Anti-Debug plugins such as ScyllaHide.

```
#ifndef _WIN64
PPEB pPeb = (PPEB)__readfsdword(0x30);
#else
PPEB pPeb = (PPEB)__readgsqword(0x60);
#endif // _WIN64
pPeb->BeingDebugged = 0;
```

For NtGlobalFlag:

Set the `NtGlobalFlag` to 0. This can be done by DLL injection. If you use OllyDbg or x32/64dbg as a debugger, you can choose various Anti-Debug plugins such as ScyllaHide.

```
#ifndef _WIN64
PPEB pPeb = (PPEB)__readfsdword(0x30);
*(PDWORD)((PBYTE)pPeb + 0x68) = 0;
#else
PPEB pPeb = (PPEB)__readgsqword(0x60);
*(PDWORD)((PBYTE)pPeb + 0xBC); = 0;
#endif // _WIN64
```

For Heap Flags:

Set the `Flags` value to `HEAP_GROWABLE`, and the `ForceFlags` value to 0. This can be done by DLL injection. If you use OllyDbg or x32/64dbg as a debugger, you can choose various Anti-Debug plugins such as ScyllaHide.

```
#ifndef _WIN64
PPEB pPeb = (PPEB)__readfsdword(0x30);
PVOID pHeapBase = !m_bIsWow64
    ? (PVOID)(*(PDWORD_PTR)((PBYTE)pPeb + 0x18))
    : (PVOID)(*(PDWORD_PTR)((PBYTE)pPeb + 0x1030));
DWORD dwHeapFlagsOffset = IsWindowsVistaOrGreater()
    ? 0x40
    : 0x0C;
DWORD dwHeapForceFlagsOffset = IsWindowsVistaOrGreater()
    ? 0x44
    : 0x10;
#else
PPEB pPeb = (PPEB)__readgsqword(0x60);
PVOID pHeapBase = (PVOID)(*(PDWORD_PTR)((PBYTE)pPeb + 0x30));
DWORD dwHeapFlagsOffset = IsWindowsVistaOrGreater()
    ? 0x70
    : 0x14;
DWORD dwHeapForceFlagsOffset = IsWindowsVistaOrGreater()
    ? 0x74
    : 0x18;
#endif // _WIN64

*(PDWORD)((PBYTE)pHeapBase + dwHeapFlagsOffset) = HEAP_GROWABLE;
*(PDWORD)((PBYTE)pHeapBase + dwHeapForceFlagsOffset) = 0;
```

For Heap Protection:

Manually patch 12 bytes for 32-bit and 20 bytes in a 64-bit environment after the heap. Hook `kernel32!HeapAlloc()` and patch the heap after its allocation.

```
#ifndef _WIN64
SIZE_T nBytesToPatch = 12;
#else
SIZE_T nBytesToPatch = 20;
#endif // _WIN64

SIZE_T nDwordsToPatch = nBytesToPatch / sizeof(DWORD);
PVOID pHeapEnd = (PBYTE)HeapEntry.lpData + HeapEntry.cbData;
for (SIZE_T offset = 0; offset < nDwordsToPatch; offset++)
    *((PDWORD)pHeapEnd + offset) = 0;
```