Investigating a problem constructing a security descriptor to deny thread-specific access rights to Everyone

devblogs.microsoft.com/oldnewthing/20151120-00

November 20, 2015

Raymond Chen

A customer had a question about how to set up a security access mask.

> How can I deny thread-specific access rights to Everyone?
>
> Specifically, I want to deny the Everyone group the following rights when I create a process:
>
> - `THREAD_SET_INFORMATION`
> - `THREAD_SET_THREAD_TOKEN`
> - `THREAD_TERMINATE`
> - `PROCESS_CREATE_PROCESS`
> - `PROCESS_SET_SESSIONID`
> - `PROCESS_VM_OPERATION`
> - `PROCESS_VM_WRITE`
>
> How do I create the access mask for this? Will this function work?

```
DWORD GetDeniedMask()
{
  DWORD accessMask = 0;
  GENERIC_MAPPING genmap;
  genmap.GenericRead = WRITE_DAC | WRITE_OWNER;
  genmap.GenericWrite = WRITE_DAC | WRITE_OWNER;
  genmap.GenericExecute = WRITE_DAC | WRITE_OWNER;
  genmap.GenericAll = WRITE_DAC | WRITE_OWNER |
    THREAD_SET_INFORMATION |
    THREAD_SET_THREAD_TOKEN |
    THREAD_TERMINATE |
    PROCESS_CREATE_PROCESS |
    PROCESS_SET_SESSIONID |
    PROCESS_VM_OPERATION |
    PROCESS_VM_WRITE;
  MapGenericMask(&accessMask, &genmap);
  return accessMask;
}
```

This question is so confused it's hard to say where to start.

> Why are you trying to deny these accesses to Everyone? Note that Everyone includes the owner of the process, which means that the owner of the process can't even terminate his own process! Furthermore, many normal operations need accesses like the ones you are denying. You are going to end up with a process that can't do much, not even to itself. For example, the `VirtualAlloc` function needs `PROCESS_VM_OPERATION` access.[1] A process that can't allocate any memory is not going to get very far. Some of these accesses are needed by the process creator in order to do things like set up the initial environment and command line. And anti-malware software is going to block the creation of any process that refuses to let the anti-malware software inspect it!

The customer explained,

> A security audit uncovered that our processes granted the rights listed above to Everyone, so we are seeing what we can do to deny those rights to Everyone while still allowing those rights to the creator and people in the right security group.
>
> Is it a security risk to grant the above listed rights to Everyone? If so, how do we deny them to Everyone while still allowing it to the right people? We assume we need to pass custom security attributes as the `lpProcessAttributes` and `lpThreadAttributes` parameters when we call the `CreateProcess` function, but we need help building those security attributes.

Since deny actions override allow actions,[2] you can't deny something to Everyone, and then grant it to a special subgroup. The deny on Everyone will override the allow on the subgroup.

The way to do this is not to deny Everyone, but rather to *stop allowing* Everyone. A security principal receives access if there is an applicable allow rule and no applicable deny rule.[2] So remove the spurious allow rule.

Actually, where is the allow rule for Everyone coming from? The default process security does not grant Everyone those accesses. The customer must be doing something unusual.

> Here is the code we are using to set the security attributes on the process. [I have converted the long C++ code into equivalent pseudo-C# code for readability. -Raymond]
>
> ```
> var acl = new AccessControlList();
>
> // Deny some accesses to AU and WD.
> var deniedMask = GetDeniedMask();
> acl.AddDenyAce(AuthenticatedUsersSid, deniedMask);
> acl.AddDenyAce(WorldSid, deniedMask);
>
> // Grant some accesses to AU and WD.
> var worldMask = GetAllowedMask();
> acl.AddAllowAce(AuthenticatedUsersSid, worldMask);
> acl.AddAllowAce(WorldSid, worldMask);
> ```

It's not clear why they denied and granted identical accesses both to Authenticated Users and to Everyone. (aka World). Since Authenticated Users are a subset of Everyone, all the rules for Authenticated Users are redundant.

We need to peel away yet another layer of the onion. What is the custom access mask being granted to Everyone?

> Here is the `GetAllowedMask` function.
>
> ```
> DWORD GetAllowedMask()
> {
>  DWORD accessMask = GENERIC_READ | GENERIC_EXECUTE;
>  GENERIC_MAPPING genmap;
>  genmap.GenericRead = GENERIC_READ |
>                       FILE_GENERIC_READ |
>                       SECTION_MAP_READ;
>  genmap.GenericWrite = 0;
>  genmap.GenericExecute = GENERIC_EXECUTE |
>                          FILE_GENERIC_EXECUTE |
>                          SECTION_MAP_EXECUTE;
>  genmap.GenericAll = GENERIC_READ |
>                      GENERIC_EXECUTE ;
>  MapGenericMask(&accessMask, &genmap);
>  return accessMask;
> }
> ```

Here we see the same confusion that started the whole thing.

The customer appears not to understand what the `MapGenericMask` function does, or what it is for.

I will pause now so you can review my earlier discussion of the `MapGenericMask` function, what it does, and its intended usage pattern.

Welcome back. If you read and understood that article, you'll observe that this customer completely misses the point of the `MapGenericMask` function. They are using it to calculate information on the client side. But if you're on the client side, you don't need to convert `GENERIC_READ` to a specific mask. That's the server's job! Just ask for generic access and go home.

Anyway, let's see what happens. The `GetWorldAccessMask` function is passing a hard-coded access mask and a hard-coded generic mapping. We can walk through the code ourselves to see what happens.

- We start with `GENERIC_READ | GENERIC_EXECUTE`.
- Since `GENERIC_READ` is set, we remove it and replace it with `genmap.GenericRead`, which is `GENERIC_READ | FILE_GENERIC_READ | SECTION_MAP_READ`, resulting in `GENERIC_READ | FILE_GENERIC_READ | SECTION_MAP_READ | GENERIC_EXECUTE`.

Hey, wait a second. We're making things worse! The whole point of `MapGenericMask` is to get rid of generic mappings, but this `genmap` structure says, "To get rid of `GENERIC_READ`, convert it to this other stuff that *includes* `GENERIC_READ`."

This is like reading some tips on how to rid your room of outdated clutter, and one of them says, “If you see an old magazine, you can get rid of it by putting a fern next to it.” Um, that’s not actually getting rid of anything. You just added more stuff.

- Since `GENERIC_WRITE` is not set, nothing is done with `GenericWrite`.
- Since `GENERIC_EXECUTE` is set, we remove it and replace it with `genmap.GenericExecute`, which is `GENERIC_EXECUTE | FILE_GENERIC_EXECUTE | SECTION_MAP_EXECUTE`, resulting in `GENERIC_READ | FILE_GENERIC_READ | SECTION_MAP_READ | GENERIC_EXECUTE | FILE_GENERIC_EXECUTE | SECTION_MAP_EXECUTE`.
- Since `GENERIC_ALL` is not set, nothing is done with `GenericAll`.
- Finally, the `MapGenericMask` function removes all generic access bits, because it promises never to return any generic access bits.

The result of all these shenanigans is that we are granting Everyone the follow access mask:

~~`GENERIC_READ`~~	~~`0x80000000`~~
`FILE_GENERIC_READ = STANDARD_RIGHTS_READ | FILE_READ_DATA | FILE_READ_ATTRIBUTES | FILE_READ_EA | SYNCHRONIZE`	`0x00120089 = 0x00020000 | 0x00000001 | 0x00000080 | 0x00000008 | 0x00100000`
`SECTION_MAP_READ`	`0x00000004`
~~`GENERIC_EXECUTE`~~	~~`0x20000000`~~
`FILE_GENERIC_EXECUTE = STANDARD_RIGHTS_EXECUTE | FILE_READ_ATTRIBUTES | FILE_EXECUTE | SYNCHRONIZE`	`0x001200A0 = 0x00020000 | 0x00000080 | 0x00000020 | 0x00100000`
`SECTION_MAP_EXECUTE`	`0x00000008`
Grand total	`0x001200AD`

Actually, this mask makes no sense. It is combining file-specific access masks and section-specific access masks. And then applying them to a process and a thread! (Background reading.)

This is like looking at the menu for a Chinese restaurant and deciding that you want the Number 21 (cashew chicken), then looking at the menu for an Indian restaurant and deciding that you want the Number 18 (saag paneer), then calling a Greek restaurant, ordering the number 21 and 18 for take-out, then calling a Thai restaurant, and ordering the number 21

and 18 for take-out. And then you wonder why the Greek restaurant gave you a moussaka and a pork souvlaki, and the Thai restaurant gave you a phad thai and a yam nua. Where's your cashew chicken and saag paneer?

Let's see what Greek food we ended up ordering by accident.

`0x00020000 | 0x00000001 | 0x00000080 | 0x00000008 | 0x00100000`	`READ DAC | PROCESS TERMINATE | PROCESS CREATE PROCESS | PROCESS VM OPERATION | SYNCHRONIZE`
`0x00000004`	`PROCESS_SET_SESSIONID`
`0x00020000 | 0x00000080 | 0x00000020 | 0x00100000`	`READ DAC | PROCESS CREATE PROCESS | PROCESS VM WRITE | SYNCHRONIZE`
`0x00000008`	`PROCESS_VM_OPERATION`

Well, that explains why the process grants so many weird accesses to Everyone: Because you're granting all these weird accesses to Everyone!

I think we can predict what Thai food we ordered by accident, but let's do the math.

`0x00020000 | 0x00000001 | 0x00000080 | 0x00000008 | 0x00100000`	`READ_DAC | THREAD_TERMINATE | THREAD_SET_THREAD_TOKEN | THREAD_GET_CONTEXT | SYNCHRONIZE`
`0x00000004`	???? undefined ????
`0x00020000 | 0x00000080 | 0x00000020 | 0x00100000`	`READ_DAC | THREAD_SET_THREAD_TOKEN | THREAD_SET_INFORMATION | SYNCHRONIZE`
`0x00000008`	`THREAD_GET_CONTEXT`

From reading the confused code, it appears that the customer wants to grant read and execute rights to Everyone, but it's not clear why. In particular, Execute rights don't have intrinsic meaning for most types of objects, aside from files (to see if you can execute them) and memory (to see if you can execute code from them). Consequently, many object types overload Execute to mean something else. For example, our Gizmo object overloads Execute to mean start/stop.

If it's the case that the customer merely wants to grant permission to execute the program to Everyone, then that's done by applying the ACL to the executable file itself.

Assuming the presumption above is true, then the solution to the customer's problem is simple: Delete all the code that tries to create a custom security descriptor and just pass `NULL` as the security descriptor for the process and thread. This creates the process with default security, which is just fine for what you want.

The customer wrote back,

> Thanks. This is code we acquired recently, and the code base is so old that nobody knows exactly what this custom security attribute is trying to do.

[1] But you luck out because `GetCurrentProcess` returns a handle with full access, so the ACLs on the process object don't get a chance to flex their muscles if the process is talking about itself.

[2] Reality is more complex than this simple statement, but the details are not important to the story. The statements are true enough.

Raymond Chen

Follow