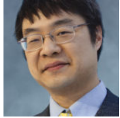
Applications and DLLs don't have privileges; users do

devblogs.microsoft.com/oldnewthing/20060818-14

August 18, 2006



Raymond Chen

I can't believe you people are actually asking for backdoors. If an end user can do it, then so can a bad guy.

In response to the requirement that all drivers on 64-bit Windows be signed, one commenter suggested adding a backdoor that permits unsigned drivers, using some "obscure registry key". Before somebody can jump up and shouts "security through obscurity!", the commenter adds this parenthetical: "(that no application has privileges to do by default)".

What does that parenthetical mean? How do you protect a registry key from an application? And if applications don't have privileges to modify a key, then who does?

The Windows security model is based on identity. Applications don't have privileges. Users have privileges. If an application is running in your user context, then it can do anything you can, and that includes setting that "obscure registry key". (This is a variation on "Your debugging code can be a security hole".) Same goes for DLLs. There's no such thing as something only an individual program/library can read/write to or do. You can't check the "identity of the calling library" because you can't trust the return address. Coming up with some other "magic encryption key" like the full path to the DLL won't help either, because a key that anybody can guess with 100% accuracy isn't much of a key.

Yes, UNIX has setuid, but that still doesn't make applications security principals. Even in UNIX, permissions are assigned to users, not to applications.

That's one of the reasons I get so puzzled when I hear people say, "Windows should let me do whatever I want with my system", while simultaneously saying, "Windows should have used ACLs to prevent applications from doing whatever they want with my system." But when you are running an application, **the application is you**. If you can do it, then an application can do it because the application is you.

Some people want to extend the concept of security principal to a chunk of code. "This registry key can be written to only by this function." But how could you enforce this? Once you let untrusted code enter a process, you can't trust any return addresses any more. How else could you identify the caller, then?

"Well, the DLL when it is created is given a magic cookie that it can use to prove its identity by passing that cookie to these 'super-secure functions'. For example,

```
// SECRET.DLL - a DLL that protects a secret registry key
HANDLE g_hMagicCookie;
// this function is called by means to be determined;
// it tells us the magic cookie to use to prove our identity.
void SetMagicCookie(HANDLE hMagicCookie)
{
 g_hMagicCookie = hMagicCookie;
}
```

and then the program can use the magic cookie to prove that it is the caller. For example, you could have `RegSetValueWithCookie(g_hMagicCookie, hkey, ...)`, where passing the cookie means 'It's me calling, please give me access to that thing that only I have access to."

That won't stop the bad guys for long. They just have to figure out where the DLL saves that cookie and read it, and bingo, they're now you.

```
// bad-guy program
int CALLBACK WinMain(...)
{
 // call some random function from SECRET.DLL
 // so it gets loaded and the magic cookie gets
 // initialized.
 SomeFunctionFromSECRETDLL();
 // experimentation tells us that SECRET.DLL
 // keeps its magic cookie at address 0x70131970
 HANDLE hMagicCookie = *(HANDLE*)0x70131970;
 RegSetValueWithCookie(hMagicCookie, hkey, ...);
 return 0;
}
```

Ta-da, we now have a program that writes to that registry key that `SECRET.DLL` was trying to protect. It does it by merely waiting for `SECRET.DLL` to receive its magic cookie, then stealing that cookie.

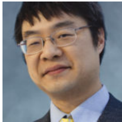
"Well, sure, but if I combine that with the check-the-return-address technique, then that'll stop them."

No, that doesn't stop anybody. All the bad guy has to do is change the `RegSetValueWithCookie(hMagicCookie, hkey, ...)` to code that hunts for a trusted address inside `SECRET.DLL` and cooks up a fake stack so that when control reaches `RegSetValueWithCookie`, everything in memory looks just like a legitimate call to the function, except that the attacker got to pass different parameters.

You can come up with whatever technique you want, it won't do any good. Once untrusted code has been granted access to a process, the entire process is compromised and you cannot trust it. Worst case, the attacker just sets a breakpoint on `RegSetValueWithCookie`, waits for the breakpoint to hit, then edits the stack to modify the parameters and resumes execution.

That's why code is not a security principal.

Corollary: Any security policy that says "Applications cannot do X without permission from the user" is flawed from conception. The application running as the user **is the user**. It's one thing to have this rule as a recommendation, even a logo requirement, but it's another thing to enforce this rule in the security subsystem.



Raymond Chen

Follow