Why are INI files deprecated in favor of the registry?

devblogs.microsoft.com/oldnewthing/20071126-00

November 26, 2007

Raymond Chen

Welcome, Slashdot readers. Remember, this Web site is for entertainment purposes only. Why are INI files deprecated in favor of the registry? There were many problems with INI files.

- INI files don't support Unicode. Even though there are Unicode functions of the private profile functions, they end up just writing ANSI text to the INI file. (There is a wacked out way you can create a Unicode INI file, but you have to step outside the API in order to do it.) This wasn't an issue in 16-bit Windows since 16-bit Windows didn't support Unicode either!
- INI file security is not granular enough. Since it's just a file, any permissions you set are at the file level, not the key level. You can't say, "Anybody can modify this section, but that section can be modified only by administrators." This wasn't an issue in 16-bit Windows since 16-bit Windows didn't do security.
- Multiple writers to an INI file can result in data loss. Consider two threads that are trying to update an INI file. If they are running simultaneously, you can get this:

Thread 1	Thread 2
Read INI file	
	Read INI file
Write INI file + X	
	Write INI file + Y

Notice that thread 2's update to the INI file accidentally deleted the change made by thread 1. This wasn't a problem in 16-bit Windows since 16-bit Windows was co-operatively multi-tasked. As long as you didn't yield the CPU between the read and the write, you were safe because nobody else could run until you yielded.

- INI files can suffer a denial of service. A program can open an INI file in exclusive mode and lock out everybody else. This is bad if the INI file was being used to hold security information, since it prevents anybody from seeing what those security settings are. This was also a problem in 16-bit Windows, but since there was no security in 16-bit Windows, a program that wanted to launch a denial of service attack on an INI file could just delete it!
- INI files contain only strings. If you wanted to store binary data, you had to encode it somehow as a string.
- Parsing an INI file is comparatively slow. Each time you read or write a value in an INI file, the file has to be loaded into memory and parsed. If you write three strings to an INI file, that INI file got loaded and parsed three times and got written out to disk three times. In 16-bit Windows, three consecutive INI file operations would result in only one parse and one write, because the operating system was co-operatively multi-tasked. When you accessed an INI file, it was parsed into memory and cached. The cache was flushed when you finally yielded CPU to another process.
- Many programs open INI files and read them directly. This means that the INI file format is locked and cannot be extended. Even if you wanted to add security to INI files, you can't. What's more, many programs that parsed INI files were buggy, so in practice you couldn't store a string longer than about 70 characters in an INI file or you'd cause some other program to crash.
- INI files are limited to 32KB in size.
- The default location for INI files was the Windows directory! This definitely was bad for Windows NT since only administrators have write permission there.
- INI files contain only two levels of structure. An INI file consists of sections, and each section consists of strings. You can't put sections inside other sections.
- [Added 9am] Central administration of INI files is difficult. Since they can be anywhere in the system, a network administrator can't write a script that asks, "Is everybody using the latest version of Firefox?" They also can't deploy scripts that say "Set everybody's Firefox settings to XYZ and deny write access so they can't change them."

The registry tried to address these concerns. You might argue whether these were valid concerns to begin with, but the Windows NT folks sure thought they were.

Commenter TC notes that the pendulum has swung back to text configuration files, but this time, they're XML. This reopens many of the problems that INI files had, but you have the major advantage that nobody *writes* to XML configuration files; they only read from them. XML configuration files are not used to store user settings; they just contain information about the program itself. Let's look at those issues again.

- XML files support Unicode.
- XML file security is not granular enough. But since the XML configuration file is read-only, the primary objection is sidestepped. (But if you want only administrators to have permission to read specific parts of the XML, then you're in trouble.)

- Since XML configuration files are read-only, you don't have to worry about multiple writers.
- XML configuration files files can suffer a denial of service. You can still open them exclusively and lock out other processes.
- XML files contain only strings. If you want to store binary data, you have to encode it somehow.
- Parsing an XML file is comparatively slow. But since they're read-only, you can safely cache the parsed result, so you only need to parse once.
- Programs parse XML files manually, but the XML format is already locked, so you couldn't extend it anyway even if you wanted to. Hopefully, those programs use a standard-conforming XML parser instead of rolling their own, but I wouldn't be surprised if people wrote their own custom XML parser that chokes on, say, processing instructions or strings longer than 70 characters.
- XML files do not have a size limit.
- XML files do not have a default location.
- XML files have complex structure. Elements can contain other elements.

XML manages to sidestep many of the problems that INI files have, but only if you promise only to read from them (and only if everybody agrees to use a standard-conforming parser), and if you don't require security granularity beyond the file level. Once you write to them, then a lot of the INI file problems return.

Raymond Chen

Follow