

Building an Anti-Virus Engine

Markus Schmall 2002-03-01

Building an Anti-Virus engine

by *Markus Schmall*

last updated March 4, 2002

The article will describe the basic ideas, concepts, components and approaches involved in developing an anti-virus program from scratch from a developer's/software engineer's point of view. It will focus on the main elements of an anti-virus engine (hereafter referred to as AV engine) and will exclude aspects like graphical user interfaces, real-time monitors, file system drivers and plug-ins for certain application software like Microsoft Exchange or Microsoft Office. Although AV engines running/scanning for single platforms (such as Palm OS or EPOC/Symbian OS) can be designed in the same way, this article will focus on designing multi-platform scanning engines, which are far more complex.

Overview

Currently, innovations in AV engines consist primarily of minor changes to existing engines. Complete redesigns of overall engine concepts are rarely seen. One exception is the highly respected [Kaspersky AntiVirus \(AVP\) version 4.0](#), which was released in early 2002.

The main parts of an AV engine are typically compiled based on the same source code for various platforms, which may have differences in the byte order (little/big endian), CPUs and general requirements on aligned code. All of these considerations must be kept in mind when developing the concept of an AV engine, as the platform on which the engine is designed to run will be a central design consideration. As well, when developing a new AV engine from the ground up, the following consideration or requirements must be considered:

- Targeted platforms
- Programming language
- File access
- Required modularity.

Targeted Platforms

A lot of platforms execute code faster when the data parts are aligned to long word (32 bit) addresses. Other platforms are not able to access 16bit/32 bit values, which are not on even addresses; for example, older Motorola CPUs like MC68020 had this limitation. The choice of programming language depends directly on the platform or platforms of implementation. Generally an AV engine should be developed in a programming language that is available for all platforms. Optimizing compilers for all platforms are available. Typical AV engines are currently developed using the programming languages C or C++. C++ is considered the more modern language but, being based on the object orientated approach, it is typically bigger and slightly slower than C code. As certain data types will be interpreted differently on different platforms (for example, as determined by long or integer variables), it is also very helpful to define data types based on standard data types, which are the same on all supported platforms.

File Access

To enable the core AV engine to be independent from the surrounding operating system, there must to be an abstraction layer between the core AV engine and the file system, which layer has to include conditional compilation for dedicated platforms. Another straightforward technique is to compile certain parts of the AV engine only for dedicated operating systems and not to use a file system layer at all. While this way approach results in faster programmed results, for the long term, it turns out to be neither easily maintainable nor expandable. An abstraction layer, comparable to the file system abstraction layer, should be also implemented for the memory interface and the graphical user interface, so that the core scan engine always has to call the same API calls to allocate memory, generate message boxes etc.

Modularity

Modularity is an important consideration in modern software development. Obviously, it is advantageous to create clean interfaces and make all program parts modular. By designing the overall AV engine with modularity in mind, single parts can be replaced later against a more powerful module by keeping the functionality the same. (This aspect will be covered in the discussion of on-line update functionalities later in this paper.) For corporate customers, it is especially important to offer a flexible management console/interface. This part obviously does not belong to the AV engine core, but should be kept in mind when designing overall interfaces, engine modules and communication matrixes. Speaking of modularity, it is also a good idea to divide the parts of the core AV engine into components, whereby the separation in a binary virus engine and a macro/script engine can be seen as a high level approach.

Pragmatic Functions

Now that some of the conceptual aspects of the AV engine design have been discussed, it would be helpful to consider some of the pragmatic functions that must be incorporated into the design of an AV engine. The following components or functions must all be taken into account in the development of a "modern" AV engine:

- Engine core
- File system layer
- File type scanners (rtf, ppt, mz, pe, etc.)
- Memory scanners
- File Decompression (e.g. ZIP archives, UPX compressed executables)
- Code emulators (e.g. Win32)
- Heuristic engines
- Update mechanisms.

AV Engine Core

The AV engine core can be seen as a straightforward framework that calls "external" scan modules and therefore can be expected to be the necessary "glue". As a result, it needs to be designed as a "registration" mechanism, so that additional components, such as a scanner for a new file format, can be registered and updated. This mechanism needs to be protected by digital certificates or similar mechanisms. Currently, there are scan engine frameworks, such as the Exchange virus protection, that offer to use between one and five different scan engines from different vendors, which will be directly called out of the framework. For example, besides their own scan technologies, [F-Secure](#) utilizes several solutions in their AV products, including F-Prot and AVP scan engines.

File System Layer

As mentioned in the previous section, it is a good idea to implement a file system layer so that all parts of the AV engine can invoke the same API calls on all platforms. The following functionalities (close to the Ansi-C standard) should be supported to enable easy access to files:

- open(filename)
- close(filehandle)

- read(file handler, buffer, length, number of read bytes)
- write(file handler, buffer, length, number of written bytes)
- seek(offset, optional fields)
- find first(handle)
- find next(handle)

In case a seek() functionality is not intended to be supported as an API call, the read/write functionality needs to be enhanced by adding a "file offset" field. The general "find first/find next" file functionality will typically only be used within the core AV engine, as this core part then passes the file pointer-like structure to the "external" scan modules for further operations.

File Type Scan

In regards to the program progression, one of the first steps is to identify the file type/archive type. For the time being, let's call this point within the engine the "entry point". This can be handled from the core AV engine or from a dedicated function call within every scanner module for a dedicated file format/type. In order to enable easy change/adaptation of a new scanner module, the latter method is preferred.

Typically, this file type check can be performed rather quickly (e.g. for Windows PE files, OLE documents etc.). In dedicated cases like PalmOS PRC files (see the SecurityFocus article [Palm OS: a Platform for Malicious Code? Part One](#)) the detection is more complex and, again, should not be placed within the core AV engine. If a compressed file is detected, decompression engine/functionality, which shall be discussed in greater detail later in this article, has to be called. More generally speaking, decompression engines can also be seen as some kind of a scanner module, which necessarily has to call back to the AV engine's entry point.

After the file type has been determined, the corresponding scanner module has to be called to perform the scan routine itself. Every module should have the ability to call back to the entry point of the AV engine. This may be required in the case of scanning embedded files within other files (for example, a Word document embedded within a PowerPoint presentation). Depending on the result of the scan, the AV engine must be able to interact with the user interface via a generic abstraction layer to show certain warnings, requests, etc.

At this point it makes sense to define what functionalities should exist within every scanner module:

- file type detection code, which checks whether the given input can be handled by the scan module;
- scan functionality (which should be able to interact with the GUI elements to show requesters etc.); and,
- removal functionality (e.g. remove link viruses from infected files or delete files completely).

The idea is to keep the interface as small and clean as possible. The scan modules should not rely on any buffers located in the core AV engine. Furthermore, the core scan module should just see file/memory pointers and work with these pointers. All underlying operations/layers should be fully transparent for the scan module.

Removal Functionality

In the case of removal functionality, it is often necessary to remove registry entries in order to disable the activation of certain malicious code. This functionality, which is obviously heavily dependent on the underlying platform, should be programmed using direct operating system functions, and should be compiled only when needed. At this point it makes no sense to implement an abstraction layer.

Memory Scanning Components

The memory scanning components (e.g. memory scanner for Windows 95/98 IFS-based malicious codes) can be placed within the same category as the registry cleaning functionalities described above. It should be noted that the memory scanning components are often not within the main focus of the development of the AV engine.

Decompression

The decompression functionality within AV engines is often seen as a small task, but it is truly a complex program. On the one hand, archives, like .zip, .tar, etc., and exchange formats, such as mime, uuencode etc., are decompressed recursively and without the need for external decompression programs. On the other hand, executable files should be able to be decompressed. Speaking of decompression of archives/exchange formats, it seems to be a good approach to decompress all files within a predefined directory and perform recursive decompression operations, if necessary. In the past we have seen a couple of attacks (see

[42]) against decompression modules, that decompressed the embedded files within memory and the system was running out of memory. The file located at [42] is a .zip archive with a total length of 42 kb. Recursively unpacked, the files archived within this file are far more than 100 MB, so that an "in memory" decompression would obviously decrease performance drastically.

Additionally it should be possible to compress the files into archives again to enable meaningful cleaning operations. The decompression operation, therefore, also needs access to the generic file system layer to store/access decompressed files.

Speaking of compressed executable files (e.g. compressed with UPX), a similar approach is possible. The decompressed file can be saved in a predefined directory and then scanned. Another typical approach is to decompress the entire file into memory and pass back the pointer and length of the file to the calling instance. The file system layer would have then to be able to address a memory range also as a file.

Finally, it should be noted that encrypted files/archives are still a major problem for decompression engines and therefore also for AV engines. Nearly all archive tools offer the possibility to encrypt the content.

Detection Engines and Techniques

Right now it is worth taking a look at detection engines and techniques beside heuristic engines (which have been discussed in another article by this author, [Heuristic Techniques in AV Solutions: An Overview](#)).

Nearly every modern AV engine contains checksum-based engines (often straight forward CRC32) and scan string-based engines. In addition to these basic techniques, script-based interpreters can often also be found in engines. By implementing these interpreters with complex instruction sets, it is possible to write detection/removal routines even for highly complex polymorphic viruses, and often without the need to change the engine/program detection code in C/C++. Obviously, these interpreters need access to emulators, memory layers and file system layers to become as powerful as possible. The interpreters typically work with precompiled code (pcode) located in the data/definition files.

Designing the On-Line Update

The core points of AV engine architecture have now been discussed. Another point to consider is the design of the "on-line update" functionality that allows users to update their AV protection. Basically there are two choices of update functionality: update data files or update data files and update executable code.

Generally speaking, all updates should be digitally signed to protect the users from installing malicious updates. It is not critical to implement this in the data file updates. Sending out only updated from previously installed versions, instead of complete update files, will keep network traffic low and, as such, is an attractive feature for users in corporate environments. To update executable code using on-line functionality is usually a more complex operation. This approach typically replaces complete modules of an AV scanner. Therefore the AV engine needs to have the functionality to register, remove, update and add modules of its own. This interface obviously needs to be protected (for example, by digital certificates), otherwise malicious codes could start to attack this registration interface and disable certain important functionality.

Conclusion

At this point it is clear that the development of a complete AV engine for a platform like Windows is an extremely complex task, one that needs to be undertaken by a group of developers. To keep an AV engine stable and maintainable over a long time is a difficult job that requires a lot of investment of money and experience in software engineering. Therefore it is not likely that the selection of independent AV solutions will increase significantly within the next years. This is unfortunate because the technical requirements on AV engines continue to grow and a greater variety of possible solutions can only help AV developers and AV users.

References:

[42] <http://www.hanau.net/fgk/downloads/42.zip>

Markus Schmall is currently working at the T-Mobile Germany in IT Security department and can be reached at markus@mschmall.de. He developed in old AMIGA days a complete AV engine, worked in the AV industry and now writes an AV engine as part of his PHD thesis.

[Privacy Statement](#)

Copyright 2006, SecurityFocus