

Heuristic Techniques in AV Solutions: An Overview

Markus Schmall 2002-02-04

Heuristic Techniques in AV Solutions: An Overview

by Markus Schmall

last updated February 4, 2002

Heuristic technologies can be found in nearly all current anti-virus (herein referred to as AV) solutions and also in other security-related areas like intrusion detection systems and attack analysis systems with correlating components. This article will offer a brief overview of generic heuristic approaches within AV solutions with a particular emphasis on heuristics for Visual Basic for Applications-based malware.

What is Heuristic Scanning?

Traditionally, AV solutions have relied strongly on signature-based scanning, also referred to as scan string-based technologies. The signature-based scan engine searches within given files for the presence of certain strings (often also only in certain regions). If these predefined strings are found, certain actions like alarms can be triggered. Modern scan string-based engines also support wildcards within the scan strings, which e.g. makes the detection of slightly polymorphic malicious codes much easier. However, signature-based scanning only detects known malware and may not detect against new attack mechanisms.

Heuristic scanning is similar to signature scanning, except that instead of looking for specific signatures, heuristic scanning looks for certain instructions or commands within a program that are not found in typical application programs. As a result, a heuristic engine is able to detect potentially malicious functionality in new, previously unexamined, malicious functionality such as the replication mechanism of a virus, the distribution routine of a worm or the payload of a trojan.

They do this by employing either weight-based systems and/or rule-based systems (both of which will be explained in greater detail later in this paper). A heuristic engine based on a weight-based system, which is a quite old styled approach, rates every functionality that is detected with a certain weight according to the degree of danger it may pose. If the sum of those weights reaches a certain threshold, also an alarm can be triggered.

Nearly all nowadays utilized heuristic approaches implement rule-based systems. This means, that the component of the heuristic engine that conducts the analysis (the analyser) extracts certain rules from a file and this rules will be compared against a set of rule for malicious code. If there matches a rule, an alarm can be triggered.

The first heuristic engines were introduced to detect DOS viruses in 1989. However, there now exist heuristic engines for nearly all classes of viruses (even for old-fashioned, nearly outdated Excel4 formula viruses like XF/ Paix). Over the years, AV development has been impressive, and the technologies utilized within heuristic engines have become more and more sophisticated. The first heuristic engines performed simple string- or pattern-matching operations to detect malicious code and were often referred to as "minimized scan string" heuristics. One example of this is evident in the following example string from VBA5 code:

```
Options.VirusProtection = 0
```

This string disables the built-in macro virus protection in Word97. A lot of heuristic engines for VBA-based macro viruses initially contained this line as a scan string. The obvious attack against this scan string was to change the representation of the "0". Another possible malicious string (as shown in a couple of macro viruses, like some W97M/Coldape variants) could be:

```
Options.VirusProtection = 1 AND 0
```

These technologies, which were introduced by virus programmers, are known as "anti-heuristic" technologies. They forced heuristic engines to scan more precisely and to analyse expressions (the logical operation 1 AND 0 results again in a 0).

Heuristic Engines and Encrypted Viruses

Historically, heuristic engines could only assess what was visible to them; as a result, encrypted viruses caused them major problems. In response to this, modern heuristic engines try to identify decryption loops, break them, and assess the presence of an encryption loop according to the additional functionality that is detected.

So how does an AV scanner identify an encryption loop (such as for M68k assembler as utilized on the current Palm OS platform)? The presence of any combination of the following conditions/instructions could indicate an encryption loop:

- initialization of a pointer with a valid memory address;
- initialization of a counter;
- memory read operation depending on the pointer;
- logical operation on the memory read result;
- memory write operation with the result from the logical operation;
- manipulation of the counter; and,
- branching depending on the counter.

A simple example for M68k assembler could look like this (assembler instructions match the above described conditions/instructions):

```

        Lea          test(pc),a0
        Move.l       #10, d0
.loop
        move.b       (a0), d1
        eor.b        #0, d1
        move.b       d1,(a0)+
        subq.l       #1,d0
        bne.s        .loop
        ...
test    dc.b         "Encryption with eor and key 0 !"
```

Of course, the example shown above is a quite trivial encryption loop, one that is quite easily detected by heuristic engines. Nevertheless, an understanding of how encryption loops can be realized is the basis of implementing a heuristic engine that is capable of detecting them. As a result we have seen a lot of viruses that try to hide the

encryption loops by inserting garbage code or making the encryption loop so long that the heuristic engine (to be more precise, the analysis component) gets confounded.

If we look at such detections, we see that, in most cases, the detection of an encryption loop is not precise enough for an exact classification, as several viruses could use the same encryption routines. In the world of binary viruses we have seen a lot of encryption engines (like TPE); generally speaking, in macro viruses these are not used in conjunction with common polymorphic engines (for example, the W97M/Pri engine is utilized quite often). Therefore, for the purpose of detection and removal, engines often communicate with and/or utilise emulator systems, which have, among other things, the ability to break and/or emulate encryption routines. After the encryption is broken (i. e., the end of the encryption loop has been reached), the heuristic analysis of the now decoded part can start. Depending on the environment, this emulation process is complicated; therefore for some platforms there exist no full emulators.

Visual Basic for Applications (VBA) and Visual Basic Script (VBS) are typical examples of complex environments in which emulators can be very helpful to break encryptions; however, a complete emulation is very complex. In most cases (such as the W97M/AntiSocial family, which utilizes encryption), a high number of encrypted instructions and the existence of typical macros like the Auto* macros or certain document handlers are already, without the usage of emulators, fully sufficient to detect this class of macro virus. This is evident in this example taken from the decryption engine of W97M/AntiSocial.D:

Line 1:

```
Private Sub Document_Open(): Application.EnableCancelKey = wdCancelDisabled
```

The definition of the private document handler Document_Open() (often inaccurately referred to as a macro) is not typical for common applications, so it should be flagged with a low priority. The next operation disables the 'ESC' key and has the same security risk level as the definition of the private document handler and, therefore, should be flagged accordingly.

Line 2:

```
For d = 6 To ThisDocument.VBProject.VBComponents.Item(1).CodeModule.CountOfLines: C$ = ""
```

This line simply initializes a 'For' loop, depending on the number of lines. Such strings should be flagged by heuristic engines, as a request to count the lines of the existing macro code is suspicious. Additionally a heuristic engine should remember that 'd' is an integer variable, the maximum value of which depends on the number of lines of code.

Line 3:

```
I = (ThisDocument.VBProject.VBComponents.Item(1).CodeModule.Lines(d, 1))
```

A line of code, depending on the counter, will be read from the macro code. The range from the counter is chosen

that way, so that every line of the malicious code can be accessed. Again, this can be seen as a memory-read operation as described above and should be flagged. Furthermore, the variable 'I' should be stored as a string variable containing line information.

Line 4:

```
f = (Mid(I, 2, 1)): For X = 3 To Len(I): B$ = Asc(Mid(I, X, 1)) - f: C$ = C$ & Chr(B$): Next X: A = C$
```

A set of operations will be done with the read content from the previous line. Actually, for the heuristic, the type of encryption that is occurring here is not really important; the existence of such a routine is suspicious enough and should be flagged. For emulation issues, the analysis of encryption functionality has to go deeper.

Line 5:

```
ThisDocument.VBProject.VBComponents.Item(1).CodeModule.ReplaceLine d, A: Next d: End Sub
```

This line replaces existing code (the parameter 'd' defines the line number and 'A' defines the actual content) and is another critical operation (equivalent to the memory-write operation mentioned above), which has to be flagged with a high security risk level. This line also contains the end of the outer 'for' loop, which is responsible for accessing all lines within a certain range of the document.

Line 6:

```
' 6Vxo|gzk&Y{h&Jui{sktzeIruyk./@&Uvzouty4Yg|kTuxsgrVxusvz&C&6
```

This line (as well as all of the following 13 lines) contains this kind of comment with encrypted code. How does the heuristic engine detect that this kind of comment is encrypted?

- the string is quite long (i.e., consists of more than forty characters) and contains no spaces;
- it is not typical to start a comment with a number; and,
- the string contains suspicious mixture of numbers, special characters and ordinary alphabet characters.

Even by looking at these six lines, it is quite obvious that this code contains suspicious operations, which is sufficient reason for a heuristic engine to issue an alert.

Nowadays, we also see engines that mix heuristic detection abilities with generic detection approaches. This means that the engines try to identify that a certain set of functionality found within a file belongs to a special class/family of malicious code. Removal capabilities are most often available for this kind of files detected by "class/family" detection.

Components of a Heuristic Engine

Depending on the environment and the technological level, the following components can be found within heuristic engines:

- variable/memory emulator;
- parser;
- flow analyzer;
- analyzer;
- disassembler/emulator; and,
- weight-based system and/or rule based system.

When looking at script-based malicious code, the first step for a detection engine, which does not necessarily have to be implemented as a part of a heuristic engine, may be to normalize the given input file and remove bad formatting, shorten irritating variable names and optionally tokenize the given script. Speaking of traditional macro viruses, there are also AV engines that work directly with the PCode (a meta code) found within the OLE file structures.

Once the file type is confirmed, the heuristic engine will check for the entry point of the given file. In case of binary files, this is fairly straightforward, as most files of this type have a clearly identifiable start point (for example, the Win32 PE entry point). For script-based malicious code, it is possible to have a couple of entry points (such as a couple of Auto* macros and document handlers within MS Word documents). The easiest approach is obviously to scan the complete program ignoring program flow. Obviously this approach will miss a lot of samples, such as those that use tricky parameter parsing between macros/function; as such, it typically has a high risk of returning inadequate results.

The main loop of every heuristic engine has to select/extract the information (typically the opcodes for the next instruction, or the next line in case of script-based malware) and pass the instruction to the core analyzer element. This analyzer element has to identify the operation and set flags according to this identification. Furthermore the communication with possible variable emulators or memory emulators is typically handled by the analyzer part. Looking back at the W97M/AntiSocial.D example, it is important for the analyzer to know that the variable 'd', as used in line 3 and 5, is actually not static or dependent on the number of code lines. The rating is obviously higher, when the variable 'd' is not static.

Rating the Found Functionality

After the complete program has been analyzed, the found functionality can be rated. This task is typically performed by weight-based systems or rule-based systems. The former system gives every found functionality a special weight and simply adds weights of the found functionalities. This type of technology is not often used in its basic form anymore, as it causes a lot of false positives. For macro viruses, traditional weight-based systems could produce a very high rating if a high number of copy operations from the current document to the global document template ("normal.dot") are found. This rating could result in a warning, even if no other malicious operation is found. So the AV programmers had to implement systems that produced an alarm only if special conditions are met, so the idea of utilizing rule-based systems within heuristic AV solutions was born.

Obviously, much better results can be reached when using rule-based systems. A rule-based system simply compares found functionality with a set of rules. If a predefined rule is found within the code, the rule-based system returns with a positive result. Depending on the exactness of the complete system, results like "generic virus" or e. g. "VBS/Loveletter variant" are realizable. Nevertheless, it should be never forgotten that heuristic engines can

cause false positives; for example, if the weight-based system is trained falsely or there are bad rules deployed within the rule-based system.

Current Situation - Why Do We Need Heuristics?

Having offered a brief overview of heuristic approaches and components of heuristic engines, we want to look more closely at why heuristic approaches are useful for both the user and the AV companies. In the last couple of years we have seen a couple of outbreaks (W97M/Melissa, VBS/Loveletter, W32/Nimda, ... just to name a few) that have illustrated how the need for protective solutions based on heuristic approaches in general have become more urgent. Additionally, we have seen a lot of malicious code that simply copies known ideas. As a result, this kind of malicious code offers perfect attack points for heuristic engines. When heuristic engines and generic approaches are capable of detecting slight variants of known malware, the AV research labs can look at other problems and optimize their time handling.

We have also encountered an increase in polymorphic/metamorphic malware, which often can be only detected by algorithmic approaches like a heuristic engine. Taking these developments into account, the usage of heuristic technologies within AV solutions is absolutely necessary. Furthermore, AV solutions are not the only area in which to utilize heuristic technologies. It is also possible to add heuristic features (e.g. utilizing rule-based systems) to intrusion detection systems and firewalls.

Markus Schmall is currently working in the IT Security department of T-Mobile Germany and can be reached at markus@mschmall.de.

[Privacy Statement](#)

Copyright 2006, SecurityFocus