How Antivirus Software Works

The antivirus software industry is huge and well-funded. As you would expect, they employ a large number of malware detection and identification personnel. (I will use the preferred term "malware" to mean all bad or malicious software including viruses, spyware, rootkits, worms, etc.)

Many of these people have a hacker or virus-making background, so the things we are doing here are not mysterious to them. (Eugene Kaspersky, the founder Kaspersky Labs, is rumored to have worked at one time as a hacker/cryptographer for Russia's KGB. Many others in the industry have a background in virus/worm creation.)

The AV industry attempts to collect as much information as they can about threats new and old and package that information into their software. The older a tool or technique is, the greater the chance they will have a mechanism to detect it.

Types of AV Detection Mechanisms:

Signature-Based

The key to making good AV software is to have a complete database of all malware signatures. These signatures are the essential part of the malware that distinguishes it from other software. That's why its key for users to keep those databases updated daily. As new malware is introduced daily into the wild, an out-of-date database of signatures can become nearly useless.

For a better concept of what these signature databases are like, you might take a look at Snort, an open-source IDS that detects malware and attacks on the wire. The beauty of working with Snort is that its signature database is open and viewable to anyone.

In the screenshot below, you can see some of the rules or signatures from the *web-attacks.rules* file from Snort. To better understand malware signatures, you may want to take some time to study these "signatures." It's important to note that these rules don't look for the entire file, but simply the single element of the malware that is unique or its "signature." Obviously, this is much more efficient.

```
S $HTTP_PORTS (msg:"WEB-ATTACKS /bin/ps command attempt"; flow:to_server,established; uricontent:"/bin/ps"; nocase; classtype:we
S $HTTP_PORTS (msg:"WEB-ATTACKS ps command attempt"; flow:to_server,established; uricontent:"ps%20"; nocase; classtype:web-appli
S $HTTP_PORTS (msg:"WEB-ATTACKS wget command attempt"; flow:to_server,established; content:"wget%20"; nocase; classtype:web-appl
S $HTTP_PORTS (msg:"WEB-ATTACKS uname -a command attempt"; flow:to_server,established; content:"uname%20-a"; nocase; classtype:w
S $HTTP_PORTS (msg:"WEB-ATTACKS /usr/bin/id command attempt"; flow:to_server,established; content:"/usr/bin/id"; nocase; classty
S $HTTP_PORTS (msg:"WEB-ATTACKS id command attempt"; flow:to_server,established; content:"|3B|id"; nocase; classtype:web-applica
S $HTTP_PORTS (msg:"WEB-ATTACKS echo command attempt"; flow:to_server,established; content:"/bin/echo"; nocase; classtype:web-ap
S $HTTP_PORTS (msg:"WEB-ATTACKS kill command attempt"; flow:to_server,established; content:"/bin/kill"; nocase; classtype:web-ap
S $HTTP_PORTS (msg:"WEB-ATTACKS chmod command attempt"; flow:to_server,established; content:"/bin/chmod"; nocase; classtype:web-
S $HTTP_PORTS (msg:"WEB-ATTACKS chgrp command attempt"; flow:to_server,established; content:"/chgrp"; nocase; classtype:web-appl
S $HTTP_PORTS (msg:"WEB-ATTACKS chown command attempt"; flow:to_server,established; content:"/chown"; nocase; classtype:web-appl
S $HTTP_PORTS (msg:"WEB-ATTACKS chsh command attempt"; flow:to_server,established; content:"/usr/bin/chsh"; nocase; classtype:we
S $HTTP_PORTS (msg:"WEB-ATTACKS tftp command attempt"; flow:to_server,established; content:"tftp%20"; nocase; classtype:web-appl
S $HTTP_PORTS (msg:"WEB-ATTACKS /usr/bin/gcc command attempt"; flow:to_server,established; content:"/usr/bin/gcc"; nocase; class
S $HTTP_PORTS (msg:"WEB-ATTACKS gcc command attempt"; flow:to_server,established; content:"gcc%20-o"; nocase; classtype:web-appl
S $HTTP_PORTS (msg:"WEB-ATTACKS /usr/bin/cc command attempt"; flow:to_server,established; content:"/usr/bin/cc"; nocase; classty
S $HTTP_PORTS (msg:"WEB-ATTACKS cc command attempt"; flow:to_server,established; content:"cc%20"; nocase; classtype:web-applicat
S $HTTP_PORTS (msg:"WEB-ATTACKS /usr/bin/cpp command attempt"; flow:to_server,established; content:"/usr/bin/cpp"; nocase; class
S $HTTP_PORTS (msg:"WEB-ATTACKS cpp command attempt"; flow:to_server,established; content:"cpp%20"; nocase; classtype:web-applic
S $HTTP_PORTS (msg:"WEB-ATTACKS /usr/bin/g++ command attempt"; flow:to_server,established; content:"/usr/bin/g++"; nocase; class
S $HTTP_PORTS (msg:"WEB-ATTACKS g++ command attempt"; flow:to_server,established; content:"g++%20"; nocase; classtype:web-applic
S $HTTP_PORTS (msg:"WEB-ATTACKS bin/python access attempt"; flow:to_server,established; content:"bin/python"; nocase; classtype:
S $HTTP_PORTS (msg:"WEB-ATTACKS python access attempt"; flow:to_server,established; content:"python%20"; nocase; classtype:web-a
S $HTTP_PORTS (msg:"WEB-ATTACKS bin/tclsh execution attempt"; flow:to_server,established; content:"bin/tclsh"; nocase; classtype
S $HTTP_PORTS (msg:"WEB-ATTACKS tclsh execution attempt"; flow:to_server,established; content:"tclsh8%20"; nocase; classtype:web
S $HTTP_PORTS (msg:"WEB-ATTACKS bin/nasm command attempt"; flow:to_server,established; content:"bin/nasm"; nocase; classtype:web
S $HTTP_PORTS (msg:"WEB-ATTACKS nasm command attempt"; flow:to_server,established; content:"nasm%20"; nocase; classtype:web-appl
S $HTTP_PORTS (msg:"WEB-ATTACKS /usr/bin/perl execution attempt"; flow:to_server,established; content:"/usr/bin/perl"; nocase; c
S $HTTP_PORTS (msg:"WEB-ATTACKS perl execution attempt"; flow:to_server,established; content:"perl%20"; nocase; classtype:web-ap
S $HTTP_PORTS (msg:"WEB-ATTACKS nt admin addition attempt"; flow:to_server,established; content:"net localgroup administrators /
S $HTTP_PORTS (msg:"WEB-ATTACKS traceroute command attempt"; flow:to_server,established; content:"traceroute%20"; nocase; classt
S $HTTP_PORTS (msg:"WEB-ATTACKS ping command attempt"; flow:to_server,established; content:"/bin/ping"; nocase; classtype:web-ap
S $HTTP_PORTS (msg:"WEB-ATTACKS netcat command attempt"; flow:to_server,established; content:"nc%20"; nocase; classtype:web-appl
S $HTTP_PORTS (msg:"WEB-ATTACKS nmap command attempt"; flow:to_server,established; content:"nmap%20"; nocase; classtype:web-appl
S $HTTP_PORTS (msg:"WEB-ATTACKS xterm command attempt"; flow:to_server,established; content:"/usr/X11R6/bin/xterm"; nocase; clas
S $HTTP_PORTS (msg:"WEB-ATTACKS X application to remote host attempt"; flow:to_server,established; content:"%20-display%20"; noc
```

The AV software companies maintain individuals and honeypots around the world that are constantly searching for new malware, and when they find it, they try to condense its signature down to the essentials. In addition, they have users all over the globe who send them suspicious malware they find.

These signatures, as I'm sure you guessed, are only effective on known malware. A zero-day attack would *not* have a signature and therefore would likely go undetected by the antivirus software.

One of the exceptions to this is that some malware production tools leave a signature on their output, so even a new piece of malware might be detected by AV if the tools used to create it have a signature that the AV developers have identified and coded for. A good example of this that many of you have found is the [msfvenom](#) module in [Metasploit](#). The template that creates the payloads has a signature, so no matter how we re-encode our payload, it still has a known signature. The key to defeat the AV with this module is to use a new template.

Heuristics

In some cases, it is not possible to have a signature for all malware, and in those cases, the AV developers attempt to deploy heuristic techniques to detect malware.

Wikipedia defines heuristic techniques as "...any approach to problem solving, learning, or discovery that employs a practical methodology not guaranteed to be optimal or perfect, but sufficient for the immediate goals. Where finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of finding a satisfactory solution."

James Whitcomb Riley once said, "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck." That principle summarizes heuristics succinctly. If it looks like malware and behaves like malware, it's probably malware.

The AV software developers look for telltale signs that the software's structure or behavior is malicious and then treat it like malware, even if no signature exists. For instance, if a file begins to replace several system files, it's probably malware. If a piece of software is trying make a TCP connection back to a known malicious IP address, it's probably malware. The beauty of this strategy is that it works with zero-day malware just as well as known malware. The drawbacks of this strategy are that it requires more compute cycles *and* the false positives that are often produced by innocuous software.

Behavior-Based

Some people create a separate category of behavior-based malware detection, but I consider it to be a subcategory of heuristics. Behavior is part of the "walks like a duck" mentioned above.

Sandbox

In some cases, suspicious software can be run in a virtual machine environment to see what it will do *before* it is installed on the system. This is referred to as "sandboxing." In this way, the AV software can study what it does or tries to do and then determine whether it is safe without endangering the entire system.

Image via [Lowe's]

Due to the time and resources necessary, this isn't a practical way of dealing with all software and files, but for particularly suspicious ones, it can be effective.