Multi-Stage Delivery of Malware

Marco Ramilli
Dipartimento di Elettronica Informatica e Sistemistica
University of Bologna
Via Venezia, 52 - 47023 Cesena â ITALY
marco.ramilli@unibo.it

Matt Bishop
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562, USA
bishop@cs.ucdavis.edu

Abstract

Malware signature detectors use patterns of bytes, or variations of patterns of bytes, to detect malware attempting to enter a systems. This approach assumes the signatures are both or sufficient length to identify the malware, and to distinguish it from non-malware objects entering the system. We describe a technique that can increase the difficulty of both to an arbitrary degree. This technique can exploit an optimization that many anti-virus systems use to make inserting the malware simple; fortunately, this particular exploit is easy to detect, provided the optimization is not present. We describe some experiments to test the effectiveness of this technique in evading existing signature-based malware detectors.

1 Introduction

Ever since Cohen's 1984 paper [6] described computer viruses in detail, a battle has raged between virus writers and anti-virus defenders. The simple computer virus has evolved into more complex stealth, polymorphic, and metamorphic engines. In parallel, anti-virus[1] systems have become more complex; no longer are simple scans for code signatures sufficient. Indeed, these systems now use techniques such as emulation, behavior analysis, sandboxing, and other forms of isolation to protect systems.

The defenses come with a price: *data objects* (which include downloaded entities such as applets on the World Wide Web, files, and email attachments) must be scanned and tested in other ways for malware. Because of the large number of different kinds of malware (over 22,000,000 as of early 2009 [5]), it is considered impractical to scan all incoming data objects for all types of malware. Thus, systems differentiate among the vectors used to put malware on systems. For example, macro viruses intended for Microsoft Word must be in Word documents to be effective, and so anti-virus programs typically do not scan incoming executables for those viruses—but they do scan any incoming Microsoft Word files for them. This creates a "gap" in protection. If, for example, a macro virus were embedded in an executable file in such a way that the executable file would ignore it when executed, but a second program could locate that virus and load it into an *existing* Microsoft Word document in such a way that the virus would be triggered when the file were opened, the anti-virus programs would not detect the macro virus' entry onto the system. The point of detection would therefore need to be the loading program.

This view of the malware attack is of a three-step process. The first step is to place the malware onto the system. The second step is to assemble the malware. The third step is to execute the assembled malware. More customary views of the process conflate the first and second steps into one, under the guise of *infection* (and the third step is the *execution* step). Anti-virus programs typically attempt to block the first two steps by detecting and preventing malware from entering the system. They require that *both* steps have taken place, because their signatures require that specific parts of the malware be detectable.

Anti-virus programs that seek to detect incoming malware use two primary techniques. The first, which we call *data signature scanning*, is to look for patterns in the incoming data objects that match known malware—*signatures*—and, when found, take some action, for example deleting the incoming data or quarantining it and notifying the user. The second, *behavior signature scanning*, emulates the data object's execution either statically (determining what instructions would be executed) or dynamically (placing it in a sandbox and executing it, with the sandboxing intercepting all system calls and possibly library calls, looking for patterns that match behavior of malware). Both techniques assume that enough of the malware is present in the data object being examined to trigger an alert. As stated above,

[1] Here, we follow the industry custom of calling anti-malware detection programs "anti-virus" programs.

these techniques all combine entry onto the system with assembly in their view of the malware life cycle on a system.

Consider an alternate view. What happens if assembly occurs *after* placement on the system? That is, portions of the malware are placed on a system, then the malware is assembled, and then it is executed—three distinct steps instead of two. This view negates the assumption that enough of the malware is present in the data object to be identified as malware. We exploit this view by partitioning our malware into multiple pieces, none of which alone contains enough of a signature to trigger an anti-virus alert. The pieces are placed onto the target system, and some time later are assembled together. This combined code is sufficient to act as malware.

The argument that this malware will then be detected by the system when it is executed, and therefore this attack is inconsequential, assumes that the system has an anti-virus engine monitoring all processes as they execute—and that the anti-virus program is correctly configured and correctly identifies all malware as such by its behavior. This assumption is of course questionable; at any rate, by that argument, no incoming data object would need to be checked for malware because all malware would be detected on execution. The magnitude of business, and the amount of research into, the detection of malware as it enters the system demonstrates that this argument is not widely accepted. Indeed, it violates the principle of separation of privilege (also known as "layers of defense") [16] because it contends that one layer of defense is sufficient.

After a brief survey of related work, we present the design of our attack, and then report on experiments. We conclude with a discussion of future directions and some ideas on how to apply this work to defeat the execution monitoring of anti-virus defenses.

2 Related Work

Multi-stage attacks are well-known. One of the earliest was the Internet worm [9], which placed a "grappling hook" on the target system. When the grappling hook was executed, the rest of the worm was pulled over. Ptacek and Newsham [15] used network hop counts to cause packets to be dropped. This fragmented attack commands into multiple packets interspersed with irrelevant data that was discarded after the intrusion detection system of the target site examined the stream for attacks, but before the stream reached the target. Other multistage attacks, often in the guise of malware (see for example [2, 4, 7, 12] are "multistage" in their activation or execution. Models [8, 14, 19] and interpretative methods such as visualization [13] have been created and applied to help understand how multistage attacks work and how they spread.

Of these attacks, the Internet worm is closest to what we describe. The main difference is that the worm uses the grappling hook to pull over an object file that must be linked to local libraries and resources in order to execute. Many existing worms work similarly, exchanging messages with other hosts and copies of the worm to propagate and to control their spread. Our attack focuses on constructing the malware from data resident on the current host.

The computer viruses Dichotomy [10] and RMNS [11] each consisted of two components. When executed, they operated as TSRs. Dichotomy intercepted the "Load_and_Execute" call, and either infected the file with the "loader" (that changed the file entry point to invoke the virus) and the virus body, or simply with the virus body. RMNS had two parts, one of which intercepted the call, and the other of which infected files. The infection part infected the file with the interception code half the time, and the infector the other half of the time. These viruses differ from our approach because we fragment malware into parts that can enter a system, and then be combined to create the malware. The components themselves need not do anything in particular, or indeed even *do* anything—until they are assembled in memory.

Sun, Ebringer, and Bostas [17] build on polymorphic malware that uses encryption to evade detection. This type of malware encrypts the unpacking routine, which is then decrypted just before execution and re-encrypted just after execution (called "multistage unpacking"). Our approach omits encryption, or indeed any obfuscation beyond the breaking up of the malware in multiple chunks that can then be reassembled and executed. A second difference is that we evade *only* detection at the injection of the malware components. Once the malware is assembled and executed, it is susceptible to detection through behavioral analysis.

Current work on evading signature-based anti-virus techniques focuses on obfuscation-based systems, including self-encrypting, polymorphic, and metamorphic malware. Self-encrypting malware was first found in the Cascade virus [3], and consisted of an initial decryption routine followed by the encrypted virus. By altering the key (based on the size of the file), the body of the virus would appear to change. The next stage grew from the need to hide the decryption routine. Polymorphism, in which instructions are replaced by equivalent instructions, helped hide those routines. Indeed, tools such as the Mutation Engine and the TridenT Polymorphic Engine automated generation of polymorphic malware [18]. However, enough non-metamorphic malware is still in use that signature-based scanning is productive. Current anti-virus engines use a variety of techniques to speed the checking of incoming data objects. Most notably, they look for malware relevant to the type of data object being analyzed. For example, the Melissa worm [1] is a worm that is loaded into Microsoft Word documents, and is then executed by the Visual Basic

interpreter. Thus, anti-virus systems typically do not check incoming executable data objects for Melissa, because executing a program will not cause Melissa to run; but editing an infected Microsoft Word document with Microsoft Word would execute (interpret) Melissa, so data objects that are Microsoft Word documents would be checked.

Packing, a technique in which malware is compressed and encrypted (often polymorphically) is closest to our method, but there are significant differences. First, packed malware typically has multiple stages (for example, the execution of the unpacker, which then unpacks and executes the malware proper) but these are typically in the same object. In our method, the malware is in multiple objects. Second, our method does not require encryption or other obfuscation (although it would of course benefit from them) because the malware is fragmented to the point that the individual components cannot be recognized. This is a form of obfuscation, but one involving breaking the malware into components each of which is too small to be recognized.

3 Design of Multi-Stage Malware

Our technique exploits the need for anti-virus scanners to look for sequences to determine whether the file contains malware—either sequences of known data (code signatures) or indicating behavior such as malware exhibits (behavior signatures). This sequence analysis assumes that the sequence is present in a single data object. This data object is the malware's infection vector. Figure 1 represents the high-level view of our attack.

The malware is broken into several *components* that are then embedded in numerous other data objects. The components are not necessarily functions or blocks of code performing well-defined actions within the malware; they may be as simple as 200-byte sequences of instructions and data in the malware. The critical feature of this fragmentation is that no single data object contains a signature that the relevant anti-virus program will flag as indicating the presence of malware. One distinguished data object (the *main data object*) contains the component (the *main actor*) that, when executed, reassembles the fragmented components into the malware and executes it.

Figure 1 summarizes this process. That figure shows n files $File^i$, each containing one of n parts p^i of the malware p. When $File^1$ is executed, it extracts the other components $p^2, ..., p^n$ of the malware from $File^2, ..., File^n$ (the figure shows this as an execution of the $Read()$ function). It then assembles these, in memory, to form a complete malware data object, which executes.

The main actor must locate the components of the malware. It can do so in a number of ways. It can look for specific flags or predetermined sequences of bytes, but this would render the *component* amenable to detection by an

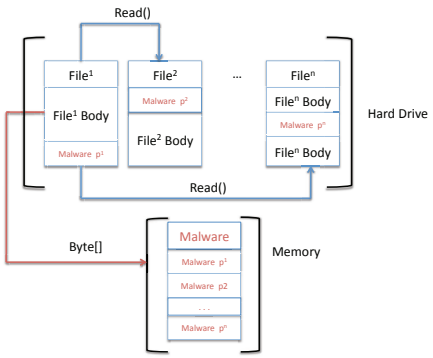

Figure 1. Injection of multi-stage malware onto a system

anti-virus signature scanner. It may also read from a predetermined location, or a location it computes based on the attributes of the containing file; in this way, the files containing the malware components will pass through the anti-virus signature scanning mechanisms.

In order to lessen the probability of a part of the malware being detected, we may exploit a common optimization of anti-virus engines. As noted earlier, most anti-virus scanners base their analysis of incoming data objects (files, applets, attachments, and so forth) upon the *type* of the file. This is usually, but not always, determined by examining the file name extension, for example ".exe" being a Windows executable file, ".doc" being a Microsoft Word document, and ".jpg" being a JPEG file. So, we simply place the components of our malware in a type of file unlikely to be scanned. If, for example, our malware is an executable, we place components in JPEG or other non-executable data objects. To summarize, the preconditions for this attack to work are:

1. The antivirus mechanisms must not flag as suspicious a file containing a portion of a malware signature;

2. The antivirus mechanisms must not flag as suspicious a program that loads multiple components into memory and executes them; and

3. The main actor must be able to locate the other parts of the malware, and execute *after* the files are resident on the system.

We discuss these in the next section.

File loader.exe received on 2010.01.06 21:55:41 (UTC)

Antivirus	Version	Last Update	Result
a-squared	4.5.0.48	2010.01.06	Trojan-Spy.Win32.Zbot.vb!IK
AhnLab-V3	5.0.0.2	2010.01.06	-
AntiVir	7.9.1.122	2009.12.31	TR/Spy.Agent.TH
Antiy-AVL	2.0.3.7	2010.01.06	-
Authentium	5.2.0.5	2010.01.06	W32/Banker.AVJI
Avast	4.8.1351.0	2010.01.06	Win32:Bancos-ASL
AVG	8.5.0.430	2010.01.04	PSW.Generic6.ACJO
BitDefender	7.2	2010.01.06	Trojan.Spy.Zeus.2.Gen
CAT-QuickHeal	10.00	2010.01.05	Win32.Trojan-Spy.Bancos.aam.4
ClamAV	0.94.1	2010.01.06	Trojan.Bancos-5978
Comodo	3490	2010.01.06	Heur.Packed.Unknown
DrWeb	5.0.1.12222	2010.01.06	Trojan.Proxy.2003
eSafe	7.0.17.0	2010.01.06	-
eTrust-Vet	35.1.7219	2010.01.06	Win32/KollahCryptozA!generic
F-Prot	4.5.1.85	2010.01.06	W32/Banker.AVJI
F-Secure	9.0.15370.0	2010.01.06	Trojan-Spy:W32/Bancos.AAM
Fortinet	4.0.14.0	2010.01.06	W32/Agent.BRN!tr
GData	19	2010.01.06	Trojan.Spy.Zeus.2.Gen
Ikarus	T3.1.1.79.0	2010.01.06	Trojan-Spy.Win32.Zbot.vb
Jiangmin	13.0.900	2010.01.06	Trojan/Tiba.puq
K7AntiVirus	7.10.940	2010.01.06	Trojan-PSW.Win32.Zbot.ch
Kaspersky	7.0.0.125	2010.01.06	Trojan-Spy.Win32.Zbot.vb
McAfee	5853	2010.01.06	Spy-Agent.bw.gen.b
McAfee+Artemis	5853	2010.01.06	Spy-Agent.bw.gen.b
McAfee-GW-Edition	6.8.5	2010.01.06	Trojan.Spy.Agent.TH
Microsoft	1.5302	2010.01.06	PWS:Win32/Zbot.ME
NOD32	4749	2010.01.06	a variant of Win32/Spy.Agent.PE
Norman	6.04.03	2010.01.06	Zbot.AM
nProtect	2009.1.8.0	2010.01.06	Trojan-Spy/W32.ZBot.41984.AP
Panda	10.0.2.2	2010.01.06	Malicious Packer
PCTools	7.0.3.5	2010.01.06	TrojanSpy.ZBot.Gen!Pac.3
Prevx	3.0	2010.01.06	-
Rising	22.29.02.06	2010.01.06	Trojan.Spy.Win32.Bancos.bay
Sophos	4.49.0	2010.01.06	Mal/Zbot-A
Sunbelt	3.2.1858.2	2010.01.06	Trojan-Spy.Win32.Zbot.gen
Symantec	20091.2.0.41	2010.01.06	Infostealer.Banker.C
TheHacker	6.5.0.3.137	2010.01.06	Trojan/Spy.Zbot.vb
TrendMicro	9.120.0.1004	2010.01.06	TSPY_BANKKYPT.X
VBA32	3.12.12.1	2010.01.06	Trojan-Spy.Win32.Zbot.wng
ViRobot	2010.1.6.2124	2010.01.06	Trojan.Win32.Tiba.41984
VirusBuster	5.0.21.0	2010.01.06	TrojanSpy.ZBot.Gen!Pac.3

File bug-feature_Zeus_MIddle.jpg received on 2010.01.06 22:02:42 (UTC)

Antivirus	Version	Last Update	Result
a-squared	4.5.0.48	2010.01.06	-
AhnLab-V3	5.0.0.2	2010.01.06	-
AntiVir	7.9.1.122	2009.12.31	-
Antiy-AVL	2.0.3.7	2010.01.06	-
Authentium	5.2.0.5	2010.01.06	-
Avast	4.8.1351.0	2010.01.06	-
AVG	8.5.0.430	2010.01.04	-
BitDefender	7.2	2010.01.06	-
CAT-QuickHeal	10.00	2010.01.05	-
ClamAV	0.94.1	2010.01.06	-
Comodo	3490	2010.01.06	-
DrWeb	5.0.1.12222	2010.01.06	-
eSafe	7.0.17.0	2010.01.06	-
eTrust-Vet	35.1.7219	2010.01.06	-
F-Prot	4.5.1.85	2010.01.06	-
F-Secure	9.0.15370.0	2010.01.06	-
Fortinet	4.0.14.0	2010.01.06	-
GData	19	2010.01.06	-
Ikarus	T3.1.1.79.0	2010.01.06	-
Jiangmin	13.0.900	2010.01.06	-
K7AntiVirus	7.10.940	2010.01.06	-
Kaspersky	7.0.0.125	2010.01.06	-
McAfee	5853	2010.01.06	-
McAfee+Artemis	5853	2010.01.06	-
McAfee-GW-Edition	6.8.5	2010.01.06	-
Microsoft	1.5302	2010.01.06	-
NOD32	4749	2010.01.06	-
Norman	6.04.03	2010.01.06	-
nProtect	2009.1.8.0	2010.01.06	-
Panda	10.0.2.2	2010.01.06	-
PCTools	7.0.3.5	2010.01.06	-
Prevx	3.0	2010.01.06	-
Rising	22.29.02.06	2010.01.06	-
Sophos	4.49.0	2010.01.06	-
Sunbelt	3.2.1858.2	2010.01.06	-
Symantec	20091.2.0.41	2010.01.06	-
TheHacker	6.5.0.3.137	2010.01.06	-
TrendMicro	9.120.0.1004	2010.01.06	-
VBA32	3.12.12.1	2010.01.06	-
ViRobot	2010.1.6.2124	2010.01.06	-
VirusBuster	5.0.21.0	2010.01.06	-

Figure 2. Analysis of Zeus in an executable and in a JPG file

4 Experiments

Define $AV(x)$ to be an anti-malware detection mechanism that returns *true* if the input to AV, namely X, is malware and *false* if not. Our question is whether we can decompose malware in such a way to avoid detection.

Our anti-virus function AV will be the set of anti-virus detectors at Virus Total, which includes most commercial anti-virus programs as well as open-source ones. We assume that Virus Total uses well configured and up-to-date AV engines. We also assume that the anti-virus tools there perform a static signature analysis on the given files.

We consider two well-known pieces of malware, Zeus (also known as Trojan.Zbot) and Spreder (also known as W32.HLLP.Spreda). As a control, we embedded Zeus into a Windows executable and then ran it through Virus Total. Figure 2 (left) shows that all but 4 anti-virus tools found the virus. Similarly, all but 8 anti-virus tools were able to detect Spreder. Thus, we know that both these pieces of malware will be detected by most anti-virus software.

Our first question is how to decompose the malware into components that will evade the anti-virus software. Preliminary to this is the question of whether we *have* to break it into components. Can we instead embed the *entire* malware into a file of the wrong type, and then have the main actor trigger its execution?

4.1 First Approach

As noted in the introduction, the large amount of malware means that scanning every incoming data object for every malware is generally prohibitively expensive—it would delay incoming data objects too long. So, modern anti-virus software makes an obvious optimization. An executable infector embedded in a JPG (image) file will not execute when the JPG file is displayed, because the bytes in the file are interpreted as a JPG image. Thus, anti-virus software will only look for malware that is triggered when the JPG image is displayed.

To verify this, we embedded Zeus in a JPG file. Figure 2 (right) shows that none of the anti-virus software products in Virus Total detected Zeus in that file. Contrast this with Figure 2 (left), where all but 4 anti-virus products detected Zeus. Interestingly, the effects of adding Zeus to the JPG file vary depend on how it is embedded. If placed immediately after the JPG header, Figure 3 (left) shows that the image is obviously corrupted. If placed just before the JPG trailer, Figure 3 (center) shows the corruption is minimal. And if placed after the JPG trailer, Figure 3 (right), no corruption is apparent. Thus, an attacker can embed malware into a file of an arbitrary type, and then inject it and the main actor into the system. If the main actor escapes detection, then it can execute the malware.

We now turn to the case where all datatypes are checked for a particular virus.

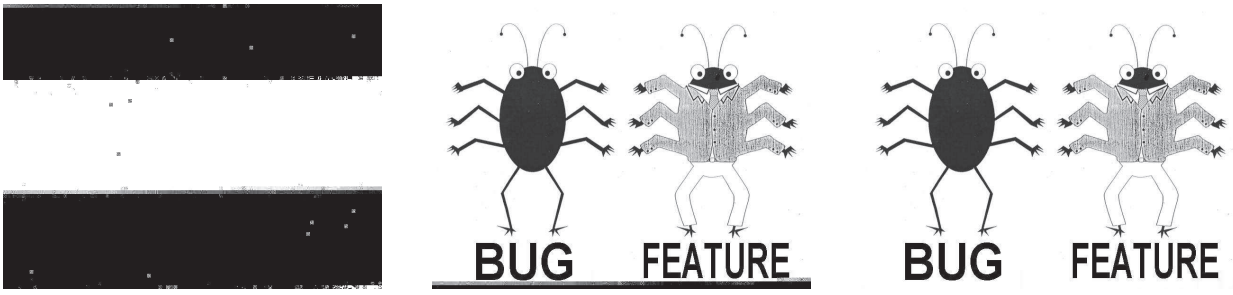

Figure 3. Image with Zeus embedded: just after the JPG header (left), just before the JPG trailer (center),after the JPG trailer (right)

4.2 Second Approach

Now we consider breaking down malware into a set of components that cannot be detected. We assume that the nature of the anti-virus software on the target system is not known; thus, we use a mechanism like Virus Total to check our components against multiple anti-virus software products. If we do know the *particular* anti-virus software on the target system, we need only consider it and not others.

We use an iterative approach. A simple program takes as input a set of files into which the malware is to be embedded, the number of components that the malware is to be broken into, and the malware. It breaks the malware into components and embeds one component into each file.

We decomposed Spreder into three parts, and embedded it in a JPG file. Only one of the anti-virus software under Virus Total detected the corruption of the container files, and that one identified the malware incorrectly (and as "suspected"); see Figure 4 (left). Rearranging the signatures by hand eliminated this alert, as shown in Figure 4 (right). Splitting Spreder into 2 components and embedding them in an MP3 file also escaped detection; Figure 5 shows the results of one such scan.

Our results for Zeus were similar. With Zeus, out of 42 anti-malware tools tested, only 7 reported potential malware on one or more of the components.

These results suggest that, for the majority of anti-virus programs in use today, this technique would enable malware to evade detection by anti-virus signature scanning.

These results indicate that, for the malware tested, at least 35 of the AV functions described above exist.

4.3 Main Actor

The main actor is a simple program. It locates the malware components and loads them into memory. The key to its success is its execution.

The main actor can be executed exactly the same way that malware is executed. Phishing, injection into a process or program, or other techniques enable this. For example, if a worm can inject specific instructions into a process through a buffer overflow, or an SQI injection attack can enable the uploading of an executable containing the main actor, then the main actor can load the components already resident on the system into memory, constructing the malware (and then executing it). Other techniques include the use of DNS cache poisoning and SEO abuse.

For demonstration purposes, we implemented this in the .NET framework. Using the common reflection technique, namely the ability of a managed code to read its own metadata for the purpose of finding assemblies, modules and type information at runtime, this program reconstructs the malware's code inside a memory buffer as shown in the following listing, and then executes it.

```
1
2  byte[] bin = new byte[stop - start + 1];
3  for (int c = 0; c <= stop - start ; c++)
4      bin[counter] = totalbin[start + c];
5  ...
6  Assembly a = Assembly.Load(bin);
7  MethodInfo method = a.EntryPoint;
8  ...
9   if (method != null){
10     object o =
11       a.CreateInstance(method.Name);
12     method.Invoke(o, null);
13 }
```

The "bin" variable collects the ordered malware components (lines 2–4). These become executable after being loaded as into memory (line 6). The *CreateInstance* method (line 11) builds the executable object from an entry point (line 7) that is activated by the *invoke* function (line 12). Figure 6 shows our main actor loaded in a Windows 32 system.

In theory, determining whether an arbitrary segment of code is the main actor is undecidable. In practice, the problem is more limited: can we characterize the main actor in such a way that it can be detected? The function of the

File bug-feature.jpg-1.jpg received on 2010.02.23 23:16:55 (UTC)

Antivirus	Version	Last Update	Result
a-squared	4.5.0.50	2010.02.23	-
AhnLab-V3	5.0.0.2	2010.02.23	-
AntiVir	8.2.1.172	2010.02.23	-
Antiy-AVL	2.0.3.7	2010.02.23	-
Authentium	5.2.0.5	2010.02.23	-
Avast	4.8.1351.0	2010.02.23	-
AVG	9.0.0.730	2010.02.23	-
BitDefender	7.2	2010.02.24	-
CAT-QuickHeal	10.00	2010.02.23	-
ClamAV	0.96.0.0-git	2010.02.23	-
Comodo	4040	2010.02.23	-
DrWeb	5.0.1.12222	2010.02.23	-
eSafe	7.0.17.0	2010.02.23	-
eTrust-Vet	35.2.7323	2010.02.23	-
F-Prot	4.5.1.85	2010.02.23	-
F-Secure	9.0.15370.0	2010.02.23	-
Fortinet	4.0.14.0	2010.02.21	-
GData	19	2010.02.23	-
Ikarus	T3.1.1.80.0	2010.02.23	-
Jiangmin	13.0.900	2010.02.23	-
K7AntiVirus	7.10.981	2010.02.23	-
Kaspersky	7.0.0.125	2010.02.23	-
McAfee	5901	2010.02.23	-
McAfee+Artemis	5901	2010.02.23	-
McAfee-GW-Edition	6.8.5	2010.02.23	-
Microsoft	1.5406	2010.02.23	-
NOD32	4891	2010.02.23	-
Norman	6.04.08	2010.02.23	-
nProtect	2009.1.8.0	2010.02.23	-
Panda	10.0.2.2	2010.02.23	-
PCTools	7.0.3.5	2010.02.23	-
Prevx	3.0	2010.02.24	-
Rising	22.34.01.03	2010.02.11	-
Sophos	4.50.0	2010.02.23	-
Sunbelt	5695	2010.02.23	-
Symantec	20091.2.0.41	2010.02.24	-
TheHacker	6.5.1.6.207	2010.02.23	-
TrendMicro	9.120.0.1004	2010.02.23	-
VBA32	3.12.12.2	2010.02.23	suspected of Exploit.Win32.EmbeddedPE.JPG
ViRobot	2010.2.23.2198	2010.02.23	-
VirusBuster	5.0.27.0	2010.02.23	-

File bug-feature.jpg-1.jpg received on 2010.02.23 23:59:20 (UTC)

Antivirus	Version	Last Update	Result
a-squared	4.5.0.50	2010.02.23	-
AhnLab-V3	5.0.0.2	2010.02.23	-
AntiVir	8.2.1.172	2010.02.23	-
Antiy-AVL	2.0.3.7	2010.02.23	-
Authentium	5.2.0.5	2010.02.23	-
Avast	4.8.1351.0	2010.02.23	-
AVG	9.0.0.730	2010.02.23	-
BitDefender	7.2	2010.02.24	-
CAT-QuickHeal	10.00	2010.02.23	-
ClamAV	0.96.0.0-git	2010.02.23	-
Comodo	4040	2010.02.23	-
DrWeb	5.0.1.12222	2010.02.23	-
eSafe	7.0.17.0	2010.02.23	-
eTrust-Vet	35.2.7323	2010.02.23	-
F-Prot	4.5.1.85	2010.02.23	-
F-Secure	9.0.15370.0	2010.02.24	-
Fortinet	4.0.14.0	2010.02.21	-
GData	19	2010.02.23	-
Ikarus	T3.1.1.80.0	2010.02.23	-
Jiangmin	13.0.900	2010.02.23	-
K7AntiVirus	7.10.981	2010.02.23	-
Kaspersky	7.0.0.125	2010.02.24	-
McAfee	5901	2010.02.23	-
McAfee+Artemis	5901	2010.02.23	-
McAfee-GW-Edition	6.8.5	2010.02.23	-
Microsoft	1.5406	2010.02.23	-
NOD32	4891	2010.02.23	-
Norman	6.04.08	2010.02.23	-
nProtect	2009.1.8.0	2010.02.23	-
Panda	10.0.2.2	2010.02.23	-
PCTools	7.0.3.5	2010.02.23	-
Rising	22.34.01.03	2010.02.11	-
Sophos	4.50.0	2010.02.23	-
Sunbelt	5695	2010.02.23	-
Symantec	20091.2.0.41	2010.02.24	-
TheHacker	6.5.1.6.207	2010.02.23	-
TrendMicro	9.120.0.1004	2010.02.23	-
VBA32	3.12.12.2	2010.02.23	-
ViRobot	2010.2.23.2198	2010.02.23	-
VirusBuster	5.0.27.0	2010.02.24	-

Figure 4. Analysis of first part of Spreder in a JPEG file: automatic (left), manually arranged (right)

File Episode2-healthClub.mp3-2.mp3 received on 2010.02.19 23:34:36 (UTC)

Antivirus	Version	Last Update	Result
a-squared	4.5.0.50	2010.02.19	-
AhnLab-V3	5.0.0.2	2010.02.19	-
AntiVir	8.2.1.170	2010.02.19	-
Antiy-AVL	2.0.3.7	2010.02.19	-
Authentium	5.2.0.5	2010.02.19	-
Avast	4.8.1351.0	2010.02.19	-
AVG	9.0.0.730	2010.02.19	-
BitDefender	7.2	2010.02.20	-
CAT-QuickHeal	10.00	2010.02.19	-
ClamAV	0.96.0.0-git	2010.02.19	-
Comodo	3994	2010.02.19	-
DrWeb	5.0.1.12222	2010.02.20	-
eSafe	7.0.17.0	2010.02.18	-
eTrust-Vet	35.2.7313	2010.02.19	-
F-Prot	4.5.1.85	2010.02.19	-
F-Secure	9.0.15370.0	2010.02.19	-
Fortinet	4.0.14.0	2010.02.18	-
GData	19	2010.02.20	-
Ikarus	T3.1.1.80.0	2010.02.19	-
Jiangmin	13.0.900	2010.02.19	-
K7AntiVirus	7.10.977	2010.02.18	-
Kaspersky	7.0.0.125	2010.02.17	-
McAfee	5897	2010.02.19	-
McAfee+Artemis	5897	2010.02.19	-
McAfee-GW-Edition	6.8.5	2010.02.19	-
Microsoft	1.5406	2010.02.19	-
NOD32	4881	2010.02.19	-
Norman	6.04.08	2010.02.19	-
nProtect	2009.1.8.0	2010.02.19	-
Panda	10.0.2.2	2010.02.19	-
PCTools	7.0.3.5	2010.02.19	-
Prevx	3.0	2010.02.20	-
Rising	22.34.01.03	2010.02.11	-
Sophos	4.50.0	2010.02.19	-
Sunbelt	5686	2010.02.19	-
Symantec	20091.2.0.41	2010.02.19	-
TheHacker	6.5.1.5.202	2010.02.19	-
TrendMicro	9.120.0.1004	2010.02.19	-
VBA32	3.12.12.2	2010.02.19	-
ViRobot	2010.2.19.2194	2010.02.19	-
VirusBuster	5.0.27.0	2010.02.19	-

Figure 5. Detection of second part of Spreder in an MPEG3 file

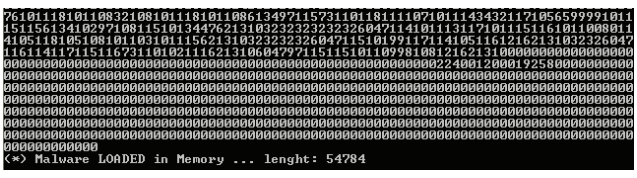

Figure 6. Screenshot of the Exploit.

main actor indicates the characteristic all main actors must share: the ability to load data from files and then execute that data. In some environments, it is not possible to distinguish between programs that do this for a benign purpose and programs that do this for a malicious purpose. For example, the above programming technique, called *reflection*, is widely used in Windows environments, and thus any anti-virus engine that flags it as a potential problem will create many false positives.

5 Conclusion and Future Work

This paper proposed an alternate view of the steps that malware uses to attack a system, and this view led to an application of the philosophy of multi-stage attacks to the delivery of a malicious payload that exploits the difficulty of correlating pieces of a malware before they are assembled into the actual malware itself.

This attack is actually a class of attacks, with many variations. For example, our experiments divided malware into roughly equal-sized parts. The malware could have been broken into random-sized parts, or the part detected as a signature could itself be fragmented, and the rest of the malware could be left intact. Or, the malware could be sent in a file of the wrong type (assuming the anti-virus engine does not check all files), and the main actor could be sent in a type of file that would be executed.

The key to creating this attack is determining how to split the malware to reduce its being detected. Clearly, breaking it into components the size of a few bytes works; in-

deed, in that case it may be possible to avoid injecting it into files, but simply load the bytes from files that happen to contain them. (In the extreme, one can conceive of a main actor constructing malware from operating system, configuration, and application files.) Scanning an executable to detect the loading of data and then the execution of that data is of course undecidable in the general case. In specific cases it can be done. However, detecting the standard hooks that enable this, such as the .NET "load binary" API, will cause many false positives because much software uses those APIs. Further, many programs that use reflection will also be flagged. Thus, this technique appears not to be amenable to detection by signature scanning.

In fact, one could be more subtle. The attack could masquerade as a buffer overflow. For this approach, the main actor would simply read data into a buffer that was of size sufficient to hold the malware. The malware is loaded, and then some extra data, designed to produce a return to the stack, overwrites the return address on the stack. When the main actor executes a "return from procedure" instruction, the malware executes. Note this only works if a buffer overflow attack can execute instructions in stack space (some systems prevent this). Behavior analysis, or analyzing the program as it executes, will detect this type of attack. Basically, once the malware is assembled in memory and executed, an anti-malware mechanism would not know *how* the malware was loaded onto the system; it simply detects its execution. So this type of attack can be thwarted with current technology, but only once the malware is resident.

Two avenues of research will determine how effective this attack is. The first is to test the attack under varying conditions. Specifically, our work used the set of anti-malware detectors at VirusTotal. Thus, we can claim only that, *against the tools as configured there*, this attack is effective. Alternate configurations might be more effective. This needs to be checked.

The second avenue is to apply this method to behavioral detection techniques. Specifically, once the malware is assembled and executed, standard behavioral detection techniques will flag the executing process as malicious. Is it possible to break the executing process up in such a way that standard behavioral analysis techniques will not detect the malicious actions? It is clear such an approach works if the analysis is done on a per-process basis; it is much less clear this method will work against analysis that examines the totality of execute of all processes in the system. This too is an area of future research.

The goal of this paper was to describe an attack that evades anti-malware mechanisms that guard against injection of malware. It suggests that a combinatorial explosion could increase the importance of detecting the *execution*, rather than the *injection* of malware.

Acknowledgements: Many thanks to Richard Ford for his assistance in tracking down information about Dichotomy and RMNS, and for his encouragement and advice.

References

[1] Melissa macro virus. CERT Advisory CA-1999-04, CERT, Pittsburgh, PA, USA, Mar. 1999.

[2] M. Abu Rajab, F. Monrose, and A. Terzis. On the impact of dynamic addressing on malware propagation. In *Proceedings of the 4th ACM workshop on Recurring malcode*, pages 51–56, New York, NY, USA, 2006. ACM.

[3] J. Aycock. *Computer Ciruses and Malware*. Advances in Information Security. Springer Science+Business Media, LLC, 2006.

[4] D. Bilar. Noisy defenses: Subverting malware's OODA loop. In *Proceedings of the 4th annual workshop on Cyber security and information intelligence research*, number 9, New York, NY, USA, 2008. ACM.

[5] G. Cluley. Av-test.org's malware count exceeds 22 million.

[6] F. Cohen. Computer viruses: Theory and experiments. In *Proceedings of the 7th DOD/NBS Computer Security Conference*, pages 240–263, Sep. 1984.

[7] W. Cui, V. Paxson, and N. C. Weaver. GQ: Realizing a system to catch worms in a quarter million places. Technical Report TR-06-004, International Computer Science Institute, Berkeley, CA, USA, Sep. 2006.

[8] K. Daley, R. Larson, and J. Dawkins. A structural framework for modeling multi-stage network attacks. In *Proceedings of the 2002 International Conference on Parallel Processing Workshops*, pages 5–10, 2002.

[9] M. Eichin and J. Rochlis. With microscope and tweezers: An analysis of the internet virus of 1988. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 326–343, May 1989.

[10] E. Kaspersky. Dichotomy: Double trouble. *Virus Bulletin*, pages 8–9, May 1994.

[11] E. Kaspersky. RMNS—the perfect couple. *Virus Bulletin*, pages 8–9, May 1995.

[12] R. W. Lo, K. N. Levitt, and R. A. Olsson. MCF: a malicious code filter. *Computers & Security*, 14(6):541–566, Nov. 1995.

[13] S. Mathew, R. Giomundo, S. Upadhyaya, M. Sudit, and A. Stotz. Understanding multistage attacks by attack-track based visualization of heterogeneous event streams. In *Proceedings of the 3rd international workshop on Visualization for computer security*, pages 1–6, New York, NY, USA, 2006. ACM.

[14] D. Ourston, S. Matzner, W. Stump, and B. Hopkins. Applications of hidden markov models to detecting multi-stage network attacks. In *Proceedings of the 36th Hawaii International Conference on Systems Sciences*, Los Alamitos, CA, USA, 2003 2003. IEEE Comput. Soc. 36th Hawaii International Conference on Systems Sciences, 6-9 January 2003, Big Island, HI, USA.

[15] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., Jan. 1998.

[16] J. J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep. 1975.

[17] L. Sun, T. Ebringer, and S. Boztas. An automatic anti-anti-vmware technique applicable for multi-stage packed malware. In *Proceedings of the 3rd International Conference on Malicious and Unwanted Software (MALWARE 2008)*, pages 17–23, Dec. 1984.

[18] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, Feb. 2005.

[19] S. J. Templeton and K. Levitt. A requires/provides model for computer attacks. In *Proceedings of the 2000 workshop on New security paradigms*, pages 31–38, New York, NY, USA, 2000. ACM.