2012 IEEE Symposium on Security and Privacy

Abusing File Processing in Malware Detectors for Fun and Profit

Suman Jana and Vitaly Shmatikov
The University of Texas at Austin

***Abstract*—We systematically describe two classes of evasion exploits against automated malware detectors. *Chameleon attacks* confuse the detectors' file-type inference heuristics, while *werewolf attacks* exploit discrepancies in format-specific file parsing between the detectors and actual operating systems and applications. These attacks do not rely on obfuscation, metamorphism, binary packing, or any other changes to malicious code. Because they enable even the simplest, easily detectable viruses to evade detection, we argue that file processing has become the weakest link of malware defense. Using a combination of manual analysis and black-box differential fuzzing, we discovered 45 new evasion exploits and tested them against 36 popular antivirus scanners, all of which proved vulnerable to various chameleon and werewolf attacks.**

I. Introduction

Modern malware detectors employ a variety of detection techniques: scanning for instances of known viruses, binary reverse-engineering, behavioral analysis, and many others. Before any of these techniques can be applied to a suspicious file, however, the detector must (1) determine the type of the file, and (2) depending on the type, analyze the file's metadata and parse the file—for example, extract the contents of an archive, find macros embedded in a document, or construct a contiguous view of executable code.

The importance of file processing grows as automated malware defense moves away from the host, with antivirus (AV) scanners and intrusion prevention systems installed at enterprise gateways and email servers, increasing popularity of cloud-based malware detection services, etc. Network- and cloud-based deployments protect multiple hosts, provide early detection capabilities and better visibility into network-wide trends, and make maintenance easier. To be effective, however, remotely deployed detectors must be able to predict how each file will be processed at its destination by the operating system and applications.

In this paper, we argue that the "semantic gap" between how malware detectors handle files and how the same files are processed on the endhosts is the Achilles heel of malware defense. We use the term *detector* generically for signature-based scanners, behavioral analyzers, or any other tool that parses and analyzes suspicious files on its own, independently of the destination endhost. The vulnerabilities we describe are unrelated to obfuscation, mutation, or any other way of hiding malicious functionality. They enable even exact, unmodified instances of malware—primitive and (otherwise) trivially detectable, as well as arbitrarily sophisticated—to evade detection simply because the detector fails to correctly process the infected file.

We introduce two new classes of evasion exploits against malware detectors. The first is *chameleon attacks*, which target the discrepancies between the heuristics used by detectors to determine the type of the file and those used by the endhosts. Contrary to a common misconception, neither is based solely on the file extension, thus our attacks are more complex than simply renaming the extension (this trick does not work against modern detectors), nor can they be solved by forcing a particular extension onto a file.

The second class is *werewolf attacks*, which exploit the discrepancies in the parsing of executables and application-specific formats between malware detectors and actual applications and operating systems.

We evaluated 36 popular AV scanners using a combination of manual analysis and differential black-box fuzzing, and discovered 45 different exploits. *All* tested scanners proved vulnerable to both chameleon and werewolf attacks. We stress that the problem is not specific to AV scanners and does not depend on known weaknesses of signature-based scanning such as the inability to handle metamorphic mutations. The actual viruses used in our testing are extremely simple. They do not employ self-encryption, polymorphism, or obfuscation, yet chameleon and werewolf attacks enable them to pass undetected through scanners whose virus databases contain their exact code. Because file processing must take place before actual malware detection, more elaborate detectors are vulnerable, too, as long as their file-processing logic differs, however slightly, from the OS and applications on any of the protected endhosts.

The problem is deeper than the anecdotally known inability of AV software to properly process archive formats. Many modern file formats are effectively archives in disguise: for example, MS Office documents contain executable macros, Compiled HTML Help (CHM) contains HTML files, etc. We discovered chameleon and werewolf attacks against *all file formats* we tested, from relatively simple archives to executable images and complex MS Office document formats. Evasion techniques based on code obfuscation are widely known and many defenses have been proposed. In contrast, our attacks involve changes only to the *meta-data* of infected files and are thus different, significantly simpler, and complementary to obfuscation-based evasion.

While each individual vulnerability may be easy to fix, file processing in malware detectors suffers from thousands of

DOI 10.1109/SP.2012.15

semantic discrepancies. It is very difficult to "write a better parser" that precisely replicates the file-parsing semantics of actual applications and operating systems: (1) many formats are underspecified, thus different applications process the same file in different and even contradictory ways, all of which must be replicated by the detector; (2) replicating the behavior of a given parser is hard—for example, after many years of testing, there are still hundreds of file-parsing discrepancies between OpenOffice and MS Office [23, 24] and between the "functionally equivalent" implementations of Unix utilities [7]; (3) the detector must be bug-compatible with all applications; (4) because applications are designed to handle even malformed files, their parsing algorithms are much looser than the format specification, change from version to version, and have idiosyncratic, difficult-to-replicate, mutually incompatible semantics for processing non-compliant files, all of which must be replicated by the detector; (5) even seemingly "safe" discrepancies—such as attempting to analyze files with invalid checksums when scanning an archive for malware—enable evasion.

Responsible disclosure. All attacks described in this paper have been reported to the public through the Common Vulnerabilities and Exposures (CVE) database.[1] In the rest of this paper, we refer to them by their candidate CVE numbers (see Tables I and II). These numbers were current at the time of writing, but may change in the future.

II. Related Work

We introduce chameleon and werewolf attacks as a generic, pervasive problem in *all* automated malware detectors and demonstrate 45 distinct attacks on 36 different detectors, exploiting semantic gaps in their processing of many archive and non-archive formats. With a couple of exceptions (e.g., a RAR archive masquerading as a Windows executable, previously mentioned in [2]), the attacks in this paper are reported and described for the first time.

There is prior evidence of malformed archive files evading AV software [2, 3, 10, 18, 38, 39]. These anecdotes are limited to archive formats only and do not describe concrete attacks. Alvarez and Zoller briefly mention that modern AV scanners need to parse a variety of formats [2] and point out the importance of correct file parsing for preventing denial of service [1]. Concrete werewolf attacks on the detectors' parsing logic for non-archive formats such as executables and Office documents have been reported in neither research literature, nor folklore. These attacks have especially serious repercussions because they are not prevented even by host-based on-access scanning (see Section IX-A).

Buffer overflow and other attacks on AV software, unrelated to file processing, are mentioned in [36, 37].

Chameleon attacks. Chameleon attacks on file-type inference heuristics are superficially similar to attacks on content-sniffing heuristics in Web browsers [19, 25, 30]. Barth et al. proposed prefix-disjoint content signatures as a browser-based defense against content-sniffing attacks [4]. The premise of this defense is that no file that matches the first few bytes of some format should be parsed as HTML regardless of its subsequent content.

Prefix-disjoint signatures do not provide a robust defense against chameleon attacks on malware detectors. Detectors handle many more file formats than Web browsers and, crucially, these formats cannot be characterized solely by their initial bytes (e.g., valid TAR archives can have arbitrary content in their first 254 bytes, possibly including signatures for other file types). Therefore, they cannot be described completely by any set of prefix-disjoint signatures.

Other semantic-gap attacks. Chameleon and werewolf attacks are an instance of a general class of "semantic-gap" attacks which exploit different views of the same object by the security monitor and the actual system. The gap described in this paper—the monitor's (mis)understanding of the type and structure of suspicious files—received much less attention than other evasion vectors against AV scanners and intrusion detection systems [16, 27, 31] and may very well be the weakest link of malware defense.

Other, complementary evasion techniques exploit networking protocols (e.g., split malware into multiple packets), obfuscate malware using mutation or packing [20], or, in the case of malicious JavaScript, obfuscate it in HTML, Flash, and PDF content. For example, Porst showed how to obfuscate scripts so that they are not recognized by AV scanners but parsed correctly by the Adobe reader [26]. HTML parsing is notoriously tricky [28], and cross-site scripting can exploit HTML-parsing discrepancies between browsers and sanitization routines [5, 35]. In contrast, we show how the most primitive viruses, which are present in standard virus databases and do not use any obfuscation, can evade detection by exploiting discrepancies in the processing of even basic file formats such as TAR and PE.

An entirely different kind of semantic gap is exploited by "split-personality" malware, whose behavior varies between monitored and unmonitored environments [8]. Such malware contains code that tries to detect virtualization, emulation, and/or instrumentation libraries. In contrast, our attacks are completely *passive*, require no active code whatsoever, and target a different feature of malware detection systems.

Semantic-gap attacks on system-call interposition exploit the gap between the monitor's and the OS's views of system-call arguments [12, 34]. These attacks typically involve concurrency and are fundamentally different from the attacks described in this paper.

Program differencing. Brumley et al. proposed to automatically detect discrepancies between different implementations of the same protocol specification by converting execution traces into symbolic formulas and comparing

[1] http://cve.mitre.org/

Table I
TESTED AV SCANNERS.

AV number	Name	AV number	Name	AV number	Name
1	ClamAV 0.96.4	2	Rising 22.83.00.03	3	CAT-QuickHeal 11.00
4	GData 21	5	Symantec 20101.3.0.103	6	Command 5.2.11.5
7	Ikarus T3.1.1.97.0	8	Emsisoft 5.1.0.1	9	PCTools 7.0.3.5
10	F-Prot 4.6.2.117	11	VirusBuster 13.6.151.0	12	Fortinent 4.2.254.0
13	Antiy-AVL 2.0.3.7	14	K7AntiVirus 9.77.3565	15	TrendMicro-HouseCall 9.120.0.1004
16	Kaspersky 7.0.0.125	17	Jiangmin 13.0.900	18	Microsoft 1.6402
19	Sophos 4.61.0	20	NOD32 5795	21	AntiVir 7.11.1.163
22	Norman 6.06.12	23	McAfee 5.400.0.1158	24	Panda 10.0.2.7
25	McAfee-GW-Edition 2010.1C	26	TrendMicro 9.120.0.1004	27	Comodo 7424
28	BitDefender 7.2	29	eSafe 7.0.17.0	30	F-Secure 9.0.16160.0
31	nProtect 2011-01-17.01	32	AhnLab-V3 2011.01.18.00	33	AVG 10.0.0.1190
34	Avast 4.8.1351.0	35	Avast5 5.0.677.0	36	VBA32 3.12.14.2

Table II
AFFECTED AV SCANNERS FOR EACH REPORTED ATTACK.

CVE	Vulnerable scanners	CVE	Vulnerable scanners	CVE	Vulnerable scanners
2012-1419	1, 3	2012-1420	2, 3, 6, 10, 12, 14, 16, 18, 20, 22, 24	2012-1421	2, 3, 5, 22
2012-1422	2, 3, 20, 22	2012-1423	2, 6, 7, 8, 9, 10, 11, 12, 14, 20, 22	2012-1424	3, 9, 13, 17, 19, 22
2012-1425	3, 5, 7, 8, 9, 13, 15, 16, 17, 20, 21, 22, 23, 25, 26	2012-1426	2, 3, 6, 10, 14, 22	2012-1427	3, 19, 22
2012-1428	3, 19, 22	2012-1429	7, 8, 23, 25, 27, 28, 29, 30, 31	2012-1430	2, 19, 23, 25, 27, 28, 29, 30, 31
2012-1431	2, 6, 10, 19, 25, 27, 28, 29, 30, 31	2012-1432	7, 8, 24, 29	2012-1433	7, 8, 24, 29, 32
2012-1434	7, 8, 24, 32	2012-1435	7, 8, 24, 29, 32	2012-1436	7, 8, 24, 29, 32
2012-1437	27	2012-1438	19, 27	2012-1439	2, 24, 29
2012-1440	22, 24, 29	2012-1441	29	2012-1442	2, 3, 13, 16, 19, 23, 24, 25, 29, 30
2012-1443	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 32, 33, 34, 35, 36	2012-1444	24, 29	2012-1445	2, 24, 29
2012-1446	2, 3, 5, 9, 13, 16, 19, 22, 23, 24, 25, 29	2012-1447	24, 29	2012-1448	3, 7, 8, 26
2012-1449	2, 20	2012-1450	7, 8, 19	2012-1451	7, 8
2012-1452	3, 7, 8	2012-1453	2, 7, 8, 13, 15, 16, 18, 19, 23, 24, 25, 26	2012-1454	2, 23, 24, 25, 29
2012-1455	2, 20	2012-1456	2, 3, 5, 7, 8, 10, 12, 15, 16, 17, 19, 20, 22, 23, 24, 25, 26, 27, 29, 33	2012-1457	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 20, 21, 22, 23, 25, 26, 28, 29, 33, 34, 35, 36
2012-1458	1, 19	2012-1459	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 31, 32, 33, 34, 35, 36	2012-1460	3, 6, 10, 13, 14, 17, 29, 36
2012-1461	2, 5, 6, 7, 8, 12, 14, 15, 16, 17, 19, 20, 22, 23, 25, 26, 28, 30, 33, 36	2012-1462	3, 5, 7, 8, 12, 16, 17, 19, 22, 29, 32, 33	2012-1463	3, 6, 10, 22, 23, 24, 27, 28, 29, 30, 31, 32

them using SMT solvers [6]. Unfortunately, this approach does not appear to be feasible for automatically discovering chameleon and werewolf attacks. The programs tested by Brumley et al. implement simple protocols like HTTP and their parsing code is very shallow, i.e., it lies close to the program's entry point. By contrast, malware scanners and applications do much more than parsing: scanners load virus signatures, match them against the file, etc., while applications perform a wide variety of operations before and after parsing. Binary differencing must be applied to the parsing code *only*, because the non-parsing functionalities of malware detectors and applications are completely different. This requires extracting the parsing code from the closed-source binaries of both detectors and applications, which is extremely difficult. Furthermore, both parsers must have the same interface, otherwise their final output states cannot be easily compared. Brumley et al. provide no method for automatically recognizing, extracting, normalizing, and comparing individual pieces of functionality hidden deep inside the binary.

Furthermore, this technique generates formulas from one execution path at a time and is less likely find bugs in rare paths. By contrast, most of the attacks reported in this paper—for example, the attack which concatenates two separate streams of gzipped data to create a single file—exploit bugs in unusual paths through the parsing code.

BEK is a new language and system for writing and analyzing string-manipulating sanitizers for Web applica-

tions [15]. BEK can compare two sanitizers for equivalence and produce a counter-example on which their outputs differ. The BEK language is specifically tailored for expressing string manipulation operations and is closely related to regular expressions. It is ill-suited for expressing file-parsing logic. For example, it cannot validate data-length fields in file headers and similar content-dependent format fields.

Development of program analysis techniques for automated discovery of chameleon and werewolf attacks is an interesting topic for future research.

III. Attacking File Processing

Figure 1 shows the main components of the file-processing logic of antivirus scanners. The basic pattern applies to other automated[2] malware detectors, both behavioral and signature-based, as long as they process files independently of the endhost's OS and applications.

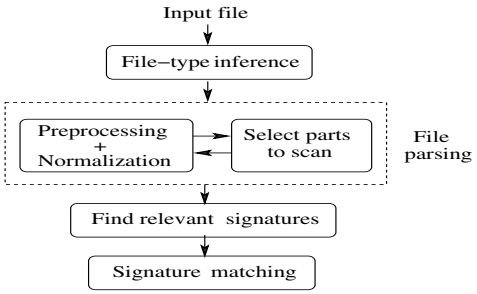

Figure 1. File processing in antivirus scanners.

The first step is **file-type inference**. The scanner must infer the file type in order to (1) parse the file correctly and (2) scan it for the correct subset of virus signatures.

The second step is **file parsing**. Files in some formats must be preprocessed before scanning (for example, the contents of an archive must be extracted). Documents in formats like HTML contain many irrelevant characters (for example, whitespace) and must be normalized. In most file formats, whether executable or application-specific, blocks of data are interspersed with meta-data. For higher performance, malware scanners parse the meta-data in order to identify and scan only the potentially "interesting" parts of the file. For example, a scanner may parse an MS Word document to find embedded macros and other executable objects and scan them for macro viruses. To detect viruses in Linux ELF executables, which can contain multiple sections, the scanner must construct a contiguous view of the executable code. This requires parsing the meta-data (ELF header) to find the offsets and sizes of code sections.

Chameleon and werewolf attacks. We will refer to attacks that exploit discrepancies in file-type inference as **chameleon attacks** because attack files appear as one type to the detector and as a different type to the actual OS or application. We will refer to attacks that exploit discrepancies in parsing as **werewolf attacks** because attack files appear to have different structure depending on whether they are parsed by the detector or the application.

Chameleon and werewolf attacks only change the *meta-data* of the file; the contents, including the malicious payload, are not modified (in contrast to code obfuscation and polymorphism). These attacks (1) start with a file that is recognized as malicious by the detector, (2) turn it into a file that is *not* recognized as malicious, yet (3) the modified file is correctly processed by the destination application or, in the case of executables, loaded by the OS. If the same file can be processed by multiple applications or versions of the same application, we consider an attack successful if at least one of them processes the modified file correctly.

Fingerprinting malware detectors and learning their logic. Because file-type inference heuristics and file-parsing logic vary from detector to detector, attacks are detector-specific and it helps to know which detector is protecting the target. This knowledge is often public—for example, Yahoo Mail scans all messages with Symantec's Norton antivirus—but even in blind testing against Gmail's unknown scanner, two chameleon and one werewolf attacks (CVE-2012-1438, 2012-1443, and 2012-1457) evaded detection.

Unknown detectors can be identified by tell-tale signs in bounced messages [22], or by using chameleon and werewolf attacks themselves. As Table II shows, different attacks work against different detectors. By trying several attacks and seeing which of them evade detection, the attacker can infer the make and model of the detector.

The logic of open-source detectors like ClamAV can be learned by analyzing their code, but the vast majority of detectors are closed-source and their logic must be learned by fuzzing and/or binary analysis. Secrecy of the file-processing logic is a weak defense, however: we report dozens of vulnerabilities in commercial scanners for which we do not have the source code, many of them discovered automatically by our black-box differential fuzzer.

IV. Generating Attacks

To test our attacks, we used VirusTotal [32], a free Web service that checks any file against multiple antivirus scanners (43 at the time of our testing). Several scanners were not available at various times due to crashes, thus for consistency we present the results for the 36 scanners that were continuously available.

VirusTotal executes the command-line versions of all AV scanners with maximum protection and all detection methods enabled. We argue that this faithfully models the level of defense provided by network-based detectors. By design, they do not observe the actual processing of files on the host and thus advanced detection techniques—for

[2]Human operators may be able to manually prevent incorrect parsing and file-type inference, but widespread deployment of human-assisted detectors is not feasible for obvious scalability reasons.

Table III
TESTED APPLICATIONS.

File type	Target application(s)
CAB	Cabextract 1.2
CHM	Microsoft HTML Help 1.x
ELF	Linux kernel (2.6.32) ELF loader
GZIP	Gzip 1.3.12, File Roller 2.30.1.1
DOC	MS Office 2007, OpenOffice 3.2
PE	Windows Vista SP2 PE loader, Wine 1.2.2 PE loader
RAR	RAR 3.90 beta 2
TAR	GNU tar 1.22, File Roller 2.30.1.1
7Z	7-Zip 9.04 beta

example, monitoring the program's execution for signs of malicious behavior—require the detector to accurately recognize the file type, parse the file, and replicate the host's execution environment. In Section IX, we explain why this is challenging to do correctly.

Attacks were also confirmed by testing against the host-based versions of AV software, where available.

We used five toy viruses from VX Heavens [33] in our tests: EICAR, Linux Bliss and Cassini, Windows Cecile, and MS Word ABC. If an exact, unobfuscated instance of such a basic virus evades detection, more sophisticated malware won't be detected, either. We count an attack as successful if the detector (1) recognizes the infection in the original file, but (2) no longer recognizes it in the modified file.

Target applications used in our testing are summarized in Table III. They were executed on laptops running Linux Ubuntu 10.04 and Windows Vista SP 2.

Black-box differential fuzzing. To find werewolf attacks automatically, we built a differential fuzzing framework that finds discrepancies between the parsing logic of applications and malware detectors. Because the source code of detectors is rarely available, our framework is *black-box*. It is implemented in Python and runs on both Linux and Windows.

The basic framework is format-independent, but format-specific components are added as plugins. Each plugin provides a parser, an output validator, and a fuzzer. The parser breaks up the format-specific header into an array of (name, offset, length) tuples, where *name* is the unique name of a header field, *offset* is its location in the file, and *length* is its size. The fuzzer modifies the content of the fields using a format-specific algorithm. The validator checks if the application still processes the modified file correctly.

Our framework takes as input two seed files in the same format. One file is parsed correctly by the destination application, the other is an infected file recognized by the detector. The framework uses the format-specific fuzzer to automatically generate modifications to the first file and the output validator to check if the application still accepts the file. If a modification is validated, the framework applies it to the second, infected file and tests whether the detector still recognizes the infection. This approach is better than directly modifying the infected file and accessing it on an endhost because (1) the host must be isolated (e.g., virtualized) in each test to prevent an actual infection, imposing a significant performance overhead on the testing framework, and (2) determining if the modified infected file is accepted by the destination application is difficult because applications are opaque and have complex side effects.

A modification is thus applied to the infected file only if the application's parser tolerates it. If the file is no longer recognized as malicious, a discrepancy between the application's and the detector's parsers has been found and an actual attack can be generated and verified by accessing the modified infected file on a secure, isolated host. We consider an infection verified if the intact malware code is extracted from the archive and/or loaded as an executable.

As a proof of concept, we implemented sample plugins for MS Cabinet (CAB), Windows executable (PE), and Linux executable (ELF) files. The fuzzer in these plugins tries a simple modification to the file's header, one field at a time: it increments the content of each field (or the first byte if the field spans multiple bytes) by 1; if this results in an overflow, it decrements the content by 1. Output validators execute destination applications on modified seed files and check the return codes and the application's output for correctness.

Once integrated with VirusTotal, our fuzzing framework found dozens of parsing bugs in 21 different detectors (Table XII). All result in actual werewolf attacks.

Of course, our simple framework cannot find all parsing discrepancies. Some parsing bugs are hidden in rarely executed paths which can only be reached through specially crafted inputs, requiring manual guidance to the fuzzer. For example, attacks involving a concatenation of two gzipped streams or a header with an incorrect checksum whose length is modified to point into the following header (see Section VI) are difficult to discover by automated fuzzing.

Another limitation is that our fuzzer does not fully "understand" the dependencies between different fields of format-specific headers and cannot automatically generate valid files if several fields must be changed consistently. For example, if file length is included in a header field, the file must be truncated or augmented whenever this field is modified.

V. CHAMELEON ATTACKS

Chameleon attacks involve specially crafted files that appear as one type to the file-type inference heuristics used by the malware detector but as a different type to the OS or application on the endhost.

The simplest chameleon attack is to hide the infected file in an archive of a type not recognized by the detector, causing it to apply generic malware signatures without extracting the contents. Even this primitive attack is surprisingly effective, as shown by Table IV.

In the rest of this section, we focus on more interesting chameleon attacks that involve a file of one type *masquerading* as a file of a different type. Masquerade attacks cause

Table IV
SUPPORT FOR 11 ARCHIVE FORMATS: 7ZIP, 7ZIP-SFX, PACK, ISO, RAR, RAR(SFX), TAR.LZOP, TAR.LZMA, TAR.RZ, TAR.XZ, AR

Scanner	Unsupported formats	Scanner	Unsupported formats	Scanner	Unsupported formats
ClamAV 0.96.4	8	Rising 22.83.00.03	9	CAT-QuickHeal 11.00	9
GData 21	7	Symantec 20101.3.0.103	10	Command 5.2.11.5	8
Ikarus T3.1.1.97.0	9	Emsisoft 5.1.0.1	8	PCTools 7.0.3.5	10
F-Prot 4.6.2.117	9	VirusBuster 13.6.151.0	10	Fortinet 4.2.254.0	9
Antiy-AVL 2.0.3.7	8	K7AntiVirus 9.77.3565	9	TrendMicro-HouseCall 9.120.0.1004	10
Kaspersky 7.0.0.125	5	Jiangmin 13.0.900	9	Microsoft 1.6402	6
Sophos 4.61.0	8	NOD32 5795	7	AntiVir 7.11.1.163	7
Norman 6.06.12	9	McAfee 5.400.0.1158	10	Panda 10.0.2.7	8
McAfee-GW-Edition 2010.1C	10	TrendMicro 9.120.0.1004	10	Comodo 7424	11
BitDefender 7.2	9	eSafe 7.0.17.0	8	F-Secure 9.0.16160.0	8
nProtect 2011-01-17.01	10	AhnLab-V3 2011.01.18.00	10	AVG 10.0.0.1190	9
Avast 4.8.1351.0	7	Avast5 5.0.677.0	7	VBA32 3.12.14.2	9

harm in several ways. First, for efficiency, detectors usually apply only a subset of analysis techniques and/or malware signatures to any given file type. If the detector is confused about the type, it may apply a wrong analysis. Second, many file types require preprocessing (e.g., unpacking) before they can be analyzed. A confused detector may apply a wrong preprocessing or no preprocessing at all.

File-type inference heuristics. File-type inference in malware scanners is not based on the file extension. Even if the endhost runs Windows, which by default relies on the extension to determine the file's type, users may override the defaults and use any program to open any file. Therefore, all tested scanners ignore the extension and attempt to determine the actual type of the file. The simple attack of renaming the extension thus does not work, but neither does the simple defense of having the scanner rewrite the extension to match the file's actual type (see Section VII).

To illustrate file-type inference heuristics, we use ClamAV v0.95.2, a popular open-source scanner [9]. The basic principles apply to other detectors, too, as evidenced by the success of chameleon attacks against all of them. For most file types, ClamAV uses fixed-offset byte-string signatures, but for certain types such as HTML or self-extracting ZIP archives, ClamAV also uses regular-expression signatures, described later in this section. Fixed-offset signatures are tuples of the form *(offset, magic-content, length)* where *offset* denotes the offset from the beginning of the file which is to be checked for this particular file type, *magic-content* is the sequence of bytes starting from *offset* that any file of this type should have, and *length* is the length (in bytes) of that sequence. Some of ClamAV's file-type signatures are shown in Table XIII in the appendix. For example, ClamAV's signature for ELF executables is *(0, 7f454c46, 4)*, thus any file which has *7f454c46* as its first four bytes will be considered as an ELF file by ClamAV.

Algorithm 1 shows a simplified version of ClamAV's algorithm for inferring the file type. The order of signatures in the list matters: once ClamAV finds a match, it does not check the list any further. In particular, if a fixed-offset signature is matched, all regex signatures are ignored. This is exploited by one of our attacks.

Algorithm 1 Simplified pseudocode of ClamAV's file-type inference algorithm.

```
Read first 1024 bytes of input file into buf
for each fixed-offset file-type signature s in the specified order do
  if !memcmp(buf+s.offset, s.magic-content, s.length) then
    if s is a file type to ignore then
      return ignore
    else
      return s.filetype
    end if
  end if
end for
Check buf for regex file-type signatures using Aho-Corasick algorithm
if buf matches a regex signature r then
  return r.filetype
else
  return unknown file type
end if
```

From version 0.96 onward, ClamAV also supports LLVM bytecode signatures, typically used to detect polymorphic malware in a file-format-aware manner. These signatures are only checked for specific file types, e.g., a signature registering $PDF\_HOOK\_DECLARE$ will only get checked if the inferred file type is PDF. Therefore, these signatures are extremely susceptible to chameleon attacks.

Requirements for file-type masquerade. Let A be the file's actual type and let B be the fake type that the attacker wants the detector to infer. For the masquerade to be successful, three conditions must hold for the file-type signatures S_A (for type A) and S_B (for type B):

1) S_A and S_B do not conflict, i.e., there are no i, j such that $0 \leq i < S_A.\text{length}$, $0 \leq j < S_B.\text{length}$, $S_A.\text{offset} + i = S_B.\text{offset} + j$, and

Table V
VULNERABLE FILE-TYPE PAIRS IN CLAMAV

Real type	Fake type	Real type	Fake type
POSIX TAR	mirc.ini	ELF	POSIX TAR
PNG	POSIX TAR	ELF	JPEG
GIF	JPEG	ELF	SIS
BMP	JPEG	MPEG	POSIX TAR
MP3	POSIX TAR	JPEG	POSIX TAR
PNG	JPEG	BMP	JPEG

S_A.magic-content$[i] \neq S_B$.magic-content$[j]$.

2) The detector matches S_B before S_A.
3) Destination OS or application correctly processes files of type A containing both S_A and S_B.

The first condition ensures that the same file may contain both S_A and S_B, the second that the detector infers type B for the file before it has a chance to infer type A. In our testing, we discovered 12 file-type pairs that satisfy all three conditions for ClamAV (Table V).

Masquerade alone is not enough. Even if the detector infers the wrong file type, it may still detect the infection by scanning the file as a "blob" or if the signatures associated with the inferred type contain the virus. That said, masquerade is a good start for exploring chameleon attacks.

Table VI
CHAMELEON ATTACKS WITH EICAR-INFECTED TAR FILES.

Actual file type	Fake file type	No. of vulnerable AVs	CVE
POSIX TAR	mirc.ini	2	2012-1419
POSIX TAR	ELF	11	2012-1420
POSIX TAR	CAB	4	2012-1421
POSIX TAR	CHM	4	2012-1422
POSIX TAR	PE	11	2012-1423
POSIX TAR	SIS	6	2012-1424
POSIX TAR	PKZIP	16	2012-1425
POSIX TAR	BZip2	6	2012-1426
POSIX TAR	WinZip	3	2012-1427
POSIX TAR	JPEG	3	2012-1428

Table VII
CHAMELEON ATTACKS WITH BLISS-INFECTED ELF FILES.

Actual file type	Fake file type	No. of vulnerable AVs	CVE
ELF	POSIX TAR	9	2012-1429
ELF	SIS	9	2012-1430
ELF	JPEG	10	2012-1431

Table VIII
CHAMELEON ATTACKS WITH CECILE-INFECTED PE FILES.

Actual file type	Fake file type	No. of vulnerable AVs	CVE
PE	Winzip	4	2012-1432
PE	JPEG	5	2012-1433
PE	SIS	4	2012-1434
PE	PKLITE	5	2012-1435
PE	LZH	5	2012-1436

Results for chameleon attacks. Our description of file-type inference logic focuses on ClamAV because its open-source code makes it easy to explain the heuristics. File-type inference based on magic strings is by no means unique to ClamAV, however. All tested scanners proved vulnerable to masquerade-based chameleon attacks. The results are summarized in Table X; the masquerade pairs for each attack and scanner are shown in Tables VI, VII, VIII, and IX. In all attacks, to masquerade as a particular type we used ClamAV's magic string if supported by ClamAV, otherwise a string from Table XIV in the appendix.

Table IX
CHAMELEON ATTACKS FOR ABC-INFECTED MS OFFICE 97 DOC FILES.

Actual file type	Fake file type	No. of vulnerable AVs	CVE
MS Office	PKSFX	1	2012-1437
MS Office	POSIX TAR	2	2012-1438

Sample attack: making a TAR archive look like mirc.ini We describe a sample exploit against ClamAV in which a POSIX TAR file masquerades as a 'mirc.ini' file. Their file-type signatures are disjoint and the signature of 'mirc.ini' is matched before the signature of TAR. It remains to ensure that adding the 'mirc.ini' signature to a TAR file does not affect the processing of the archive by the `tar` program.

Table XIII says that the signature of 'mirc.ini' begins at offset 0 and ends at offset 9. Thus the $0 - 9$ bytes of the input TAR file must be changed to '5b616c69617365735d'. Because the initial 100 bytes contain the name of the first file, the first 9 bytes of this name will change as a side effect. This does not affect the archive's contents and any virus infecting any file in the archive will be free to spread.

We converted this masquerade exploit into a working chameleon attack using the test EICAR virus [11], which is detected by all antivirus scanners, including ClamAV. If the first 9 bytes of a TAR archive containing 'eicar.com' are directly changed to '5b616c69617365735d' ('[aliases]' in ASCII), the `tar` application considers the archive corrupted because the names of all member files are part of a checksum-protected header block. To avoid this issue, it is sufficient to rename 'eicar.com' to '[aliases].com' and put it inside the TAR archive. ClamAV does not recognize the file as an archive and scans it as a "blob," looking for the EICAR signature only at offset 0 and failing to detect it in the middle of the file. Another approach is to use a TAR manipulation library to change the checksum, but this is not necessary here because the fake file-type signature '[aliases]' only contains ASCII characters.

Sample attack: user-repaired archive. The application on the endhost is often capable of repairing the modified file (this is common in archiving programs). In some cases, it may prompt the user whether she wants to repair the file. Most users answer 'Yes' to these questions.

Given a RAR archive with an EICAR-infected file, we changed the first two bytes to "MZ," which is the magic identifier for Windows executables. *None* of the tested scanners detected the infection in the modified archive, yet

Table X
SUCCESSFUL CHAMELEON ATTACKS.

Format type	File format	No. of attacks	CVE
non-archive	ELF	3	2012-1429, 2012-1430, 2012-1431
	PE	5	2012-1432, 2012-1433, 2012-1434, 2012-1435, 2012-1436
	MS Office 97	2	2012-1437, 2012-1438
archive	TAR	10	2012-1419, 2012-1420, 2012-1421, 2012-1422, 2012-1423, 2012-1424, 2012-1425, 2012-1426, 2012-1427, 2012-1428
	RAR	1	2012-1443

the RAR program on the endhost repaired it and correctly extracted the infected file. This is especially surprising because this particular attack is anecdotally known [2].

Sample attack: exploiting regex-based file-type inference. To recognize certain formats, ClamAV uses regular expressions to match magic strings that can occur anywhere in a file, in addition to looking for magic strings at fixed offsets. We describe two sample attacks on this combination of fixed-offset and regex-based inference (tested against ClamAV only and thus not counted in Table X).

ZIP archives may start with arbitrary bytes. To recognize ZIP files, ClamAV uses both a fixed-offset signature '504b0304' at offset 0 and a regex signature '504b0304' which can appear at any offset within the input file. Once a format has been recognized according to the fixed-offset signature, ClamAV does not do any further inference—even if there are regex matches inside the file. To exploit this, we created a ZIP file containing a Cassini-infected executable and prepended the string '504b0304' to it. ClamAV matched the fixed-offset signature at offset 0 but failed to notice the regex signature at offset 4, was thus unable to extract the contents correctly, and declared the archive clean. The destination application (`unzip`) ignored the initial bytes and extracted the infected file.

The second attack exploits the fact that ClamAV ignores files of certain types (e.g., MPEG video and Ogg Vorbis audio) because they are not affected by any major viruses. We created a CAB archive containing a Cassini-infected file and prepended the string '000001b3' to it, which is the fixed-offset MPEG signature. ClamAV's file-type database contains a regex signature for CAB format—'4d534346' anywhere in the file, which matches CAB files even with garbage prepended—but ClamAV does not apply regex signatures once the fixed-offset signature has been matched. Therefore, ClamAV infers MPEG type for the file and does not scan it, while the destination application (`cabextract`) correctly extracts the infected file.

VI. WEREWOLF ATTACKS

Werewolf attacks tamper with the file's meta-data, causing the detector to parse it incorrectly and/or incompletely. In contrast, the OS or application on the endhost usually "understands" the format much deeper (see Section VIII-A) and processes the file correctly.

Table XI shows that every scanner we tested is vulnerable to multiple file-parsing attacks. Table XII summarizes the header-parsing discrepancies found by our differential fuzzing framework. All of them result in successful werewolf attacks; the rest were found by manual analysis.

Below, we explain some of the attacks, using ClamAV as an example of an open-source detector and McAfee as an example of a closed-source detector.

Table XI
SUCCESSFUL WEREWOLF ATTACKS.

Format type	File format	No. of vulnerable AVs	CVE
non-archive	ELF	12	2012-1463
	CHM	2	2012-1458
archive	ZIP	20	2012-1456
	TAR	29	2012-1457
	TAR	35	2012-1459
	TGZ	8	2012-1460
	TGZ	20	2012-1461
	ZIP	12	2012-1462

Table XII
HEADER-PARSING ERRORS (ALL RESULT IN WEREWOLF ATTACKS).

Format type	Format	Header fields	No. of vuln. AVs	CVE
non-archive	ELF	padding	4	2012-1439
		identsize	5	2012-1440
		class	11	2012-1442
		abiversion	4	2012-1444
		abi	4	2012-1445
		encoding	14	2012-1446
		e_version	4	2012-1447
		ei_version	6	2012-1454
	PE	e_minalloc + 13 others	2	2012-1441
		e_ip and e_res	1	2012-1441
archive	CAB	cbCabinet	5	2012-1448
		vMajor	2	2012-1449
		reserved3	3	2012-1450
		reserved2	2	2012-1451
		reserved1	3	2012-1452
		coffFiles	14	2012-1453
		vMinor	2	2012-1455

A. Sample werewolf attacks on archive files

Wrong checksum. In a POSIX TAR archive, each member file has a 512-byte header protected by a simple checksum. All headers also contain a file length field, which is used by the extractor to locate the next header in the archive. Most scanners do not use the checksum field when parsing an archive. This is reasonable because a virus may lurk even in an archive whose checksum is wrong, but in this case the scanners are too smart for their own good.

Our sample attack uses a TAR archive with two files: the first one is clean, while the second is infected with the test EICAR virus. The length field in the header of the first, clean file has been modified to point into the middle of the header of the second, infected file (see Figure 2). Scanners that do not verify the checksum field are unable to find the beginning of the second header. 35 of the 36 tested scanners fail to detect the infection in the modified archive (the only exception is eSafe 7.0.17.0).

In contrast, `tar` on Linux discovers that the checksum is invalid, prints out a warning, skips the first header, finds the second, infected file by searching for the magic string "ustar," and proceeds to extract it correctly.

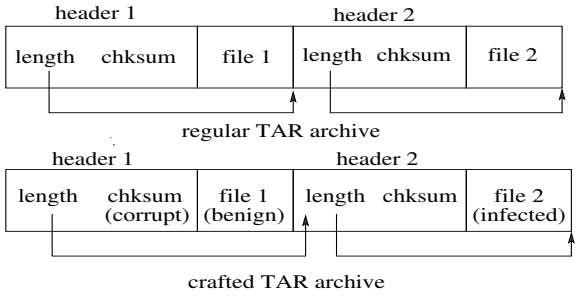

Figure 2. A crafted TAR archive with the modified length field in the first header.

Misleading length. If the length field in the header of a file included into a TAR archive is greater than the archive's total length ($1,000,000$+original length in our experiments), 29 out of 36 scanners fail to detect the infection.

One of the vulnerable scanners is McAfee, which has the default upper limit of 1 MB on memory for loading a file. Since the size specified in the header is greater than 1 MB, McAfee declares the file clean. GNU `tar` prints a warning but extracts the infected contents correctly.

Multiple streams. GZIP files can contain multiple compressed streams, which are assembled when the contents are extracted. This feature can be used to craft a `.tar.gz` file with the EICAR test virus broken into two parts. 20 out of 36 scanners fail to detect the infection. When the contents are extracted, the infected file is correctly reassembled.

For example, McAfee simply ignores all bytes after the first stream of compressed data. Even if another infected file is appended to a GZIP file containing multiple compressed streams, McAfee fails to detect the infection.

Random garbage. If a ZIP archive has garbage bytes in the beginning, the `unzip` program skips these bytes and still extracts the contents correctly (we used Zip 3.0 and UnZip 6.00 in Ubuntu 10.04 for this test). 12 out of 36 scanners fail to detect the infection in a file consisting of 1024 bytes of random garbage followed by an EICAR-infected ZIP file. Note that the file still has the proper `.zip` extension.

Random garbage at the end of a valid GZIP archive does not affect the `gzip` program, which simply ignores it when extracting the contents. Yet, given an EICAR-infected `.tar.gz` file with 6 random bytes appended, 8 out of 36 scanners fail to detect the infection.

Ambiguous files conforming to multiple formats. Flexibility of many file formats enables an attacker to create werewolf files that can be correctly parsed according to more than one format and produce different results. Given that `zip` correctly parses ZIP archives with garbage prepended, while `tar` correctly parses TAR archives with garbage appended, we created a werewolf file consisting of a TAR archive followed by a virus-infected ZIP archive. This file can be processed either by `tar`, or by `zip` and different contents will be extracted depending on which program is used. 20 out of 36 scanners fail to detect the infection.

Other werewolf files that can be parsed according to multiple formats are CAB-TAR, ELF-ZIP, and PE-CAB. Some of these pairs include non-archive formats!

B. Sample werewolf attacks on non-archive files

Werewolf attacks are also effective against executables, Office documents, and CHM files. Many modern file formats are similar to archives because they can can contain embedded objects of different types. This makes parsing much harder and opens the door to werewolf attacks.

Fake endianness. In most ELF files, the 5th byte of the header indicates endianness: 01 for little-endian, 02 for big-endian. Linux kernel, however, does not check this field before loading an ELF file. If the 5th byte of a Bliss-infected, little-endian ELF file is changed to 02, 12 out of 36 scanners fail to detect the infection.

Empty VBA project names. MS Word files may contain embedded objects such as executable macros, images, etc. Because viruses can exploit the auto-execution feature, detectors try to recognize and scan macros in the document. In this example, we focus on how ClamAV does this.

In MS documents, macros are generally stored inside VBA (Visual Basic for Application) projects. A group of VBA projects is identified by a two-byte signature, "cc61"; each project in a group has an unique unicode name. ClamAV first iterates through the VBA project names treating the data as little-endian and checks if the resulting names are valid (have positive length and begin with "*\g", "*\h", "*\", or "*\d"). If an invalid name is found, ClamAV stops. ClamAV stores the number of valid project names it found in the first pass and repeats the same process, but now assuming that the data are big-endian. Finally, ClamAV compares the number of strings found during the two passes. If the first number is greater than the second, ClamAV treats the file as little-endian, otherwise, as big-endian.

We created an ABC-infected Word file in which the first VBA project name is empty but the other names are intact.

When parsing project names, ClamAV calculated the valid name count to be 0 in both little-endian and big-endian cases and failed to detect the infection. On the other hand, destination applications (MS Office 2007 and OpenOffice) open the document correctly and execute the infected macros even though the first project name is empty.

Incorrect compression reset interval. A Windows Compiled HTML Help (CHM) file is a set of HTML files, scripts, and images compressed using the LZX algorithm. For faster random accesses, the algorithm is reset at intervals instead of compressing the entire file as a single stream. The length of each interval is specified in the LZXC header.

If the header is modified so that the reset interval is lower than in the original file, the target application (in this case, Windows CHM viewer `hh.exe`) correctly decompresses the content located before the tampered header. On the other hand, several detectors (including ClamAV) attempt to decompress the entire CHM file before scanning it for malware. When they fail to decompress the contents located after the tampered header, they declare the file to be clean.

Bypassing section-specific signatures. ClamAV uses section-specific hash signatures when scanning Windows executables. They contain the offset and the length of a section of the file and the value of its MD5 hash. 85% of signatures in ClamAV's current database are of this type. If ClamAV's parser mistakenly believes that the executable is corrupt or contains some unsupported features, ClamAV skips the section-specific hash signatures, enabling evasion.

VII. Defenses Against Chameleon Attacks

Simplistic solutions such as changing the order in which magic strings are matched may address the specific vulnerabilities we found but will undoubtedly introduce new ones. One generic defense against all chameleon attacks is to recognize files that match multiple types and process them for all matching types. This may open the door to denial of service if the attacker floods the detector with files containing a large number of magic strings. To prevent this, the detector should reject files with an abnormally high number of possible types, but this only works for fixed-offset magic strings. Checking whether the file matches more than a certain number of regular expressions can still impose an unacceptable overhead on the detector.

Another approach is *normalization*: the detector can modify the file to ensure that the endhost's interpretation of its type matches the detector's. In Windows, file extension determines by default which program is used to open it. In Linux, desktop managers such as KDE and GNOME use extensions as the first heuristic and fall back on magic strings if the file has an unknown extension or no extension at all.

Unfortunately, rewriting the extension is not a failproof defense against chameleon attacks because it does not guarantee that the endhost's behavior matches the detector's expectations. First, users may override the default settings in both Windows and Linux and choose any program to open any file. Second, for endhosts running Linux, the detector must be aware of the list of known extensions: if the normalized extension is not on the list, chameleon attacks may still succeed even with the default settings.

VIII. Network-Based Defenses Against Werewolf Attacks

No matter what technique a network-based detector is using—scanning for virus signatures, emulated execution, behavioral analysis, etc.—it must first recognize the type of the file and parse it correctly. Even behavioral detection does not help if the detector is unable to find executable code in a maliciously crafted file and thus cannot execute it. Because network-based detectors do not observe the actual processing and/or execution of the file on the endhost, they must guess how the endhost may process the file. If a network-based detector is protecting multiple endhosts, it must guess correctly for all of them. In the rest of this section, we argue that this is extremely error-prone.

A. *"Write a better parser"*

The obvious defense against werewolf attacks is to ensure that the malware detector parses each file exactly the same way as the file's destination application or OS, thus eliminating the "semantic gap." Note, however, that detectors deployed on mail servers, network gateways, as a cloud-based service, etc. aim to benefit from economies of scale and typically protect many hosts with a diverse set of applications installed on them.

To prevent werewolf attacks, the malware detector must parse each file in multiple ways, one for every possible destination application and OS. The detector must (1) know all applications that may possibly be used on any of the endhosts to access the file, (2) know every application's parsing logic, (3) precisely replicate this logic within the detector for all possible inputs, including damaged and non-compliant files, (4) replicate every known and unknown bug in every application's parsing algorithm, and (5) be promptly updated with a new algorithm whenever an application is installed or updated on any endhost.

Format-compliant parsing is not enough. Format specifications prescribe how to parse correctly formatted files. In practice, however, many files do not fully conform to the specification, thus (1) applications are designed to handle even non-compliant files, and (2) blocking "malformed" files is likely to render the detector unusable because of false positives. Many formats do not define what it means for a file to be "well-formed," causing files created by legitimate applications to appear malformed. For example, up to 68% of PE executable images in the wild have structural anomalies and do not conform to the PE format specification [29]. Formats like PDF have no universally recognized notion of validity

and even conformance benchmarks accept malformed and corrupt files [14]. Furthermore, every application parses non-compliant files in its own idiosyncratic way, resulting in different outputs for the same input (see Fig. 3).

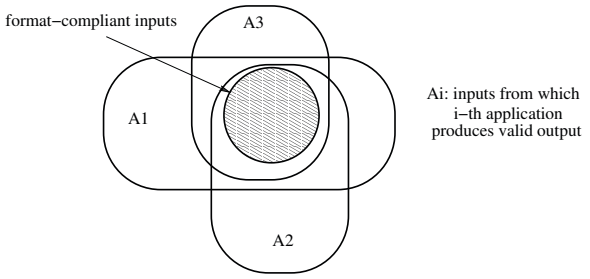

Figure 3. Parsing discrepancies.

Let I be the set of all possible file inputs, O the set of possible outputs, and $S_f : I_S \rightarrow O_S$ the specification for format f, where $I_S \subset I$ and $O_S \subset O$. An ideal program P_{ideal} would only produce an output for compliant inputs:

$$P_{ideal}(x) = \begin{cases} S_f(x) & \text{if } x \in I_S \\ Error & \text{if } x \in I - I_S \end{cases}$$

In practice, however, applications have to deal with non-compliant inputs that lie in $I - I_S$. Any real program P_{real} has its idiosyncratic way S_d of parsing non-compliant files:

$$P_{real}(x) = \begin{cases} S_f(x) & \text{if } x \in I_S \\ S_d(x) & \text{if } x \in I_d \text{ where } S_d : I_d \rightarrow O_S \\ Error & \text{if } x \in I - (I_S \cup I_d) \end{cases}$$

Suppose there are n programs $P_1, P_2, \ldots, P_n$ for processing format f. The detector AV does not know which of them will be used on the endhost and must produce:

$$AV(x) = \begin{cases} S_f(x) & \text{if } x \in I_S \\ S_{d_1}(x) \ldots \cup S_{d_n}(x) & \text{if } x \in I_{d_1} \ldots \cup I_{d_n} \\ Error & \text{if } x \in I - (I_S \cup I_{d_1} \\ & \ldots \cup I_{d_n}) \end{cases}$$

Building such a parser is very difficult. For example, specifications of archive formats usually do not say what to do when some member files are damaged or malformed (e.g., have invalid checksums). Some applications extract only the valid files, others generate an error, yet others attempt to repair the damage or simply ignore the invalid checksums. Critically, *many applications produce usable outputs even for the input files that are invalid according to the format specification*.

Detectors do not parse in the same way as applications. First, the parsing functionality of applications is much richer than that of malware detectors. Detectors only implement the bare minimum needed to analyze a file for malware. In the above example, many detectors ignore checksums in archives because their goal is to find hidden malware code, not verify file integrity. At first glance, a parser that ignores checksums seems safe because, in theory, it should accept strictly more inputs than a parser that verifies checksums. As we show in Section VI, this is not true! Ignoring checksums introduces subtle parsing discrepancies between the parser and the application and enables werewolf attacks.

Second, applications often have bugs in their file-parsing code. The detector must replicate every known and unknown parsing bug in every application that could be used on any endhost to handle the file.

Third, many format specifications are incomplete and/or nondeterministic. As a consequence, different applications make different choices and parse even the same *compliant* file in different ways. For example, parsing of PDF files is notoriously loose [14, 26].

Fourth, specifications of proprietary file formats are often closed-source and change with every release of the application, making it infeasible for the implementors of malware detectors to keep up.

It is hard to replicate the application's parsing logic. Even with access to the application's parser, it is difficult to write another parser that exactly replicates its behavior on all possible inputs. For example, after 12 years of bug-fixing there are still many file-parsing discrepancies between the "functionally equivalent" busybox and coreutil versions of Unix utilities [7]. 238 out of 743 compatibility bugs between OpenOffice and MS Office are caused by file processing [24] and even after a significant reverse-engineering effort, faithful replication of parsing remains a challenge [23].

In general, complete replication of the input-output behavior is infeasible for most non-trivial systems. Non-parsing examples include differences between OS implementations of the same network protocol stack (exploited by `Nmap`) and differences between monitored and unmonitored execution environments (exploited by split-personality malware [8]).

Same file can be parsed according to different, contradictory formats. Many file formats are flexible enough that an attacker can craft a single file which is valid according to multiple file formats and can be correctly parsed in multiple ways. For example, as mentioned in Section VI, a werewolf file consisting of a valid TAR archive followed by a valid ZIP archive can be processed either by `tar`, or by `zip` and will produce different results depending on which program is used. Similar attacks are possible for other format pairs, such as ELF and ZIP or PE and CAB.

The detector must determine all possible formats with which the file may be compatible, and, for each format, parse the file in all possible ways supported by all applications dealing with this format. Even if this were feasible, it is likely to impose an unacceptable performance overhead.

Detector must keep an up-to-date list of all applications on all protected endhosts. Even if the detector were capable of accurately replicating hundreds of different file-parsing algorithms, it must know *which* algorithms to apply. To do

this, the detector must know which applications may handle the file of any of the protected endhosts at any given time, and its parsing logic must be promptly updated whenever a new version of any application is installed on any endhost. In many cases—for instance, when the detector is running on a mail server—the complete set of protected applications may not even be known.

For example, one of our werewolf attacks involves a TAR archive with a single file and a malformed header specifying a significantly larger length than the actual size of the file. We tested three different Linux applications: GNU tar 1.22, 7-Zip 9.04 beta, and File Roller 2.30.1.1. 7-Zip was not able to extract the contents. GNU tar extracted the contents with an "unexpected EOF" warning. Surprisingly, File Roller, which is a GUI front-end for GNU tar, failed to extract the contents. Further examination revealed that even though GNU tar extracts correctly, it returns 2 instead of the usual 0 because it reached the end of file much earlier than it was expecting based on the length field of the header. This causes File Roller to produce an error message.

File parsing is version-dependent even in the same application. For example, GNU tar up to version 1.16 supported ustar type N header logical records, but later versions of GNU tar no longer do.

It is hard to update parsing code. Adding or modifying a file parser is not nearly as simple as adding a new virus signature. All signatures share a common format, thus a generic signature-matching engine is usually capable of handling both old and new signatures. Adding new signatures does not require any changes to the signature format or the scanning code. Parsers, on the other hand, must be implemented by hand and manually updated after any change in the parsing logic of any of the protected applications.

Normalization is no easier than parsing. Normalization—rewriting a non-compliant file so that it complies with the format specification—may help in defeating werewolf attacks. Unfortunately, it requires parsing the file first and thus faces all the problems outlined above.

For example, consider normalizing an archive to remove invalid files. The detector must parse the archive to find individual files and determine their validity according to the specification of each file's format. This is extremely error-prone. Suppose that per specification, the 5th byte of the file contains the format version number. Now the detector must keep track of valid version numbers for each format, and so on. The notion of validity varies dramatically from file to file, with different parts of the header and content used for this purpose in different formats. This makes normalization infeasible for all but the simplest formats.

B. Do not parse files in the detector

An alternative to parsing is to submit each file to a virtual environment that lets it be parsed by the actual application or loaded by the guest OS, then tries to detect malware from outside the OS (e.g., using virtual-machine introspection [13]). This approach defeats chameleon and werewolf attacks only if all of the following hold: (1) the guest OS and applications are *exact* replicas of the protected endhost; (2) if there are multiple endhost configurations (e.g., if different hosts may use different applications or versions of the same application to access a given file), every configuration is replicated exactly; (3) the analysis environment exactly replicates human behavior, including user responses to "repair corrupted file?" messages; and (4) the environment is not vulnerable to split-personality evasion [8]. Production deployment of network- or cloud-based malware detectors that satisfy all of these requirements is a hard problem beyond the scope of this paper.

C. Defend in depth

Many attacks are detector-specific, thus applying multiple detectors to the same file—as advocated by CloudAV [21]—may provide better protection, at a significant performance cost. Some of our attacks, however, evaded all 36 tested AV scanners. Furthermore, several non-interfering attacks can be combined in a single file, enabling it to evade multiple detectors.

IX. Host-Based Defenses Against Werewolf Attacks

One of the main purposes of network-based deployment of malware detectors is to reduce the need for host-based detection. Nevertheless, we discuss host-based defenses for completeness. Host-based techniques—such as continuously scanning the memory for signs of malicious behavior—are effective because the detector operates during or after the file has been processed and thus does not need to independently replicate the results of file processing. Therefore, host-based detectors are better equipped to deal with chameleon and werewolf attacks. In practice, however, many are vulnerable to the same attacks as their network-based versions.

A. On-access scanning

A typical on-access scanner intercepts file-open, file-close, and file-execute system calls and scans their targets for infection. On-access scanners are effective against werewolf attacks on *archive formats only* because they do not need to parse archives. After the user has extracted the contents, she will try to open and/or execute them. At this point, the scanner intercepts the open/execute system call and detects the virus before any harm is done. This is a special case where the results of parsing (i.e., the extracted files) are stored in the file system and thus accessible to the scanner.

Unfortunately, as we show in this paper, werewolf attacks affect not only archive formats, but also ELF, PE, and MS Office (among others). For these formats, existing on-access scanners do not have access to the internal data

representation after the OS or application has parsed the file and must rely on their own parsing, opening the door to werewolf attacks. For example, on-access scanning in ClamAV uses a Linux kernel module called Dazuko that scans the target files of open, close, and exec system calls. In our experiments, ClamAV successfully detected an infected file unpacked from a malformed archive into the monitored directory, but failed to detect an infected Word file with an empty VBA project name (see Section VI-B) even when opened by OpenOffice from the same directory.

B. Tight integration with applications

When running on the host, a malware detector can benefit from tighter integration with the file-processing logic of the OS and applications. One plausible approach is for the OS and application implementors to refactor their code so that the detector can be invoked in the middle of file processing and given access to the results of parsing. Unfortunately, this approach is insecure against malware that exploits vulnerabilities in the parsing code itself. For example, a detector that waits until the JPEG library has parsed a JPEG file before checking that the file is safe cannot protect the library from malicious JPEGs that use bugs to take it over, defeating the purpose of malware detection. Furthermore, tight integration between applications and external functionality which is not integral to the their operation adds complexity and is contrary to the principles of modular system design.

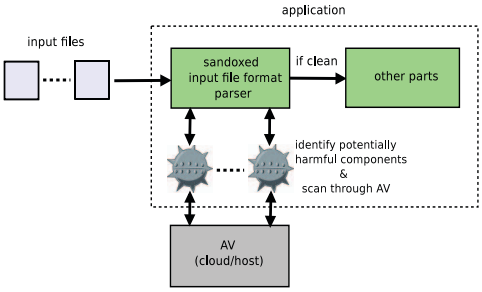

Figure 4. Application refactoring to mitigate werewolf and chameleon attacks on host- and cloud-based malware scanners.

Privilege separation can help solve this "chicken and egg" dilemma if the application is refactored so that the parsing code runs at lower privilege than the rest of the application. The parser can invoke a host- or even cloud-based malware detector and send the results of parsing for scanning, as shown in Fig. 4. After the detector declares them clean, they are passed on to the rest of the application. This architecture avoids the need to replicate application-specific parsing in the detector. Even if malware exploits a vulnerability in the parser, it will only gain the ability to perform low-privilege operations that the parser is allowed to perform.

Implementing this architecture requires that the antivirus vendors support a standardized interface through which applications can submit parsed data for analysis. Some existing archiving applications such as WinRAR and WinZip support invocation of command-line antivirus scanners, but this functionality is not yet available in non-archiving applications.

Another approach is for the application and the detector to use the same parsing code, e.g., by employing the same parsing library. For instance, multiple-streams and random-garbage attacks do not work against ClamAV because ClamAV uses the `libz` library for parsing GZIP files. The origins of `libz` are similar to `gzip`, thus ClamAV effectively uses the same parsing code as the application. This approach suffers from most of the flaws outlined in Section VIII-A—the detector must know exactly which parsing code is used by the application and must be updated whenever the application's parsing logic changes—but these flaws may be easier to mitigate in a host-based deployment.

X. Conclusion

We presented two classes of practical attacks against automated malware detectors. They enable even unobfuscated, easily recognizable malware to evade detection by placing it in specially crafted files that are processed differently by the detector and the endhost. All 36 antivirus scanners in our experimental testing proved vulnerable to these attacks, yielding a total of 45 different exploits, almost all of which are reported here for the first time. The rest have been known only anecdotally and never been systematically analyzed.

We argue that semantic gaps in file processing are a fundamental flaw of network- and cloud-based malware detectors, regardless of the actual detection technique they use. As long as the detector analyzes files on its own, independently of the actual operating systems and applications on the endhosts, it faces the insurmountable challenge of correctly replicating their file-processing logic on every possible input. Development of malware detectors that do not suffer from this gap—for example, if they operate on exact virtual copies of protected systems that process each file using the actual applications and faithfully emulate human response, or if they are integrated with the parsing logic of actual applications—is an interesting topic for future research.

Acknowledgments. The research described in this paper was partially supported by the NSF grants CNS-0746888 and CNS-0905602, Google research award, and the MURI program under AFOSR Grant No. FA9550-08-1-0352.

References

[1] S. Alvarez. Antivirus insecurity. http://events.ccc.de/camp/2007/Fahrplan/attachments/1324-AntivirusInSecuritySergioshadownAlvarez.pdf, 2007.

[2] S. Alvarez and T. Zoller. The death of AV defense in depth? - revisiting anti-virus software. http://cansecwest.com/csw08/csw08-alvarez.pdf, 2008.

[3] avast! Anti-virus engine malformed ZIP/CAB archive virus detection bypass. http://secunia.com/advisories/17126/, 2005.

[4] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *S&P*, 2009.

[5] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side XSS filters. In *WWW*, 2010.

[6] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *USENIX Security*, 2007.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[8] X. Chen, J. Andersen, Z. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *DSN*, 2008.

[9] ClamAV. http://www.clamav.net.

[10] http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=evasion, 2012.

[11] EICAR — The Anti-Virus or Anti-Malware Test File. http://www.eicar.org/anti_virus_test_file.htm.

[12] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS*, 2003.

[13] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, 2003.

[14] M. Gavin. Recognizing corrupt and malformed PDF files. http://labs.appligent.com/presentations/recognizing_malformed_pdf_f.pdf.

[15] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *USENIX Security*, 2011.

[16] M. Hypponen. Retroviruses - how viruses fight back. http://www.hypponen.com/staff/hermanni/more/papers/retro.htm, 1994.

[17] G. Kessler. File signatures table. http://www.garykessler.net/library/file_sigs.html, 2012.

[18] McAfee VirusScan vulnerability. http://www.pc1news.com/news/0665/mcafeevirusscanvulnerability-allow-compressed-archives-to-bypass-the-scan-engine.html, 2009.

[19] J. Nazario. Mime sniffing and phishing. http://http://asert.arbornetworks.com/2009/03/mime-sniffing-and-phishing/, 2009.

[20] J. Oberheide, M. Bailey, and F. Jahanian. PolyPack: An automated online packing service for optimal antivirus evasion. In *WOOT*, 2009.

[21] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-version antivirus in the network cloud. In *USENIX Security*, 2008.

[22] J. Oberheide and F. Jahanian. Remote fingerprinting and exploitation of mail server antivirus engines. http://jon.oberheide.org/files/umich09-mailav.pdf, 2009.

[23] Microsoft patch breaks Impress/PowerPoint compatibility. http://user.services.openoffice.org/en/forum/viewtopic.php?t=36515, 2010.

[24] OpenOffice-MS interoperability bugs. http://openoffice.org/bugzilla/buglist.cgi?keywords=ms_interoperability, 2011.

[25] W. Palant. The hazards of MIME sniffing. http://adblockplus.org/blog/the-hazards-of-mime-sniffing, 2007.

[26] S. Porst. How to really obfuscate your PDF malware. http://www.recon.cx/2010/slides/recon_2010_sebastian_porst.pdf, 2010.

[27] T. Ptacek and T. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection, 1998.

[28] T. Scholte, D. Balzarotti, and E. Kirda. Quo vadis? A study of the evolution of input validation vulnerabilities in Web applications. In *FC*, 2011.

[29] C. Sheehan. Pimp my PE: Parsing malicious and malformed executables. http://research.sunbelt-software.com/ViperSDK/Pimp%20My%20PE.ppt, 2007.

[30] IE content-type logic. http://blogs.msdn.com/b/ie/archive/2005/02/01/364581.aspx, 2005.

[31] P. Ször and P. Ferrie. Hunting for metamorphic. http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf.

[32] Virus Total. http://www.virustotal.com.

[33] VX Heavens. http://vx.netlux.org/vl.php.

[34] R. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *WOOT*, 2007.

[35] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. A systematic analysis of XSS sanitization in Web application frameworks. In *ESORICS*, 2011.

[36] A. Wheeler and N. Mehta. 0wning antivirus. http://www.blackhat.com/presentations/bh-europe-05/bh-eu-05-wheeler-mehta-up.pdf, 2005.

[37] F. Xue. Attacking antivirus. http://www.blackhat.com/presentations/bh-europe-08/Feng-Xue/Whitepaper/bh-eu-08-xue-WP.pdf, 2008.

[38] Anti-virus software may not properly scan malformed zip archives. http://www.kb.cert.org/vuls/id/968818, 2005.

[39] Musing on information security. http://blog.zoller.lu/search/label/Advisory, 2009.

APPENDIX

Table XIII
EXAMPLES OF CLAMAV'S FIXED-OFFSET "MAGIC STRINGS" IN THE ORDER THEY ARE CHECKED.

Order	Offset	Length	File type	Magic content	Order	Offset	Length	File type	Magic content
1	0	9	mirc.ini	5b 61 6c 69 61 73 65 73 5d	25	0	5	EVS mail	582d455653
2	257	5	TAR-POSIX	7573746172	26	0	17	Mail	582d4170706172656e746c792d546f3a20
3	0	5	RTF	7b5c727466	27	0	4	Mail	546f3a20
4	0	14	SIP log	5349502d48495420285349502f48	28	0	9	Mail	5375626a6563743a20
5	8	4	SIS	19040010	29	0	4	compress. exed	535a4444
6	6	4	JPEG	4a464946	30	0	13	Maildir	52657475726e2d706174683a20
7	6	4	JPEG	45786966	31	0	13	Maildir	52657475726e2d506174683a20
8	0	3	MP3	fffb90	32	0	10	Raw mail	52656365697665643a20
9	0	3	JPEG	ffd8ff	33	0	4	RAR	52617221
10	0	8	OLE2 container	d0cf11e0a1b11ae1	34	0	4	RIFX	52494658
11	0	8	CryptFF	b6b9acaefeffffff	35	0	4	RIFF	52494646
12	0	4	PNG	89504e47	36	0	8	ZIP	504b3030504b0304
13	0	4	ELF	7f454c46	37	0	4	ZIP	504b0304
14	0	4	TNEF	789f3e22	38	0	4	Ogg Stream	4f676753
15	0	14	VPOP3 (DOS)	763a0d0a52656365697665643a20	39	0	12	Mail	4d6573736167652d49643a20
16	0	13	VPOP3 (UNIX)	763a0a5265636569766 5643a20	40	0	12	Mail	4d6573736167652d49443a20
17	0	6	UUencoded	626567696e20	41	0	2	MS-EXE	4d5a
18	0	2	ARJ	60ea	42	0	4	MS CAB	4d534346
19	0	8	Mail	582d5549444c3a20	43	0	4	MS CHM	49545346
20	0	11	Symantec	582d53796d616e7465632d	44	0	3	MP3	494433
21	0	9	Mail	582d53696576653a20	45	0	26	Qmail bounce	48692e20546869732069732074686520716d61696c2d73656e64
22	0	11	Mail	582d5265616c2d546f3a20	46	0	3	GIF	474946
23	0	15	Mail	582d4f726967696e616c2d546f3a20	47	0	6	Exim mail	46726f6d3a20
24	0	17	Mail	582d456e76656c6f70652d46726f6d3a20	48	0	5	MBox	46726f6d20

Table XIV
"MAGIC STRINGS" FOR FILE TYPES NOT SUPPORTED BY CLAMAV (SOURCE: [17]). MULTIPLE OFFSETS SEPARATED BY "," INDICATE THAT THE MAGIC CONTENT CAN APPEAR AT ANY OF THEM.

Offset	Length	File type	Magic content
0	8	MS Office files	D0 CF 11 E0 A1 B1 1A E1
0	2	TAR.Z (LZW)	1F 9D
0	2	TAR.Z (LZH)	1F A0
0	8	AR, MS Coff	21 3C 61 72 63 68 3E 0A
0	4	PACK	50 41 43 4B
2	3	LZA,LZH	2D 6C 68
0	6	7Zip	37 7A BC AF 27 1C
526	5	PKZIP SFX	50 4B 53 70 58
29,152	6	WinZip	57 69 6E 5A 69 70
30	6	PKLITE	50 4B 4C 49 54 45
0	4	PKZIP	50 4B 03 04
0	4	Zoo	5A 4F 4F 20
4	8	Quicktime MOV	6D 6F 6F 76
0,30	23	EPS	25 21 50 53 2D 41 64 6F 62 65 2D 33 2E 30 20 45 50 53 46 2D 33 20 30
32769, 34817, 36865	5	ISO	43 44 30 30 31