Revealing Packed Malware

Wei Yan
Trend Micro

Zheng Zhang
McAfee

Nirwan Ansari
New Jersey Institute of Technology

The past few years have witnessed a significant increase in malware threats to computer users, threats that also pose a serious risk to the Internet's integrity. Malware exploits software vulnerabilities to compromise computers and help attackers steal users' private data. To evade malicious content detection, malware authors use *packers*, binary tools that instigate code obfuscation. By using executable packers, modern malware can completely bypass personal firewalls and antivirus (AV) scanners. Thus, security researchers are facing a great challenge in overcoming malware's complexity. Reverse engineering (RE) has become an important approach to analyzing a program's logic flow and internal data structures, such as system call functions. Security researchers and AV products must be able to unpack and inspect the payloads hidden within the packed programs using RE tools.

The packer problem

Packers are software programs that compress and encrypt other executable files in a disk and restore the original executable images when the packed files are loaded into memories. A packed file is a type of archived file, so we can't say that just because a file is packed, it's bad. Some commercial packers help protect Windows applications against modern cracking tools by putting those applications into a strong protection "shell." (Anticracking technology includes encrypting code areas, verifying licenses, and protecting codes from decompiling.) Software vendors aiming to save storage space also use packers to compress their products, which can reduce download time and save customers Internet bandwidth.

However, viruses have used packers widely to avoid detection, and packers are increasingly incorporated into some malware families. Reportedly, among 735 malwares collected for the WildList in March 2006, more than 92 percent were packed by crypters and packers from 30 different families.[1] AV vendors must mitigate an astronomical number of packers every day.

To identify known malware, existing commercial security applications search a computer's binary files for predefined signatures, but obfuscated viruses use software packers to protect their internal code and data structures from detection. AV scanners act like file filters, inspecting suspicious file loading and storing activities, but with obfuscated content, malicious programs can bypass AV scanners and are ultimately unpacked and executed in the victim system. If malware intends to infect more files or propagate to other computers, it might re-encrypt itself to bypass the AV filter again.

Gabor Szappanos studied possible approaches for blacklisting custom packers that only malware uses.[2] However, AV scanners must still unpack and scan the internal original content for samples packed by unknown or "grey" packers, which both good and bad files could potentially use. AV scanners will ideally detect real viruses within packed files no matter which packers attackers have used, but a false positive will occur if a packed file has no Trojan but is detected as malware by the scanner. To decrease the false positive rate, AV scanners must be able to unpack the samples and retrieve the unpacked data. Packed malware must unpack itself at runtime before it executes, and so security researchers can use RE tools to find the exact moment and location where the original data will be uncompressed and available. However, AV vendors lack the time to learn how each packer works. Consequently, some AV scanners simply report all executable files compressed by the same packer as viruses, causing false alarms. AV vendors must consider their legal liability and the compensation the benign compressed executables might require in the event of false positive damage.

How packers work

Executable files are specially formatted file objects that operating systems understand and execute. Modern executable formats include Portable Executable (PE)

format[3] for Windows, Executable and Linkable Format (ELF) for Linux, and Mach Object (Mach-O) for Mac OS. Here, we'll focus on the PE format because it's the most popular format for executables, libraries, and drivers in Windows. PE tools facilitate easily viewing, analyzing, and editing WIN32 PE files.

A PE file comprises various sections and headers that describe the section data, import table, export table, resources, and so on. As Figure 1 shows, a PE file starts with the DOS *executable header*, followed by the *PE header*, which begins with the signature bits "PE." The PE header also includes some general file properties, such as the number of sections, machine type, and time stamp. The *optional header* contains several important information segments and is followed by the *section table headers*, which summarize each section's raw size, virtual size, section name, and so on. Finally, at the end of the PE file is the section data, which contains the file's *original entry point* (OEP)—that is, the entry point where file execution begins. To search a PE file for malware, a scanner typically scans the segments for the known signatures at certain offsets from OEP.

Most PE packers work only on executable files and dynamic link libraries (DLLs). They can be written in different programming languages, such as C++, Delphi, Visual Basic, or even Assembly. Aside from shrinking the original file size, packing is an efficient way to obfuscate a file's original contents, and as of publication time, packers are malware authors' favored binary tools for obscuring their codes.

Code obfuscation has evolved from simple compression/encryption to polymorphism/metamorphism and finally to packing. Based on their purposes and behaviors, we can broadly classify packers into four categories:

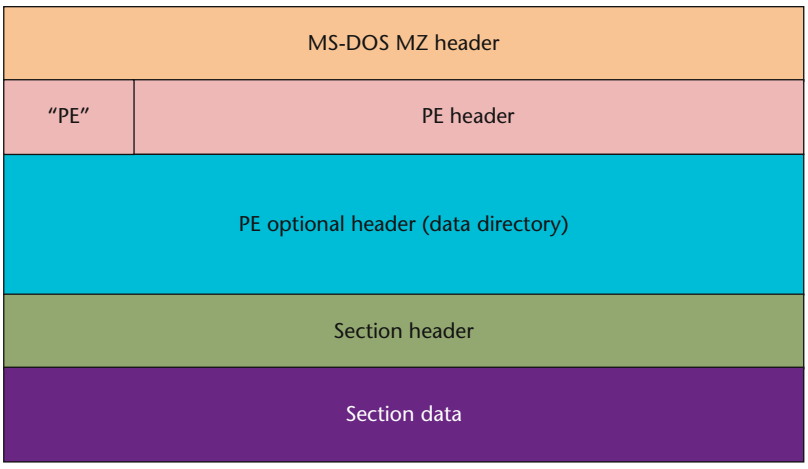

Figure 1. PE file format. This format is used by Windows executables. It consists of the "PE" signature, optional and section headers, and the section data.

- *Compressors* shrink file sizes through compression with little or no anti-unpacking tricks. Popular compressors include the Ultimate PE Packer (UPack; www.wex.cn/dwing), Ultimate Packer for Executables (UPX; http://upx.sourceforge.net), and ASPack (www.aspack.com).
- *Crypters* encrypt and obfuscate the original file contents and prevent the files from being unpacked without any compression. Malware developers widely use crypters such as Yoda's Crypter (http://yodap.sourceforge.net/) and PolyCrypt PE (www.jlabsoftware.com).
- *Protectors* combine features from both compressors and crypters. Some commercial protectors, such as Armadillo (www.siliconrealms.com/) and Themida (www.oreans.com), also include comprehensive license-management and antipiracy functions.
- *Bundlers* pack a software package of multiple executable and data files into a single bundled executable file, which unpacks and accesses files within the package without extracting them to disk. Examples of typical PE bundlers include PEBundle (www.bitsum.com/pebundle.asp) and MoleBox (www.molebox.com).

To perform packing, a packer first parses PE internal structures. Then, it reorganizes PE headers, sections, import tables, and export tables into new structures and attaches a code segment that the malware will invoke before the OEP. This code is called the stub, and it decompresses the original data and locates the OEP.

During packing, a packer compresses and encrypts the code and resource sections using the *compression and encryption libraries*. With randomization, the packer can also generate different variants of a single file every time the file is packed. For some powerful packers, the polymorphism engine also adds a protection layer against RE and debugging. Generally, when a computer invokes a packed file, the packer stub will first be invoked to unpack the file in the memory, and then the codes in the original file will get executed. There are several steps the stub engine needs to follow:

- save the register context at the entry point (usually with a `PUSHA` [push all general registers] instruction);
- decrypt and decompress the code and data sections;
- load and link the libraries and

APIs that the original executable imported;
- restore the register context saved at the entry point (usually with a `POPA` [pop all general registers] instruction); and
- continue to execute the instructions at the OEP, usually with an intersection jump instruction.

Another obfuscation technology is *API call redirection*, which aims to make an executable file smaller and prevent it from running if a security application doesn't unpack it correctly. To hide Windows API function calls, a packer usually destroys the original import table. To unpack a packed file, the stub decompresses the data, acquires each API's address, and rebuilds the import table. Reconstructing an import table scrambled by the polymorphism code image is very difficult. In addition, malware authors have developed various anti-unpacking techniques to prevent packed files from being unpacked and cracked, for example,

- calculating the CRC checksum of the packed executable file to detect file patching;
- inserting useless junk codes between the useful instructions to fool a static decompiler;
- triggering various exceptions to detect dynamic debugging; or
- redirecting and mutating the original executable's instructions with different but equivalent ones to prevent memory dumping.

Differences exist between a self-extracting archived file and a packed one. Users must extract an archived file onto hard disks before they can access it. So, AV programs might still be able to detect the "unarchived" temporary files, even if they can't unpack the archived ones. (One popular archive tool on the Windows platform is Winzip; www.winzip.com). On the other hand, a packed file will be unpacked only in memory, and users can't stop its execution once the file starts to run. Thus, AV applications must have a built-in functionality to unpack the packed malware.

Unpacking malware

Because packing has become the most common method for malware authors to obfuscate code, it's vital for security researchers and AV products to be able to unpack and inspect payloads hidden within packed programs.

Unpacking methods

Unpacking is the process of stripping the packer layer (or layers) off packed executables to restore the original contents so that AV programs and security researchers can inspect and analyze the original executable signatures. We can use three different techniques to unpack a packed file: *manual unpacking*, *static unpacking*, or *generic unpacking*.

Security researchers and hackers commonly use manual unpacking to execute the packed programs by using native debuggers—for example, SoftIce (www.compuware.com) and Ollydbg (www.ollydbg.de)—to analyze the packer layers' encryption and decompression algorithms and manually restore the original files. This process is time consuming and requires deep understanding of kernel and assembly programming, but with sufficient time and knowledge, researchers can fully reverse obfuscated viruses' underlying logic and, interestingly, can often discover nonobvious bugs hidden within the programs. Owing to manual packings' highly skilled requirements and manual nature, only knowledgeable researchers within controlled environments can carry it out.

To automate packer detection in the field, AV programs usually develop static unpackers, which are dedicated routines to decompress/decrypt executables packed by specific packers without actually executing the suspicious programs. Sample static unpackers include Heaventools' UPX and Upack unpacker plugins (www.heaventools.com/peexplorer-upack-unpacker.htm). Static unpacking is very efficient for unpacking files packed by known packers, but virus developers can bypass them using unforeseen or custom packers. Thus, generic unpacking—which uses programs to execute or emulate unknown packed executables until they're fully decrypted in memory—is becoming increasingly important for AV providers wanting to decrypt unknown samples.. IDA Pro (www.datarescue.com/idabase/index.htm) provides a debugger-based universal unpacker, which can unpack many simple packers. Tobias Graf presented an emulator-based generic unpacking engine.[4] Despite its flexibility and potential, AV products don't widely use generic unpacking, mainly owing to complexities inherent in implementing a secure and efficient generic unpacking engine, and also because deciding when the packed files are fully unpacked is difficult.

Aside from some rare exceptions, most obfuscated programs require an intersection long jump to transit the execution flow from the packer section to the section containing OEP. If a generic unpacking engine can capture the intersection jumps, the AV engine could use the following heuristics to determine whether the OEP has arrived:

- *Instruction pointer rule*—IDA Pro's universal unpacker plugin tracks the destination instruction pointer (referred to as EIP for Intel IA32 processors) and assumes that the OEP has been reached once EIP falls within a section located before the packer layer and that the packed file has been fully unpacked before the OEP jumps.

- *Stack pointer rule*—to ensure that the original executable executes correctly, most packers will restore the stack level (referred to as ESP for Intel IA32 processors) to the value it had when the packer codes start to execute.
- *Signature rule*—Graf proposed searching for popular compilers' entry signatures,[4] for example, Microsoft Visual C++, GNU C++, or Delphi, whose signatures are relatively static among all executables these compilers generate.
- *Behavior rule*—Graf also proposed stopping the OEP searching at some Windows API functions,[4] such as `CreateWindowA`, which aren't usually called by the packer codes.

By applying these four rules together, the generic unpacking engine can reliably distinguish the original executable codes from the packer codes.

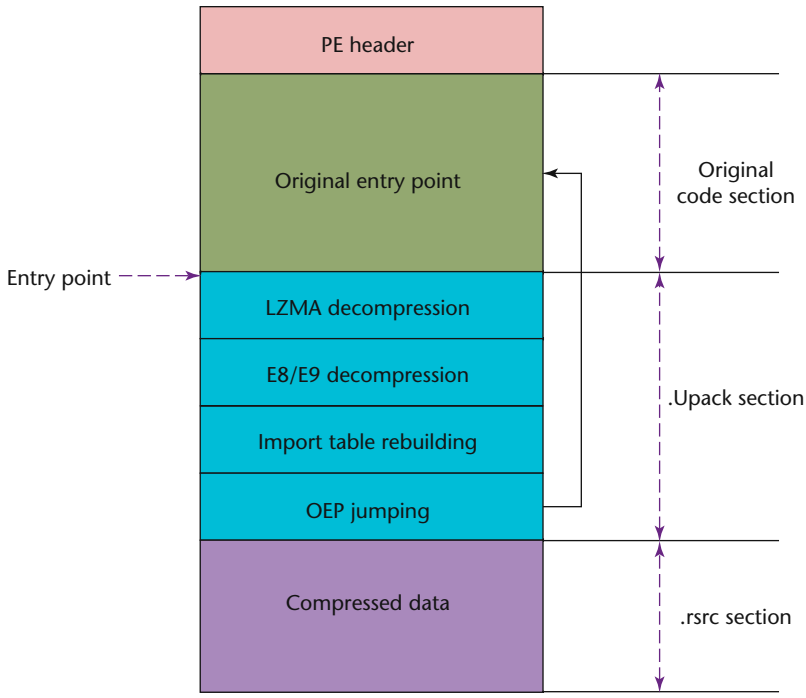

Figure 2. UPack workflow. UPack stores the compressed data in the .rsrc section and decompresses them into a new section using the Lempel-Ziv-Markov algorithm (LZMA). The Upack section includes LZMA parameters and decompression codes.

Unpacking UPack

UPack is a Windows-based compression packer that compresses PE-formatted files with very high compression rates. Various viruses and worms have used it to avoid detection, such as W32/Zotob (http://vil.nai.com/vil/content/v135433.htm) and W32/Mytob (http://vil.nai.com/vil/content/v132158.htm). UPack uses a modified version of the Lempel-Ziv-Markov algorithm (LZMA) as the compression engine. A UPacked file consists of two sections, ".Upack" and ".rsrc." At the beginning of the ".Upack" section is the output data buffer for decompression stub, followed by the import table data. The ".rsrc" section contains the compressed source data.

As Figure 2 shows, UPack's unpacking process involves four consecutive steps: modified LZMA decompression, E8/E9 decompression, import table rebuilding, and jumping to OEP. However, UPack changes LZMA parameters to modify the normal LZMA decoders without affecting compression performance. E8/E9 (jumping instructions) decompression can increase the compression ratio for some data types, such as short jumps, which can increase the compression ratio by 5 to 10 percent. The import table rebuilding and jumping to OEP stages are similar to other packers. In the import table building stage, UPack extracts the DLL names, followed by the trunk table addresses and APIs. After that, UPack jumps to OEP. For files with relocation tables, UPack stores the Relative Virtual Address (RVA) of the relocatable data blocks, which will be relocated when the relocation table needs rebuilding. The UPacked executables' behavior meets the EIP and ESP rules defined in the previous sections. Thus, it's relatively easy for a generic unpacking engine to accurately detect the OEP.

Evolving packers

These days, more malware is packed, and the AV industry has witnessed malware's shift in emphasis to virtual machine (VM) protectors. VM protectors have become the new generation of packers. They convert assembly instructions into bytecodes and use VM to interpret those codes, which are extremely difficult to disassemble using traditional RE tools. Every VM protector has a different VM, which means AV vendors have a hard time keeping up with the new packers.

A VM protector normally includes a compiler, interpreter, and handler. When packing, a protector replaces the assembly codes with its own bytecodes using the compiler, and so original codes of malware will never appear in the file. During execution, the packed malware can execute the VM handler via the bytecode interpreter. One possible way to defeat VM protectors is to divide them into function

modules. For each module, security researchers can compare the VM context differences before and after that module's execution. Then, they can guess its function via both static and dynamic analysis. The major concern remains: Who has the time to reverse all the bytecodes given that security researchers are already preoccupied with a large backlog of malware?

Today's AV industry devotes much effort to combating packed malware. Various new emerging technologies let AV software detect packers undergoing modifications. At the same time, however, hackers are launching unknown malware, which most AV software can't detect. This trend will continue into the future. □

References

1. T. Brosch and M. Morgenstern, "Runtime Packers: The Hidden Problem?" keynotes from Black Hat USA 2006 Briefings and Training, www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf.
2. G. Szappanos, "Exepacker Blacklisting: Theory and Experiences," *Proc. 2nd Int'l Computer AntiVirus Researchers Organization Workshop*, (CARO), 2008; www.datasecurity-event.com/uploads/gszappanos.ppt.
3. M. Pietrek, "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format," *Microsoft Systems J.*, Mar. 1994, pp. 15–34.
4. T. Graf, "Generic Unpacking—How to Handle Modified or Unknown PE Compression Engines," *Proc. 2005 Virus Bulletin Conf.*, Virus Bulletin, 2005.

Wei Yan *is a senior threat researcher in advanced threats research at Trend Micro. His research interests include malware detection and classification, signature automatic generation, and intrusion detection. Yan has a PhD in computer engineering from the New Jersey Institute of Technology. He is a member of the IEEE and Usenix. Contact him at wei_yan@trendmicro.com.*

Zheng Zhang *is a research scientist in AVERT Labs at McAfee. His research interests include software unpacking and decryption, malware sandboxing, and network intrusion detection. Zhang has a PhD in electrical engineering from the New Jersey Institute of Technology. Contact him at zzhang@avertlabs.com.*

Nirwan Ansari *is a professor of electrical and computer engineering at the New Jersey Institute of Technology. His research interests include various aspects of broadband networks and multimedia communications. Ansari has a PhD in electrical engineering from Purdue University. Contact him at nirwan.ansari@njit.edu.*