OmniUnpack: Fast, Generic, and Safe Unpacking of Malware

Lorenzo Martignoni
Università degli Studi di Milano
lorenzo@security.dico.unimi.it

Mihai Christodorescu
IBM Research[*]
mihai@us.ibm.com

Somesh Jha
University of Wisconsin
jha@cs.wisc.edu

Abstract

Malicious software (or malware) has become a growing threat as malware writers have learned that signature-based detectors can be easily evaded by "packing" the malicious payload in layers of compression or encryption. State-of-the-art malware detectors have adopted both static and dynamic techinques to recover the payload of packed malware, but unfortunately such techniques are highly ineffective. In this paper we propose a new technique, called OmniUnpack, to monitor the execution of a program in real-time and to detect when the program has removed the various layers of packing. OmniUnpack aids malware detection by directly providing to the detector the unpacked malicious payload. Experimental results demonstrate the effectiveness of our approach. OmniUnpack is able to deal with both known and unknown packing algorithms and introduces a low overhead (at most 11% for packed benign programs).

1. Introduction

Packed malicious programs (malware) pose a significant problem in malware analysis, detection, and forensics. Such programs consist of a decompression or decryption routine that extracts the garbled payload from memory and then executes it. We use the term *packed* and its variations to refer to malware whose payload is either compressed or encrypted. This unpacking routine can be invoked once, in which case the whole payload is extracted to memory in a single step, or multiple times, when parts of the payload are extracted to memory at different times. For a security analyst, this means that the program has to be executed in a contained yet accurate environment before an analysis of the payload can be performed. For a malware detector, this means that the scanning for malicious code has to be postponed until after the start of execution, i.e., when the program has unpacked its payload.

Malware writers have learned that binary packers are effective at bypassing signature-based detectors and at keeping the malware undetected for longer [16]. The numerous packers available online, each with a variety of settings, can generate many variants from the same executable. The percentage of new malware that is packed is on the rise, from 29% in 2003 to 35% in 2005 up to 80% in 2007 [5, 6, 12]. This situation is further complicated by the ease of obtaining and modifying the source code for various packers (e.g., UPX [11]). Modifications to the source code can introduce changes in the compression or encryption algorithm, create multiple layers of encryption, or add protection against reverse engineering. Currently, new packers are created from existing ones at a rate of 10–15 per month [16]. As a result, malware writers have a large selection of tools to pack their malware, to the point that more than 50% of malware samples are simply repacked versions of existing malware [16].

The traditional approach adopted by malware detectors is to use specific unpacking routines to recover the original program, one routine per packing algorithm, before scanning it for malicious code [18] (e.g., with ClamAV [4]). This approach is of course limited to a fixed set of known packers. A technique built on process emulation was then introduced to perform generic unpacking, i.e., without requiring knowledge of the specific packer. As any other dynamic-analysis technique, emulation places a time limit on the execution of the packed program and is restricted by the fidelity of the emulation environment [18, 16]. Other forms of generic unpacking use instruction-level tracing inside a real system to determine when the unpacked code is executed, but incur several orders of magnitude

Desirable feature	OmniUnpack	PolyUnpack	ClamAV
Fast for interactive use	✓	-	✓
Handles unknown packers	✓	✓	-
Incremental detection	✓	✓	-
Resilient to anti-debugging	✓	-	✓
Resilient to SEH attacks	✓	-	✓

Table 1: Unpacker feature comparison.

[*]The work presented in this paper was done at University of Wisconsin.

slowdown due to the use of single-step debugging (e.g., PolyUnpack [15]) or whole-system emulation [3].

We present an unpacking technique called OmniUnpack that addresses the shortcomings of existing systems and integrates efficiently with any malware-detection engine (Table 1). Our approach is generic, being able to handle any type of packer and any type of self-modifying code. OmniUnpack monitors the program execution and tracks written as well as written-then-executed memory pages. When the program makes a potentially damaging system call, OmniUnpack invokes a malware detector on the written memory pages. If the detection result is negative (i.e., no malware found), execution is resumed. We achieve near-native efficiency by tracking memory accesses at the page level (using non-executable pages or equivalent hardware mechanisms) instead of the instruction level. The resulting low overhead of OmniUnpack means that it can be used for continuous monitoring of a production system. Furthermore, OmniUnpack handles any number of unpacking and self-modification layers, each time communicating to the malware-detection engine only the newly generated code that needs to be scanned. The unpacker is implemented with a limited amount of changes to the operating system and does not use debugging, virtual machine, or emulation, making it immune to the vast majority of self-protection tricks employed by malware [18].

This paper makes the following contributions:

- A fast, general-purpose unpacker resilient to anti-debugging, anti-VM, and anti-emulation techniques. Our unpacker integrates with any operating system (e.g., Microsoft Windows) and any malware detector (e.g., ClamAV) without trading off efficiency or unpacking capability.
- An in-memory malware detection strategy independent of packing and self-modification. Our unpacker requires the malware detector to decide whether the program memory contains malicious code. The malware detector needs no knowledge of any packing approaches.
- A set of experimental results indicate the 20 times speedup over existing unpacking tools. Our unpacker imposes only a small overhead (approximately 6%) on programs which are not packed and a slightly higher overhead (approximately 11%) on packed programs.

2. Overview

Our algorithm follows a simple strategy to handle packed code. All memory writes and the program counter are tracked. If the program counter reaches a written memory address, we know that some form of unpacking, self-modification, or code generation occurs in the program. All written-then-executed (or written-and-about-to-be-executed) memory locations should then be analyzed by a malware detector. Starting from this strategy, the goal of our research is to design an unpacking algorithm that achieves low overhead yet does not compromise the security of the system.

Consider a program that was packed three times. An execution trace of the program can be divided into four different stages, one per unpacking routine and one for the original code. Figure 1(a) illustrates such an execution trace, with the unpacking stages in various shades of gray and the original code indicated by ▨. OmniUnpack monitors the execution of the program and records the memory locations that were written and, of those, the memory locations that were written and then executed. As long as there are no memory locations that are written and executed, there is no reason to invoke a malware detector. If a memory location is written and is about to be executed, it is a candidate for malware detection. We avoid the overhead of monitoring individual memory locations by tracking accesses to memory pages, which group multiple locations into one logical unit (e.g., memory pages contain 4096 bytes on most Intel IA-32-based operating systems).

The monitoring of memory accesses is done efficiently inside OmniUnpack through the use of hardware mechanisms. On systems that provide virtual memory capabilities, the hardware manages the translation between virtual and physical addresses, while the operating system (OS) manages the address mapping used by the hardware. Additionally, the hardware provides memory-protection facilities at the page level. When the virtual-physical address mapping needs to be updated or when the memory protection is violated, the hardware signals to the OS through an exception and allows the OS to repair the memory state before continuing execution. We use these existing features to intercept the first moment when a page is written and the first moment when a page is about to be executed after a write.

Imprecision of page-level tracking. Page-level tracking decreases the granularity of monitoring while significantly reducing the overhead of memory-access tracking. As a downside, it is less precise, often resulting in spurious detected unpacking stages. The spurious unpacking stages are caused by multiple layers of packing and by anti-disassembly and anti-static analysis techniques (which build on self-modifying code and thus appear to mark the end of an unpacking stage). Furthermore, code that executes from the same page on which it writes, even though non-self-modifying, also generates multiple spurious unpacking stages. Figure 2

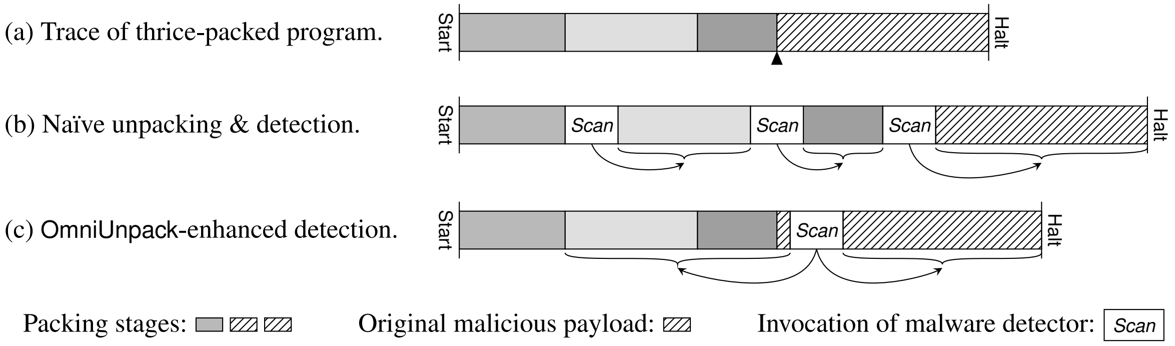
(a) Trace of thrice-packed program.
Start
Halt
(b) Naïve unpacking & detection.
Start
Scan
Scan
Scan
Halt
(c) OmniUnpack-enhanced detection.
Start
Scan
Halt
Packing stages: ▨ ▨ ▨
Original malicious payload: ▨
Invocation of malware detector: Scan

Figure 1: Execution traces for a packed executable: (a) unmonitored, (b) monitored by a simple detector, and (c) monitored by OmniUnpack. The arrow on trace (a) indicates the ideal location for invoking the malware detector. By approximating this ideal location, OmniUnpack reduces the number of detector invocations and the overall overhead.

shows a comparison of the number of unpacking stages detected using fine-grained memory-access tracking (i.e., every machine instruction and every memory byte) and those detected using coarse-grained memory-access tracking (every memory page). It would be unnecessarily expensive to invoke the malware detector every time a written memory page is executed, because such an event (written-then-executed) is frequent. *Written-then-executed pages are indicative of unpacking but not indicative of the end of unpacking*. In our example from Figure 1(a), the ideal moment to invoke the malware detector is at end of all three unpacking stages, marked by the arrow. Unfortunately, the problem of determining when unpacking is completed is undecidable [15].

Approximation of the end of an unpacking stage. We approximate the solution using the following heuristic: if the current execution trace indicates unpacking (i.e., memory pages were written and then executed), and if the program is about to invoke a dangerous system call, then we assume that an unpacking stage has completed and we invoke the malware detector. The effect on the thrice-packed program is illustrated in Figure 1(c), where the malware detector is invoked only once to scan all written memory pages. The overhead is thus reduced by invoking the detector only once, not three times as in the case of the naïve strategy that scans the memory every time a written-then-executed page occurs (Figure 1(b)).

Choice of dangerous system calls. A *dangerous system call* is a system call whose execution can leave the system in an unsafe state. To achieve its malicious goal, the malware has to interact with the system. System calls are the mechanism offered by the OS to interact with the host system. For example, a malware that wants to be executed automatically at system boot has to replicate itself into a new file and create the appropriate registry keys to run the new file at every boot. All the system calls in this sequence are mandatory to achieve the goal, but only a small subset of them are dangerous to the system. For example it is not dangerous to open and read files, but it is dangerous to create and set registry keys. Table 2 lists examples of Microsoft Windows system calls we define as dangerous and monitor. Any system call that modifies OS state was considered dangerous in our evaluation.

Continuous monitoring of the execution. Because of the possibility of multiple unpacking stages and of the approximation we are using to detect them, it is insufficient to monitor and scan the program only once during an execution. OmniUnpack implements a continuous-monitoring approach, where the execution is observed in its entirety. We note that this is a necessary departure

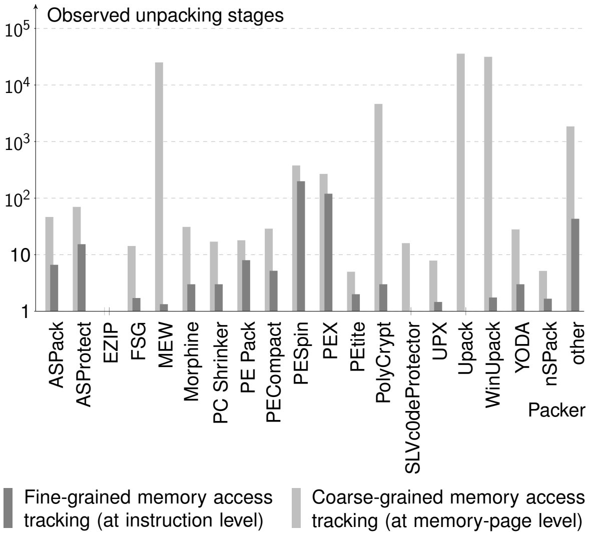

Figure 2: Comparison of the number of unpacking stages per packer, observed using fine- and coarse-grained memory-access tracking. The numbers represent averages over 272 packed samples.

System call	Description
NtWriteFile	Write data into a file
NtSetValueKey	Set the value of a registry key
NtCreateProcess	Create a new process
NtDeleteFile	Delete a file
NtDeleteKey	Delete a registry key
NtWriteVirtualMemory	Write data into memory (can be used to write in the memory of another process)
...	

Table 2: Microsoft Windows dangerous system calls.

from the traditional view of unpacking and scanning as separate, one-time stages of the malware detection process. Attempts to protect the malicious code with multiples layers interleaved with the execution of dangerous system calls or attempts to delay the execution of the unpacked code are worthless with our system. As our experimental results show (Section 5), the low overhead of OmniUnpack allows for continuous monitoring in an end-user environment.

3. The OmniUnpack Algorithm

We now give a detailed description of the algorithm sketched in Section 2. The algorithm ensures that every code fragment reached during a program execution is checked by a malware detector before the host system might be irreparably harmed. The OmniUnpack algorithm monitors the execution of the program and invokes the malware detector on newly generated code. The program is stopped while the malware detector is executed and resumed if no malicious code is found.

Intuitively, newly generated code is available for scanning when an unpacking stage is concluded. Unfortunately, in presence of a high number of spurious unpacking stages, invoking the detector at the end of each observed unpacking stage would become prohibitively expensive. Using the fact that the host system cannot be damaged without executing a dangerous system call, we can postpone malware detection until such a system call is invoked. The advantage of postponing detection is that the output of multiple unpacking stages is accumulated and analyzed only once. A pseudo-code version of OmniUnpack is given in Algorithm 1.

The algorithm tracks the memory pages that were written (the set W) and the memory pages that were written and executed (the set WX). Obviously, $WX \subseteq W$. When the program invokes a dangerous system call after executing code from a previously written memory page (line 11), the algorithm triggers malware detection on all the pages written (i.e., those recorded in the set W). Unless the process is found to be malicious and thus terminated (line 14), the system call is allowed and execution is resumed. After a scan, the sets W and WX are reset to ensure that the same pages will not be scanned again (unless modified once again before execution). At the end of an unpacking stage we do not scan only the executed-after-write pages, but all the written pages. Therefore, if unpacking is concluded, no further memory scans are required during the rest of the execution.

Algorithm 1: OmniUnpack

```
Input: An execution trace $\mathscr{T} = \langle e_0, e_1, \ldots \rangle$, where a trace
       event $e_i$ is either a write access $w(p)$ to a memory
       page $p$, an instruction execution $x(p)$ from a memory
       page $p$, or the system-call invocation $s$.
1  $W \leftarrow \emptyset$;                  /* written pages */
2  $WX \leftarrow \emptyset$; /* pages written then executed */
3  foreach $e \in \mathscr{T}$ do
4      switch the value of $e$ do
5          case write access $w(p)$
6              $W \leftarrow W \cup \{p\}$
7          case instruction execution $x(p)$
8              if $p \in W$ then
9                  $WX \leftarrow WX \cup \{p\}$
10         case system-call invocation $s$
11             if $s \in \{$dangerous system calls$\} \wedge WX \neq \emptyset$ then
12                 foreach $p \in W$ do
13                     $r \leftarrow$ Scan($p$)
14                     if $r =$ MALICIOUS then
15                         halt execution
16                         return
17             invoke the system-call handler for $s$
18             $W \leftarrow \emptyset$
19             $WX \leftarrow \emptyset$
```

We note that OmniUnpack does not need to observe all memory-page accesses. It is sufficient to observe the first memory access in an uninterrupted sequence of accesses of the same type. For example, only the first write to a page is useful, subsequent writes to the same page do not impact the result of the algorithm and can be ignored. In other words, the information we need is the first execution of a page after write accesses to the same page. The implementation makes use of this observation to attain near-native performance, with minimal overhead.

4. Implementation

We have implemented OmniUnpack as a kernel driver for Microsoft Windows XP executing on a Intel IA-32 processor, where we simulate non-executable pages in software. This setting allows us to measure the performance and efficacy of unpacking in a real-world scenario, i.e., on unmodified hardware. Our implementation is derived from OllyBone [17], a plugin for the well known debugger OllyDbg, which allows for page-level break-on-execute by adopting an approach similar to that used by PaX PAGEEXEC [13]. The user-space

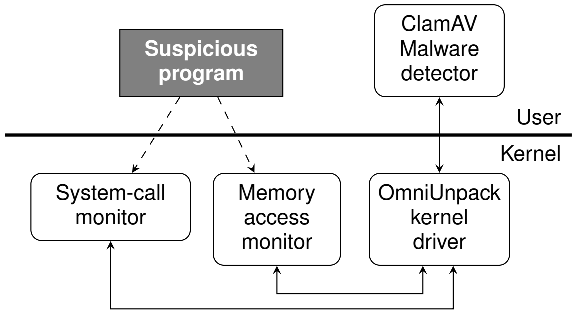

Figure 3: Architecture of the OmniUnpack-based malware-detection system.

malware detector is based on the ClamAV open source anti-virus [4].

4.1. Architecture Overview

Our system is composed of a kernel driver and a user-space component (Figure 3). The kernel component is responsible for (I) tracking memory accesses to detect the end of an unpacking stage (Section 4.2), (II) detecting when a dangerous interaction between the suspicious program and the system occurs (Section 4.3), and (III) triggering malware detection. The user-space component consists of a malware detector responsible for on-demand scanning of memory of the suspicious program (Section 4.4).

The kernel component keeps track of the memory locations that can potentially contain newly generated malicious code. Before allowing any dangerous interactions with the system, it notifies the user-space malware detector to analyze these locations. If the malware detector does not find any malware, the execution is resumed, otherwise the program is terminated.

4.2. Monitoring Memory Accesses

Coarse-grained memory-access tracking is implemented by leveraging the memory protection mechanism offered by the hardware. Attempts to execute the code generated during unpacking are detected by enforcing a *write-xor-execute* policy ($W \oplus X$) on the memory pages of the suspicious program (i.e., a memory page can be either writable or executable).

When an unpacking routine writes the unpacked code to memory, the destination page is marked as writable but not executable. At the end of the unpacking stage, when the program accesses the same page for execution, the lack of execution permission causes a protection exception that is reported to the operating system (OS). We intercept and process such exceptions. Further accesses for execution will not cause any protection exception because the execution permission has been granted since the first access for execution. A subsequent attempt to modify the code stored in the executed page will not cause a new protection exception unless the write permissions have been removed again at the end of the unpacking stage.

The $W \oplus X$ policy is only temporarily applied on the original page permissions to allow for the interception of write and execution attempts. The real permissions are recorded and restored, when necessary, to guarantee the correct behavior of the monitored program. During the execution of the program when a protection exception is raised, we check the original page permissions to detect whether the protection exception is real (i.e., the access to the page violates the original permissions) or it is caused by the $W \oplus X$ policy. In the first case we restore the original page permissions and we invoke the OS page fault handler. In the second case we update the page permissions to authorize the new type of access and to prevent the complementary type of access. Note that, once a page has been marked as written-then-executed, there is no need anymore to enforce the $W \oplus X$ policy on the page until the end of the current unpacking stage.

Many hardware architectures (e.g., Intel IA-64, Sun Sparc, Alpha) offer facilities to enforce the $W \oplus X$ policy through support for read, write, and execute permissions at the page level. Unfortunately, the architecture targeted by the vast majority of malicious programs (Intel IA-32) lacks such facilities. On this architecture non-executable pages can be simulated via a software mechanism with only a minimal overhead, as demonstrated by the PaX PAGEEXEC project [13]. We employ a mechanism similar to PaX PAGEEXEC for the purpose of tracking memory write and execute accesses. The idea is to simulate non-executable pages by marking them as not accessible from user space, while granting only temporarily read/write access permissions by exploiting the presence of two separate caches on the CPU for the virtual-to-physical address translation (one for data and the other for instructions).

When the program is loaded in memory and when a new page is allocated in memory (e.g., as a result of a page miss or of dynamic memory allocation), we record the original page permissions and we remove the write permission. We handle explicit requests from the program to update page permissions similarly. Algorithm 2 is responsible for enforcing the $W \oplus X$ policy and for passing any errors to the OS page fault handler. The algorithm is invoked every time a memory protection error is raised by the CPU during the execution of the suspicious program. If the access type complies with the original page permissions, the algorithm updates the permissions according to the access type. Otherwise, the algorithm restores the original permissions and invokes the OS page fault handler to gracefully handle the

```
Algorithm 2: Track page accesses
Input: The address $a$ of the page fault and the type
       $t \in \{READ, WRITE, EXECUTE\}$ of the attempted
       access that caused the page fault.
$p \leftarrow$ Page($a$)
if $t$ authorized by the original permissions of $p$ then
    if $t = EXECUTE$ then
        remove write permission from $p$
        grant execution permission to $p$
        emit event $x(p)$
    else if $t = WRITE$ then
        remove execution permission from $p$
        grant write permission to $p$
        emit event $w(p)$
else
    restore the original permissions of $p$
    invoke the OS page fault handler
```

error. If the error is due to the $W \oplus X$ policy, the algorithm generates the appropriate event for processing by the OmniUnpack-based malware detector.

Because the granularity of our memory-access monitoring is at page level, on the first faulty access the required page permissions are granted such that further accesses of the same type do not cause any fault. This makes our system significantly faster than techniques that track memory accesses at instruction granularity. A downside of our monitoring scheme is that it is not possible to tell which addresses are really accessed on the page. We have to assume that memory accesses potentially affect all addresses on a page. This assumption could lead to the identification of spurious unpacking stages if a page contains non-overlapping writable data and code. Nonetheless, our experience shows that OmniUnpack is overall significantly faster than existing fine-grained unpackers (see the experimental evaluation of Section 5).

4.3. Monitoring System Calls

We intercept the system calls of interest (the ones considered dangerous) through interposition on the system-call dispatch table. Although by enforcing stricter permissions on memory pages we could determine the program point where this invocation originated, we do not distinguish between dangerous system calls invoked from existing program code and those invoked from newly generated code. If newly generated code is present in the program memory, the code is executed, and the execution is followed by a dangerous system call, we consider the unpacking stage concluded and we trigger malware detection. The system call is consequently allowed only if no malicious code is detected. If, instead, a dangerous system call is invoked before a previously-written page is executed, memory scanning will not be triggered because no execution of previously generated code has been observed so far. To guarantee the ability to intercept further unpacking stages, at the end of scanning, write permissions are revoked from all scanned pages.

4.4. User-Space Malware Detector

The system does not depend on a particular detection technique. The only desirable quality is performance, because scanning is performed during the execution of the program. OmniUnpack only provides a detector with data to analyze and then authorizes, or denies, the execution of such data according to the response of the detector. If the detector adopted fails to identify the presence of malicious code in memory, our system will resume the execution because the response of the detector is assumed to be correct.

The advantage of running the malware detector in user-space is that a bug in the malware detector will not compromise the stability of the whole system. This setup is compatible with current commercial virus scanners, which could be thus used in combination with OmniUnpack without significant reengineering. Because it is run as a separate process, the malware detector does not have direct access to the memory of the suspicious program. To avoid the overhead of copying the data to scan from the suspicious program to the malware detector, we share the memory containing such data between the two processes. Before invoking the user-space detector, the kernel component maps in the virtual address space of the malware detector process the pages for which the analysis is required and communicates to the detector the address and the amount of data to analyze. The malware detector then scans the memory region part of its address space.

The detection is triggered during the execution to scan only the pages that potentially contain the newly generated malicious code. Therefore, pages executed without prior modification do not trigger detection. Such pages are analyzed after the program is loaded in memory, but before the execution is started. The main advantage of such approach is that, if the program is not packed or self-modifying (i.e., most benign programs), its execution is not slowed down beyond an initial one-time memory scan. Thus, the performance of the OmniUnpack-based malware detector is comparable to that of current virus detectors that scan the program at the beginning of execution.

4.5. Requirements for Signatures

We identify three requirements that the signatures used by the malware detector have to satisfy to guarantee accurate detection.

First, the signature needs to characterize the real malicious payload, not the packed code or the unpacking routine. An attacker can easily modify the malware by applying a new layer of packing, by substituting the unpacking routine with an equivalent one (e.g., adopting polymorphism), or by modifying the packed payload (e.g., encrypting it with a different key). To effectively identify all the new instances of the malware generated using such techniques, the signature must capture the original malicious payload, which remains unchanged during all these obfuscations. As the malware detector is invoked to scan both the code as it is loaded in memory and the code generated at run-time, the OmniUnpack-based malware detector guarantees that the original malicious payload will be scanned.

Second, the signature must characterize the fragments of the original code of the malware that are available at the time a dangerous system call is made. Standard malware detectors identify malicious programs off-line, before the execution. Our system instead performs real-time detection and therefore it is essential that we detect that a program is malicious before it causes any damage to the system. To guarantee that the malware is blocked in time, the signature should describe code that is present in memory when a dangerous system call is about to damage the system.

Third, the signature should characterize a fragment of the malware payload that is not modified between the time it executes and the time the malware detector scans it. Because we delay scanning until a dangerous system call is invoked, unpacked code (part of the payload) could be executed up to that invocation. The unpacked code can erase itself right before the system-call invocation, thus evading detection. To improve the effectiveness of the detector and to avoid any evasion attempts, the signature should describe code that is necessarily present when the system call is made.

We suspect that some of the signatures used by malware detectors do not satisfy the requirements of our system. For example, most ClamAV [4] signatures describe the data representing the packed code instead of the original malicious payload. For our evaluation we developed a custom database of appropriate signatures. We could not analyze the signatures used by the commercial malware detectors as their format is not known. If those signatures do not satisfy the requirements of our system they have to be replaced with others that satisfy our requirements.

5. Experimental Evaluation

To evaluate the effectiveness and the performance of OmniUnpack we performed three different experiments. To assess performance, we compared the unpacking time for OmniUnpack, for PolyUnpack [15], and for the ClamAV unpacker [4]. To assess effectiveness, we measured the detection rate of standard ClamAV against the detection rate of the OmniUnpack-based ClamAV. To determine whether OmniUnpack is practical for continuous, real-time use, we measured its overhead on benign programs.

The authors of PolyUnpack could not provide us with a command-line implementation of their tool because of intellectual-property reasons. The PolyUnpack version that is available online (Stand-Alone Version 0.2 from http://polyunpack.cc.gt.atl.ga.us/polyunpack.zip) is prohibitively slow because of its continuous GUI updates (for some packers, it took several *hours* to unpack the program). We implemented a simplified version of PolyUnpack, with the goal of making it as fast as possible. We used the Microsoft Windows Debugging API to single-step the program execution automatically. No other analysis or computation was done by our PolyUnpack implementation before or during unpacking.

ClamAV is an open-source virus scanner that includes a set of specialized unpacking algorithms. At the time of writing this paper, it handled the following packers: UPX (all versions), FSG versions 1.3, 1.31, 1.33, and 2.0, Petite 2.x, NsPack, wwpack32 1.20, MEW, Upack, SUE, and Y0da Cryptor 1.3. We isolated the unpacking components of ClamAV 0.90.2 and used them in our tests.

Our test platform was an Intel Pentium4 3GHz with 2GB of RAM running GNU/Linux 2.6.9. All experiments were performed in a Qemu virtual machine with no KQemu acceleration [2]. To accurately model the IA-32 behavior, our custom version of Qemu implements separated TLBs for data and instructions. The guest OS was a standard installation of Microsoft Windows XP SP2 equipped with OmniUnpack.

Our results can be summarized as follows:

- OmniUnpack was significantly faster than PolyUnpack, performing unpacking in a few seconds (within one order of magnitude of the fastest unpacker, ClamAV). This effectively reduced unpacking time from more then 100 seconds to 5 seconds.
- OmniUnpack handled 80% of packed malware, while ClamAV handled only about 15%. The unhandled cases were due to current implementation limitations of OmniUnpack.
- The overhead for benign programs was relatively small: 6% for non packed programs and 11% for packed programs.

5.1. Time to Unpack

We evaluated the performance of OmniUnpack with respect to two current techniques for malware unpacking, execution monitoring through debugging and algo-

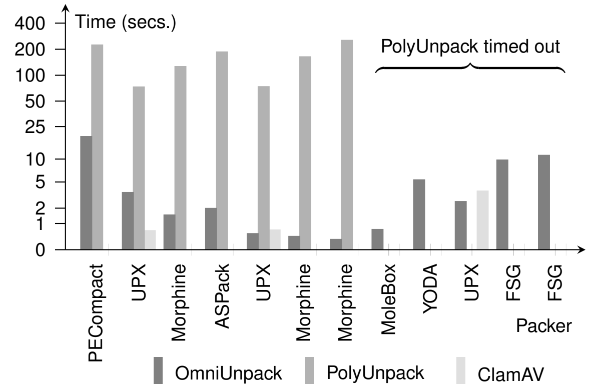

Figure 4: Comparison of the performance of OmniUnpack with respect to PolyUnpack and ClamAV. Note that the Y axis shows time in seconds on a logarithmic scale.

rithmic unpacking. We chose PolyUnpack [15] as a representative implementation of unpacking based on execution monitoring through debugging, and ClamAV [4] for algorithmic unpacking.

The evaluation proceeded as follows: for each packed malware sample in our test set, we executed it under OmniUnpack and recorded the program point at the end of each unpacking stage and the time to reach this program point. Each malware under OmniUnpack execution was bounded by a 90-second timeout. We then executed the same malware sample under our implementation of PolyUnpack and recorded the time it took PolyUnpack to reach the same program point, with a timeout of 300 seconds. We also measured the time required by the algorithmic unpackers of ClamAV. All times reported here are wall-clock (total) time.

The test set was composed by malicious programs collected from the *Offensive Computing* website [1] and contained programs packed with standard packers, programs packed with customized versions of the publicly available packers, and programs packed with unknown packers (i.e., that no tool could identify).

The experimental evaluation shows that OmniUnpack is able to correctly monitor all the malicious programs and to detect the end of each unpacking stage, independent of the packing algorithm. Furthermore, the average number of unpacking stages per malware detected was 7.14, while the average number of unpacking stages detected using naïve coarse-grained memory-access tracking was 2089.70 (cf. Figure 2).

The results of the comparison of OmniUnpack, PolyUnpack, and the ClamAV unpacker are reported in Figure 4. For the ease of presentation we report only the time requested to reach the end of the first unpacking stage. Consider the chart entry for UPX (the second entry from the left). The left bar shows the time it took OmniUnpack to reach the end of the first unpacking stage, the middle bar shows the time for PolyUnpack to reach the same point, and the right bar indicates the total time ClamAV took to unpack the program. In this test, OmniUnpack was 20.56 times faster than PolyUnpack, and only 5 times slower than ClamAV. These numbers reflect the overall trend observed in our experiments, where OmniUnpack significantly outperforms PolyUnpack and is at most one order of magnitude slower than ClamAV.

In several cases PolyUnpack was not able to complete unpacking before the timeout expired. Similarly, ClamAV failed to unpack some of the test samples. OmniUnpack handled all samples efficiently, with times close to optimal (i.e., to ClamAV). There were few cases when ClamAV appeared to be slower than OmniUnpack (e.g., for packer UPX in Figure 4). This is due to our result-reporting methodology, which shows the OmniUnpack and PolyUnpack times for unpacking one stage and the ClamAV time for unpacking all stages. For example, in the case of UPX, OmniUnpack took longer than ClamAV to unpack all stages, although in the first stage it was quicker than ClamAV.

5.2. Ability to Handle Unknown Packing Algorithms

The benefit of a general unpacking technique such as OmniUnpack is that it supports binaries packed with any arbitrary algorithms applied any number of times. This is in contrast with specialized unpacking algorithms, which can only manipulate binaries packed with particular versions of particular packers. We compared the detection rate of ClamAV (used in its full capacity of malware detector) with that of ClamAV enhanced with OmniUnpack. The OmniUnpack-enhanced ClamAV applies the OmniUnpack algorithm to detect newly generated code and then invokes ClamAV on the memory regions containing the new code.

The test set was composed by a toy application that we packed with 20 different packers we found on the web. The signature we used for the detection was manually generated and described the body of the `main` function of our toy application, before packing. The same signature was used for both ClamAV and OmniUnpack-enhanced ClamAV.

The results of the comparison are reported in Table 3. ClamAV unpacked 5 of the packed instances and only 3 of these matched the signature. We believe that the unpacking algorithms contained a bug that prevented the correct recovery of the original payload. On the other hand, OmniUnpack successfully unpacked and matched the original payload of 16 of the 20 packed instances. The instances packed with Armadillo and CExe were not correctly identified because both pack-

Packer	ClamAV		OmniUnpack
	Unpacked	Detected	Detected
ACprotect	-	-	✓
Armadillo	-	-	IMPL
ASpack	-	-	✓
ASprotect	-	-	✓
CExe	-	-	IMPL
ExeStealth	-	-	✓
FSG	✓	✓	✓
MEW	✓	✓	✓
MoleBox	-	-	✓
Morphine	-	-	✓
nPack	-	-	N/A
nSPack	✓	-	✓
PKLite	-	-	✓
RLPpack	-	-	✓
teLock	-	-	IMPL
Themida	-	-	✓
UPX	✓	-	✓
WinUpackE	✓	✓	✓
Xcompw	-	-	✓
Xpackw	-	-	✓
Rate	25%	15%	80%

Table 3: Comparison of the detection rate of ClamAV and OmniUnpack-enhanced ClamAV. IMPL denotes a limitation of the current OmniUnpack implementation and N/A denotes programs that did not execute correctly after packing, with or without OmniUnpack.

ers unpack the original payload, or an intermediate executable, on disk and then execute this executable. Our prototype did not follow process creation. nPack generated an invalid executable which crashed with or without OmniUnpack and was thus not detected. The instance packed with teLock was not identified because we ran the experiment within a Qemu virtual machine and probably the packer used an instruction not correctly implemented in the virtual machine and the program failed to unpack. We believe this problem would not manifest itself with real hardware.

5.3. Overhead for Benign Programs

Another performance concern is related to the overhead OmniUnpack introduces during the execution of benign programs. Because OmniUnpack changes the malware-detection approach from one-time to continuous scanning, it is possible that benign programs incur overheads when monitored by OmniUnpack. We compared the execution times of benign programs with OmniUnpack disabled and enabled.

The test set was composed of several common command-line applications (agrep, bison, bzip2, cksum, compress, egrep, tar, uniq, wc, and wget) and by their packed copy (packed with UPX). The test set also included an application (tester), as well as its packed copy, that performs different types of memory accesses repeatedly (e.g., page read, page write, and then page execution).

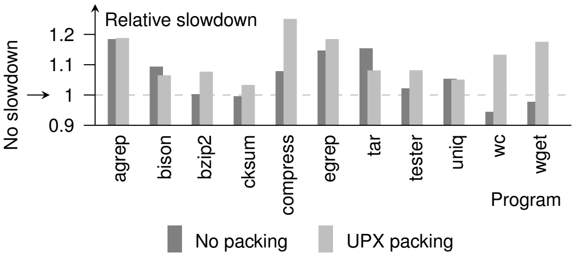

Figure 5: Relative slowdown introduced by OmniUnpack.

The average overhead of OmniUnpack (computed on three runs with different arguments) is reported in Figure 5. For the ease of presentation we show only the total overhead introduced by our system. The total overhead is the sum of the user time, the system time, the time to scan the program when first loaded in memory, and the time requested to scan the written memory pages at the end of an unpacking stage. For non-packed applications the average overhead introduced by OmniUnpack was about 6%, while for packed applications it was about 11%. In two cases, wc and wget, the execution time with OmniUnpack was smaller then that measured with OmniUnpack disabled. During the initial setup of OmniUnpack all memory pages of the programs are accessed to load them in memory. That avoided most of the page faults due to non-present pages during the execution and resulted in faster execution with OmniUnpack.

6. Related Work

Packed malware has been a known problem for a long time. The first polymorphic virus that encrypted its payload appeared in 1989 and tools to handle such malware followed quickly. Ször presents a comprehensive survey of unpacking techniques [18]. He covers most algorithmic unpacking approaches, including heuristics such as *static decryptor detection* and *x-raying*. Stepan discussed specialized unpackers and their limitations, in particular the fact these specialized routines need to be updated " 'after the fact' to accomodate new packer versions" [16].

Generic unpacking is the solution to the threat of diverse packing techniques. Emulation and sandboxing technologies, which integrated emulation into the malware detector, have been proposed for the detection of polymorphic malware [7, 8]. This approach is open to resource-consumption attacks and is prone to false negatives, since the execution time in the sandbox has to be restricted for performance reasons [9, 10]. In contrast, OmniUnpack imposes little overhead and as a result can monitor the whole program execution without affecting the user experience. A faster approach to generic unpacking uses dynamic code generation to speed up emulation [16], but it is still too slow for interactive use. A further improvement is introduced by PolyUnpack, which executes the program inside a debugger until it reaches an instruction sequence that does not appear in the static disassembly of the program [15]. PolyUnpack suffers from limitations inherent in static disassembly, limitations that we avoid by using a purely dynamic technique. Additionally, the use of a debugger leads to significant overhead that relegates this tool to the forensic field.

OmniUnpack avoids both the pitfalls of algorithmic unpacking (too specific) and of emulation- and debugging-based unpacking (too time intensive). Our unpacking technique builds on common hardware features that allow us to monitor the status of memory pages (e.g., written vs. executed), making it fast, generic, and widely applicable. Similar uses of hardware features for security purposes have been proposed by PaX PAGEEXEC [13] (to prevent some classes of memory error exploits) and Ray *et al.* [14] (to integrate virus scanning with the virtual memory manager). We enhanced these mechanisms to create OmniUnpack as a fast malware defense that is resilient to packing, self-protection, and self-modification.

7. Conclusion

The current approaches to packed malicious programs have several limitations that make analysis and accurate detection both complex and time consuming. Dynamic approaches based on debugging and emulation are slow and can be easily detected. Static approaches are fast but require a priori knowledge of the packing algorithm. In this paper we have presented a new unpacking technique, called OmniUnpack, that addresses the aforementioned shortcomings. Our technique is implemented directly in the operating system and constantly monitors the execution of a suspicious program to detect the end of the unpacking process and to notify a malware detector about the presence of new unpacked code in memory. Our experimental evaluation has shown that our technique is able to deal with known and unknown unpacking algorithms and that it is drastically faster than the dynamic approaches currently available and at most an order of magnitude slower than static approaches. Moreover, our technique introduces little overhead, making OmniUnpack amenable to continuous monitoring of any program on production systems.

Future work includes the improvement of our prototype implementation (e.g., support for monitoring multiple processes) and the automatic generation of signatures that satisfy the requirements imposed by our system.

References

[1] Offensive Computing. http://www.offensivecomputing.net/.

[2] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. http://fabrice.bellard.free.fr/qemu/.

[3] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith. Malware normalization. Technical Report 1539, University of Wisconsin, Madison, WI, USA, Nov. 2005.

[4] T. Kojm. Clam AntiVirus. http://www.clamav.net.

[5] Y. Mashevsky. Watershed in malicious code evolution. Published online at http://www.viruslist.com/en/analysis?pubid=167798878 (last accessed on May 15, 2007), July 29, 2005.

[6] McAfee Avert Labs. McAfee Avert Labs unveils predictions for top ten security threats in 2007 as hacking comes of age. Press release published online at http://www.mcafee.com/us/about/press/corporate/2006/20061129_080000_f.html (last accessed on May 15, 2007), Nov. 29, 2006.

[7] C. Nachenberg. Polymorphic virus detection module. *United States Patent # 5,696,822*, Dec. 1997.

[8] C. Nachenberg. Polymorphic virus detection module. *United States Patent # 5,826,013*, Oct. 1998.

[9] K. Natvig. Sandbox technology inside AV scanners. In *Proceedings of the 2001 Virus Bulletin Conference*, pages 475–487. Virus Bulletin, Sept. 2001.

[10] K. Natvig. Sandbox II: Internet. In *Proceedings of the 2002 Virus Bulletin Conference*, pages 1–18. Virus Bulletin, 2002.

[11] M. F. Oberhumer and L. Molnár. The Ultimate Packer for eXecutables (UPX). Published online at http://upx.sourceforge.net/. Last accessed on 14 Jan. 2007.

[12] Panda Research. Mal(ware)formation statistics. Published online at http://research.pandasoftware.com/blogs/research/archive/2007/05/28/Mal_2800_ware_2900_formation-statistics.aspx (last accessed on May 29, 2007).

[13] PaX Team. PaX PAGEEXEC: paging based non-executable pages. http://pax.grsecurity.net/docs/pageexec.txt.

[14] K. D. Ray, M. Kramer, P. England, and S. A. Field. On-access Scan of Memory for Malware. *United States Patent # 2006/0200863 A1*, Sept. 2006.

[15] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of 2006 Annual Computer Security Applications Conference (ACSAC)*, pages 289–300, Washington, DC, USA, 2006. IEEE Computer Society.

[16] A. Stepan. Improving proactive detection of packed malware. *Virus Bulletin*, pages 11–13, Mar. 2006.

[17] J. Stewart. OllyBonE. http://www.joestewart.org/ollybone/.

[18] P. Szor. *The Art of Computer Virus Research and Defense*, chapter 11, pages 425–494. Addison-Wesley, 2005.