Denial-of-Service Attacks on Host-Based Generic Unpackers*

Limin Liu[1], Jiang Ming[2], Zhi Wang[2,3], Debin Gao[2], and Chunfu Jia[3]

[1] State Key Lab of Information Security, Graduate University of CAS, China
lmliu@is.ac.cn
[2] School of Information Systems, Singapore Management University, Singapore
{jiangming, zwang, dbgao}@smu.edu.sg
[3] College of Information Technology and Science, Nankai University, China
cfjia@nankai.edu.cn

Abstract. With the advance of packing techniques, a few generic and automatic unpackers have been proposed. These unpackers are designed to automatically unpack packed binaries without specific knowledge of the packing techniques used. In this paper, we present an automatic packer with which packed malware forges spurious unpacking behaviors that lead to a denial-of-service attack on host-based generic unpackers. We present the design, implementation, and evaluation of the proposed packer and malware produced using the proposed packer, and show the success of denial-of-service attacks on host-based generic unpackers.
Keywords: generic unpacker, denial-of-service attack, spurious unpacking behavior

1 Introduction

Most malware is packed in order to evade malicious content detection. It has been reported that among 20,000 malware samples collected in April 2008, more than 80% were packed by packers from 150 different families [10]. This is further complicated by the ease of obtaining and modifying the source code of various packers. Currently, new packers are created from existing ones at a rate of 10 to 15 per month [17].

A few generic and automatic unpacking techniques have been proposed to unpack packed binaries without specific knowledge of the packing technique used, e.g., OmniUnpack [9], Justin [5], Renovo [8], PolyUnpack [14] and others[1]. A

* This research was mostly done when the first three authors, Limin Liu, Jiang Ming, and Zhi Wang, were researchers working in Singapore Management University. It was partially supported by National Science Foundation (NSF) China under the agreements 90718005, 70890084/G021102, and 60573015.

[1] For example, OllyBonE from National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov; Deroko, http://deroko.phearless.org/rce.html; RL!Depacker, http://ap0x.jezgra.net/; QuickUnpack, http://qunpack.ahteam.org/; and Universal PE Unpacker, http://www.hex-rays.com/idapro/unpack_pe/unpacking.pdf.

difficulty that most of these generic unpackers face is to determine the end of the unpacking routine in the execution of the packed malware, which has been proved to be an undecidable problem [14]. Intuitively, the malware might be packed with multiple layers of packers, which makes it difficult to differentiate the end of the unpacking and the start of the execution of the original malware. Many generic unpackers adopt various heuristics to handle this difficulty. However, they may be wrong in determining the end of unpacking [9].

Using the heuristics does not necessarily expose the unpacker and its operating machine to attacks like denial-of-service attacks we propose in this paper, especially if the unpacker runs on a controlled and protected environment, e.g., emulators or virtual machines. However, when the unpacker is running on a real host machine that has to be protected by, e.g., anti-virus software, a denial-of-service attack becomes possible.

Some of the latest and most sophisticated unpackers operate on real host machines instead of emulators or virtual machines to improve efficiency [5, 9]. Since they do not run under a controlled and protected environment, anti-virus software is invoked at *all* potential end-of-unpacking points to protect the host machine and to detect the unpacked malware. For example, OmniUnpack [9] invokes anti-virus software to scan the malware whenever a page is written and then executed, and the execution involves a dangerous system call made.

Our proposed denial-of-service attack exploits the requirement of host-based generic unpackers to invoke anti-virus software at all potential end-of-unpacking points during the execution of the packed malware, and the fact that the anti-virus scanning time dominates the overhead of the unpacker (see Section 2). The idea is for the DoS-packed malware to forge a large number of spurious unpacking routines that lead to a large number of invocations of the anti-virus software, which costs lost of resources on the unpacker including CPU cycles, memory, and time.

The automatic packer we propose in this paper attaches the DoS attack code into the executable of malware. The resulting packed malware has the following properties.

- benign: the DoS attack code we add to the malware (not including the original malware) does no harm to the operating system, which makes the packed malware and its partially packed version evade the anti-virus scanning.
- environment-aware: the DoS attack only targets the unpacker and is not executed in normal machines (victims of the original malware).
- semantic-preserving: our packer does not change the semantics of the original malware.
- light-weighted: the overhead when executing on normal machines is small.

In this paper, we present a security vulnerability of host-based generic unpackers, and propose an automatic packer with which packed malware forges spurious unpacking behaviors that lead to a denial-of-service attack on host-based generic unpackers. We implement this packer and try it out on a number of malware. Experiments show that the packed malware successfully differenti-

ates the unpacker and normal environments, and launches DoS attacks on the target unpacker.

In the rest of this paper, we first describe the two types of unpackers available and an analysis of the host-based unpackers (see Section 2). In Section 3, we detail the design of our denial-of-service attack and the implementation our automatic packer that realizes the attack. Section 4 presents the evaluation of the proposed attack and its implementation.

2 Generic unpackers and their heuristics

When a packed malware is executed, it first dynamically unpacks its malware payload and then executes the malware. Generic unpackers try to extract the malware payload after detecting the end of unpacking. However, determining the end of unpacking is proved to be an undecidable problem [14]. Therefore, generic unpackers either 1) extract the malware payload after it is executed, or 2) have to introduce various heuristics to approximate the end of unpacking. The approach adopted by an unpacker depends on its executing environment. Some unpackers execute on emulators or virtual machines, while others execute directly on host machines for better efficiency.

2.1 Unpackers on controlled environments

A controlled environment may use emulators (e.g., Qemu, Bochs) or virtual machines (e.g., VMware, VirtualPC). Unpackers on controlled environments first dynamically execute the packed code for a period of time that is sufficient for completing the unpacking, and then extract the real payload. It is usually tricky to choose the time period which is sufficient to extract the actual payload. If the malware payload gets executed within this time period, the executing environment of the unpacker runs into an unsafe mode. However, this is not an issue as the controlled environment guarantees the safety of the host machine. Even if the emulated or virtual machine is infected with viruses, it can be restored by, e.g., previous and clean snapshots.

Examples in this category include PolyUnpack [14], which compares the executing instructions with static disassembly code by monitoring the program's execution. Malware Normalizer [2], Renovo [8], Azure [13], Saffron [11], Paradyn[2], and Pandora [1] unpack the program based on the fact that the packed code has to write the instruction in memory and then execute these instructions. Renovo and Pandora use emulators to monitor the memory access, Saffron and Paradyn use dynamic instrumentation, and Azure uses Intel's VT extension. Eureka [15] monitors two system calls, `NtTerminateProcess` and `NtCreateProcess`, to find the malicious payload.

Using emulators and virtual machines have two challenges in general. One is that running a program in emulators or virtual machines is much slower (up

[2] Paradyn, `http://pages.cs.wisc.edu/~paradyn/`.

to several hundreds of times [17]) than running it on a host machine. Another is due to the various anti-emulation and anti-vm techniques [3, 4, 12] that may terminate the program's execution once the existence of a controlled environment is detected.

2.2 Host-based unpackers

Running the unpacker on a host machine improves the performance but might leave the system in an unsafe state. Therefore, it is important for unpackers to terminate the execution once the unpacking routine is completed and before the malicious payload gets executed. Heuristics are introduced to approximate the end of unpacking. Fig. 1 gives an overview of host-based generic unpackers.

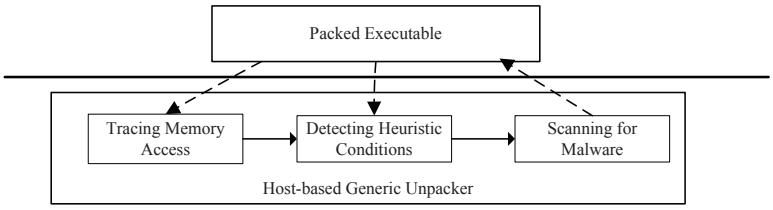


Fig. 1. Host-based unpackers

In general, the unpacker monitors memory accesses of the packed code and detects satisfying conditions of the heuristics. Whenever the conditions are met, the unpacker invokes anti-virus software to scan the (partially) unpacked code to decide the end of unpacking. The calling of the anti-virus software is important as the unpacker needs to protect its operating environment not to be infected by the malware.

Both OmniUnpack [9] and Justin [5] monitor events when a memory page is written and subsequently executed, although they use different heuristics. OmniUnpack uses a heuristic that when the malware gets executed, it needs to interact with the operating system using dangerous system calls. Justin incorporates other heuristics including unpacker memory avoidance, stack pointer checking, and command-line argument checking. In order to improve performance, both of them monitor memory accesses at the page level.

Note that these heuristics adopted by host-based unpackers might not be always accurate. For example, an invocation of a dangerous system call is not a sufficient condition to indicate end of the unpacking or execution of the malware.

2.3 Overhead of host-based unpackers

We used our implementation of a previously proposed unpacker (see Section 4) to unpack a number of packed malware (Amanda, BirdWatcher, Aimbot, Arus, BlackEyes, Adroar, Aidid and DenisBee) and analyzed the breakdown of the

time spent. Fig. 2 shows the average time spent in each stage of the unpacking. It shows that the time taken by the anti-virus software dominates the overhead of the unpacker. This result is consistent with another report on the time spent by anti-virus software [6], in which it was reported that scanning a binary or system file of 1 Mbytes takes at least 0.02 seconds for more than 40 AV-scanners (the average size of the malware we tested is 61 Kbytes, which corresponds to 0.01953 second per Mbytes).

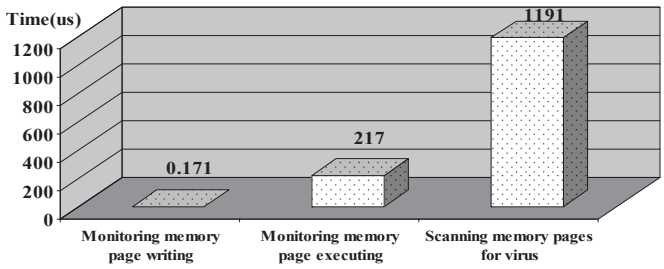


Fig. 2. Overhead of host-based unpackers

Inspired by this analysis, we designed packed malware that results in a large number of invocations of the anti-virus software in order to make effective denial-of-service attacks on host-based unpackers.

3 Denial-of-service attack and its implementation

Fig. 3 shows an overview of our denial-of-service attack on a particular host-based generic unpacker, OmniUnpack. The packed malware first detects the existence of the unpacker in the executing environment (see Section 3.1), then forges (a loop of) spurious behavior if an unpacker is detected or executes the original malware otherwise. Each iteration of the spurious behavior tries to satisfy the heuristic condition of the unpacker, which, for example, includes writing to a memory page, executing the memory page, and making a dangerous system call.

The spurious behavior satisfying the heuristic conditions of the unpacker will result in an invocation of the anti-virus software to scan for viruses. T_W, T_E, and T_S represent the overhead introduced by monitoring `write` behavior, `execute` behavior, and scanning for malware by the anti-virus software, respectively.

Designing this denial-of-service attack in a packed malware with all the properties shown in Section 1 has a few challenges. First, the packed malware needs to be able to detect the existence of an unpacker. This ensures that the original malware will be executed when the packed code is run on a normal machine (the victim of the original malware). Second, the code that facilitates the denial-of-service attack, including the forged spurious behavior, should not be flagged as a virus by the anti-virus software. If it is, it will be detected the first time the

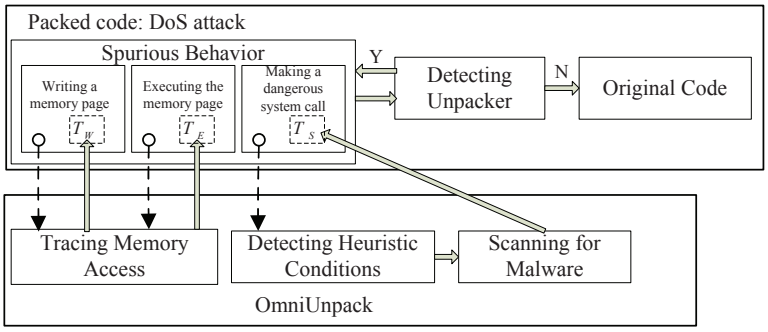


Fig. 3. Overview of the DoS attack on OmniUnpack

anti-virus software is triggered, which in term will result in the success of the unpacker in detecting the malware, and the denial-of-service attack will fail. On the other hand, as long as this code is benign, the anti-virus software will have to be invoked for every iteration of the spurious behaviors, which forms the base of our denial-of-service attack.

3.1 Detecting the existence of an unpacker

Generic unpackers introduce overhead in, e.g., monitoring memory access and invoking anti-virus software. To distinguish between executing environments with and without a generic unpacker, we monitor the time-stamp counter of the processor (TSC) to measure the number of CPU clock cycles in executing spurious unpacking routines. Figure 4 shows the details of such a measurement.

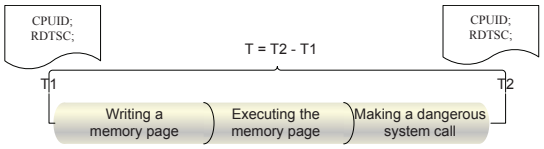


Fig. 4. Measuring CPU cycles to detect the existence of an unpacker

To implement this checking, we used the `RDTSC` instruction to measure the number of CPU cycles. We measure T (the difference between $T2$ and $T1$) many times (each of which consists of 5 iterations of the execution of a spurious unpacking routine to reduce the cache warm-up effect) to find its upper bound. By applying standard statistical techniques to our experiments, we arrived at a threshold value of $5,000$ clock cycles, i.e., if T exceeds $5,000$, it is most likely that the process is being executed with an unpacker. Please refer to Section 3.4 for some implementation details.

One possible way for the unpacker to confuse this measurement is to fake the `RDTSC` instruction by hooking the Interrupt Descriptor Table (IDT)[3]. However this needs a driver to be executed in supervisor mode (ring 0) and may affect the system stability. Dealing with this issue is beyond the scope of this paper.

3.2 Spurious unpacking behavior

As mentioned in Section 2.2, host-based generic unpackers adopt various heuristics to approximate the end of unpacking. These heuristics help identifying the potential points in the execution of the packed code that are likely to be the end of unpacking. However, these potential points in the execution of the packed code might not be the correct ones. The host-based generic unpackers understand the potential inaccuracy and invoke the anti-virus software at all these potential points of execution. Our denial-of-service attack forges a large number of the spurious unpacking behaviors so that the unpacker invokes the anti-virus software many times.

To make the discussion more concrete, here we take OmniUnpack as an example. As discussed in Section 2.2, OmniUnpack monitors events when a memory page is written and subsequently executed, and uses a heuristic that when the malware gets executed, it needs to interact with the operating system using dangerous system calls. Our denial-of-service attack first writes a memory page p with predefined instructions which do no meaningful operation (e.g., `nop`), and then executes instructions in p. After that, it invokes a dangerous system call. Note that as discussed at the beginning of Section 3, the forged spurious behavior should not be flagged as a virus by the anti-virus software. To satisfy this requirement, we choose to first create a file and then invoke the dangerous system call `NtDeleteFile` to delete it. We tested this with a number of anti-virus software and found that none of them flags such behavior as malicious.

3.3 Our automatic packer

To implement a packer that takes as input unpacked malware and outputs packed code that performs our denial-of-service attack on a host-based generic unpacker, we took a packer from aPLib[4] and modified its compressor engine. Our modified packer is not very different from the original one in their construction of packed code, i.e., both will first load the original malware binary into memory, compress the data from different sections (e.g., `.text`, `.data` and `.rsrc`), and obfuscate the import table to make reverse engineering difficult, and then add the code needed for unpacking (decompressing the packed code and rebuilding the import table). The difference is that our modified packer also inserts the DoS attack stub, which gets executed before the original binary is decompressed. Fig. 5 presents the structure of the packed code produced by our automatic packer.

[3] Fake RDTSC, ARTeam, `http://deroko.phearless.org/ring0.html`.
[4] aPLib compression library, `http://www.ibsensoftware.com/`.

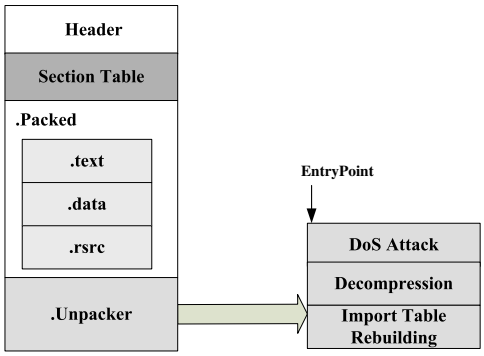


Fig. 5. Structure of the packed code

At load time the DoS attack stub is first executed to detect whether a host-based generic unpacker exists. If an unpacker does not exist, the code starts decompressing the packed malware and rebuilding the original imports using addresses obtained from APIs `LoadLibrary` and `GetProcAddress`. Control is then transferred to the entry point of the decompressed code, and the original malware binary gets executed. If an unpacker exists, the DoS attack code starts iterations of the spurious behavior.

3.4 Implementation details

In order to make our measurement of TSC more accurate, we took into consideration several issues. First, because Intel CPU supports out-of-order execution, we flush the instruction pipeline to prevent early termination of the measurement. The `CPUID` instruction is used for serializing execution [7]. We also subtract the overhead of instructions `RDTSC` and `CPUID` from our results (350 to 400 clock cycles). Second, cache effect might bias our results. It takes a large number of cycles to load data and code into the cache when execution starts. To reduce the cache warm-up effect, we repeat the measurement at least five times and discount the first measurement. Third, time-stamp counters on different cores may not be synchronized with each other. Instruction `SetThreadAffinityMask` is used to force execution on only one core.

Context switch and memory page swapping sometimes confuse our detection of the existence of an unpacker, because they introduce big overhead even if an unpacker does not exist. In order to reduce the false-positive rate of detecting an unpacker, we choose to detect the existence of an unpacker every time the spurious unpacking behavior is generated. The overhead introduced by this checking is very small compared to the overhead of the spurious behavior generated, but it lowers the false-positive rate significantly. In our experiments, our packed malware always manages to detect the existence of an unpacker correctly with at most two rounds of spurious behaviors.

4 Evaluation

In order to evaluate our proposed denial-of-service attack and the packer we implemented, we chose one of the latest and most sophisticated host-based generic unpackers, OmniUnpack [9], as the target. Since we do not have the OmniUnpack source code from the authors, we implemented it ourselves based on the descriptions from the paper and some help from the OmniUnpack authors on a computer with an Intel CPU of 3 GHz and 1 Gbytes of memory running Windows XP SP3.

There are three parts in our evaluation. First, we use anti-virus software to scan a few pieces of malware packed using existing packers and our packer. Second, we execute malware packed using our packer on normal computers without unpackers. Third, we execute malware packed using our packer on machines with our implementation of the OmniUnpack.

4.1 Anti-virus software detection

This part of the evaluation tries to see if the code added by our packer can escape detection by various anti-virus software. As described in Section 3.3, our packer is implemented by adding our denial-of-service attack code to an existing packer from aPLib. We first measure the size of the packed code with the original packer from aPLib and the one with our packer. Results are shown in Table 1. We find that our packer adds roughly 0.843% overhead compared to packed code with the original packer.

	Amanda	BirdWatcher	Aimbot	Arus	BlackEyes	Adroar	Aidid	DenisBee
Original packer	40,960	77,824	20,480	186,368	20,480	77,824	20,480	41,472
Our packer	41,472	78,336	20,992	186,880	20,992	78,336	20,992	41,984

Table 1. Size of the packed code (bytes)

In order to find whether the original packer from aPLib or our denial-of-service attack code contributes to anti-virus detection, we use anti-virus software to scan both the malware packed using the original packer from aPLib and packed using our packer. Table 2 shows the result.

Columns and rows in Table 2 show different malware samples and anti-virus software we have tested, respectively. Results show that the denial-of-service attack code introduced by our packer does not contribute to the detection by anti-virus software, because in all the eight cases where malware packed with our packer is detected, the same malware packed using the original packer is also detected. Also note that there are a few cases where malware packed using the original packer is detected, while the one packed with ours is not.

	Amanda	BirdWatcher	Aimbot	Arus	BlackEyes	Adroar	Aidid	DenisBee
Kaspersky								
ClamAV								
Mcafee	⊗	⊗	⊗			⊗ ☒	⊗ ☒	⊗ ☒
Microsoft Protection	⊗				⊗		⊗	⊗
NOD32	⊗	⊗ ☒	⊗			⊗	⊗	⊗
Norman		⊗						
TrendMicro					⊗ ☒			
nProtect	⊗ ☒	⊗	⊗ ☒		⊗ ☒			

⊗: malware packed with the original packer is detected
☒: malware packed with our packer is detected

Table 2. Packed malware escaping detection of anti-virus software

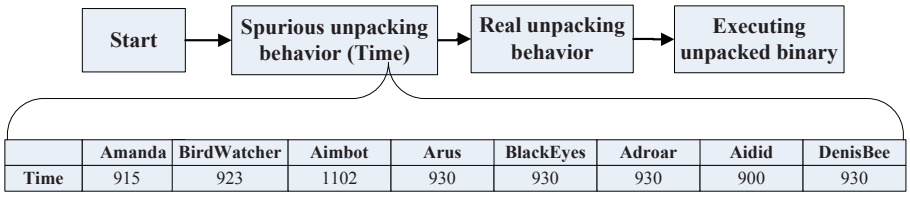
4.2 Executing our packed malware

In this part of the evaluation, we executed malware packed using our packer on two different environments, one with an unpacker (our implementation of OmniUnpack) and one without an unpacker (victim of the malware). We used the same anti-virus software OmniUnpack uses, i.e., ClamAV[5]. OmniUnpack claims that it only scans modified pages since the last dangerous system call. However according to a later report [5], this scanning strategy is not compatible with most existing commercial AV-scanners. Instead, we dumped the whole memory image and scan it with ClamAV just as what Justin [5] does. We also instrumented our implementation of the OmniUnpack to record T_W, T_E, and $T_S$in every spurious unpacking behavior found.

We ran eight malware samples (the same as those reported in Table 2) packed with our packer. Fig. 6 and Fig. 7 show our observations of the execution of these eight malware samples with and without OmniUnpack in the executing environment, respectively. Numbers shown in the figures are number of CPU cycles. We found that all eight packed malware samples succeeded in the denial-of-service attack on OmniUnpack by forging a large number of spurious unpacking behaviors (Fig. 7). At the same time, they all managed to unpack the original malware with only five spurious unpacking iterations on a normal executing environment (Fig. 6 only shows the last spurious iteration, and the first four are used to reduce the cache warm-up effect, see Section 3.1).

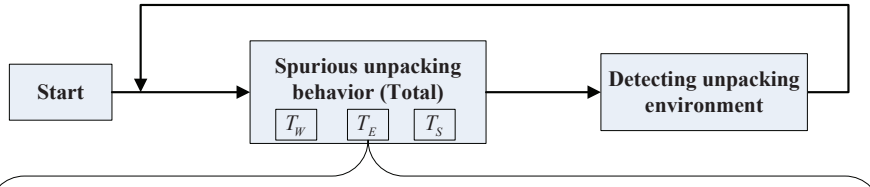
A closer look at Fig. 6 reveals that only about $1,000$ CPU cycles were spent in the first spurious unpacking iteration, which is well below the threshold used in detecting the executing environment (see Section 3.1). However, this number is several order of magnitude larger in the presence of OmniUnpack (see Fig. 6). This result verifies that our packed malware detects the existence of a host-based generic unpacker correctly.

[5] Clam AntiVirus, http://www.clamav.net/.



	Amanda	BirdWatcher	Aimbot	Arus	BlackEyes	Adroar	Aidid	DenisBee
Time	915	923	1102	930	930	930	900	930

Fig. 6. Malware execution without host-based generic unpackers



	Amanda	BirdWatcher	Aimbot	Arus	BlackEyes	Adroar	Aidid	DenisBee
T_W	806	497	556	607	492	584	1125	515
T_E	7.30×10^5	8.29×10^5	6.36×10^5	8.07×10^5	8.68×10^5	1.07×10^6	7.20×10^5	7.14×10^5
T_S	1.51×10^{10}	1.32×10^{10}	1.33×10^{10}	1.34×10^{10}	1.33×10^{10}	1.49×10^{10}	1.24×10^{10}	1.25×10^{10}
Total	1.51×10^{10}	1.32×10^{10}	1.33×10^{10}	1.34×10^{10}	1.33×10^{10}	1.49×10^{10}	1.24×10^{10}	1.25×10^{10}

Fig. 7. Malware execution with OmniUnpack

4.3 Applicability of our attack and limitations

Unpackers like OmniUnpack could introduce a counter to monitor the number of spurious unpacking behaviors and use a threshold to fight against our denial-of-service attack. However, this does not solve the problem as it is hard to set such a threshold and it comes with the price of failure in unpacking many packed code. This is because the idea of our attack is based on 1) end of unpacking is undecidable and heuristics have to be used for approximation; and 2) host-based generic unpackers need to protect itself by invoking anti-virus software in all potential execution points of the end of unpacking. We do not believe a simple solution exists for practical host-based generic unpackers in fighting against our denial-of-service attack.

However, our attack is not applicable to the unpackers which adopt very different heuristics. For example, Eureka [15] introduces statistical n-gram analysis to help find the end of unpacking. Hump-and-dump [18] creates a histogram of the addresses of executed instructions ordered by the last time the address is executed. Our attack will not work on these unpackers.

5 Related Work

Ferrie introduced various anti-unpacking tricks, including anti-dumping, anti-debugging, anti-emulating, and anti-intercepting techniques [4]. Since the real

payload will eventually be unpacked to the memory, unpackers dump the memory when they detect the existence of the payload. Anti-dumping techniques are designed to prevent the unpacker from dumping the process memory. Several unpackers make use of a debugger or emulator to monitor the unpacking behavior. Anti-debugging and anti-emulating tricks are used to circumvent the unpacking whenever there is a debugger or an emulator running. Some unpackers use page intercepter to monitor memory page accesses. Anti-intercepting techniques are used to detect the existence of an interceptor and evade it.

There have also been works to detect the existence of an emulator [3, 12], including CPU bugs, model-specific registers, and alignment checking. These techniques can be adopted to evade emulator-based unpackers.

Dual-mapping is used to evade unpackers which dynamically unpack the program by detecting memory execution [16]. In this work, physical memory regions are mapped into two virtual ones, one for written and another for execution.

Our work is different from these previous work in that we use timing information for detecting the existence of an unpacker.

6 Conclusion

In this paper, we present a denial-of-service attack on host-based generic unpackers. Malware samples packed using our automatic packer detects the existence of a host-based generic unpacker and forges spurious unpacking behaviors, and these spurious unpacking behaviors result in a denial-of-service attack on the unpacker. Our experimental evaluation demonstrates the effectiveness of this attack.

Acknowledgements

We thank the authors of OmniUnpack for their help in our implementation and sharing of information in the dangerous system call table.

References

1. L. Bohne. *Pandora's Bochs: Automatic Unpacking of Malware*. PhD thesis, University of Mannheim, 2008.
2. M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith. Malware normalization. Technical report 1539, University of Wisconsin, Madison, WI, USA, 2005.
3. P. Ferrie. Attacks on virtual machine emulation. In *AVAR Conference*. Symantec Advanced Threat Research, 2006.
4. P. Ferrie. Anti-unpacker tricks. In *Proceedings of the 2nd International CARO Workshop*, 2008.
5. F. Guo, P. Ferrie, and T.C. Chiueh. A study of the packer problem and its solutions. In *Proceedings of 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 98–115, 2008.

6. J. Hawes. Comparative review. In *Virus Bulletin*, pages 14–27, 2009.
7. Intel Corporation. *Using the RDTSC Instruction for Performance Monitoring*, 1997.
8. M.G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode (WORM)*, pages 46–53, 2007.
9. L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the 2007 Annual Computer Security Applications Conference(ACSAC)*, pages 431–441. IEEE Computer Society, 2007.
10. M. Morgenstern and A. Marx. Runtime packer testing experiences. In *Proceedings of the 2nd International CARO Workshop*, 2008.
11. D. Quist. Covert debugging: Circumventing software armoring techniques. In *Black Hat*, 2007.
12. T. Raffetseder, C. Krügel, and E. Kirda. Detecting system emulators. In *Proceedings of 10th International Conference on Information Security (ISC)*, pages 1–18, 2007.
13. P. Royal. Alternative medicine: The malware analyst's blue pill. In *Black Hat*, 2008.
14. 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. IEEE Computer Society, 2006.
15. M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee. Eureka: A framework for enabling static malware analysis. In *Proceedings of 13th European Symposium on Research in Computer Security (ESORICS)*, pages 481–500, 2008.
16. Skape. Using dual-mappings to evade automated unpackers. *Uninformed Journal*, 2008.
17. A. Stepan. Improving proactive detection of packed malware. In *Virus Bulletin*, pages 11–13, 2006.
18. L. Sun, T. Ebringer, and S. Boztas. Hump-and-dump: Efficient generic unpacking using an ordered address execution histogram. In *Second International CARO Workshop*. Department of Computer Science and Software Engineering, The University of Melbourne, Australia, 2008.