# The Nepenthes Platform: An Efficient Approach to Collect Malware

Paul Baecher[1], Markus Koetter[1], Thorsten Holz[2], Maximillian Dornseif[2], and Felix Freiling[2]

[1] Nepenthes Development Team
nepenthesdev@gmail.com
[2] University of Mannheim
Laboratory for Dependable Distributed Systems
{holz, dornseif, freiling}@informatik.uni-mannheim.de

**Abstract.** Up to now, there is little empirically backed quantitative and qualitative knowledge about self-replicating malware publicly available. This hampers research in these topics because many counter-strategies against malware, e.g., network- and host-based intrusion detection systems, need hard empirical data to take full effect.

We present the *nepenthes* platform, a framework for large-scale collection of information on self-replicating malware in the wild. The basic principle of nepenthes is to emulate only the *vulnerable* parts of a service. This leads to an efficient and effective solution that offers many advantages compared to other honeypot-based solutions. Furthermore, nepenthes offers a flexible deployment solution, leading to even better scalability. Using the nepenthes platform we and several other organizations were able to greatly broaden the empirical basis of data available about self-replicating malware and provide thousands of samples of previously unknown malware to vendors of host-based IDS/anti-virus systems. This greatly improves the detection rate of this kind of threat.

**Keywords:** Honeypots, Intrusion Detection Systems, Malware.

## 1 Introduction

*Automated Malware Collection.* Software artifacts that serve malicious purposes are usually termed as *malware.* Particularly menacing is malware that spreads automatically over the network from machine to machine by exploiting known or unknown vulnerabilities. Such malware is not only a constant threat to the integrity of individual computers on the Internet. In the form of botnets for example that can bring down almost any server through distributed denial of service, the combined power of many compromised machines is a constant danger even to uninfected sites.

We describe here an approach to *collect* malware. Why should this be done? There are two main reasons, both following the motto “know your enemy”: First of all, investigating individual pieces of malware allows better defences against these and similar artifacts. For example, intrusion detection and anti-virus systems can refine their list of signatures against which files and network traffic are matched. In general, the better and more we know about what malware is currently spreading in the wild, the better can our defenses be. The second reason why we should collect malware is that, if we do it in a large scale, we can generate statistics to learn more about attack patterns, attack trends, and attack rates of malicious network traffic today, based on live and authentic data.

Collecting malware in the wild and analyzing it is not an easy task. In practice, much malware is collected and analyzed by detailed forensic examinations of infected machines. The actual malware needs to be dissected from the compromised machine by hand. With the increasing birth rate of new malware this can only be done for a small proportion of system compromises. Also, sophisticated worms and viruses spread so fast today that hand-controlled human intervention is almost always too late. In both cases we need a very *high degree of automation* to handle these issues.

*Honeypot technology.* The main tool to collect malware in an automated fashion today are so-called *honeypots*. A honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource. The idea behind this methodology is to lure in attackers such as automated malware and then study them in detail. Honeypots have proven to be a very effective tool in learning more about Internet crime like credit card fraud [10] or botnets [6]. The literature distinguishes two general types of honeypots:

– *Low-interaction honeypots* offer limited services to the attacker. They emulate services or operating systems and the level of interaction varies with the implementation. The risk tends to be very low. In addition, deploying and maintaining these honeypots tends to be easy. A popular example of this kind of honeypots is *honeyd* [14]. With the help of low-interaction honeypots, it is possible to learn more about attack patterns and attacker behavior.
– *High-interaction honeypots* offer the attacker a real system to interact with. More risk is involved when deploying a high-interaction honeypot, e.g., special provisions are done to prevent attacks against system that are not involved in the setup. They are normally more complex to setup and maintain. The most common setup for this kind of honeypots is a *GenIII honeynet* [3].

Low-interaction honeypots entail less risks than high-interaction ones. In addition, deploying and maintaining low-interaction honeypots tends to be easy, at least much easier than running high-interaction honeypots, since less special provisions have to be done to prevent attacks against the system that runs the honeypot software. However, high-interaction honeypots still allow us to study attackers in more detail and learn more about the actual proceeding of attackers than low-interaction honeypots. The differences between low-interaction and high-interaction honeypots manifest a tradeoff: high-interaction honeypots are *expressive*, i.e., they offer full system functionality which is in general not supported by low-interaction honeypots. However, low-interaction honeypots are much more *scalable*, i.e., it is much easier and less resource-intensive to deploy them in a large-scale.

*Contribution.* In this paper we introduce *nepenthes*, a new type of honeypot that inherits the scalability of low-interaction honeypots but at the same time offers a high degree of expressiveness. Nepenthes is not a honeypot *per se* but rather a platform to deploy honeypot modules (called *vulnerability modules*). This is the key to increased expressiveness: Vulnerability modules offer a highly flexible way to configure nepenthes into a honeypot for many different types of vulnerabilities. In classical terms, nepenthes still realizes a low-interaction honeypot since it *emulates* the vulnerable services. However, as we argue in this paper, emulation and the knowledge about the expected attacker behavior is the key to automation. Furthermore, the flexibility of nepenthes allows to deploy unique features not available in high-interaction honeypots. For example, it is possible to emulate the vulnerabilities of different operating systems and computer architectures on a *single machine* and during a *single attack* (i.e., an emulation can mimic the generic parts of a network conversation and depending on the network traffic decide whether it wants to be a Linux or a Win32 machine for example). This improves the scalability. We report on experiments showing that nepenthes is also scalable by emulating more than 16.000 different IP addresses on a single physical machine. Furthermore, through its flexible reporting mechanisms, nepenthes can be deployed in a hierarchical manner increasing scalability even further. Automation is further supported through the modularity of nepenthes, which offers the possibility to add specialized analysis and reporting modules.

With the help of the nepenthes platform, we are able to collect malware that is currently spreading in the wild on a large-scale. Since we focus on malware that is currently spreading, we can carry out a vulnerability assessment based on live data. Furthermore, the collected malware samples enable us to examine the effectiveness of current anti-virus engines. Furthermore, since we collect malware on a large-scale, we can also detect new trends or attack patterns. We will present more results in Section 3.

In summary, nepenthes is a unique novel combination of expressiveness, scalability and flexibility in honeypot-based research.

*Related work.* Large-scale measurements of malicious network traffic have been the focus of previous research. With the help of approaches like the *network telescope* [11] or *darknets* [4] it is possible to observe large parts of the Internet and monitor malicious activities. In contrast to nepenthes, these approaches *passively* collect information about the network status and can infere further information from it, e.g., inferring the amount of Distributed Denial-of-Service attacks [12]. By not responding to the packets, it is not possible to learn more about full attacks. Slightly more expressive approaches like the *Internet Motion Sensor* [2] differentiate services by replying to a TCP SYN paket with TCP SYN-ACK pakets. However, their expressiveness is also limited and only with further extensions it is possible to also learn more about spreading malware.

*honeyd* [14] is a prominent example of a low-interaction honeypot. This daemon creates virtual hosts on a network. It simulates the TCP/IP stack of arbitrary operating systems and can be configured to run arbitrary services. These services are generally small scripts that emulate real services, and offer only a limited expressiveness. Honeyd can simulate arbitrary network topologies including dedicated routes and routers, and can be configured to feign latency and packet loss. In summary, this tool can emulate complex networks by simulating different hosts with any kind of services and help to learn about attacks from a high-level point of view. In contrast to nepenthes, honeyd does not offer as much expressiveness since the reply capabilities of honeyd are limited from a network point of view. Nepenthes can be used as a subsystem for honeyd, however. This extends honeyd and enables a way to combine both approaches: nepenthes acts then as a component of honeyd and is capable of dealing with automated downloading of malware.

The *Collapsar platform* [9] is a virtual-machine-based architecture for network attack detention. It allows to host and manage several high-interaction virtual honeypots in a local dedicated network. Malicious traffic is redirected from other networks (*decentralized honeypot presence*) to this central network which hosts all honeypots (*centralized honeypot management*). This enables a way to build a *honeyfarm*. Note that the idea of a honeyfarm is not tied to the notion of a high-interaction honeypot: It is also possible to deploy nepenthes as a honeyfarm system by redirecting traffic from remote locations to a central nepenthes server.

*Internet Sink* (*iSink*) [23] is a system that passively monitors network traffic and is also able to actively respond to incoming connection requests. The design is stateless and therefore the expressiveness of the responses is limited. Similarly, *HoneyTank* [19] is a system that implements stateless responders to network probes. This allows to collect information about malicious activties to a limited amount. Statelessness implies that the expressiveness is limited. In contrast to these systems, nepenthes implements a finite state machine to emulate vulnerabilities. This allows us to collect more detailed information about an attack.

Closest to our work is the *Potemkin* virtual honeyfarm by Vrable *et al.* [20]. Potemkin exploits virtual machines, aggressive memory sharing, and late binding of resources to emulate more than 64,000 high-interaction honeypots using ten physical servers. This approach is promising, but has currently several drawbacks compared to nepenthes: Firstly, each honeypot within Potemkin has to be a fixed system in a fixed configuration. In contrast to this, the vulnerability modules of nepenthes allow greater flexibility. As mentioned above, nepenthes can react for example on exploitation attempts against Windows 2000 and Windows XP, even regardless of service pack. It would even be possible to emulate on a single nepenthes honeypot vulnerabilities for different operating systems and even different processor architectures. Secondly, the scalability of nepenthes is at least as good as the scalability of Potemkin. Thirdly, there are currently only preliminary results for the scalability of Potemkin. In [20], the authors give only results for a representative 10 minutes period. Since the implementation of Potemkin is not publicly available, we can not verify these results. In contrast to this, nepenthes runs stable for weeks and the source code is available under the GNU General Public License.

*Roadmap.* This paper is outlined as follows: Section 2 presents the nepenthes platform in detail and in Section 3 we show the results of our work, especially focusing on the effectiveness of this approach. We give an overview of future work in Section 4 and conclude the paper in Section 5.

## 2 The Nepenthes Platform

In this section we introduce the *nepenthes* platform in detail. We show how the concept of low-interaction honeypots can be extended to effectively develop a method to collect malware. In addition, this platform can be used to learn more about attack patterns. Moreover, we present a technique to use this platform in a distributed way, similar to the concepts introduced by Collapsar [9].

The main idea behind nepenthes is emulation of vulnerable services. Currently, there are two main concepts in this area: honeyd scripts simply emulate the necessary parts of a service to fool automated tools or very low-skilled attackers. This allows a large-scale deployment with thousands of low-interaction honeypots in parallel. But this approach has some limits: with honeyd it is not possible to emulate more complex protocols, e.g., a full emulation of FTP data channels is not possible. In contrast to this, high-interaction GenIII honeypots use a real system and thus do not have to emulate a service. The drawback of this approach is the poor scalability. Deploying several thousand of these honeypots is not possible due to limitations in maintenance and hardware requirements. Virtual approaches like Potemkin [20] are in an early stage of development and it is currently not clear how they will perform in real-world scenarios, although preliminary results look very promising.

The gap between these two approaches can be filled with the help of the nepenthes platform. It allows to deploy several thousands of honeypots in parallel with only moderate requirements in hardware and maintenance. This platform enables us to efficiently deploy thousands of honeypots in parallel and collect information about malicious network traffic.

### 2.1 Architecture of the Nepenthes Platform

nepenthes is based upon a very flexible and modularized design. The core – the actual daemon – handles the network interface and coordinates the actions of the other modules. The actual work is carried out by several modules, which register themselves in the nepenthes core. Currently, there are several different types of modules:

- *Vulnerability modules* emulate the vulnerable parts of network services.
- *Shellcode parsing modules* analyze the payload received by one of the vulnerability modules. These modules analyze the received shellcode, an assembly language program, and extract information about the propagating malware from it.
- *Fetch modules* use the information extracted by the shellcode parsing modules to download the malware from a remote location.

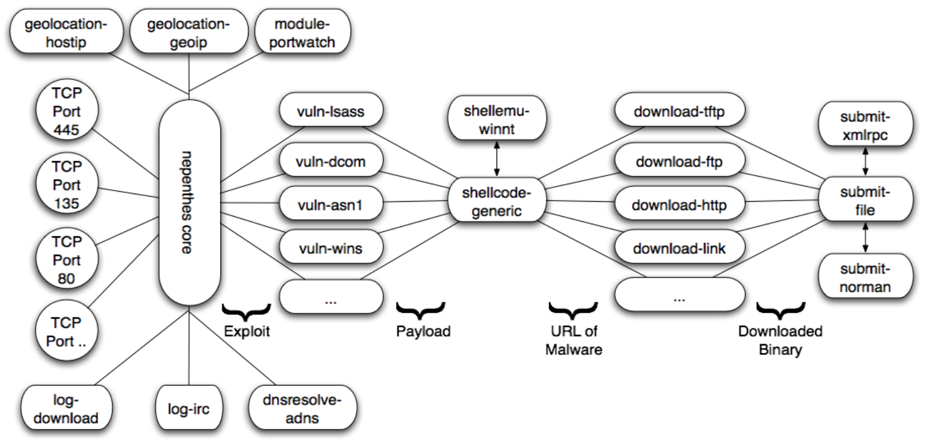


**Fig. 1.** Concept behind nepenthes platform

– *Submission modules* take care of the downloaded malware, e.g., by saving the binary to a hard disc, storing it in a database, or sending it to anti-virus vendors.
– *Logging modules* log information about the emulation process and help in getting an overview of patterns in the collected data.

In addition, several further components are important for the functionality and efficiency of the nepenthes platform: *shell emulation*, a *virtual filesystem* for each emulated shell, *geolocation modules*, *sniffing modules* to learn more about new activity on specified ports, and *asynchronous DNS resolution*.

The schematic interaction between the different components is depicted in Figure 1 and we introduce the different building blocks in the next paragraphs.

*Vulnerability modules* are the main factor of the nepenthes platform. They enable an effective mechanism to collect malware. The main idea behind these modules is the following observation: in order to get infected by autonomous spreading malware, it is sufficient to only emulate the *necessary* parts of a vulnerable service. So instead of emulating the whole service, we only need to emulate the relevant parts and thus are able to efficiently implement this emulation. Moreover, this concepts leads to a scalable architecture and the possibility of large-scale deployment due to only moderate requirements on processing resources and memory. Often the emulation can be very simple: we just need to provide some minimal information at certain offsets in the network flow during the exploitation process. This is enough to fool the autonomous spreading malware and make it believe that it can actually exploit our honeypot. This is an example of the deception techniques used in honeypot-based research. With the help of vulnerability modules we trigger an incoming exploitation attempt and eventually we receive the actual payload, which is then passed to the next type of modules.

*Shellcode parsing modules* analyze the received payload and extract automatically relevant information about the exploitation attempt. Currently, only one shellcode parsing module is capable of analyzing all shellcodes we have found in the wild. The module works in the following way: first, it tries to decode the shellcode. Most of the shellcodes are encrypted with an *XOR encoder*. An XOR decoder is a common way to encrypt the actual shellcode in order to evade intrusion detection systems and avoid string processing functions. Afterwards the module decodes the code itself according to the computed key and then applies some pattern detection, e.g., `CreateProcess()` or generic URL detection patterns. The results are further analyzed (e.g., to extract credentials) and if enough information can be reconstructed to download the malware from the remote location, this information is passed to the next kind of modules. A shellcode module that parses shellcodes in an even more generic way by emulating a Windows operating system environment is currently in development.

*Fetch modules* have the task of downloading files from the remote location. Currently, there are seven different fetch modules. The protocols TFTP, HTTP, FTP and csend/creceive (an IRC-based submission method) are supported. Since some kinds of autonomous spreading malware use custom protocols for propagation, there are also fetch modules to handle these custom protocols. Fetching files from a remote location implies that the system running nepenthes contacts other machines in the Internet. From an ethical point of view, this could be a problem since systems not under our control are contacted. A normal computer system that is infected by autonomous spreading malware would react in the same way, therefore we have no concerns fetching the malware from the remote location. However, it is possible to turn off the fetch modules. Then the system collects information about exploitation attempts and can still be useful as some kind of warning system.

Finally, *submission modules* handle successfully downloaded files. Currently there are four different types of submission modules:

- A module that stores the file in a configurable location on the filesystem and is also capable of changing the ownership.
- A module that submits the file to a central database to enable distributed sensors with central logging interface.
- A module that submits the file to another nepenthes instance to enable a hierarchical structure of nepenthes sensors.
- A module that submits the file to an antivirus vendor for further analysis.

Certain malware does not spread by downloading shellcodes, but by providing a shell to the attacker. Therefore it is sometimes required to spawn and emulate a Windows shell. nepenthes offers *shell emulation* by emulating a rudimentary Windows shell to enable a shell interaction for the attacker. Several commands can be interpreted and batch file execution is supported. Such a limited simulation has proven to be sufficient to trick automated attacks. Based on the collected information from the shell session, it is then possible to also download the corresponding malware.

A common technique to infect a host via a shell is to write commands for downloading and executing malware into a temporary batch file and then execute

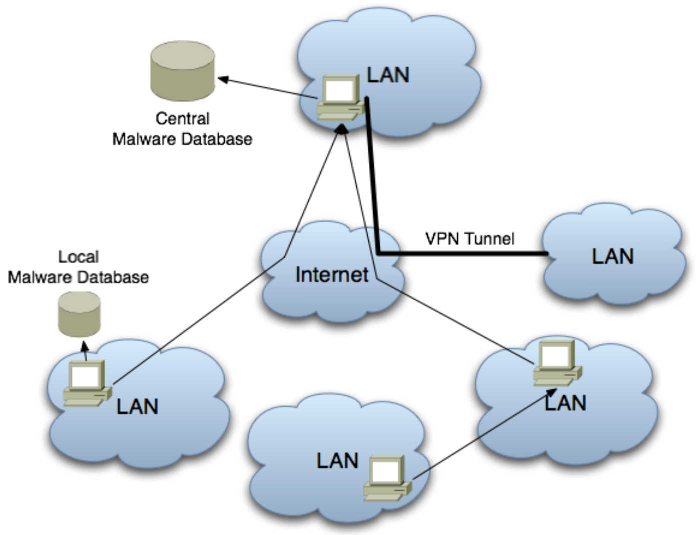


**Fig. 2.** Setup of distributed nepenthes platform

it. Therefore, a *virtual filesystem* is implemented to enable this type of attacks. This helps in scalability since files are only created on demand, similar to *copy-on-write*: when the incoming attack tries to create a file, this file is created on demand and subsequently, the attacking process can modify and access it. All this is done virtually, to enable a higher efficiency. Every shell session has its own virtual filesystem, so that concurrent infection sessions using similar exploits do not infere with each other. The temporary file is analyzed after the attacking process has finished and based on this information, the malware is downloaded from the Internet automatically. This mechanism is similar to *cages* in Symantec's ManTrap honeypot solution [18].

Nepenthes has several advantages compared to other solutions to automatically collect malware. On the one hand, nepenthes is a very stable architecture. A wrong offset or a broken exploit will not lead to crashes, as opposed to other attempts in this area. On the other hand, nepenthes scales well to even a large number of IP addresses in parallel. By hierarchical deployment, it is very easy to cover even larger parts of the network space with only limited resources.

### 2.2 Flexible Deployment

Nepenthes offers a very flexible design that allows a wide array of possible setups. The most simple setup is a local nepenthes sensor, deployed in a LAN. It collects information about malicious, local traffic and stores the information on the local hard disc. More advanced uses of nepenthes are possible with a distributed approach. Figure 2 illustrates a possible setup of a distributed nepenthes platform: a local nepenthes sensor in a LAN collects information about suspicious traffic there. This sensor stores the collected information in a local database and also forwards all information to another nepenthes sensor.

A second setup is a hierarchical one (depicted in the middle of Figure 2): a distributed structure with several levels is build and each level sends the collected information to the sensor at the higher level. In such a way, load can be distributed across several sensor or information about different network ranges can be collected in a central and efficient way.

Thirdly, traffic can be re-routed from a LAN to a remote nepenthes sensor with the help of a VPN tunnel (depicted on the right). This approach is similar to the network setup of the Collapsar project [9]. It enables a flexible setup for network attack detention. Furthermore, it simplifies deployment and requires less maintenance.

## 2.3 Capturing New Exploits

An important factor of a honeypot-based system is also the ability to detect and respond to *zero-day* (*0day*) attacks, e.g., attack that exploit an unknown vulnerability or at least a vulnerability for which no patch is available. The nepenthes platform also has the capability to respond to this kind of threat. The two basic blocks for this ability are the *portwatch* and *bridging* modules. These modules can track network traffic at network ports and help in the analysis of new exploits. By capturing the traffic with the help of the portwatch module, we can at least learn more about any new threat since we have already a full network capture of the first few packets. In addition, nepenthes can be extended to really *handle* 0day attacks. If a new exploit targets the nepenthes platform, it will trigger the first steps of a vulnerability module. At some point, the new exploit will diverge from the emulation. This divergence can be detected and then we perform a switch (*hot swap*) to either a real honeypot or some kind of specialized system for dynamic taint analysis, e.g. Argos [13]. This second system is an instance of the system nepenthes is emulating vulnerabilities for and shares the internal state with it. This approach is similar to *shadow honeypots* [1].

With the help of the nepenthes platform, we can efficiently handle all known exploits. Once something new is propagating in the wild, we switch from our emulation to a real honeypot to capture all aspects of the new attack. From the captured information, we are also able to respond to this new threat and automatically extract response patterns. The mechanism behind this is rather simple, but effective. We record the network flow and extract from this flow the necessary information to build a full vulnerability module. The whole mechanism could presumably also be extended to build a fully automated system to respond to new threats. Since the honeypot has by definition no false positives, we can assume that all traffic is malicious. For known malicious traffic, we can respond with the correct replies. For unknown malicious code, we need to learn the correct replies with the help of a shadow honeypot. Based on the correct replies, a learning algorithm could be used to extract all dynamic data inside the replies (e.g., timestamps) and a correct vulnerability module could be built on-the-fly. These ideas are currently in development.

### 2.4 Limitations

We also identified several limitations of the nepenthes platform which we present in this section. First, nepenthes is only capable of collecting malware that is *autonomously* spreading, i.e., that propagates further by scanning for vulnerable systems and then exploits them. This is a limitation that nepenthes has in common with most honeypot-based approaches: a web site that contains a browser exploit which is only triggered when the web site is accessed will not be detected with ordinary honeypots due to their passive nature. The way out of this dilemma is to use client-side honeypots like HoneyMonkeys [22] or Kathy Wang's honeyclient [21] to detect this kind of attacks. The modular architecture of nepenthes would enable this kind of vulnerability modules, but this is not the aim of the nepenthes platform. The results in Section 3.2 show that nepenthes is rather able to collect many different types of bots [7].

Secondly, malware that propagates by using a *hitlist* to find vulnerable systems [17] is hard to detect with nepenthes. This is a limitation that nepenthes has in common with all current honeypot-based systems and also other approaches in the area of vulnerability assessment. Here, the solution of the problem would be to *become part of the hitlist.* If for example the malware generates its hitlist by querying a search engine for vulnerable systems, the trick would be to smuggle a honeypot system in the index of the search engine. Currently it is unclear how such an *advertisement* could be implemented within the nepenthes platform.

Thirdly, it is possible to remotely detect the presence of nepenthes: since a nepenthes instance normally emulates a large number of vulnerabilities and thus opens many TCP ports, an attacker could become suspicious during the reconnaissance phase. Current automated malware does not check the plausibility of the target, but future malware could do so. To mitigate this problem, the stealthiness can be improved by using only the vulnerability modules which belong to a certain configuration of a real system, e.g., only vulnerability modules which emulate vulnerabilities for Windows 2000 Service Pack 1. The tradeoff lies in reduced expressiveness and leads to fewer samples collected. A similar problem with the stealthiness appears if the results obtained by running nepenthes are published unmodified. To mitigate such a risk, we refer to the solution outlined in [16].

Moreover, nepenthes is not exhaustive in terms of analyzing which exploits a particular piece of malware is targeting. This limitation is due to the fact that we respond to an incoming exploitation attempt and can just react on these network pakets. Once we have downloaded a binary executable of the malware, static or dynamic analysis of this binary can overcome this limitation. This is, however, out of the scope of the current nepenthes implementation.

## 3 Results

Vulnerability modules are one of the most important components of the whole nepenthes architecture since they take care of the emulation process. At the time of this writing, there are 21 vulnerability modules in total. Table 1 gives an overview of selected available modules, including a reference to the related security advisory or a brief summary of its function.

**Table 1.** Overview of selected emulated vulnerable services

| Name | Reference |
|---|---|
| vuln-asn1 | ASN .1 Vulnerability Could Allow Code Execution (MS04-007) |
| vuln-bagle | Emulation of backdoor from Bagle worm |
| vuln-dcom | Buffer Overrun In RPC Interface (MS03-026) |
| vuln-iis | IIS SSL Vulnerability (MS04-011 and CAN-2004-0120) |
| vuln-kuang2 | Emulation of backdoor from Kuang2 worm |
| vuln-lsass | LSASS vulnerability (MS04-011 and CAN-2003-0533) |
| vuln-msdtc | Vulnerabilities in MSDTC Could Allow Remote Code Execution (MS05-051) |
| vuln-msmq | Vulnerability in Message Queuing Could Allow Code Execution (MS05-017) |
| vuln-mssql | Buffer Overruns in SQL Server 2000 Resolution Service (MS02-039) |
| vuln-mydoom | Emulation of backdoor from myDoom/Novarg worm |
| vuln-optix | Emulation of backdoor from Optix Pro trojan |
| vuln-pnp | Vulnerability in Plug and Play Could Allow Remote Code Execution (MS05-039) |
| vuln-sasserftpd | Sasser Worm FTP Server Buffer Overflow (OSVDB ID: 6197) |
| vuln-ssh | Logging of SSH password brute-forcing attacks |
| vuln-sub7 | Emulation of backdoor from Sub7 trojan |
| vuln-wins | Vulnerability in WINS Could Allow Remote Code Execution (MS04-045) |

This selection of emulated vulnerabilities has proven to be sufficient to handle most of the autonomous spreading malware we have observed in the wild. As we show in the remainder of this section, these modules allows us to learn more about the propagating malware. However, if a certain packet flow cannot be handled by any vulnerability module, all collected information is stored on hard disc to facilitate later analysis. This allows us to detect changes in attack patterns, is an indicator of new trends, and helps us to develop new modules. In case of a *0day*, i.e., an vulnerability for which no information is publicly available, this can enable a fast analysis since the first stages of the attack have already been captured. As outlined in Section 2.3, this can also be extended to really handle 0day attacks. A drackback of this approach is that an attacker can send random data to a network port and nepenthes will store this data on hard disc. This can lead to a Denial-of-Service condition if the attacker sends large amount of bogus network traffic, however we did not experience any problems up to now. In addition, this problem can be mitigated by implementing upper bounds on the amount of traffic stored on hard disk.

Developing a new vulnerability modules to emulate a novel security vulnerability or to capture a propagating 0day exploit is a straightforward process and demands only little effort. On average, writing of less than 500 lines of C++ code (including comments and blank lines) is required to implement the needed functionality. This task can be carried out with some experience in a short amount of time, sometimes only requiring a couple of minutes.

As an example, we want to present our experience with the recent *Zotob* worm: in MS05-039, Microsoft announced a security vulnerability in the Plug and Play service of Windows 2000 and Windows XP at August 09, 2005. This vulnerability is rated critical for Windows 2000 since it allows remote code execution, resulting in a remote system compromise. Two days later, a proof-of-concept exploit for this vulnerability was released. This exploit code contains enough information to implement a vulnerability module for nepenthes, so that malware propagating with the help of MS05-039 can be captured with this module. Without the proof-of-concept exploit, it would have been possible to build a vulnerability module only based on the information provided in the security advisory by Microsoft. But this process would be more complex since it would require the development of an attack vector, which could then be emulated as a vulnerability module. Nevertheless, this is feasible. After all, attackers also implemented a proof-of-concept exploit solely on the basis of the information in the security bulletin. Another three days after the release of the proof-of-concept exploit – at August 14 – a worm named *Zotob* started to exploit this vulnerability in the wild. So only five days after the release of the security advisory, the first bot propagated with the help of this vulnerability. But at this point in time, nepenthes was already capable of capturing such a worm. Similarly, the process of emulating the vulnerability in Microsoft Distributed Transaction Coordinator (MSDTC), published in Microsoft security bulletin MS05-051, took only a small amount of time.

### 3.1 Scalability

In this section, we want to evaluate the scalability of the nepenthes platform. With the help of several metrics we investigate, how effective our approach is, and how many honeypot systems we can emulate with our implementation.

As noted in [20], a "key factor to determine the scalability of a honeypot is the number of honeypots required to handle the traffic from a particular IP address range". To cover a /16 network, a naive approach would be to install over 64,000 honeypots to cover the whole network range. This would of course be a waste of resources, since only a limited amount of IP addresses receives network traffic at any given point in time. The low-interaction honeypot honeyd is reported to be able to simulate a whole /16 network on just a single computer. The expressiveness of this tool is low since it only emulates the TCP/IP stack of an arbitrary operating system. In contrast to this, nepenthes is capable of emulating several vulnerabilities at application level.

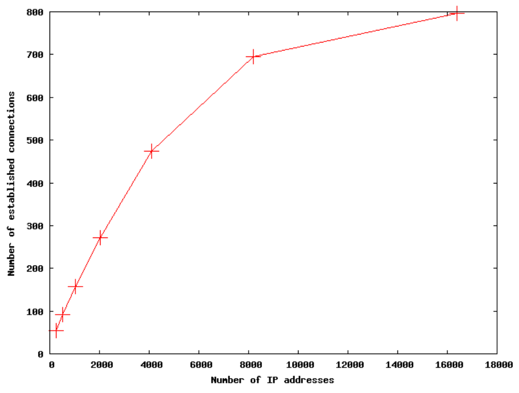
To evaluate the scalability of nepenthes, we have used the following setup: the testbed is a commercial off-the-shelf (COTS) system with a 2.4GHz Pentium III, 2 GB of physical memory, and 100 MB Ethernet NIC running Debian Linux 3.0 and version 2.6.12 of the Linux kernel. This system runs nepenthes 0.1.5 in default configuration. This means that all 21 vulnerability modules are used, resulting in a total of 29 TCP sockets on which nepenthes emulates vulnerable services.

We tested the implementation with a varying number of emulated systems, ranging from only 256 honeypots up to 32,000 emulated honeypots. For each configuration, we measured the number of established TCP connections, the system load, and the memory consumption of nepenthes, for a time interval of one hour. We repeated this measurement several times in different order to cancel out statistical unsteadiness. Such an unsteadiness could for example be caused by diurnal properties of malware epidemics [5] or bursts in the network traffic. The average value of all measurements is then an estimation of the specific metric we are interested in. Figures 3 (a) and (b) give an overview of our results. In each figure, the x-axis represents the number of IP addresses assigned to nepenthes running on the testbed machine. The y-axis reprents the number of established TCP connections (a) and the average system load (b), respectively. We forbear from plotting the memory consumption since it is low (less than 20 MB for even a large number of simulated IP addresses), and nearly independent from the number of established TCP connections. In the first figure we see that the scalability is nearly linear up to 8,192 IP addresses. This corresponds to the system load, which is below 1 (figure b). Afterwards, the number of established TCP connections is degreasing, which is caused by a system load above 1, i.e., the system is fully occupied with I/O operations.

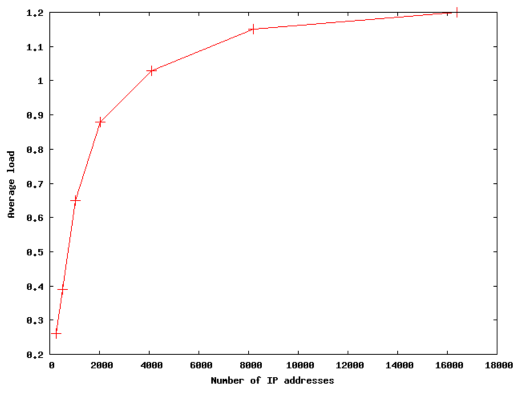
In the following, we take a closer look at the long-time performance of the nepenthes platform emulating a whole /18 network, i.e., about 16,000 IP addresses. We have this setup up and running for more then five months and it runs quite stable. There are seldom kernel crashes, but these are caused by instabilities in the Linux kernel handling such a large amount of IP addresses in parallel. Apart from this, nepenthes itself is a mature system. To get an overview of the overall performance of this platform, we present some statistics on the performance first. In Figure 4 (a) we see the five minute average of established TCP connections for an instance of nepenthes running on a /18 network for 30 hours. The number of established TCP connections is on average 796, with peaks of up to 1172. The lowest values are around 600 concurrent established connections, so the volatitlity is rather high. Our experience shows that burst of more than 1300 concurrent established TCP connections are tolerable on this system. Even more connections could be handled with better hardware: currently, the average load of the system is slightly above 1, i.e., the processor is never idle. For a one hour period, we observed more than 180,000 SYN packets, which could potentially be handled by nepenthes.

Figure 4 (b) depicts the five minute average of network throughput. Green is the amount of incoming traffic, with an average of 308.8 kB/s and a maximum of 369.7 kB/s. The outgoing traffic is displayed with a blue line. The average of outgoing traffic is 86.6 kB/s, whereas the peak lies at 105.4 kB/s. So despite a rather high volatility in concurrent TCP connections, the network throughput is rather stable.

We now take a closer look at the long-time performance of this nepenthes instance regarding the download of new samples collected. A five week period is the data foundation of the following statistics. Figure 5 depicts the daily number of download attempts and successful downloads.



(a)



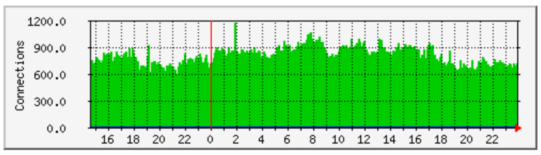
(b)

**Fig. 3.** Number of concurrent established TCP connections (a) and system load (b) in relation to number of IP addresses assigned to nepenthes

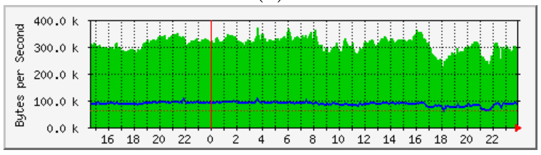
### 3.2 Statistics for Collected Malware

In this section, we analyze the malware we have collected with our honeynet platform. Since nepenthes is optimized to collect malware in an automated way, this is the vast amount of information we collect with the help of this tool. A human attacker could also try to exploit our honeynet platform, but he would presumably notice quickly that he is just attacking a low-interaction honeypot since we only emulate the necessary parts of each vulnerable service and the command shell only emulates the commands typically issued by malware. So we concentrate on automated attacks and show how effective and efficient our approach is.

With the help of the nepenthes platform, we are able to automatically collect malware on a large-scale basis. We are running nepenthes in several different networks and centrally store the malware we have downloaded. Figure 5 (a) and (b) show the cumulative number of download attempts and successful downloads

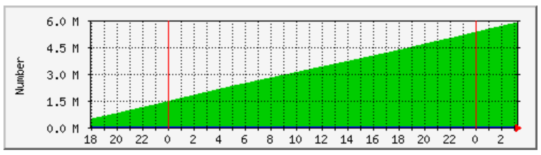


(a)

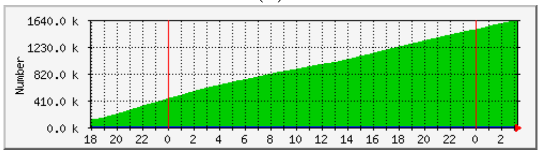


(b)

**Fig. 4.** Five minute average of established TCP connections (a) and network throughput (b) for nepenthes running on a /18 network in a period of 33 hours



(a)



(b)

**Fig. 5.** Number of malware download attempts (a) and successful downloaded files (b) for nepenthes running on a /18 network in a period of 33 hours

for a nepenthes platform assigned to a /18 network. Within about 33 hours, more than 5.5 million exploitation attempts are effectively handled by this system (see Figure 5 (a)). That means that so often the download modules are triggered to start a download. Often these download attempts fail, e.g., because the malware tries to download a copy of itself from a server that is meanwhile taken down. Figure 5 (b) depicts the number of successful download, i.e., nepenthes successfully download a piece of malware. Within these 33 hours, about 1.5 million binaries are downloaded. Most of these binaries are duplicates, but nepenthes has to issue a download and is only afterwards able to decide whether the binary is actually a new one. In this particular period, we were able to download 508 new unique binaries.

In a four month period, we have collected more than 15,500 unique binaries, corresponding to about 1,400 MB of data. Uniqueness in this context is based on different MD5 sums of the collected binaries. All of the files we have collected are

**Table 2.** Detection rates of different antivirus engines

| | AV engine 1 | AV engine 2 | AV engine 3 | AV engine 4 |
|---|---|---|---|---|
| Complete set (14,414 binaries) | 85.0% | 85.3% | 90.2% | 78.1% |
| Latest 24 hours (460 binaries) | 82.6% | 77.8% | 84.1% | 73.1% |

**Table 3.** Top ten types of collected malware

| Place | Name according to ClamAV | Number of captured samples |
|---|---|---|
| 1 | Worm.Padobot.M | 1136 |
| 2 | Trojan.Gobot-3 | 906 |
| 3 | Worm.Padobot.N | 698 |
| 4 | Trojan.Gobot-4 | 639 |
| 5 | Trojan.Poebot-3 | 540 |
| 6 | Trojan.IRCBot-16 | 501 |
| 7 | Worm.Padobot.P | 497 |
| 8 | Trojan.Downloader.Delf-35 | 442 |
| 9 | Trojan.Mybot-1411 | 386 |
| 10 | Trojan.Ghostbot.A | 357 |

PE or MZ files, i.e., binaries targeting systems running Windows as operating system. This is no surprise since nepenthes currently focuses on emulating only vulnerabilities of Windows.

For the binaries we have collected, we found that about 7% of them are broken, i.e., some part of the header- or body-structure is corrupted. Further analysis showed that this is mainly caused by faulty propagation attempts. If the malware for examples spreads further with the help of TFTP (Trivial File Transfer Protocol), this transfer can be faulty since TFTP relies on the unreliable UDP protocol. Furthermore, a download can lead to a corrupted binary if the attacking station stops the infection process, e.g., because it is disconnected from the Internet.

The remaining 14,414 binaries are analyzed with different antivirus (AV) engines. Since we know that each binary tried to propagate further, we can assume that each binary is malicious. Thus a perfect AV engine should detect 100% of these samples as malicious. However, we can show that the current signature-based AV engines are far away from being perfect. Table 2 gives an overview of the results we obtained with four different AV engines. If we scan the whole set of more than 14,000 binaries, we see that the results range between 80 and 90 %, thus all AV solutions are missing a significant amount of malware. If we scan only the latest files, i.e., files that we have captured within the last 24 hours, the statistics get even worse. Table 2 gives also an overview of the detection rate for 460 unique files that were captured within 24 hours. We see that the detection rates are lower compared to the overall rate. Thus "fresh" malware is often not detected since the AV vendors do not have signatures for this new threats.

Table 3 gives an overview of the top ten malware types we collected. We obtained this results by scanning the malware samples with the free AV engine

ClamAV. In total, we could identify 642 *different* types of malware. The table shows that bots clearly dominate the samples we collect. This is mainly caused by the large number of botnets in the wild and the aggressive spreading of the individual bots. Interestingly is also the number of captured samples compared to the malware name. Please remember that we classify a samples as unique with the help of the MD5 sum. This means that 1136 different samples are detected as Worm.Padobot.M.

## 4 Future Work

In this section we want to give an overview of further work in the area of nepenthes and large-scale honeynet deployments. An extension of the nepenthes platform to support UDP-based exploits is straightforward. Most of these exploits are "single-shot" attempts that just send one UDP packet. Therefore it is only necessary to capture the payload and analyze it, we do not need to emulate any service at all. However, if the exploit requires interaction with the honeypot, we can use the same concept as for TCP-based exploits: we just emulate the necessary parts and trick the exploit.

The current nepenthes platform is another building block towards an automated system to effectively stop remote control networks. Such networks are needed by attackers to coordinate automated activity, e.g. to send commands to a large number of compromised machines. An example of such a remote control network is a botnet, i.e., a network of compromised machines that can be remotely controlled by an attacker. The whole process of stopping such a network is depicted in Figure 6. With the help of nepenthes, we can now automate step 1 to a high degree. Without supervision, this platform can collect malware that currently propagates within a network. We are currently working on step 2 - an automated mechanism to extract the sensitive information of a remote control network from a given binary. With the help of honeypots, we can automate this step to a certain degree. In addition, we explore possible ways to use sandbox-like techniques to extract this information during runtime. Thirdly, we can use static binary analysis, but it seems like this approach cannot be automated easily. Step 3 in the whole process can be automated as outlined in [6]: we impersonate as a legal victim and infiltrate the network. This allows us to study the attacker and his techniques, collect more information about other victims, or learn about new trends. Finally, step 4 can be automated to a limited degree with the help of techniques such as stooping the communication channel between victims and remote control server, or other ways to shut-down the main server itself [8]. This step also needs some further research, but it seems viable that this can also be automated to a high degree. The whole process would then allow us to automatically defend against these kind of attacks in a pro-active manner. An automated system is desirable since this kind of attacks is a growing threat within the attacker community.

We are currently in the process of deploying a network intrusion detection system (NIDS) based on nepenthes. In cooperation with SurfNET, we want to explore feasible ways of using honeypots as a new kind of IDS. The goals

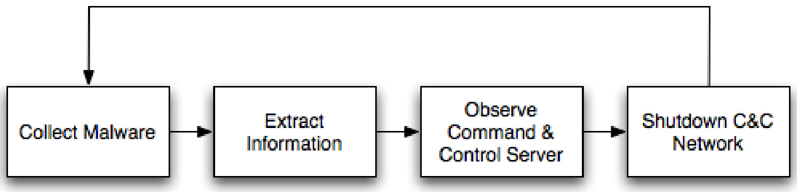


**Fig. 6.** Four steps to stop remote control networks

of this project are manifold: one the one hand, the system should enable us to understand the types and amount of malicious traffic within a LAN. In addition, it should stop spreading worms and other kinds of malware. The literature in this field shows some ways how to achieve this goal with honeypots [14]. On the other hand, the solution must be scalable and easy to manage and maintain. Zero-maintenance of the individual sensors is desirable and a missing feature of existing solutions. Our current experience shows that nepenthes scales well to a couple of thousand honeypots with just one physical machine. In addition, a hierarchical setup can be used to distribute load if an even larger setup is needed. The nepenthes platform can also scale to high-speed networks due to its limited amount of memory resource and only moderate amount of processing resources needed. Furthermore, the proposed NIDS should have close to no false positives. Up to now, we did not have any false positives with our nepenthes setup, so this goal seems to be reachable. This is mainly due to the assumption of honeypots: all network traffic is suspicious. False negatives of our platform generate a log-entry and all captured information about network traffic that could not be handled are saved. This way, all possible information to help in avoiding false negatives is already available for analysis by a human.

Finally, an empirical analysis of the effectiveness of a distributed nepenthes setup is desirable. Nepenthes offers the possibility of distributed deployment as outlined in Section 2.2 and a recent study concludes that distributed worm monitoring offers several advantages in regards to detection time [15]. Those results are obtained with the help of captured packet traces. With the help of nepenthes, the results could be verified on live data. Additionally, such a study would reveal to what degree a certain piece of malware spreads locally.

## 5 Conclusion

In this paper we introduced the nepenthes platform. This is a new kind of honeypot-based system that specializes in large-scale malware collection. Nepenthes inherits the scalability of low-interaction honeypots but at the same time offers a high degree of expressiveness. This goal is reached by emulating only the *vulnerable* parts of a service. This leads to an efficient and effective solution that offers many advantages compared to other honeypot-based solutions. The main advantage is the flexibility: an ordinary honeypot solution has to use a fixed configuration. If an incoming exploit targets another configuration, this exploit will fail. In contrast to this, one instance of nepenthes can be exploited by a wide array of exploits since nepenthes is flexible in the emulation process. It can decide during runtime which offset is the correct one to get successfully exploited. Several other factors like *virtual filesystem* and *shell emulation* contribute further to the enhanced scalability. With only one physical machine we are able to listen to more than 16,000 IP addresses in parallel.

We have collected millions of malware samples currently spreading in the wild. A further examination of more than 14,000 unique and valid binaries showed that current anti-virus engines have some limitations and fail to detect all malware propagating in the wild. Moreover, we presented some ideas how nepenthes could be used as the basic block of an automated system to stop botnets or as part of a next-generation network intrusion detection system.

## References

1. K. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. In *Proceedings of the 14th USENIX Security Symposium*, 2005.
2. Michael Bailey, Evan Cooke, Farnam Jahanian, Jose Nazario, and David Watson. The Internet Motion Sensor: A Distributed Blackhole Monitoring System. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 05)*, 2005.
3. Edward Balas and Camilo Viecco. Towards a Third Generation Data Capture Architecture for Honeynets. In *Proceeedings of the 6th IEEE Information Assurance Workshop*, West Point, 2005. IEEE.
4. Team Cymru: The Darknet Project. Internet: `http://www.cymru.com/Darknet/`, Accessed: 2006.
5. David Dagon, Cliff Zou, and Wenke Lee. Modeling Botnet Propagation Using Time Zones. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS 06)*, 2006.
6. Felix Freiling, Thorsten Holz, and Georg Wicherski. Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks. In *10th European Symposium On Research In Computer Security, ESORICS05, Milano, Italy, September 12-14, 2005, Proceedings*, Lecture Notes in Computer Science. Springer, 2005.
7. Thorsten Holz. A Short Visit to the Bot Zoo. *IEEE Security & Privacy*, 3(3):76–79, 2005.
8. Thorsten Holz. Spying With Bots. *USENIX ;login:*, 30(6):18–23, 2005.
9. Xuxian Jiang and Dongyan Xu. Collapsar: A vm-based architecture for network attack detention center. In *Proceedings of 13th USENIX Security Symposium*, 2004.
10. Bill McCarty. Automated Identity Theft. *IEEE Security & Privacy*, 1(5):89–92, 2003.
11. David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage. Network Telescopes. Technical Report TR-2004-04, CAIDA, 2004.
12. David Moore, Geoffrey M. Voelker, and Stefan Savage. Inferring Internet Denial-of-Service Activity. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.
13. Georgios Portokalidis. Argos: An Emulator for Capturing Zero-Day Attacks. Internet: `http://www.few.vu.nl/~porto/argos/`, Accessed: 2006.

14. Niels Provos. A Virtual Honeypot Framework. In *Proceedings of 13th USENIX Security Symposium*, pages 1–14, 2004.
15. Moheeb Abu Rajab and Andreas Terzis. On the Effectiveness of Distributed Worm Monitoring. In *Proceedings of the 14th USENIX Security Symposium*, 2005.
16. Yoichi Shinoda, Ko Ikai, and Motomu Itoh. Vulnerabilities of Passive Internet Threat Monitors. In *Proceedings of the 14th USENIX Security Symposium*, 2005.
17. Stuart Staniford, David Moore, Vern Paxson, and Nicholas Weaver. The Top Speed of Flash Worms. In *ACM Workshop on Rapid Malcode (WORM)*, 2004.
18. Symantec. Mantrap. Internet: `http://www.symantec.com/`, Accessed: 2006.
19. Nicolas Vanderavero, Xavier Brouckaert, Olivier Bonaventure, and Baudouin Le Charlier. The HoneyTank : a scalable approach to collect malicious Internet traffic. In *Proceedings of the International Infrastructure Survivability Workshop*, 2004.
20. Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2005.
21. Kathy Wang. Honeyclient. Internet: `http://honeyclient.org`, Accessed: 2006.
22. Yi-Min Wang, Doug Beck, Chad Verbowski, Shuo Chen, Sam King, Xuxian Jiang, and Roussi Roussev. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the 13th Network and Distributed System Security Symposium (NDSS 06)*, 2006.
23. Vinod Yegneswaran, Paul Barford, and Dave Plonka. On the Design and Use of Internet Sinks for Network Abuse Monitoring. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.