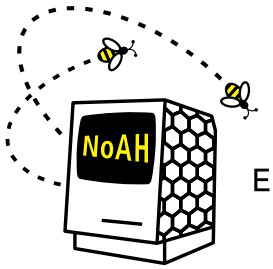
SIXTH FRAMEWORK PROGRAMME

Structuring the European Research Area Specific Programme

RESEARCH INFRASTRUCTURES ACTION



European Network of Affined Honeypots

Contract No. RIDS-011923

D1.3 Containment Environment Design

Abstract: This document describes the design of the NoAH containment environment. The environment, known as Argos, is intended for use in a high-interaction honeypot that runs real services on any operating system. Unlike most other systems, we do not require the traffic arriving at the honeypot to be suspect to begin with, as Argos is designed to detect zero-day attacks.

Contractual Date of Delivery	M12
Actual Date of Delivery	M13
Deliverable Security Class	Public

The NoAH Consortium consists of:

FORTH	Coordinator	Greece
VU	Principal Contractor	The Netherlands
TERENA	Principal Contractor	The Netherlands
FORTHnet	Principal Contractor	Greece
DFN-CERT	Principal Contractor	Germany
ETHZ	Principal Contractor	Switzerland
VTRIP	Principal Contractor	Greece
ALCATEL	Principal Contractor	France

Contents

1 Introduction **7**
1.1 NoAH in a nutshell 7
1.2 Overview 7

2 Argos: the NoAH Containment Environment **9**

3 Argos Design **13**
3.1 Tagging, tracking and triggering 13
3.2 Signature generation and use 15
3.2.1 A simple signature generator for NoAH Containment . . . 16
3.2.2 More sophisticated signature generators 17
3.2.3 Experience with the use of signatures 17
3.3 Discussion about signature generation methods 17

4 Prototype evaluation **19**
4.1 Performance 19
4.2 Effectiveness 21
4.3 Detailed evaluation 22

5 Summary **27**

List of Figures

3.1 Argos: High-Level Overview. (1) Network data arrives and is both logged and sent to the emulator. (2) The emulator tags data from the network as tainted. (3) An alert is triggered if tainted data is used in ways that violate the security policy and forensics about the process under attacks is logged. (4) The memory log, forensics data, and network trace are correlated and (5) a signature is generated. 14

4.1 Performance Benchmarks 20
4.2 Vulnerable program "`vuln1.c`" 23
4.3 Vulnerable program "`vuln2.c`" 24
4.4 Vulnerable program "`vuln3.c`" 25

Chapter 1

Introduction

The NoAH project proposes to use a network of honeypots to (i) intercept zero-day attacks, and (ii) generate signatures for such attacks. It explicitly intends to protect multiple operating systems (OSs) and all applications without modification. The implication is that source code should not be required. Indeed, NoAH intends to protects any OS and any application in a transparent fashion.

1.1 NoAH in a nutshell

The NoAH architecture and the project's approach consist of a number of (conceptually) centralised honeypot farms that receive traffic from networks that are associated with NoAH. These networks contain redirectors that obtain one or more IP addresses in their local domain and tunnel all traffic to these IP addresses to the honeypots. Conceptually closer to the core of the NoAH infrastructure, low-interaction honeypots (LI-HPs) will form a first-pass filter for the core. All traffic that can be handled by the LI-HPs will not be forwarded to the core.

The core itself consists of farms of high-interaction honeypots (HI-HPs) running real services (such as webservices, databases, etc) that engage in real interaction with the outside world. The core is heavily instrumented with an eye on detecting sophisticated zero-day exploits as they occur. The core is made up out of various components, such as firewalls with white lists and black lists, reverse firewalls that prevent attacks originating in the core to spread to the outside world, and so on. The central component of the core is what we have termed the *containment environment* in the NoAH project. The containment environment is the run time system that runs the vulnerable operating system and its services.

1.2 Overview

In this document we describe the design of the NoAH Containment Environment which has been coined Argos. Significant sections of this deliverable are based on

a paper about NoAH Containment that was published in the Eurosys 2006 conference [11]. We show how we plan to provide full system protection for any OS and any application. We also explain our plans for attaching the containment environment to signature generators and other components.

NoAH Containment is developed in collaboration with the Dutch NWO/STW/EZ Sentinels project known as 'DeWorm'. Since the projects share some goals and one of the NoAH partners participates in both projects, we had the opportunity to (1) broaden the scope of the containment environment, and (2) increase the impact of the NoAH architecture.

The remainder of this document is organised as follows. In Chapter 3 we explain the background and motivation that led to the NoAH Containment design. In Chapter 3 we describe the design of NoAH Containment in more detail and we evaluate a prototype implementation in Chapter 4. A summary can be found in Chapter 5.

Chapter 2

Argos: the NoAH Containment Environment

The rate at which self-propagating attacks spread across the Internet has prompted a wealth of research in automated response systems. We have already encountered worms that spread across the Internet in as little as ten minutes, and researchers claim that even faster worms are just around the corner [15]. For such outbreaks human intervention is too slow and automated response systems are needed. Important criteria for such systems in practice are: (a) reliable *detection* of a wide variety of zero-day attacks, (b) reliable generation of *signatures* that can be used to stop the attacks, and (c) *cost-effective* deployment.

Existing automated response systems tend to incur a fairly large ratio of false positives in attack detection and use of signatures [17, 16, 5, 12, 7]. A large share of false positives violates the first two criteria. Although these systems may play an important role in intrusion detection systems (IDS), it is problematic to apply them in fully automated response systems.

One approach that attempts to avoid false positives altogether is known as dynamic taint analysis. Briefly, untrusted data from the network is tagged and an alert is generated (only) if and when an exploit takes place, e.g., when data coming from the network are executed. This technique proves to be very reliable and to generate few, if any, false positives. It is used in current projects that can be categorised as (i) hardware-oriented full-system protection, and (ii) OS- and process-specific solutions in software. These are two rather different approaches, and each approach has important implications. For our purposes, the two most important representatives of these approaches (and the most closely related to NoAH Containment) are Minos [2] and Vigilante [1], respectively. While we single out these projects to high-light some of the design decisions of NoAH Containment, we emphasise that they represent a broader category of projects.

Minos is a hardware approach to IDS that detects misuse of untrusted data throughout the entire system, including the kernel and user processes. It is currently unable to generate ready-to-use signatures, and for cost-effective deploy-

ment it relies on implementation in hardware, although an evaluation version is implemented on the Bochs emulator. A major disadvantage of Minos regarding the issues of detection and signature generation (the first two criteria) is that its design as a hardware solution limits its flexibility. For instance, unable to look beyond physical addresses, it may *detect* certain exploits, such as a register spring attacks [14], but requires an awkward hack to even determine where the attack originated [3], and cannot directly handle physical to virtual address translation at all. This is not a trivial artefact that only shows up in register spring attacks, but a fundamental problem that occurs whenever virtual addresses are needed.

In a sense, Vigilante is the complete opposite of Minos as it represents a per-process solution that works with virtual addresses. Again, this is a design decision that limits its flexibility. In the detection and signature generation phase, for instance, Vigilante is unable to deal with complex memory operations such as DMA and memory mapping, which is due to its choice for virtual addresses and per-process protection. To a lesser extent, the issue of cost-effectiveness also exists for Vigilante, which requires instrumentation and modification of individual services and does not protect the OS kernel at all. Unfortunately, kernel attacks have become a reality and are expected to be more common in the future [6]. For signature generation Vigilante relies on replaying the attack which is often not possible due to challenge/response interaction with nonces, random numbers, etc.

It is not our intention to dismiss either of these solutions out of hand. On the contrary, our design was inspired by both projects and we consider them important milestones in the design of automated response systems. However, we do believe that they are too limited in all three aspects mentioned in the beginning of this section. It is our design to present a third approach that combines the best of both worlds and meets all of the criteria.

In this document we propose NoAH Containment which will explore an extreme in the design space for automated response systems. First, like Minos we will offer whole-system protection in software by way of a modified x86 emulator which runs our own version of dynamic taint analysis [8]. In other words, we automatically protect any (unmodified) OS and all its processes, drivers, etc. Second, NoAH Containment will take into account complex memory operations, such as memory mapping and DMA (commonly ignored by other projects), and will at the same time be capable of handling complex exploits (such as register springs). This is to a large extent due to our ability to handle both virtual and physical addresses. Third, buffer overflow and format string / code injection exploits will trigger alerts that result in the automatic generation of signatures based on the correlation of the exploit's memory footprint and its network trace. While this document is not about signature generation *per se*, we will sketch an approach in the design chapter. Fourth, while the system is designed to be OS- and application-neutral, when an attack is detected, we can inject OS-specific forensics shellcode. In other words, we can exploit the code under attack as the attack is happening to extract additional information about the attack which is subsequently used in signature generation.

True to the goals of NoAH, we focus on attacks that are orchestrated remotely (like worms) and do not require user interaction. Approaches that take advantage of misconfigured security policies are not addressed. Even though such attacks constitute an ample security issue, they are beyond the scope of our work and require a different approach. Specifically, we will focus on *exploits* rather than attack *payloads*, i.e., we capture the code that triggers buffer overflows and injects code in order to gain control over the machine, and not the behaviour of the attack once it is in. In our opinion, it is more useful to catch and block exploits, because without the exploits the actual attack will never be executed. Moreover, in practice the same exploit is often used with different payloads, so the pay-off for stopping the exploit is potentially large[1]. In addition, exploits are less mutable than attack payloads and may be more easily caught even in the face of polymorphism.

NoAH Containment is designed as an 'advertised honeypot', i.e., a honeypot that runs real services and differs from normal honeypots in that we don't need to hide it. Rather, we may actively link to it and 'advertise' its IP address(es) in the hope of making it visible to attackers employing hitlists rather than random IP scanning to identify victims. The price we pay for this is that unlike conventional honeypots we expect to receive a fair amount of legitimate traffic (e.g., crawlers). On the other hand, since NoAH Containment is targeted as a honeypot, we do not require our solution to perform as well as unprotected systems. Also, the low-interaction honeypots may filter out a large portion of the traffic containing known attacks. Nevertheless, it should be fast enough to run real services and have reasonable response time.

[1] Nevertheless, we do dump the entire attack to file for manual analysis.

Chapter 3

Argos Design

Argos, being the containment environment in the NoAH project sits at the core of the NoAH infrastructure. It is used to run high-interaction honeypots in the larger NoAH architecture as discussed in deliverable D1.1. In this chapter, we will discuss some details of the NoAH Containment design, while the next chapter presents some preliminary results obtained with an Argos prototype.

An overview of the NoAH Containment architecture is shown in Figure 3.1. The full execution path consists of five main steps, indicated by the numbers in the figure which correspond to the circled numbers in this section. Incoming traffic will be both logged in a trace database, and fed to the unmodified application/OS running on our emulator ①. In the emulator we employ dynamic taint analysis to detect when a vulnerability is exploited to alter an application's control flow ②. This will be achieved by identifying illegal uses of possibly unsafe data such as the data received from the network [8]. There are three steps to accomplish this:

- tag data originating from an unsafe source as tainted;
- track tainted data during execution
- identify and prevent unsafe usage of tainted data;

3.1 Tagging, tracking and triggering

Data originating from the network is marked as tainted, whenever it is copied to memory or registers, the new location is tainted also, and whenever it is used, say, as a jump target, we raise an alarm. Thus far this is similar to approaches like [1] and [8]. As mentioned earlier, NoAH Containment differs from most existing projects in that we trace physical addresses rather than virtual addresses. As a result, the memory mapping problem disappears, because all virtual address space mappings of a certain page, refer to the same physical address.

There are various options for determining which behaviour should trigger an alert. In NoAH Containment we intend to opt for illegitimate use of tainted data,

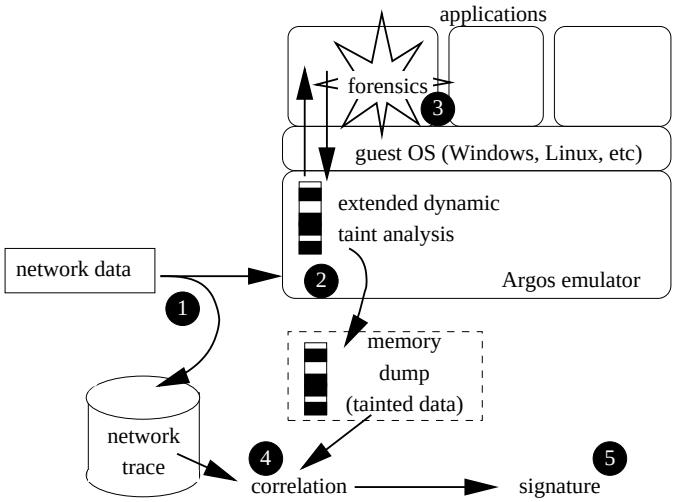


Figure 3.1: Argos: High-Level Overview. (1) Network data arrives and is both logged and sent to the emulator. (2) The emulator tags data from the network as tainted. (3) An alert is triggered if tainted data is used in ways that violate the security policy and forensics about the process under attacks is logged. (4) The memory log, forensics data, and network trace are correlated and (5) a signature is generated.

including loading the processor's program counter register (EIP) with a tainted value, or loading EIP with an address whose contents are tainted. Unsafe usage of tainted data is prevented because Argos detects the exploit before it can take control of the vulnerable program. We also plan to check the arguments of some system calls for use of tainted data and may add other methods later. When a violation is detected, an alarm will be raised which leads to a signature generation phase ③-⑤. To aid signature generation, NoAH Containment will first dump all tainted blocks and some additional information to file, with markers specifying the address that triggered the violation, the memory area it was pointing to, etc. Since we have full access to the machine, its registers and all its mappings, we will be able to translate between physical and virtual addresses as needed. The dump therefore contains registers, physical memory blocks and specific virtual address, as explained later, and in fact contains enough information not just for signature generation, but for, say, manual analysis as well.

In addition, we plan to employ a novel technique to automate forensics on the code under attack. Recall that NoAH Containment is OS- and application-neutral, i.e., we are able to work out-of-the-box with any OS and application on the IA32 instruction set architecture (no modification or recompilation required). When an attack is detected, we may not even know which process is causing the alarm. To unearth additional information about the application (e.g., process identifier, executable name, open files and sockets, etc.), we plan to inject our own shellcode to perform forensics ③. In other words, we 'exploit' the code under attack with

our own shellcode. Obviously there is no need to use a real exploit in the form of network packets to inject the forensics shellcode in the address space of the process under attack. Instead, we simply copy a small program into a read-only page of the process. So rather than trying to cleverly exploit some vulnerability in the process, we simply inject the code we want executed in the process' execution environment 'like it or not'.

We emphasise that even without the shellcode, which by its nature may contain OS-specific features, NoAH Containment still works, albeit with reduced accuracy. In our opinion, an OS-neutral framework with OS-specific extensions to improve performance is a powerful model, as it permits a generic solution without necessarily paying the price in terms of performance or accuracy. To the best of our knowledge, we are the first to propose employing the means of attack (shellcode) for defensive purposes.

The dump of the memory blocks (tainted data, registers, etc.) plus the additional information obtained by our shellcode will then be used for correlation with the network traces in the trace database ④. In case of TCP connections, we will have to reconstruct flows prior to correlation. We will use the correlation to generate signatures. We should observe that many different signature generation can be plugged in the NoAH Containment back-end. We will describe one such engine in later sections, but stress that this is a rather simplistic approach. In the remainder of this project we plan to develop new methods for generating signatures. In particular, we aim to cater to zero-day polymorphic attacks.

3.2 Signature generation and use

While slightly beyond the scope of this deliverable, this section briefly discusses signature generation. The reason for this is that it shows the sort of information that needs to be provided by the containment environment.

The signature generator may be described as NoAH Containment' back-end. By designing the system in a modular way, we should be able to experiment with different types of back-end. We have already experimented with signature generators and consider the design of a reliable signature generator for polymorphic, zero-day attacks as one of the most challenging aspects of the NoAH project.

Note that simple signature generators are not new. For instance, in the *honeycomb* project, a rather crude signature generator sits on top of the *honeyd* low-interaction honeypot. It scans different instances of network traffic arriving at the honeypot for longest common substrings (LCSs) and uses the LCSs as signatures. While an interesting attempt, the method can only be described as crude and inaccurate.

The SweetBait system developed at the Vrije Universiteit Amsterdam is an attempt to refine such LCS signature generation method [10]. It also builds on the *honeyd/honeycomb* combination, but with important differences. First, it employs white-listing of known traffic to avoid fingerprinting benign traffic. Second, sig-

natures that are generated at different hosts are compared and if there is sufficient overlap they are combined, creating a smaller, more accurate signature. SweetBait on top of the *honeyd/honeycomb* combination generates Snort rules that have shown to be quite powerful. Nevertheless, we do not believe crude measures based on LCS of different streams is appropriate for modern attacks.

Even so, SweetBait is a useful tool for refining signatures based on patterns in a more generic setting. For instance, below we will sketch how SweetBait was also applied to a signature generator for NoAH Containment to refine generator's initial signatures.

3.2.1 A simple signature generator for NoAH Containment

Given NoAH Containment, a simple way for generating signatures is to match the attack's memory footprint against the network trace. In an experimental implementation we have used the address that was used to attack (e.g., the address that overwrites the `RET` address in a stack buffer overflow) as a key to search for in the traffic stream. When the address is found in the stream, we scan the bytes before and after the address (both in memory and in the network trace) to find the LCS between the memory footprint and the network trace. This process eventually yields the best match between memory and network trace that contains the address. In NoAH Containment, this simple signature generator is known as CREST. Note that CREST is clearly inspired by the LCS work in *honeycomb*.

The signature generator produces Snort-like signatures based on the LCS that are, in principle, ready to be used for filtering. However, signature generation on this basis is rather simplistic and it will often be the case that the signature is not optimal (e.g., too long). We therefore try to improve it using the same techniques as sketched earlier. In essence, NoAH Containment also submits the signatures to the SweetBait subsystem, which correlates signatures from different sites, and refines signatures based on similarity [9]. For instance, a signature consisting of the exploit plus the IP address of the infected host, would look slightly different at different sites. SweetBait will notice the resemblance between two such signatures, and generate a shorter, more specialised signature that is then used in subsequent filtering.

It should be mentioned that refinement in this case is more likely to yield good results than in the *honeyd/honeycomb* combination, for the following reason. In *honeyd/honeycomb* we only have the LCS as a basis for a comparison. As a result, two signatures really must look very similar indeed, before we conclude that they correspond to the same attack. For NoAH Containment signatures, we can be much more relaxed as we are able to just compare signatures with the same 'key' value (e.g., the address that was used in an overflow). As all attacks using the same key value are almost certain to be the same exploit, we can be much more aggressive in finding similarity between two signatures.

We have experimented with this approach and the results are promising for a large class of attacks. Still, in our opinion, the signature generation process out-

lined above is only suited for less sophisticated, monomorphic attacks. Something better is needed to handle polymorphic, zero-day attacks. Indeed, we emphasise again that there are many ways to generate signatures and we intend to keep NoAH Containment modular and extensible.

3.2.2 More sophisticated signature generators

The LCS methods are fairly crude and powerless in the face of encrypted attacks[1]. However, what we learn is that we also want methods that do not rely on exact matches. While beyond the scope of this paper, we speculate that protocol-specific signatures that look at anomalies in the protocol fields might handle polymorphism without incurring too many false positives. For instance, if we knew that an attack occurred in the URL field of an HTTP GET request, we could look at the length of the URL, or the byte distribution. This leads to a hard requirement for NoAH Containment: it should be able to determine exactly where an attack occurred. In the section 3.3, we will list some more requirements.

3.2.3 Experience with the use of signatures

The final step is the automated use of the signatures. In this section we describe some of the experiments we conducted in this area to get a feel for what is required. It does not talk about the design in detail.

Attached to NoAH Containment (and or SweetBait) are intrusion detection and prevention systems (IDS and IPS), that NoAH Containment (and or SweetBait) provide with signatures of traffic to block or track. As a first stab at IDS we have already experimented with sensors based on the well-known open source network IDS *Snort* [13] and for this purpose, SweetBait generates rules in *Snort* rule format. We have also experimented with an IPS which is a relatively simple homegrown solution that employs the Aho-Corasick pattern matching algorithm to match network signatures. Although not very sophisticated, we have implemented it as a Linux kernel module that can be used directly with SweetBait. In a separate effort, one of the authors developed a high-speed version of the IPS on Intel IXP1200 network processors that could be used as an alternative [4]. The current version of SweetBait is intelligent in the sense that it distinguishes between virulent attacks (e.g., many incidence reports) and rare events, and circulates the signatures accordingly. This is analogous to the way in which the police puts out APBs for dangerous criminals rather than for, say, pickpockets.

3.3 Discussion about signature generation methods

The problem with the methods for signature generation and use of the signatures with which we have experimented so far is that they are fairly limited in what

[1] Note, however, that polymorphism in exploits is rare and by necessity not too complex.

they can be used for. For instance, because we search for exact matches, they are powerless in the face of polymorphic attacks which are likely to be more common in the future. So, while we have already built and evaluated SweetBait we think that neither its signature generation method, nor indeed the Snort-like signatures are very future proof. The focus of this deliverable is primarily on steps ①-④ of Figure 3.1 and we will therefore try to draw up some requirements based on our experience with the crude signature generators.

In the NoAH project we will develop new methods for stopping more complex attacks than what is possible with the SweetBait/Snort-like approach. The exercise in signature generation has been quite useful, however, in determining the information that should be produced by NoAH Containment for successful signature generation. In our opinion, we should strive to make NoAH Containment provide the following information:

1. Name and version of the operating system;
2. Name of the application under attack (version number can then also be retrieved);
3. Names of the dynamic libraries this application links against;
4. Attacked port and ideally the exact stream under attack;
5. All tainted memory pages for the process under attack;
6. The address used in the attack;
7. A dump of the data at this address;
8. A dump of the pages where the address originates.
9. Ideally, an exact mapping of each tainted word in memory to offsets in the network trace.

Chapter 4

Prototype evaluation

While Argos is very much a work in progress, we did implement a NoAH Containment prototype on top of the Qemu open source emulator[1]. In this section we will briefly discuss some preliminary results with the prototype. A detailed discussion of the prototype will be given in deliverable D2.1, while a more thorough evaluation is postponed until Work Package 3.

We evaluate NoAH Containment along two dimensions: performance and effectiveness. While performance is not critical for a honeypot, it needs to be fast enough to generate signatures in a timely fashion.

4.1 Performance

Performance evaluation was carried out by comparing the observed slowdown at guests running on top of various configurations of Argos and unmodified Qemu, with the original host. The host used during these experiments was an AMD Athlon™ XP 2800 at 2 GHz with 512 KB of L2 cache, 1 GB of RAM and 2 IDE UDMA-5 hard disks, running Gentoo Linux with kernel 2.6.12.5. The guest OS ran SlackWare Linux 10.1 with kernel 2.4.29, on top of Qemu 0.7.2 and Argos. To ameliorate the guest's disk I/O performance, we did not use a file as a hard disk image, but instead dedicated one of the hard disks.

To quantify the observed slowdown we used `bunzip2` and `apache`. `bunzip2` is a very CPU intensive UNIX decompression utility. We used it to decompress the Linux kernel v2.6.13 source code (approx. 38 MB) and measured its execution time using another UNIX utility `time`. `Apache`, on the other hand, is a popular web server that we chose because it enables us to test the performance of a network service. We measured its throughput in terms of maximum processed requests per second using the `httperf` HTTP performance tool. `httperf` is able to generate high rates of single file requests to determine a web server's maximum capacity.

In addition to the above, we used BYTE magazine's UNIX benchmark. This benchmark, `nbench` for brevity, executes various CPU intensive tests to produce

[1] `http://fabrice.bellard.free.fr/qemu/`

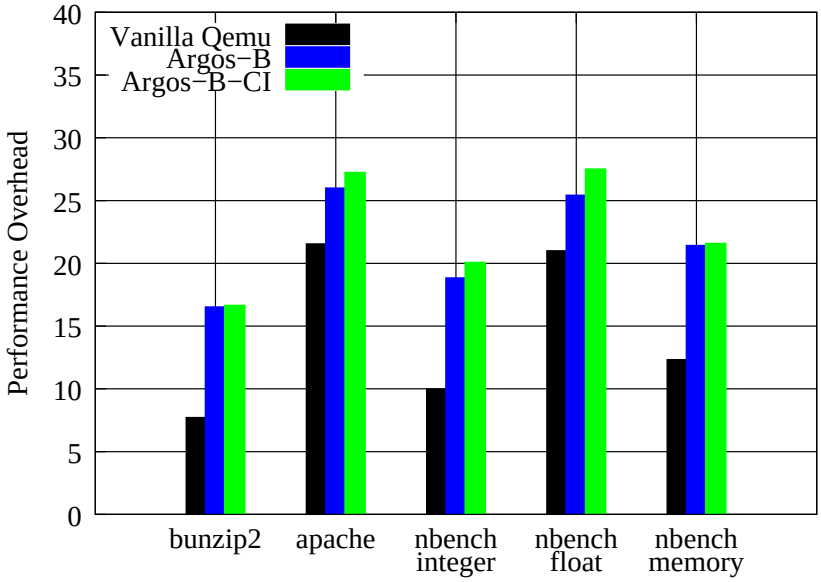


Figure 4.1: Performance Benchmarks

Configuration	Served Requests/Sec.
Native	499.9
Vanilla Qemu	23.3
Argos-B	18.7
Argos-B-CI	18.3

Table 4.1: Apache Throughput

three indexes. Each index corresponds to the CPU's integer, float and memory operations and represents how it compares with an AMD K6™ at 233 MHz.

Figure 4.1 shows the results of the evaluation. We tested the benchmark applications at the host, at guests running over the original Qemu, and at different configurations of NoAH Containment: using a bytemap, and using a bytemap with code-injection detection enabled. These are indicated in the figure as Vanilla QEMU, Argos-B, and ARGOS-B-CI respectively. The y-axis represents how many times slower a test was, compared with the same test without emulation. The x-axis shows the 2 applications tested along with the 3 indexes reported by `nbench`. Each colour in the graph is a configuration tested, which from top to bottom are: unmodified Qemu, NoAH Containment using a bytemap for memory tagging, and the same with code-injection detection enabled. `Apache` throughput in requests served per second is also displayed in Table 4.1.

Even in the fastest configuration, NoAH Containment is at least 16 times slower than the host. Most of the overhead, however, is incurred by Qemu itself. NoAH Containment with all the additional instrumentation is at most 2 times slower than vanilla Qemu. In the case of `apache` and float operations specifically, there is only an 18% overhead. This is explained by the lack of a real network interface, and a hardware FPU in the emulator, which incurs most of the overhead. In addition, we emphasise that we have not used any of the optimisation modules available for Qemu. These modules speed up the emulator to a performance of roughly half that of the native system. While it is likely that we will not quite achieve an equally large speed-up, we are confident that much optimisation is possible.

4.2 Effectiveness

To determine how effective NoAH Containment is in capturing attacks, we launched multiple exploits against both Windows and Linux operating systems running on top of it. For the Windows 2000 OS, we used the Metasploit framework[2], which provides ready-to-use exploits, along with a convenient way to launch them. We tested *all* exploits for which we were able to obtain the software. In particular, all the attacks were performed against vulnerabilities in software available with the professional version of the OS, with the exception of the War-FTPD ftp server which is third-party software. While we have also successfully run other OSs on NoAH Containment (e.g., Windows XP), we have only just started its evaluation. For the Linux OS, we crafted two applications containing a stack and a heap buffer overflow respectively and also used `nbSMTP`, an SMTP client that contains a remote format string vulnerability that we attacked using a publicly available exploit.

A list of some the tested exploits along with the underlying OS and their associated worms is shown in table 4.2. For Windows, we have only listed fairly well-known exploits. All exploits were successfully captured by NoAH Containment and the attacked processes were consequently stopped to prevent the exploit payloads from executing.

Observe that Argos even managed to detect the fairly complex WMF exploit (an exploit, for which it proved quite difficult to generate a good Snort rule). This vulnerability is critical because false alarms might be expected due to the nature of this vulnerability. This is because the WMF image format is basically a container comprised of references of functions and their arguments. A WMF image is rendered by the execution of these function calls which results for instance in displaying a bitmap or drawing a line. A false alarm might be raised because of inappropriate tagging of image data containing either the references of functions or their arguments. However, our test image has been displayed without problems in the Argos containment environment. In addition, an alarm was raised correctly when using the WMF image that contained the exploit data.

[2]The Metasploit Project `http://www.metasploit.com/`

Finally, we should mention that during the performance evaluation, as well as the preparation of attacks, NoAH Containment did not generate any false alarms about an attack. A low number of false positives is crucial for automated response systems. Even though the results do not undeniably prove that NoAH Containment will *never* generate false positives, considering the large number of exploits tested, it may serve as an indication that NoAH Containment is fairly reliable. For this reason, we decided for the time being to use the signatures as is, rather than generating self-certifying alerts (SCAs [1]). However, in case we incur false positives in the future, NoAH Containment is quite suitable for generating SCAs.

4.3 Detailed evaluation

In this section the ability of Argos to detect exploits for buffer overflow vulnerabilities is demonstrated. This class of vulnerabilities is of special interest because corresponding zero-day exploits can be expected to be more severe in comparison to other classes of vulnerabilities, including the recent code injection vulnerabilities in PHP applications.

Buffer overflow vulnerabilities have shown to be the majority of vulnerabilities reported in the security advisories issued by the DFN-CERT in 2004 and 2005 and affect all common operating systems. Since the core components (e.g., the kernel) of all common operating systems are written in C or C++ a zero-day exploit for buffer overflows and related vulnerabilities affects commonly a very large number of systems at the same moment. This is in contrast to the aforementioned vulnerabilities in PHP applications that allows to overwrite critical variables or to include code. In comparison to the Code-Red, Blaster and Slammer worms (all of whom use buffer overflow exploits) the number of infected systems by worms abusing PHP vulnerabilities (e.g., the Santy worm and variants) and the overall damage was much lower.

For testing the Argos containment environment DFN-CERT handcrafted several C programs which are designed to simulate typical buffer overflow vulnerabilities. For providing these programs with data sent to the Argos environment we use the ”netcat” tool which listened on a specific TCP port. The received data was sent to the programs by using an Unix pipe. All programs are derived from the `vuln.c` program shown in Figure 4.2. The programs all contain a buffer overflow vulnerability.

Stack-based buffer overflows affecting the array `overflow_me` can be caused in the programs `vuln1.c`-`vuln3.c` in Figures 4.2-4.4. These vulnerabilities could be exploited to modify the execution control flow of the vulnerable program and exploitation should be detected by the containment environment.

The vulnerability in the program `vuln1.c` is caused by the insecure use of the strcpy() function which does not care about array boundaries. This function is directly or indirectly involved in a very large number of buffer overflow vulnerabilities. By overflowing the buffer in `overflow_me` the return address of this

```
#include <stdio.h>

int
main(void)
{
    char overflow_me[16];
    static char buffer[64];
    int c;

    int i = 0;
    while ( ( c = getc(stdin)) != EOF) {
        buffer[i] = c; // unsafe - not checked for bounds
        i++;
    }
    buffer[i + 1] = (char) 0;

    strcpy( overflow_me, buffer ); // unsafe - not checked for bounds
    printf("input = '%s' \n", overflow_me);
    return 0;
}
```

Figure 4.2: Vulnerable program "`vuln1.c`"

```
#include <stdio.h>

int
main(void)
{
    char overflow_me[16];
    static char buffer[64];
    int c;

    int i = 0;
    while ( ( c = getc(stdin)) != EOF) {
        buffer[i] = c + (char) 1;
        i++;
    }
    buffer[i + 1] = (char) 0;

    strcpy( overflow_me, buffer );
    printf("input = '%s' \n", overflow_me);
    return 0;
}
```

Figure 4.3: Vulnerable program "`vuln2.c`"

function can be overwritten by an arbitrary value, e.g., to redirect the execution control flow to injected shell code. As expected, Argos raised an alert in which the EIP contained tainted input data.

The programs `vuln2.c` and `vuln3.c` (Figures 4.3 and 4.4, respectively) were intended to simulate the encoding or decoding of input data. This is done for instance by common web-servers to encode the URI of the received HTTP GET request. It has been shown that Argos was able to detect the encoded exploit data with which the EIP has been overwritten. Only the more complex decoding function in `vuln3.c` modified the input data in such a way that the tagging of the data got lost. As a consequence Argos was not able to raise an alarm concerning the overwritten EIP. However, even if Argos is not able to detect the first step of the attack directly it is unlikely that further steps would be remain undetected. This is because almost all exploits rely on executing injected shell code. In addition, if the attacker is not able to control all bytes of the EIP (e.g., if the input 8-bit data is encoded in the UTF 16-bit characterset) alternative exploit methods that do not need to execute shell code (e.g., return-into-lib-c exploits) are very unlikely successful.

```
#include <stdio.h>

int
main(void)
{
    char overflow_me[16];
    static char buffer[64];
    char c;
    int tc;

    int i = 0;
    static char ascii[256];

    for ( i = 0; i < 256; i++) {
        ascii[i] = (char) ((i / 4) + 65);
    }

    i = 0;
    while ( ( c = getc(stdin)) != EOF) {
        tc = (int) c ^ ascii[c];
        if (tc < 65) tc += 65;
        if ( tc > 128) tc -= 128;
        buffer[i] = (char) tc;
        buffer[i + 1] = 'a';
        i += 2;
    }
    buffer[i + 1] = (char) 0;

    strcpy( overflow_me, buffer );
    printf("input = '%s' \n", overflow_me);
    return 0;
}
```

Figure 4.4: Vulnerable program "`vuln3.c`"

Vulnerability	**OS**
Apache Chunked Encoding Overflow (`Scalper`)	Windows 2000
Microsoft IIS ISAPI .printer Extension Host Header Overflow (`sadminD/IIS`)	Windows 2000
Microsoft Windows WebDav ntdll.dll Overflow (`Welchia`, `Poxdar`)	Windows 2000
Microsoft FrontPage Server Extensions Debug Overflow (`Poxdar`)	Windows 2000
Microsoft LSASS MS04-011 Overflow (`Sasser`, `Gaobot.ali`, `Poxdar`)	Windows 2000
Microsoft Windows PnP Service Remote Overflow (`Zotob`, `Wallz`)	Windows 2000
Microsoft ASN.1 Library Bitstring Heap Overflow (`Zotob`, `Sdbot`)	Windows 2000
Microsoft Windows Message Queueing Remote Overflow (`Zotob`)	Windows 2000
Microsoft Windows RPC DCOM Interface Overflow (`Blaster`, `Welchia`, `Mytob-CF`, `Dopbot-A`, `Poxdar`)	Windows 2000
War-FTPD 1.65 USER Overflow	Windows 2000
WMF exploit	Windows 2000, XP and others
nbSMTP v0.99 remote format string exploit	Linux 2.4.29
Custom Stack Overflow	Linux 2.4.29
Custom Heap Corruption Overflow	Linux 2.4.29

Table 4.2: Some exploits Captured by Argos

Chapter 5

Summary

In this deliverable we have described NoAH Containment, an emulator for detecting zero-day attacks of worms. It is intended to form the heart of the Noah infrastructure. Since its inception prototypes of NoAH Containment have been released which have been enthusiastically picked both by the academic community and industry. The response so far has been very positive.

NoAH Containment is founded on two pillars: (efficient) emulation and dynamic taint analysis. The NoAH Containment emulator is a modified version of an open source emulator known as Qemu which essentially performs just-in-time compilation of binary code with caching of translated blocks. Performance is quite reasonable (although a far cry from native performance). NoAH Containment essentially augments the emulator with tagging and taint analysis. In other words, it keeps track of what data originates in the network. Such data is tagged as tainted. Whenever such memory is copied,, or used in computations, the destination is also tainted. It triggers an alert when tainted data is used in a way that violates security policies. For instance, any attempt to use tainted data as a jump target leads to an intrusion alert, which causes a memory dump to be made of all tainted blocks that belong to the violating process.

The dumps are exceedingly useful for post-mortem analysis. Moreover, the design aims at elaborate forensics when an attack is detected. For instance, it injects forensics shellcode into the process under attack to discover the process identity, the ports and protocols on which it is listening, etc. This is invaluable information for fingerprinting the attack (be it manually or automated).

Besides the memory dumps NoAH Containment also logs all network traffic, with an eye on permitting correlation of network traffic and memory logs. The ultimate goal is to auto-generate signatures that can be used to block attacks in other sites. Some early attempts in this directions are described in a paper that is presented at the Eurosys conference in Leuven in April 2006.

Finally, some initial experiments with signature generation have provided us with a useful list of requirements for the NoAH Containment containment environment. While signature generation is the topic of a separate deliverable, the

experiments are important for the design of the containment environment as they provide fairly hard requirements for the architecture.

Note that NoAH Containment is the result of a collaboration between the NoAH project and the Dutch STW DeWorm project.

Bibliography

[1] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *SOSP'05*, Brighton, UK, October 2005.

[2] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proc. of the 37th annual International Symposium on Microarchitecture*, pages 221–232, 2004.

[3] J. R. Crandall, S. F. Wu, and F. T. Chong. Experiences using Minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In *Intrusion and Malware Detection and Vulnerability Assessment: Second International Conference (DIMVA05)*, Vienna, Austria, July 2005.

[4] H.Bos and K. Huang. Towards software-based signature detection for intrusion prevention on the network card. In *Proc of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[5] K. Hyang-Ah and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proc. of the 13th USENIX Security Symposium*, 2004.

[6] B. Jack. Remote windows kernel exploitation - step into the ring 0. eEye Digital Security Whitepaper, `www.eeye.com/~data/publish/whitepapers/research/OT20050205.FILE.pdf`, 2005.

[7] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *2nd Workshop on Hot Topics in Networks (HotNets-II)*, 2003.

[8] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[9] G. Portokalidis and H. Bos. SweetBait: Zero-Hour Worm Detection and Containment Using Honeypots, (An extended version of this report was accepted by Elsevier Journal on Computer Networks, Special Issue on Security through Self-Protecting and Self-Healing Systems), TR IR-CS-015. Technical report, Vrije Universiteit Amsterdam, May 2005.

[10] G. Portokalidis and H. Bos. Sweetbait: Zero-hour worm detection and containment using low- and high-interaction honeypots. *Elsevier Computer Networks (to appear)*, tba(tba):tba, tba 2006.

[11] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, Leuven, Belgium, April 2006.

[12] D. D. X. Qin, G. Gu, W. Lee, J. L. Julian Grizzard, and H. Owen. Honeystat: Local worm detection using honeypots. In *RAID*, 2004.

[13] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proc. of LISA '99: 13th Systems Administration Conference*, 1999.

[14] D. Spyrit. Win32 buffer overows (location, exploitation, and prevention). Phrack 55, 1999.

[15] V. P. Stuart Staniford and N. Weaver. How to 0wn the internet in your spare time. In *Proc. of the 11th USENIX Security Symposium*, 2002.

[16] G. V. Sumeet Singh, Cristian Estan and S. Savage. Automated worm fingerprinting. In *Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 45–60, 2004.

[17] M. M. Williamson. Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code. In *Proc. of ACSAC Security Conference*, Las Vegas, Nevada, 2002.