Argos: an Emulator for Fingerprinting Zero-Day Attacks

for advertised honeypots with automatic signature generation

Georgios Portokalidis Asia Slowinska Herbert Bos

Department of Computer Science
Faculteit der Exacte Wetenschappen
Vrije Universiteit Amsterdam
De Boelelaan 1081
1081 HV Amsterdam, Netherlands
{porto, jmslowin, herbertb}@few.vu.nl

ABSTRACT

As modern operating systems and software become larger and more complex, they are more likely to contain bugs, which may allow attackers to gain illegitimate access. A fast and reliable mechanism to discern and generate vaccines for such attacks is vital for the successful protection of networks and systems. In this paper we present *Argos*, a containment environment for worms as well as human orchestrated attacks. *Argos* is built upon a fast x86 emulator which tracks network data throughout execution to identify their invalid use as jump targets, function addresses, instructions, etc. Furthermore, system call policies disallow the use of network data as arguments to certain calls. When an attack is detected, we perform 'intelligent' process- or kernel-aware logging of the corresponding emulator state for further off-line processing. In addition, our own *forensics shellcode* is injected, replacing the malevolent shellcode, to gather information about the attacked process. By correlating the data logged by the emulator with the data collected from the network, we are able to generate accurate network intrusion detection signatures for the *exploits* that are immune to *payload* mutations. The entire process can be automated and has few if any false positives, thus rapid global scale deployment of the signatures is possible.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software

General Terms

Security, Design, Experimentation, Performance

EuroSys'06, April 18–21, 2006, Leuven, Belgium.

"And [Hera] set a watcher upon her [Io], great and strong Argos, who with four eyes looks every way. And the goddess stirred in him unwearying strength: sleep never fell upon his eyes; but he kept sure watch always." - Homerica, Aegimius

1. INTRODUCTION

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 [40]. 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 [43, 37, 23, 15, 27]. A large share of false positives violates the first two criteria. Although these systems may play an important role in intrusion detection systems (IDS), they are not suitable for fully automated response systems.

An 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 from the network are executed. This technique proves to be 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 are Minos [12] and Vigilante [28], respectively.

Minos does not generate signatures at all and for cost-effective deployment relies on implementation in hardware. Moreover, by looking at physical addresses only, it may *detect* certain exploits, such as a register spring attacks [39], but requires an awkward hack to determine where the attack originated [13]. Also, it cannot directly handle physical to virtual address translation at all.

In contrast, Vigilante represents a per-process solution that works with virtual addresses. Again, this is a design

decision that limits its flexibility, as it is not able to handle DMA or memory mapping. Also, the issue of cost-effectiveness arises as Vigilante must instrument 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 [26]. For signature generation it relies on replaying the attack which is often not possible due to challenge/response interaction with nonces, random numbers, etc.

We believe that both hardware-oriented full-system solutions, and OS- and process-specific software solutions 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 paper we describe *Argos* which explores another extreme in the design space for automated response systems. First, like Minos we offer whole-system protection in software by way of a modified x86 emulator which runs our own version of dynamic taint analysis [31]. In other words, we automatically protect any (unmodified) OS and all its processes, drivers, etc. Second, *Argos* takes into account complex memory operations, such as memory mapping and DMA (commonly ignored by other projects), and is at the same time quite 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 trigger alerts that result in the automatic generation of signatures based on the correlation of the exploit's memory footprint and its network trace. Fourth, while the system is OS- and application-neutral, when an attack is detected, we inject OS-specific forensics shellcode. In other words, we exploit the code under attack as the attack is happening to extract additional information about the attack which is subsequently used in signature generation. Fifth, by comparing signatures from multiple sites, we refine *Argos*' signatures automatically. Sixth, signatures are auto-distributed to remote intrusion detection and prevention systems (IDS and IPS).

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 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 payload and may be more easily caught even in the face of polymorphism.

Argos is designed as an 'advertised honeypot', i.e., a honeypot that runs real services and differs from normal honeypots in that we don't hide it. Rather, we actively link to it and 'advertise' its IP address 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 *Argos* is targeted as a honeypot, we do not require our solution to perform as well as unprotected systems. Nevertheless, it should be fast enough to run real services and have reasonable response time.

The remainder of this paper is organised as follows. While related work is discussed mainly throughout the text, we summarise various approaches in Section 2. In Section 3 we describe the design of *Argos*. Implementation details are discussed in Section 4. The system is evaluated in Section 5. Conclusions are in Section 6

2. BACKGROUND AND RELATED WORK

For an attacker to compromise a host, it is necessary to divert its conventional control flow to execute his own instructions, or replace elements of the host's control flow with his own. To accomplish this, an attacker needs to overwrite values such as jump targets, function addresses and function return addresses. Alternatively, he can also overwrite a function's arguments or even its instructions. Such attacks have been prominent the last years and can be classified to the following major categories:

- *Stack smashing attacks* [3] involve the exploitation of bugs that allow an attacker to overflow a stack buffer to overwrite a function's return address, so that when the function returns, arbitrary code can be executed;
- *Heap Corruption attacks* [34, 11] exploit heap overflows that allow an attacker to overwrite an arbitrary memory location, and as a result execute arbitrary code;
- *Format string attacks* [21] are the most versatile type of attack. They exploit a feature in the `printf()` family of functions, which allows the number of characters printed to be stored in a location in memory. When a user supplied string is used as a format string, an attacker can manipulate the string to overwrite any location in memory with arbitrary values. These attacks offer more options to the orchestrator, including overwriting function arguments, such as the file to be executed of the `execve()` system call;

The attack methods described above have been the subject of research by the security community for years. Both Stackguard, Stackshield and gcc extensions have been used to protect against stack smashing attacks [8, 25]. Later research has suggested that many of these methods can be easily bypassed [7]. There exist patches for most OSs to make the stack non-executable, but this introduces other problems (e.g., trampolines[2] rely on stacks being executable, and, in Linux at least, so do signals) and can sometimes be bypassed also. Buffer overflow detection and protection methods exist in abundance [38, 10, 30, 36]. They make it difficult/impossible to overwrite specific addresses so as to divert the control flow, e.g., by modifying the way in which code and data are stored in memory. In contrast, we desire a method that permits the overflow, but triggers an alert whenever the control flow diversion is attempted. Beyond that, the addresses should not be changed, as we aim to generate a reliable signature of the actual attack.

Some existing format string protection methods afford safety by way of a patch to `glibc` which counts arguments

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

[2]`http://gcc.gnu.org/onlinedocs/gccint/Trampolines.html`

to `printf()` [9] and by using type qualifiers [41]. Both approaches require recompilation of the code. Code injection problems have been addressed by instruction set randomisation and detection of attempts to perform syscalls from illegitimate addresses [16, 18, 24]. As we want to generate signatures, instruction set randomisation is not very useful for our purposes. The syscall protection offered by Dome [24] is potentially interesting, but was rejected for its limited scope (syscalls) and inconvenience (static analysis of every application is required to determine legitimate addresses for performing syscalls).

A different approach is to guard against overflows and attacks in hardware. For instance, StackGhost [29] protects the stack on Sparc architectures. Similarly, [17] uses dynamic information flow analysis to guard against overflows and some format string attacks. None of these mechanisms are widely available for the most commonly used processor/OS combinations and indeed, to the best of our knowledge [17] has not progressed beyond simulation. Instead of real machines Dunlap and Garfinkel suggest virtual machines [19, 20]. Similar work is presented in [42] which uses a modified version of the Dynamo dynamic optimiser. While *Argos* is different from these projects in many respects, we do follow a similar approach in that we employ an *emulator* of the x86 architecture.

Most closely related to our work are Minos [12] and Vigilante [28]. Like *Argos* both employ taint analysis to discover illegitimate use of 'foreign' data [31]. The differences with *Argos*, however, are manifold. Briefly, Minos is a hardware project that in the current software implementation on Bochs can only be deployed at a great cost in performance (up to several orders of magnitude slowdown[3]). Once misbehaviour is detected, Minos also makes no attempt to generate signatures. One of the reasons for this is that by aiming at a hardware solution, Minos has had to sacrifice flexibility for (potential) performance, as the amount of information available at the hardware level is very limited. For instance, since the hardware sees physical addresses it is difficult to deal with complex attacks requiring virtual addresses such as register spring attacks. Moreover, it seems that generating signatures akin to the self-certifying alerts (SCAs) in Vigilante would be all but impossible for Minos. In contrast, while *Argos* works with physical addresses also, we explicitly target emulation in software to provide us with full access to physical-to-virtual address mapping, registers, etc.

Vigilante differs from *Argos* in at least three ways: (a) it protects individual *processes* (requiring per-process management and leaving the kernel and non-monitored services in a vulnerable position), (b) it is OS-specific, and (c) it deals with *virtual* addresses only. While convenient, the disadvantage of virtual addresses is that certain things, like memory mapped data, become hard to check. After all, which areas in which address spaces should be tainted is a complex issue. For this reason, Vigilante and most other projects are unable to handle memory mapped areas. By positioning itself at the application-level, approaches like Vigilante also cannot monitor DMA activity. In contrast, *Argos* uses physical addresses and handles memory mapping as well as DMA.

3. DESIGN

[3]Indeed, the Minos authors mention that in the future they may replace Bochs by Qemu (which is already used by *Argos*): `wwwcsif.cs.ucdavis.edu/~crandall/DIMVAMinos.ppt`.

An overview of the *Argos* architecture is shown in Figure 1. The full execution path consists of six main steps, indicated by the numbers in the figure which correspond to the circled numbers in this section. Incoming traffic is 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 is achieved by identifying illegal uses of possibly unsafe data such as the data received from the network [31]. 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;

In other words, 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 [28] and [31]. As mentioned earlier, *Argos* 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.

When a violation is detected, an alarm is raised which leads to a signature generation phase ③-⑥. To aid signature generation, *Argos* first dumps 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 are 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 employ a novel technique to automate forensics on the code under attack. Recall that *Argos* 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 inject our own shellcode to perform forensics ③. In other words, we 'exploit' the code under attack with our own shellcode.

We emphasise that even without the shellcode, which by its nature contains OS-specific features, *Argos* 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 employ 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 is then used for correlation with the network traces in the trace database ④. In case of TCP connections, we reconstruct flows prior to correlation. The result of the correlation phase

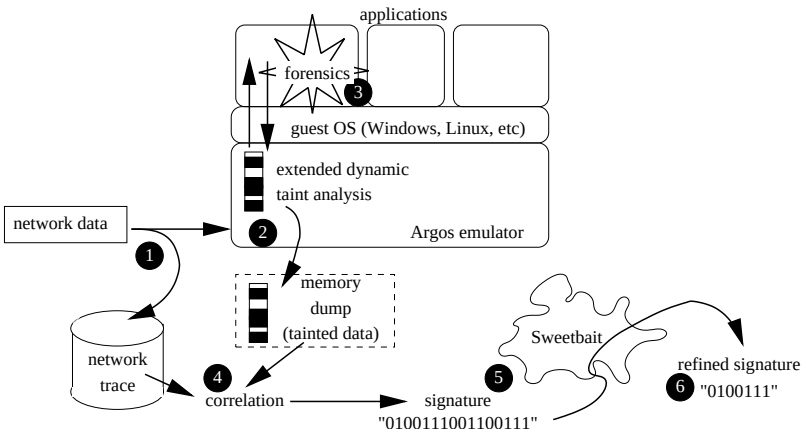

Figure 1: Argos: High-Level Overview

is a real signature that is, in principle, ready to be used for filtering. However, we do not consider the signature optimal and therefore try to refine it. For this purpose, *Argos* submits the signature to a subsystem known as *SweetBait*, which correlates signatures from different sites, and refines signatures based on similarity [32]. For instance, a signature consisting of the exploit plus the IP address of the infected host, would look slightly different at different sites. *SweetBait* notices the resemblance between two such signatures, and generates a shorter more specialised signature that it is then used in subsequent filtering.

The final step is the automated use of the signature ⑥. Attached to *SweetBait* are intrusion detection and prevention systems (IDS and IPS), that *SweetBait* provides with signatures of traffic to block or track. As IDS we use sensors based on the well-known open source network IDS *snort* [35] and for this purpose, *SweetBait* generates rules in *snort* rule format. The IPS 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 [22]. *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.

The focus of this paper is primarily on steps ①-④ and we will limit ourselves to summarising the *SweetBait* implementation. Interested readers are referred to [32] for details.

4. IMPLEMENTATION

Argos extends the *Qemu* [5] emulator by providing it with the means to taint and track memory, and generate memory footprints in case of a detected violation. *Qemu* is a fast and portable dynamic translator that emulates multiple architectures such as `x86`, `x86_64`, `POWER-PC64`, etc. Unlike other emulators such as Bochs[4], *Qemu* is not an interpreter. Rather, entire blocks of instructions are translated and cached so the process is not repeated if the same instructions are executed again. Furthermore, instead of providing the software equivalent of a hardware system, *Qemu* employs various optimisations to improve performance. As a result, *Qemu* is significantly faster than most emulators.

Our implementation extends *Qemu*'s Pentium architecture. In the remainder of this paper, it will be referred to simply as the `x86` architecture. For the sake of clarity we will also use the terms guest and host to distinguish between the emulated system and the system hosting *Qemu*.

We divide our implementation of *Argos* in two parts. The first contains our extended dynamic taint analysis which we used both to secure *Qemu* and to enable it to issue alerts whenever it identifies an attack. The second part covers the extraction of critical information from the emulator and the OS to generate a signature.

4.1 Extended Dynamic Taint Analysis

The dynamic taint analysis in *Argos* resembles that of other projects. However, there are important differences. In this section we discuss the implementation details.

4.1.1 Tagging

An important implementation decision in taint analysis concerns the granularity of the tagging. In principle, one could tag data blocks as small as a single bit, up to chunks of 4 KB or larger. We opted for variable granularity; per byte tagging of physical memory, while at the same time using a single tag for each CPU register. Per byte tagging of memory does not incur any additional computational costs i.e. over per double word tagging, and provides higher accuracy. On the other hand, per byte tagging of registers would introduce increased complexity in register operations, which is unacceptable. It is worth noting that altering *Argos* to employ a different granularity is trivial. For reasons of performance and to facilitate the process of forensics at a later stage, the nature of the memory and register tags is also different.

Register tagging. There are eight general purpose registers in the `x86` architecture [2], and we allocate a 4 B tag for each of them. The tag is used to store the physical memory address from where the contents of the register originate. Segment registers and the instruction pointer register (`EIP`) are not tagged and are always considered *untainted*. Since

[4]`http://bochs.sourceforge.net/`, by Kevin Lawton et al.

they can only be altered implicitly and because of their role, they belong to the protected elements of the system. The EFLAGS register is also not tagged and is considered *untainted*, because it is frequently affected by operations involving untrusted data, and tagging it would make it impossible to differentiate between malicious and benevolent sources. By default, MMX and FPU registers are treated similarly, although *Argos* is *able* to tag them if required. We implemented tagging for these registers as an option only, since they are involved in very specific operations that are rarely, if ever, involved in attacks. For the sake of performance, we ignore them by default.

Memory tagging. Since we do not store any additional data for physical memory tags, a binary flag for tagging would suffice. Nevertheless, one could also use a byte flag increasing memory consumption in exchange for performance enhancement. This might seem costly, but recall that we tag *physical* rather than virtual memory. While virtual memory space may be huge (e.g., 2^{64} on 64-bit machines) the same is not true for physical memory, which is commonly on the order of 512 MB - 1GB. Moreover, the guest's 'physical' RAM need not correspond to the physical memory at the host, so the cost in hardware resources can be kept fairly small. The scheme to be used can be configured at compile time. Following, we will discuss the two tagging schemes in more detail.

A *bitmap* is a large array, where every byte corresponds to 8 bytes in memory. The index idx of any physical memory address $paddr$ in the bitmap can be calculated by first shifting the address right by 3 ($idx = paddr \gg 3$) to locate the byte containing the bit flag ($map[idx]$). The individual bit flag is retrieved by using the lower 3 bits of $paddr$ ($b = map[idx] \bigoplus (0x01 \ll (paddr \bigoplus 0x07))$). The $size$ of the bitmap is an eighth of the guest's total addressable physical memory $RAMSZ$ ($size = \frac{RAMSZ}{8}$), i.e. the bitmap for a guest system of 512 MB would be 64 MB.

Similarly, a *bytemap* is also a large array, where each byte corresponds to a byte in memory. The physical address $paddr$ of each byte is also the index idx in the bytemap. Its total $size$ is equal to the guest's total addressable physical memory $RAMSZ$ ($size = RAMSZ$).

Finally, incoming network data are marked as *tainted*. Since the entire process does not involve OS participation the tagging is performed by the virtual NE2000 NIC emulated by *Qemu*. OSs communicate with peripherals in two ways: port I/O and memory mapped I/O. *Qemu*'s virtual NIC though, supports only port I/O, which in x86 architectures is performed using instructions IN and OUT. By instrumenting these instructions the registers loaded with data from the NE2000 are tagged as *tainted* while all other port I/O operations result in clearing the destination register's tag.

4.1.2 Tracking

Qemu translates all guest instructions to host native instructions by dynamically linking blocks of functions that implement the corresponding operations. Tracking *tainted* data involves instrumenting these functions to manipulate the tags, as data are moved around or altered. Besides registers and memory locations, available instruction operands include immediate values, which we consider to be *untainted*. We have classified instrumented functions in the following categories:

- *2 or 3 operand* ALU *operations;* these are the most common operations and include ADD, SUB, AND, XOR, etc. If the destination operands are not *tainted*, they result in copying the source operands tags to the destination operands tags.
- *Data move operations;* these operations move data from register to register, copying the source's tag to the destination's tag.
- *Single register operations;* shift and rotate ops belong to this category. The tag of the register is preserved as it is.
- *Memory related operations;* all LOAD, STORE, PUSH and POP operations belong here. These operations retrieve or store the tags from or to memory respectively.
- FPU, MMX, *or* SSE *operations;* as explained above, these are ignored by default (tagging of the corresponding registers is optional in *Argos*), unless their result is stored in one of the registers we track or to memory. In these cases, the destination is cleared. More advanced instructions such as SSE2 and 3DNow! are not supported by *Qemu*.
- *Operations that do not directly alter registers or memory;* some of these ops are NOP, JMP, etc. For most of these we do not have to add any instrumentation code for tracking data, but for identifying their illegal use instead, as we describe in the following section.
- *Sanitising operations;* certain fairly complex instructions result in always cleaning the tags of their destination operands. This was introduced to marginalise the possibility of false positives. Such instructions are rotate left/right (ROR, ROL), BCD and SSE instructions, as well as double precision shifts.

Fortunately, we do not have to worry about special instruction uses such as xor eax,eax or sub eax, eax. These are used in abundance in x86's to set a register to zero, because unlike RISC there is no zero register available. *Qemu* makes sure to translate these as a separate function that moves zero to the target register. When this function is compiled it follows the native architecture's idiom of zeroing a register.

Modern systems provide a mechanism for peripherals to write directly to memory without consuming CPU cycles, namely direct memory access (DMA). When using DMA OSs instead of reading small chunks of data from peripherals they allocate a larger area of memory and send its address to the peripheral, which in turn writes data directly in that area without occupying the CPU. *Qemu* emulates DMA for components such as the hard disk. Whenever a DMA write to memory is performed in *Argos*, it is intercepted and the corresponding memory tags are cleared.

4.1.3 Preventing Invalid Uses of Tainted Data

Most of the observed attacks today gain control over a host by redirecting control to instructions supplied by the attacker (e.g., shellcode), or to already available code by carefully manipulating arguments (return to libc). For these attacks to succeed the instruction pointer of the host must be loaded with a value supplied by the attacker. In the x86 architecture, the instruction pointer register EIP is loaded by the following instructions: call, ret and jmp.

By instrumenting these instructions to make sure that a *tainted* value is not loaded in EIP, we manage to identify all attacks employing such methods. Optionally, we can also check whether a *tainted* value is being loaded on model specific registers (MSR) or segment registers, but so far we have not encountered such attacks and we are not aware of their existence.

While these measures capture a broad category of exploits, they alone are not sufficient. For instance, they are unable to deal with format string vulnerabilities, which allow an attacker to overwrite any memory location with arbitrary data. These attacks do not directly overwrite critical values with network data, and might remain undetected. Therefore, we have extended dynamic taint analysis to also scan for code-injection attacks that would not be captured otherwise. This is easily accomplished by checking that the memory location loaded on EIP is not *tainted*.

Finally, to address attacks that are based solely on altering arguments of critical functions such as system calls, we have instrumented *Qemu* to check when arguments supplied to system calls like execve() are *tainted*. To reliably implement this functionality we require a hint about the OS being run on *Argos*, since OSs use different system calls. The current version of *Argos* supports this feature solely for the Linux OS, but we plan to extend it to support FreeBSD, and MS Windows operating systems.

4.2 Signature Generation

In this section we explain how we extract useful information once an attack is detected, how signatures are generated, and how they are specialised by correlating memory and network traces. In addition we show how we refine signatures with an eye on obtaining small signatures containing an exploit's nucleus. Also, unlike related projects like [28], we intentionally investigated signature generation methods that do not require attacks to be replayed. Replaying attacks is difficult, e.g., because challenge/response authentication may insert nonces in the interaction. While we know of one attempt to implement replay in the face of cookies and nonces [14], we don't believe current approaches are able to handle most complex protocols.

We emphasize that the signature generation methods described in this section are only a first stab and mainly serve to demonstrate how the wealth of information generated by *Argos* can be exploited by suitable back-ends. We are currently exploring more advanced methods. In our opinion, the ability to plug in different back-ends (signature generators) is quite useful.

4.2.1 Extracting Data

An identified attack can become an asset for the entire network security community if we generate a signature to successfully block it at the network level. To achieve this, *Argos* exports the contents of 'interesting' memory areas in the guest for off-line processing. To reduce the amount of exported data we dynamically determine whether the attack occurred in user- or kernel-space. This is achieved by retrieving the processor's privilege ring bits from *Qemu*'s hidden flags register. The kernel is always running on privileged ring 0, so we can distinguish processes from the kernel by looking at the ring in which we are running.

Additionally, every process is sharing its virtual address space with the kernel. OSs accomplish this by splitting the address space. In the case of Linux a 3:1 split is used, meaning that three quarters of the virtual address space are given to the process while one quarter is assigned to the kernel. Windows on the other hand is using a 2:2 split. The user/kernel space split is predefined in most OS configurations, so we are able to use static values as long as we know which OS is being run. We take advantage of this information to dump only relevant data.

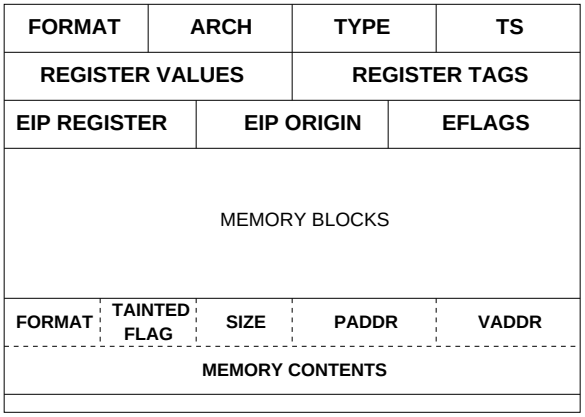

Figure 2: Memory dump format

To determine which physical memory pages are of interest and need to be logged, we traverse the page directory installed on the processor. In x86 architectures the physical memory address of the active page directory is stored in control register 3 (CR3). Note that because we traverse the virtual address space of processes, physical pages mapped to multiple virtual addresses will be logged multiple times (one for each mapping).

By locating all the physical pages accessible to the process / kernel, and making sure that we do not cross the user / kernel space split, we dump all *tainted* memory areas as well as the physical page pointed to by EIP regardless of its tags state. The structure of the dumped data is shown in Figure 2. For each detected attack the following information is exported: the log's format (FORMAT), the guest architecture (ARCH could be i386 or x86_64), the type of the attack (TYPE), the timestamp (TS), register contents and tags (including EIP and its origin), the EFLAGS register, and finally memory contents in blocks. Each memory block is preceded by the following header: the block's format (FORMAT), a *tainted* flag, the size of the block in bytes, and the physical (PADDR) and virtual (VADDR) address of the block. The actual contents of the memory block are written next. When all blocks have been written, the end of the dump is indicated by a memory block header containing only zeroes.

All of the above are logged in a file named 'argos.csi.RID', where RID is a random ID that will be also used in advanced forensics discussed in the following section.

The data extracted from *Argos* serve for more than signature generation. By logging all potentially 'interesting' data, thorough analysis of the attack is made possible. Consider for example techniques such as register springs, which do not directly alter control flow to injected code. By also logging the legitimate code that is used for the spring, and by exploiting the presence of both physical and virtual addresses in the log, a security specialist can effectively reverse engineer

most, if not all, attacks.

4.2.2 Advanced Forensics

An intrinsic characteristic of *Argos* is that it is process agnostic. This presents us with the problem of identifying the target of an attack. Discovering the victim process, provides valuable information that can be used to locate vulnerable hosts, and assist in signature generation. To overcome this obstacle, we came up with a novel idea that enables us to execute code in the process's address space, thus permitting us to gather information about it.

Currently, most attacks hijack processes by injecting assembly code (shellcode) and diverting control flow to its beginning. Inspired by the above, we inject our own shellcode into a process's virtual address space. After detecting an attack and logging state, we place forensics shellcode *directly* into the process's virtual address space. The location where the code is injected is crucial, and after various experiments we chose the last `text` segment page at the beginning of the address space. Placing the code in the `text` segment is important to guarantee that it will not be overwritten by the process, since it is read-only. It also increases the probability that we will not overwrite any critical process data. Having the shellcode in place we then point `EIP` to its beginning to commence execution.

As an example, we implemented shellcode that extracts the `PID` of the victim process, and transmits it over a TCP connection along with the `RID` generated previously. The information is transmitted to a process running at the guest, and the code then enters a loop that forces it to sleep forever to ensure that while it does not terminate, it remains dormant. At the other end, an information gathering process at the guest receives the `PID` and uses it to extract information about the given process by the OS. Finally, this information is transmitted to the host, where it is stored.

The forensics process retrieves information about the attacked process by running `netstat`, or if that is not available `OpenPorts` [1]. The above tools offer both the name of the process, as well as all the associated ports. The set of ports can be used to restrict our search in network traces (as discussed in Section 4.2.3) by discarding traffic destined to other ports. Currently, forensics are available for both Linux and Win32 systems. In the future, we envision extracting the same or more information without employing a third process at the guest.

4.2.3 Information Correlation

The memory fingerprint collected from the guest, along with the information extracted using advanced forensics are subsequently correlated with the network trace of data exchanged between the guest and the attacker. We capture traffic using `tcpdump` and store it directly in a trace database that is periodically garbage collected to weed out the old traffic streams. In the next version of *Argos* we capture traffic using the home-grown FFPF framework which allows us to dump different flows in different traces [6].

The collected network traces are first preprocessed by re-assembling TCP streams to formulate a continuous picture of the data sent to the guest. For stream reassembly we build on the open source `ethereal` library[5]. This enables us to detect attacks that are split over multiple packets either intentionally, or as part of TCP fragmentation.

[5]http://www.ethereal.com/

The current version of *Argos* uses the attacked port number provided by forensics to filter out uninteresting network flows. In addition, the dumped memory contents are also reduced. The tag value of `EIP` is used to locate in the network trace the *tainted* memory block that is primarily responsible for the attack. This block along with the remaining network flows are processed to identify patterns that could be used as signatures. *Argos* uses two different methods to locate such patterns: (i) longest common sub-string (LCS), and (ii) critical exploit string detection (CREST).

(i) LCS is a popular and fast algorithm for detecting patterns between multiple strings also used by other automatic signature generation projects [27]. The algorithm's name is self-explanatory: it finds the longest substring that is common to memory and traffic trace. Along with the attacked port number and protocol we then generate a Snort signature. While this method appears promising, it did not work so well in our setup, as the common substring between the trace and memory is (obviously) huge. While we are still improving the LCS signature generation, we achieved the best results so far with CREST.

(ii) CREST is a novel algorithm. The incentive behind its development was the fact that the output of *Argos* offers vital insight about the internal workings of attacks. The dumped information allows us to generate signatures targetting the string that triggers the *exploit*, and that may therefore be very accurate and immune to techniques such as polymorphism. Using the physical memory origin (O_{EIP}) and value of `EIP` (V_{EIP}) we can pin-point the memory location that acts as the attacker's foothold to take control of the guest. The advantage of CREST is that it captures the very first part of an attack, which is less mutable. The current version of CREST yields signatures based on this exploit string. However, we are now working on a more advanced implementation of CREST, in which we attempt to isolate exactly the (minimal) part of the network trace that causes the exploit.

The current implementation of CREST is fairly simple. Essentially, we locate V_{EIP} in the network traces corresponding to the application's port and then extend the trace to the left and to the right. In other words, we match up individual bytes above and below the O_{EIP} in the memory dump with bytes before and after the location of V_{EIP} in the network trace. We stop when we encounter bytes that are different. *Argos* uses the resulting byte sequence and combines it with the port number and protocol to generate a signature in snort rule format. Signatures generated in this way were generally of reasonable size, a few hundred bytes, which makes them immediately usable. Moreover, as we show in Section 4.2.4, the signatures are later refined to make them even smaller.

Note that although we currently use only a small amount of it, for signature generation we are able to work with a wealth of information. In practice, *Argos* produces significantly more information than other projects [12, 28], because we have full access to physical and virtual memory addresses, registers, EIP, etc. So even though it proves to be very effective even in its current form, CREST should be considered a work-in-progress and the current implementation as a proof-of-concept. For instance, CREST would fail to generate a signature for a format string attack, or any attack that manages to point EIP to a tainted memory area without tainting the register itself.

In the next release of our system, we plan to keep track

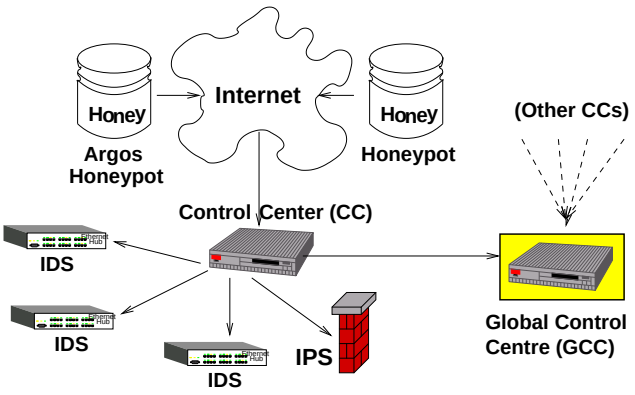

Figure 3: Architecture of the SweetBait Subsystem

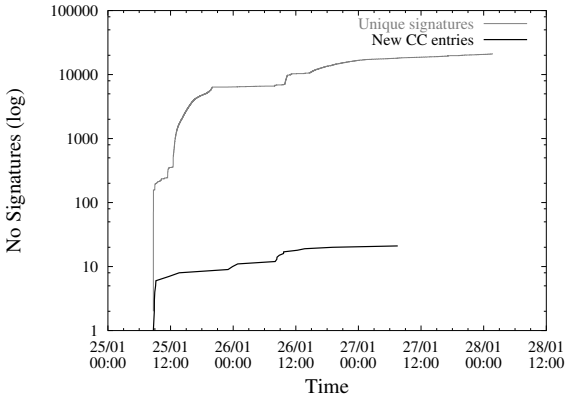

Figure 4: SweetBait Sig. Specialisation Results

of the exact origin in the network trace(s) of each word of tainted memory. As a result, we will no longer need to scan the trace(s) for the occurrence of specific byte patterns. Rather, we will be able to pinpoint problematic bytes in the network traces directly, greatly improving speed and accuracy of signature generation methods.

A final feature of *Argos*' signature generation is that it is able to generate both flow and packet signatures. Flow signatures consist exactly of the sequence of bytes as explained above. For packet signatures, on the other hand, *Argos* maps the byte sequence back to individual packets. In other words, if a signature comprises more than one packet, *Argos* will split it up in its constituent parts. As we keep track of the contributions of individual packets that make up the full stream, we are even able to handle fairly complex cases, such as overlapping TCP segments. Packet signatures are useful for IDS and IPS implementations that do not perform flow reassembly.

4.2.4 SweetBait

SweetBait is an automated signature generation and deployment system [32]. It collects snort-like signatures from multiple sources such as honeypots and processes them to detect similarities. Even though *Argos* is its main input for this project, we have also connected *SweetBait* to low-interaction honeypots based on *honeyd* [33] and honeycomb [27]. It should be mentioned that to handle signatures of different nature, *SweetBait* types them to avoid confusion. The *SweetBait* subsystem is illustrated in Figure 3.

The brain of the *SweetBait* subsystem is formed by the control centre (CC). CC maintains a database of attack signatures that is constantly updated and it pushes the signatures of the most virulent attacks to a set of IDS and IPS sensors according to their signature budgets, as explained later in this section. In addition to the IDS/IPS sensors we also associate a set of *Argos* honeypots with each CC. Honeypots send their signatures to their CC over SSL-protected channels. The signatures are gathered by the CC and compared against known signatures. In essence, it uses LCS to find the amount of overlap between signatures. If two signatures are sufficiently alike, it concludes that the LCS represents a more specialised signature for the attack and installs a new signature version that deprecates the older one. In this way, we attempt to locate the immutable part of signatures and remove the parts that vary, such as target IPs, host names, attack payloads etc. Doing so minimises the number of collected signatures to a manageable size. For example, we employed *SweetBait* with low-interaction (honed/honeycomb) honeypots (since we had a much larger set of signatures for these honeypots than for *Argos*) and were able to reduce the thousands of signatures generated during the period of three days to less than 30 (Figure 4).

The specialisation process is mainly governed by three parameters. The minimum match parameter m represents the minimum amount of overlap that two signatures should have before the CC decides that they are variations of the same signature. The value m is expressed as a percentage of the size of the known signature. For instance, $m = 90\%$ means that the new signature should match at least 90% of the signature that is already in the database for it to be classified as a variation of this signature. The minimum and maximum length parameters L and M represent the minimum and maximum length of an acceptable signature respectively. For instance, $L = 10$ and $M = 1500$ means that for a signature to be accepted and stored in the database it should be longer than 10 bytes and shorter than 1500.

The optimal value for these parameters varies with the nature of the signatures. For instance, if the signatures are likely to be unrelated, such as the signatures generated by *honeyd/honeycomb*, m should be large to ensure that the signatures really are related. While in this case the optimal choice of parameters is a matter of careful tuning, we are in a much better position when dealing with *Argos* signatures. After all, here we may force the subsystem to compare only signatures that are known to be related. For instance, by comparing only signatures with the same V_{EIP} during specialisation, we know that only similar exploits will be considered. In essence, we can set m and M to an arbitrarily low and high value respectively, and have L govern the process entirely. The value of L was determined by looking at the size of real signatures used by the snort framework. In snort, the content fields of most rules are fairly small, often less than ten bytes. By choosing a value slightly greater than that, the signatures are likely to be both accurate and small. In practice we use $L = 12$ for the signatures generated by *Argos* and make sure that *Argos* generates signatures that are related (e.g., that have the same V_{EIP}) in a separate bucket to be processed as a separate group by *SweetBait*.

SweetBait deploys the final versions of signatures to network IDSs and IPSs. To warrant increased performance levels of

the connected IDSs and IPSs and deal with heterogeneous capacities of IDSs and IPSs, a signature budget in number of bytes can be specified, so that the number of signatures pushed to the sensors does not exceed a certain level.

To determine the signatures that will be pushed to the sensors, SweetBait uses network IDS sensors to approximate the virulence of the corresponding attacks. The density of generated alerts by the IDS is used as an indicator of aggression, which in turn determines whether a signature should be pushed to the prevention system or still be monitored[6]. In other words, a signature that is reported frequently by many sites will have a higher virulence estimation than one that is reported less frequently and by a smaller number of honeypots, and is therefore more likely to be pushed to the IDS/IPS sensors. Additionally, signatures can be manually tagged as valid, or invalid to increase the level of certainty. Whether IPS sensors automatically block traffic based on signatures that are not manually tagged as valid is a configurable parameter. Details about the IPS sensors are beyond the scope of this paper and can be found in [32] and [22].

An important feature of the *SweetBait* subsystem is its ability to exchange signatures on a global scale. Global scale collaboration is necessary for identifying and preventing zero-day attacks, and SweetBait makes this partially feasible by means of the global control centre (GCC). The GCC collects signatures and statistics in a similar way to a CC, with the main difference being the lack of a signature budget when pushing signatures to CCs.

The CC periodically exchanges information with the GCC. This includes newly generated signatures, as well as activity statistics of known signatures. The statistics received by the GCC are accumulated with the ones generated locally to determine a worm's aggressiveness. This accumulation ensures that the CC is able to react to a planetary outbreak, even if it has not yet been attacked itself, achieving immunisation of the protected network. Again, we secured all communication between CC and GCC using SSL.

5. EVALUATION

We evaluate *Argos* 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.

5.1 Performance

For realistic performance measurements we compare the speed of code running on *Argos* with that of code running without emulation. We do this for a variety of realistic benchmarks, i.e., benchmarks that are also used in real-life to compare PC performance. Note that while this is an honest way of showing the slowdown incurred by *Argos*, it is not necessarily the most relevant measure. After all, we do not use *Argos* as a desktop and in practice hardly care whether results appear much less quickly than they would without emulation. The only moment when slowdown becomes an issue is when attackers decide to shun slow hosts, because it might be a honeypot. To the best of our knowledge such worms do not exist in practice.

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.

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

Table 1: Apache Throughput

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 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 5 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 *Argos*: 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*, *Argos* 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 1.

Even in the fastest configuration, *Argos* is at least 16 times slower than the host. Most of the overhead, however, is incurred by *Qemu* itself. *Argos* 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

[6]Specifically, we use an exponentially weighted moving average over the number of reports per sensor.

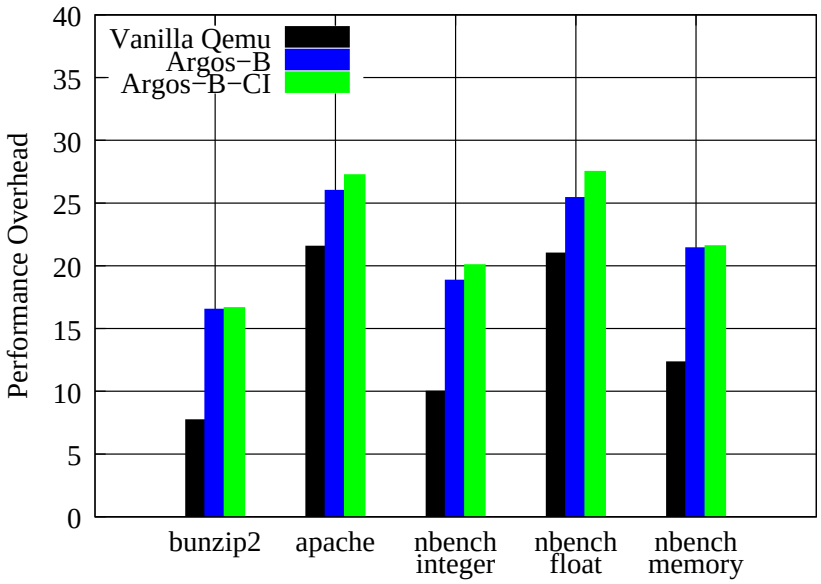

Figure 5: Performance Benchmarks

is possible.

Moreover, even though the performance penalty is large, personal experience with *Argos* has shown us that it is tolerable. Even when executing graphics-intensive tasks, the machine offers decent usability to human operators who use it as a desktop machine. Moreover, we should bear in mind that *Argos* was not designed as a desktop system, but as a platform for hosting advertised *honeypots*. Performance is not our main concern. Still, we have plans to introduce novelties that will further improve performance in future versions of *Argos*. A related project that takes a similar approach, but focuses on performance with an eye on protecting desktops is described in [4].

5.2 Effectiveness

To determine how effective *Argos* 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[7], 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 *Argos* (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 the tested exploits along with the underlying OS and their associated worms is shown in table 2. For Windows, we have only listed fairly well-known exploits. All exploits were successfully captured by *Argos* and the attacked processes were consequently stopped to prevent the exploit payloads from executing. In addition, our forensics shellcode executed successfully, providing us with process names, IDs, and open port numbers at the time of the attack.

Finally, we should mention that during the performance evaluation, as well as the preparation of attacks, *Argos* 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 *Argos* will *never* generate false positives, considering the large number of exploits tested, it may serve as an indication that *Argos* 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 [28]). However, in case we incur false positives in the future, *Argos* is quite suitable for generating SCAs.

5.3 Signatures

The final part of the evaluation involves signature generation. To illustrate the process, we explain in some detail the signature that is generated by *Argos* for the Windows RPC DCOM vulnerability listed in Table 2.

We use the Metasploit framework to launch three attacks with different payloads using the same exploit mentioned above, against 3 distinct instances of *Argos* hosting guests with different IPs. The motivation for doing so is to force *Argos* to generate varying signatures for the same exploit. In this experiment, we employ the CREST algorithm (Section 4.2.3) to generate the signatures, and consequently submit them to the *SweetBait* subsystem.

During the correlation, CREST searches through the network trace and reconstructs the byte streams of relevant TCP flows. Note that the logs that are considered by the signature generator are generally fairly short, because we are able to store separate flows in separate files by using the home-grown FFPF framework [6]. As a result, CREST may ignore flows that finished a long time ago and flows to ports other than the one(s) reported by forensics. The signature generation times including TCP reassembly for logs of various sizes is

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

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
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 2: Exploits Captured by Argos

shown in Figure 6.

SweetBait was configured to perform aggressive signature specialisation as explained in Section 4.2.4. Examining its database after the reception of all signatures, we discovered that it successfully classified them as being part of the same attack and generated a single specialised signature based on their similarities. The size of the signatures was effectively reduced from approximately 180 bytes to only 16 as it is shown in Figure 7. The figure shows the payload part of the original signatures, generated by *Argos* without the *SweetBait* subsystem, as well as *SweetBait*'s specialisation. The signatures are represented in the way content fields are represented in snort rules, i.e., series of printable characters are shown as strings, while series of non-printable bytes are enclosed on the left and right by the character '|' and represented by their hexadecimal values. Observe that the specialised signature generated by *SweetBait* is found in each of the original signatures, as shown by the boxes in Figure 7.

Furthermore, we used the specialised signature to scan a benevolent network trace for the possibility of it generating false positives. Besides homegrown traces, we used the RootFu DEFCON[8] competition network traces that are publicly available for research purposes. We first verified that the exploit was not present in the traces, by scanning the trace with open source community snort rules, using rules obtained from bleeding snort[9]. Next, we scanned it with the signature generated by *Argos*. Again, there were no (false) alerts.

Even though our signature generation algorithm is fairly simple, we are able to automatically generate signatures with a very small probability of false positives, by means of the *SweetBait* subsystem and deployment at multiple *Argos* sites.

[8] `http://www.shmoo.com/cctf/`

[9] `http://www.bleedingsnort.com`

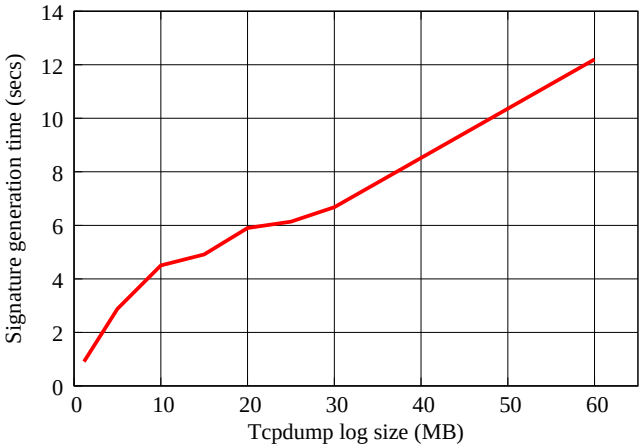

Figure 6: Signature generation

6. CONCLUSION

In this paper we have discussed an extreme in the design space for automated intrusion detection and response system: a software-only whole-system solution based on an `x86` emulator that uses dynamic taint analysis to detect exploits and protects unmodified operating systems, processes, etc. By choosing a vantage point that incorporates attractive properties from both the hardware level (e.g., awareness of physical addresses, memory mapping and DMA) and also the higher-levels (virtual addresses, per-process forensics), we believe our approach is able to meet the demands of automated response systems better than existing solutions.

The system exports the tainted memory blocks and additional information as soon as an attack is detected, at which point it injects forensics shellcode into the code under attack to extract additional information (e.g., executable name and process identifier). Next, it correlates the memory blocks with the network traces to generate a signature. Similar signatures from multiple sites are later refined to generate smaller, more specialised signature that are subsequently transmitted to intrusion prevention systems. Performance without employing any of the emulator's optimisation modules is significantly slower than code running without the emulator. Even so, as our intended application domain is (advertised) honeypots, we believe the overhead is acceptable. More importantly, the system proved to be effective and was used to capture and fingerprint a range of real exploits.

Future work

Our focus throughout the project was on detecting real attacks and generating usable network signatures. As we have not yet encountered false positives, the signatures are meant to be used *directly*, for instance in IPSs such as our own [22] and IDSs such as snort [35]. In case of false positives, we plan to generate signatures as self-certifying alerts [28]. Future research also addresses more advanced signature generation and automated analysis of the attack.

Original signatures:

```
|98 91|KCBJ7J|99 98|G|F8|@HHA?IK7N|9B F8|N|97 9F 9F|
?JIO|EB 10 EB 19 9F| u| 18 00| #7| F3| w| EB E0 FD 7F|
A|F5|A|FC|F|90 9B|C?|D6 98 91 FC 93 98 92 F9|K|FC|J
|9B 92|H|FC 99 D6|A|96|OJ|93|N|FC|O|FD|CC|97 96|J|91
92|JAKI?|27|B@|99|G|99 F5|I|93 F8|C|D6 27|O7|90 91|
7O|D6 99|@H?|FD 27 91|BI|F9 97|H|D6 96 98|?|91 93 97
F8 FD EB 04 FF FF FF FF|J|92|G|93 92|7|9F 98 EB 04
EB 04 92 9F|?@|EB 04 FF FF FF FF 97|CI|F8 F5 FC|FKK@
OJHF|96|GHN|92 9B|K|93|F7|OA|
```

```
|98 99|?B|99|H|99 99|I|96 93|J|F8 D6 F5 90|NKAJ|FC 97
90 91 D6|OA|27 F5 F9 92|EB 10 EB 19 9F| u18 00|#7| F3
| w| EB E0 FD 7F|@N|9F 27 96|JH|FC|N|FC|F|90 D6 90
90 F9 97 9B|J7KO|91 D6|KKG|93 F9 9B 92 92|?KGF|FC|N|
93|F|9F 90 F9 98 92 98 96|A7C|97 99|J|FC|HI7|27|G|98
99|?F|D6 F9 98|@@|9F D6 98|@A|F8 92 93|IB|F8|BFH|98|
NHC|96 90 EB 04 FF FF FF FF|J|98 F8|J|92 9B 90|A|EB
04 EB 04|JKH|91 EB 04 FF FF FF FF F8 FD|J|FD|IH|96|?
?|93 91|C|D6|@NIHI|9F|@|F8 F5|G|D6 OA|
```

```
F?F|9B|C?|F5 98|F|27|IO|F9|?|FD|BB?|90 9B F5|?|FC|A
|9B|F|D6 97|CH|F5|EB 10 EB 19 9F| u| 18 00| #7| F3| w
| EB E0 FD 7F|9B|N|9F 27|?GC|F9|JH|F8|B@FICN|99 F9
97|B|9F 90 90 92|?|99 D6|JAB|90|ACO|93 27|N|FD|C|90|
O|96 F5 F9 90|H|98 90|?|93|A|99 93 FC 91 F8|O|9F 93
9B F9|I|D6 92|K@NH|F9 91|F|91|J7A?I|9B 98 93|N7A?|92
27|N|EB 04 FF FF FF FF|HIN7|99|N|98|G|EB 04 EB 04 99|
K|FC D6 EB 04 FF FF FF FF|AACK|98 90|@|92|77|93|?C
|9B|BF|9F 90 F5|A|FD 90 9B 9B OA OA|
```

SweetBait specialised signature:

```
|EB 10 EB 19 9f|u|18 00|#7|F3|w|EB E0 FD 7F|
```

Figure 7: Signature Specialisation (snort format)

Acknowledgements

This research is partly funded by the Sentinels *DeWorm* project of the national research foundation of The Netherlands, and the EU FP6 NoAH project. We are grateful to Kostas Anagnostakis of the Institute for Infocomm Research in Singapore, Steve Uhlig of Université Catholique de Louvain in Belgium, and Rogier Spoor of SURFnet in the Netherlands, for their comments on earlier drafts of this paper.

7. REFERENCES

[1] Diamondcs openports. `http://www.diamondcs.com.au/openports/`.

[2] *Basic Architecture*, volume 1 of *Intel Architecture Software Developer's Manual*. Intel Corporation, 1997.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49), November 1996.

[4] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *Proc. of the 1st EuroSys Conference*, Arpil 2006.

[5] F. Bellard. QEMU, a fast and portable dynamic translator. In *In Proc. of the USENIX Annual Technical Conference*, pages 41–46, April 2005.

[6] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. FFPF: Fairly Fast Packet Filters. In *Proceedings of OSDI'04*, San Francisco, CA, December 2004.

[7] bulba and Kil3r. Bypassing Stackguard and Stackshield. *Phrack Magazine*, 10(56), January 2000.

[8] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the 7th USENIX Security Symposium*, 1998.

[9] C. Cowan, M. Barringer, S. Beattie and G. Kroah-Hartman. FormatGuard: Automatic protection from printf format string vulnerabilities. In *In Proc. of the 10th Usenix Security Symposium*, August 2001.

[10] C. Cowan, S. Beattie, J. Johansen and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *In Proc. of the 12th USENIX Security Symposium*, pages 91–104, August 2003.

[11] M. Conover. w00w00 on heap overflows. `http://www.w00w00.org/articles.html`, January 1999.

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

[13] 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.

[14] W. Cui, V. Paxson, N. Weaver, and R. Katz. Protocol-independent adaptive replay of application dialog. In *The 13th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2006.

[15] D. Dagonand, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine and Henry Owen. HoneyStat: Local worm detection using honeypots. In *In Proc. of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.

[16] E. G. Barrantes, D.H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovix and D.D. Zovi. Randomized instruction set emulation to disrupt code injection attacks. In *In Proc. of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 281–289, October 2003.

[17] G. E. Suh, J. W. Lee, D. Zhang and S. Devadas. Secure program execution via dynamic information flow tracking. *ACM SIGOPS Operating Systems Review*, 38(5):86–96, December 2004. SESSION: Security.

[18] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *In Proc. of the ACM Computer and Communications Security (CCS) Conference*, pages 272–280, October 2003.

[19] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *In Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[20] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion

detection. In *In Proc. of the 10th ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, February 2003.

[21] gera and riq. Advances in format string exploitation. *Phrack Magazine*, 11(59), July 2002.

[22] 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.

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

[24] J. C. Rabek, R. I. Khazan, S. M. Lewandowski and R. K. Cunningham. Detection of injected, dynamically generated, and obfuscated malicious code. In *In Proc. of the ACM workshop on Rapid Malcode*, 2003.

[25] J. Etoh. GCC extension for protecting applications from stack-smashing attacks. Technical report, IBM, June 2000.

[26] 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.

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

[28] M. Costa, J. Crowcroft, M. Castro, A Rowstron, L. Zhou, L. Zhang and P. Barham. Vigilante: End-to-end containment of internet worms. In *In Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, October 2005.

[29] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *In Proc. of the 10th USENIX Security Symposium*, pages 55–66, August 2001.

[30] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overlows in C. In *In Proc. of the ACM Conference on Object-Oriented Programming, Systems, Languages and Application*, October 2003.

[31] 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.

[32] 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.

[33] N. Provos. A virtual honeypot framework. In *Proc. of the 13th USENIX Security Symposium*, 2004.

[34] rix. Smashing C++ VPTRS. *Phrack Magazine*, 10(56), January 2000.

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

[36] S. Bhatkar, D.C. Du Varney and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *In Proc. of the 12th USENIX Security Symposium*, pages 105–120, August 2003.

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

[38] S. Sidiroglou and A. D. Keromytis. Using execution transactions to recover from buffer overflow attacks. Cucs-031-04, Columbia University, 2004.

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

[40] 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.

[41] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *In Proc. of the 10th USENIX Security Symposium*, pages 201–216, August 2001.

[42] V. Kiriansky, D. Bruening and S. Amarasinghe. Secure execution via program shepherding. In *In Proc. of the 11th USENIX Security Symposium*, 2002.

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