

Eudaemon: A Good Spirit to Protect Processes from Internet Attacks

Georgios Portokalidis and Herbert Bos

Department of Computer Science
Vrije Universiteit Amsterdam
1081 HV Amsterdam, Netherlands
porto, herbertb@few.vu.nl

ABSTRACT

Eudaemon is a technique that aims to blur the borders between protected and unprotected applications, and brings together honeypot technology and end-user intrusion detection and prevention. *Eudaemon* is able to attach to any running process, and redirect execution to a user-space emulator that will dynamically instrument the binary by means of taint analysis. Any attempts to subvert the control flow to malicious code will be detected and averted. When desired *Eudaemon* can reattach itself to the emulated process, and return execution to the native binary. In other words, applications can be moved between protected mode and native mode at will. The entire procedure is performed within the process context, offering a sufficient degree of isolation, and can be initiated whenever the user sees fit, i.e. when within a critical section from a security perspective, or when spare CPU cycles are available.

“Greeks divided daemons into good and evil categories: eudaemons (also called kalodaemons) and kakodaemons, respectively. Eudaemons resembled the Abrahamic idea of the guardian angel; they watched over mortals to help keep them out of trouble. (Thus eudaemonia, originally the state of having a eudaemon, came to mean “well-being” or “happiness”).” - Wikipedia

1. INTRODUCTION

Sophisticated high-interaction honeypots like Argos [24], Minos [10], and Vigilante [20] all use dynamic taint analysis (as pioneered by Newsome and Song in TaintCheck [22]) to capture zero-day attacks. In essence, dynamic taint analysis uses an emulator to tag data from suspicious origins (e.g., data coming from the network) and raises an alert when such data is used to divert the control flow. For performance reasons, such honeypots are ill-suited for full-time deployment on every desktop PC. The slow-down incurred by the emula-

tors may range from one to several orders of magnitude, an unacceptable penalty for production machines. In addition, current honeypots suffer from other disadvantages [16, 32]:

1. Honeypot avoidance. An attacker may create a hitlist containing all hosts that are not honeypots and attack only those machines.
2. Configuration divergence. The configuration of honeypots often does not match exactly the configuration of production machines. For instance, users may have installed different versions of the software, or additional programs. Honeypots only reflect a limited set of configurations. Indeed, high interaction honeypots typically have a single configuration.
3. Management overhead. Honeypots require administrators to manage at least two installations: one for the real systems, and one for the honeypot.
4. Limited coverage. Even if a honeypot covers a sizable number of IP addresses, it may take a while before it gets infected. This is especially true if the honeypot only covers dark IP space. Moreover, the address space that is covered is limited by the amount of traffic that can be handled by a single honeypot.
5. Server-side protection. Most honeypots mimic servers. They sit in dark IP space and wait for attackers to contact them. Unfortunately, the trend for attackers is to move away from the servers in favour of client-side attacks.

The last issue is perhaps the most alarming. Since 2003, client-side attacks are increasingly common. Such attacks often exist of hackers taking over client machines and grouping them into botnets, that are subsequently used for unwanted and illegal activities, such as spamming, on-line fraud, distributed denial of service (ddos) attacks, and harvesting of passwords and credit cards. In this case, clients may be infected because they access servers with malicious content. Such attacks are not caught by most current honeypots. Client honeypots are much less common, and for the few that do exist (e.g., [30, 21]), the other problems remain.

Less expensive methods than taint analysis for intrusion detection exist, but suffer from other problems. For instance,

measures like StackGuard [11], and address / instruction set randomisation [7, 15], while cheap, provide little information about the attack. In other words, such approaches do not allow for signature generation or convenient analysis. Once detected, it has been suggested to replay attacks against more heavily instrumented machines [20, 29], but the problem of replaying is still a research issue in the presence of challenge/response authentication [12, 18]. Moreover, heavily instrumented machines that serve as target for many alerts may not scale easily.

To solve both the issues related with current honeypots, as well as the shift to client-side exploits, we propose turning a host into a honeypot in its spare time, and at user’s request respectively. Doing so represents a radically new way of employing honeypots and may change the way we think about network security. We briefly sketch both usage scenarios and the way they solve each issue.

Idle-time honeypots

Studies suggest that PCs tend to be idle more than 85% of the time [14]. While not all of this time may be harvested for the honeypot, normal machines, and certainly client machines tend to be idle more or less continuously for over 16 hours a day with many long breaks even in the ‘active’ hours. Much like a screen-saver kicks in after a configurable amount of time of keyboard inactivity, we switch to honeypot mode in idle time. Honeypot mode would take the existing, running software on the machine and run it in an instrumented fashion. Moreover, it uses the machine’s normal, live IP address(es).

If at any time, any host can be a honeypot, the rules of the game for the attackers change significantly. For instance, they can no longer harvest a set of IP addresses in advance, because what appears to be a suitable target now may be a heavily instrumented honeypot by the time you attack it. As long as some machines in the set run as honeypot, the attacker risks being exposed. As a result, important classes of attack are rendered obsolete.

Honey-on-demand

An alternative application of *Eudaemon* is sometimes popularly referred as ‘the big red button’, i.e., a button that users may press when they are about to access an unknown and possibly suspicious website, or when they open attachments, view images or movies, etc. Pressing the button will make the application run in honeypot mode, heavily instrumented and safe against client-side exploits.

The button is of course a metaphor that represents an interface for determining which application should be protected when. Besides end users, the interface could be used by applications. For instance, mail readers could demand to be run in emulation mode when opening an email, while running at native speed the rest of the time.

Also, it may be used for servers. Often, when a vulnerability is announced, there is not yet a patch available. Even if there is a patch available, administrators may be reluctant to apply it right away. If the server is not too heavily loaded, *Eudaemon* may be used to run the server in safe mode, thus buying precious time until the patch can be ap-

plied. Such usage of *Eudaemon* escapes the honeypot and client-side exploits domains, and enters the area of intrusion prevention.

Eudaemon

In this paper, we present the design and implementation of *Eudaemon*, a ‘good spirit’ capable of temporarily possessing unmodified applications at runtime to protect them from evil. The paper focuses mainly on the *techniques* for possession, protection, and release, rather than on the applications that may make use of them. So far, *Eudaemon* has only been implemented on Linux, but a port to Windows of the possession and release code is almost completed. As a contribution, this paper presents a novel idea for security which can be applied in different ways and also shows in detail how such a switch to and from honeypot mode works in a modern operating system.

In summary, the technique on Linux works as follows. When *Eudaemon* receives the order to possess a process, it attaches to the process using the Unix `ptrace` system call. The call provides a means by which a process may observe and control the execution of another process, and examine and change its core image and registers. We temporarily pause the execution of the victim process, save its processor state, and inject a small amount of shellcode in its address space. The shellcode calls a modified version of the Argos emulator which is linked in as a library. The emulator is started with the processor state that was previously saved. From that point onwards, execution of the process code continues, except that Argos provides full-blown taint analysis, and raises an alert whenever data with suspicious origins (e.g., the network) is used in a way that violates the security policy. When *Eudaemon* receives the order to release the process, it halts the process temporarily, removes itself from the process and resumes the process in native mode. In other words, network applications (e.g., peer-to-peer or FTP download systems), besides being inactive for a few milliseconds, are not interrupted for the possession period.

The remainder of this paper is organised as follows. In Section 2 we place our work in the context of related work. Section 3 describes the modified version of our Argos emulator with details about how we folded the original system-wide emulator in a userspace library and how we do the tainting. The process of possession and release is discussed in Section 4. Section 5 evaluates both performance and functionality. Conclusions are drawn in Section 6.

2. RELATED WORK

While taint analysis is used by many projects (TaintCheck [22], Minos [10], Vigilante [20], and Argos [24]), most of the existing work assumes deployment on dedicated honeypots, for performance reasons. This is equally true for client-side honeypots [30, 21]. As a result, these techniques suffer from most of the problems identified in Section 1.

One of the first attempts to make emulator-based taint analysis suitable for deployment on production machines is the work by Ho et al [4] which describes practical means of speeding up a Xen/Qemu-based honeypot. In their paper, they describe a Xen virtual machine that switches execution to a modified Qemu to perform fine grained tracking of

network data, and consequently detect attacks. Their approach resulted in a slowdown of a factor 2 up to a factor of 100 times compared to native execution, which prohibits its usage in a real systems. Furthermore, it requires that Xen is installed at the host, which excludes the application of the system by the majority of desktop users. Finally, while providing a system-wide solution provides a higher lever of security overall, the performance and administration overheads deem it inapplicable on desktop systems, and as such an application based solution is more appropriate.

We deal with performance penalties by running in slow mode on demand. In essence, we slice up program execution in the temporal domain. A different way of slicing program execution is described in [19]. This approach assumes a software monoculture and suggests that each node that runs a particular application, executes a part of it in emulated mode. In other words, they slice up the application in the spatial domain. As a result, a distributed system is needed to cover the full application and information is exchanged when a vulnerability is found. In contrast, *Eudaemon* directly benefits individual installations. Nevertheless, exchanging information about attacks among hosts seems attractive for our purposes also and is considered future work. Another advantage of our work is that we do not depend so much on a software monoculture. In practise, we think that the OS used by communities tends to be quite uniform, but variation exists in applications, due to plug-ins, customisations and other extensions.

Yet another interesting way of coping with the slowdown (and indeed, a way of slicing in the temporal domain for servers) is known as shadow honeypots [5]. The idea is to use a fast method for classifying certain traffic as potentially malicious with few false negatives, while allowing for some false positives. Such requests are then handled by the honeypot rather than by the main server. The technique requires a careful tradeoff between the number of false positives and false negatives generated by the pre-classification, as false positives may overload the server, while false negatives would endanger the main server. In addition, shadow honeypots suffer from the problems of configuration and management overhead identified in Section 1. Nevertheless, the idea of pre-classification could well be applied to *Eudaemon*.

Other means of protection at the client-side include advanced filters (e.g., as generated by Vigilante or VSEF [20, 8]), firewalls [31], and intrusion prevention systems on the network card [13]. As these approaches use existing filters, none of these cater to zero-day attacks. Vigilante will not be run on the host because of the overhead. This is also true for TaintCheck and similar approaches [22].

To some extent the problem of zero-day attacks also holds for virus scanners [28], except that modern virus scanners do emulate some of the data (e.g., some email attachments) received from the network. However, remote exploits are typically beyond their capabilities.

Protection mechanisms such as StackGuard [11], PointGuard [9], and address space and instruction set randomisation [7, 15] protect against certain classes of attack, but are unable to

generate much analysis information about the attack, let alone generate signatures.

Many groups have tried to use a fast detection method like address space randomisation and perform more detailed instrumentation on a different host by replaying the attack [29]. In our opinion, replaying arbitrary traffic automatically is still very difficult, although promising results have been attained [12]. Problems include challenge/response authentication and cookies. Also, the heavily instrumented machines that are meant to do the analysis may become a bottleneck if many attacks are detected. *Eudaemon* inherently scales because it employs the users' machines.

Process hijacking is a fairly common technique in the black-hat community [2, 25]. By injecting code into a live processes, such attacks are fairly hard to detect, as no separate process is created and no attack can be detected at the file-system level.

To conclude this section, in 2005 Butler Lampson proposed to partition the world into two zones: green (safe) and red (unaccountable) [17] and use a VM to isolate the two parts. While more work is clearly needed in this area, we believe *Eudaemon* might be a small step toward having the two zones while maintaining an integrated view on the applications.

3. THE PROCESS EMULATOR LIBRARY

Eudaemon is based upon the existence of an emulator that is able to run individual applications, and at the same time protect them against exploits targeting unknown vulnerabilities. It is also desired from the emulator that some data about the detected attack are exported, to enable information sharing on new attacks. In this paper, we investigate the use of a user-space emulator derived from the Argos system emulator.

3.1 Argos

Argos is a secure x86 based system emulator designed for use in honeypots. It is based upon QEMU [6], an open source emulator that employs dynamic translation to achieve fairly good emulation speed¹. The emulated system is secured by applying extended dynamic taint analysis, which in short involves the tagging of network input as "tainted", and consequently tracking the tainted values during execution to identify exploit attempts.

The system emulated, also referred to as guest system, consists of virtual CPU, memory management unit (MMU), and peripherals (i.e. network interfaces, video adaptor, USB hubs, etc). Data entering the system through the virtual network devices are tagged as tainted. The tagging granularity is variable, memory data are tagged per byte, while a single tag is assigned to each of the 8 CPU registers. MMX registers are treated as memory, and have byte granularity as well. Three different models for storing tags assigned to memory data are defined: *BYTEMAP*, *PAGEMAP*, and *PGD*

- *BYTEMAP* uses a byte-sized tag for each byte of mem-

¹Argos is available from: <http://www.few.vu.nl/argos>

ory. An array of equal size to the guest system’s RAM is allocated at start-up, and tags are accessed by using physical RAM addresses as indexes into that array. It is by far the most costly option in terms of memory space, but the fastest in performance. Taking into account the maximum size of a process’s address space on 32-bit Linux, which is usually 3 GB a system of almost 1.5 GB of virtual RAM can be emulated.

- *PAGEMAP* also uses byte sized tags, but instead of reserving all needed space at start-up, it partitions the memory space in pages. When data belonging to a memory page are tainted, tags for that page are allocated and the corresponding tags asserted. The allocated pages are indexed using a single array. Even though the maximum RAM size we can emulate becomes a little less, due to the additional indexing array, the amount of memory practically used is also less. This is due to the fact that some memory pages will never be tainted, because for example they hold program instructions. *PAGEMAP* represents a solution that reconciles mediocre memory usage with minor performance penalties.
- *PGD* is the slowest memory tagging option, but requires the least memory space. Paging is again employed, but instead of using a single level indexing array, a two level look-up procedure is used, similar to the one in x86 CPUs. In more detail, at the first level an array called *page directory* is used to index the second level of arrays named *page tables*. The tags pages themselves are indexed by the page tables. Using such a configuration effectively reduces the amount of memory space needed at the beginning down to 4K, and is independent of the guest’s RAM size. Moreover, a single bit is used to represent each tag, decreasing the size of a tags page from 4K to 512B (assuming a typical Linux page size in x86 systems of 4096 bytes.). *PGD* allows us to emulate a system with more than 2G of RAM.

Tracking of network data is achieved by instrumenting the guest’s instructions to propagate tags. Consider for example the arithmetic ALU operations ADD, SUB, and etc. These are instrumented to taint their result, whenever they are used with a tainted operand. In a similar way, MMU operations such as load and store, copy tag values between registers and memory. As a result, every byte originating or being dependant on network data can be monitored.

Detecting attacks is based upon the observation, that most exploits attempt to either redirect control to attacker supplied instructions (shellcode), or to already available code (libc). This can be accomplished by either loading an attacker supplied value on the instruction pointer (EIP), or by injecting instructions within a program’s control flow. In x86 architectures, manipulation of EIP can be performed using one of the *call*, *jmp*, and *ret* instructions. Argos monitors these instructions, and checks that none of them is used with tainted arguments, or results in EIP pointing to tainted data. Even in the case where EIP is not directly pointed to a tainted location, “walking in” an area with tainted code will eventually cause an alert since attackers are bound to

use a checked instruction (such as *jmp*, *call*, or *ret*). It is, as such, able to detect most overwriting and code injecting exploits.

After an attack is detected Argos generates an alert and logs the state of the emulator to persistent storage. Alerts can be sent to standard output, or alternatively redirected to a socket. The generated logs contain information related to the attacked process that can assist the creation of anti-measures. Argos scans the victim process’s memory and logs all locations that have been marked tainted, as well as the virtual CPU’s registers, and the type of the offending instruction. The produced data can be used to analyse the attack, as well as to generate a network intrusion detection (NID) signature [23] for systems such as Snort [26].

In later versions of Argos, an extension performing more intensive data tracking was introduced. Virtual network devices were extended to log inbound network traffic, and assign a 32-bit index to every received byte. This value is then propagated during emulator execution using enlarged 32-bit tags. This extension increases memory consumption considerably, and introduces performance overhead, but it produces very extensive attack logs that bind every tainted memory byte to its corresponding source byte in the network. Slowinska et al [27] have developed an algorithm that uses such detailed attack logs to pinpoint exactly which bytes contributed to an attack and, based on this information, create very accurate signatures.

3.2 Creating a User-Space Emulator

Argos provides a good starting point for us to develop a user-space process emulator. The dynamic translator performing the instruction instrumentation, as well as the main execution routines can be kept almost intact. Furthermore, Argos shares its code base with QEMU, which already includes a user-space emulator.

3.2.1 Securing the User-Space Emulator

To produce a secure user-space emulator, we applied the extended dynamic tainting principles used in the Argos system emulator on the QEMU user-space emulator. These principles are: network data tainting, tracking tainted data throughout execution, and monitoring the use of tainted data in critical operations. Data tracking has been left unchanged, since the corresponding code is shared between the user-space and system emulator. The sole difference in tracking is the usage of virtual memory addresses, because the emulator now resides in user-space. This section will elaborate on applying the remaining two principles.

Data tainting

Data tainting in the system emulator, is performed by the virtual network devices, an element not present in user-space. Instead data are read using the *read()* system call. This leads to an inability to distinguish network data from other forms of input, since the same system call is used to read data from all descriptors including files, pipes, and sockets. To overcome this limitation we introduced a bitmap used to mark certain descriptors as tainted. Subsequent calls to *read()* result in data tainting only if a tainted descriptor was used. The following system calls are monitored to track descriptor allocation:

- *socket()* is used to create a new socket for network communication. Returned descriptors are all marked tainted. Even in the case of netlink and UNIX domain sockets, it is safe to mark socket descriptors because binaries are not read through them.
- *accept()* is used upon a listening network socket to accept new connections from clients. The newly allocated descriptors are also marked tainted.
- *open()* returns a file descriptor, which can be used to access a file. Normally, this call should be ignored, but instead we allow the user to decide whether some files should also be treated as unsafe input. This enables the emulator also to capture exploits located within files. Consider for example a malicious image or video file that was received as an e-mail attachment, and triggers a vulnerability in the image viewer or movie player. Scanning the path name provided to *open*, we can determine whether the file being opened is located in a declared unsafe directory such as the temporary directory */tmp* or the user home directory */home/...*, and mark the returned descriptor as tainted.
- *pipe()* creates a pair of descriptors that can be used for inter-process communication. We consider the input provided by another process as unsafe, since it is of unknown origin, and taint both descriptors.
- *dup()* and *dup2()* create a copy of a descriptor. If the original descriptor is tainted, then its copy is also marked tainted.

Programs can also use other methods, besides *read()*, to access input. These methods can be categorised in message passing and memory sharing. Messages can be exchanged either over a network socket, or a message queue. In both cases, it is fairly trivial to monitor the message receiving system calls to taint incoming data. However, memory sharing operates in a different manner. A program can request that a file is memory mapped into its address space, or that a memory area is shared with other programs. This implies that simply tainting the memory area involved in these operations is not sufficient. Succeeding memory operations could clean the area's tags, ignoring future updates made by other processes. This is valid for both mapped files and shared memory. To overcome this issue we have extended memory tags to include a sticky flag for every tainted page. Asserting this flag ensures that the page will be always considered tainted ignoring all the writes performed by the process, until it is unmapped or not shared anymore.

Between the three available models for keeping tainted memory tags, only *PGD* is suitable for use in user-space. A model that partitions address space in pages is necessary to inexpensively track the sticky page flag mentioned above. Furthermore, *BYTEMAP* and *PAGEMAP* use a fixed size array which needs to be allocated at start-up and depends on the size of tracked memory. Processes have a large and usually sparse address space, which would make the use of any of the last two impractical and expensive.

User-space presents us with yet another limitation concerning data tainting. The process and the emulator reside in the

same address space, and as such contend for memory. Emulator memory requirements increase as more tainted data are stored in memory. This could lead to a situation where the emulator cannot allocate any more space for tags, resulting in unsafe data left untainted. To bypass such a case, we enforce a limit on the maximum size of the process's data segment. We do not foresee that this would become a problem, since our target space does not include large server applications such as database and application servers. As an alternative, tag pages that have not been accessed recently can be swapped out to disk, but such an approach would penalise performance significantly.

Monitoring the use of tainted data in critical operations remains the same as in the system emulator. Nevertheless, being in user-space offers us the chance to expand monitored critical operations to include certain system calls. In theory, we could apply policies concerning tainted arguments to all system calls, but in practise it only makes sense for the *exec()* system call. *exec()* executes a file by replacing the image of the currently run process with the one in the file. It has been frequently exploited by attackers, who overwrite its arguments with arbitrary executables' names. The Argos user-space emulator is thus shielding programs against such attacks as well.

Generating alerts works in a similar fashion to the system emulator. The alert is sent either to standard output or preferably to a socket, and a log is generated. The attack log has the same format as defined by the Argos system emulator, which we will not expand here for brevity. The only observable difference is the lack of physical addresses in the tainted memory blocks, since these cannot be accessed from user-space.

Finally, we need to protect the emulator's memory against malicious or accidental accesses. The user-space emulator does not contain an MMU, but instead memory is accessed directly. This leaves an opening that would allow the emulated code to access internal emulator state, such as memory tags, since emulator code and vulnerable code reside in the same address space. We once more "abuse" the *PGD* used to store memory tags, to protect the emulator's memory. We insert an "emulator memory flag" in every page table entry, which we use to mark pages that have been allocated or used by the emulator. We then reference the *PGD* on every memory access to check that the translated code is not attempting to use these pages. An additional check is performed on every system call to ensure that pointers passed to the kernel, do actually point to valid areas in memory.

3.2.2 Eudaemon Support

The Argos user-space emulator as described so far, can be used to run applications securely, but it cannot be used with *Eudaemon* yet. Further extensions are necessary to enable the transition of a process from native to emulated execution within Argos. Primarily, it needs to be in a form which can be dynamically included in any process. Dynamic shared libraries or DSOs provide exactly that, namely pieces of code that can be included in programs.

Compiling Argos as a dynamic shared library is trivial, but it requires a simple interface to interconnect with *Eudaemon*.

The following consists a basic interface for interconnection with *Eudaemon*:

- *bool argos_isrunning()*; this function receives no arguments. It returns a boolean value that specifies whether the emulator is active at the moment the function was called.
- *argos_initandrun(struct user_regs_struct *regs, struct user_fpxstate_struct *fpx)*; it consists the library's main entry point. It initialises the emulator with the provided arguments and commences emulation. Its first argument is a structure containing the registers' state², while the second argument is a structure containing floating-point and MMX state.
- *bool argos_stop()*; this function requests that the emulator stops, and consequently *argos_initandrun()* returns. On success *true* is returned, while otherwise, in the case that Argos is not actually running, *false* is returned. Calling this function does not cause the emulator to exit immediately, instead we wait until the virtual CPU reaches a state that it is safe to return.

The emulator also needs to take special care of signal handlers. Upon entering the emulator, the original signal handlers set up by the application are saved, and replaced with a single signal reception point. This is necessary because the application code cannot be called directly. The common signal handler is responsible for setting up a signal delivery context that will eventually transfer control to the translated and instrumented handler. Signal handler updates during emulation are also taken care of, and they are all restored when the process returns to native execution mode.

Finally, the *exec()* system call needs to be further modified. Compiling Argos as a library means that if the current process image is replaced with a different executable, as it is done by *exec()*, we will have to re-attach to it and switch to emulation mode, or let the newly called binary execute natively. The latter case is the default, while to support the first we have *exec()* optionally signalling a process, notifying it of the event. The signalled process can then use *Eudaemon* to force the target into emulation mode once more.

4. EUDAEMON

This section will describe the procedure of *possessing* and *releasing* a system process. The terms *possession* and *release* will be used to describe the act of switching a process's execution to emulated and native mode respectively. Process possession and release are two distinct operations that are independent from each other in the sense that no state needs to be preserved between the two. In other words, a possessed process holds all the information needed for its release. The only prerequisite for these operations is that the emulator library is already present within the target process's address space.

²Structure *user_regs_struct* is defined in the Linux kernel to provide a user-space interface to a process's state, including general purpose registers, EIP, EFLAGS, and segment registers

4.1 Overview

We exploit the shared library pre-loading mechanism in Linux to transparently load the emulator library within the address space of every process. In detail, Linux (as well as other Unix based systems) supports the pre-loading of dynamic shared libraries in applications using dynamic linking. This is accomplished by either defining the environment variable *LD_PRELOAD* to include the desired library, or by including it in the file */etc/ld.so.preload* (the exact file location might vary depending on Linux distribution).

Our technique is not limited to systems that support such a mechanism. For instance, in Windows we employ dynamic linked library (DLL) injection to accomplish our goal. There are more than one ways to force a program to load a DLL in Windows, but we will not elaborate. Briefly, one option is to inject some assembly code (shellcode) within the target process, which will in turn load the DLL in memory. In the rest of this section we will develop our technique assuming the underlying operating system (OS) is Linux.

Eudaemon exploits the Linux system call *ptrace()*, which was originally intended mainly for debugging purposes. Using techniques based on those used by debuggers, we use *ptrace()* to achieve possession and release without process and OS cooperation.

In summary, *ptrace()* allows one process to attach itself to another, assuming the role of its parent. The target is stopped and the attaching process receives control over it. The attaching process is then able to read the target's state, such as register values, floating point (FP) and MMX state, as well as memory data. It is also able to resume execution of the target process, while receiving notification of events such as system call execution and signal reception. This allows *Eudaemon* to access process state, and to inject the instructions needed to perform the switch from native execution to emulation and vice versa.

Possession, and later on release, of a process is performed by solely providing a process identifier (PID). In Linux, there is some ambiguity concerning PIDs, emerging from the fact that in multi-threaded programs PIDs and thread identifiers (TIDs) are in many cases equal. In fact, unless a process belongs to a thread group these two are always equal. This equivocality does not affect *Eudaemon*, since in practise the PIDs reported by Unix utilities such as *ps* represent TIDs and uniquely identify a process throughout its lifetime. Furthermore, both procedures are thread-safe in the sense that concurrent operation upon two or more threads of a process is possible, since -as we will describe further on- there is no use of its currently used memory space.

To initialise and start the emulator library effectively, we inject shellcode into the target process. This is not alike methods used by attackers to conceal their own code within system processes [2]. Since the entire procedure has to be thread-safe, the shellcode cannot make use of target memory. For this reason it does not contain any variables, instead we set the variables using the process's registers, which we can modify at will using *ptrace()*. Consider for example a function call that in x86 assembly can be expressed as *call eax*. To setup *eax* in this case, we modify the target's regis-

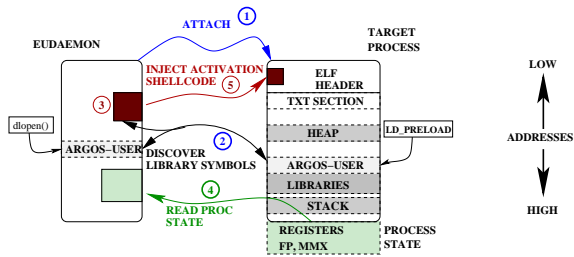


Figure 1: Process possession: phase 1

ters placing the address of the desired function in *eax*, and then allowing it to resume.

The location where we inject the shellcode is the beginning of the target executable’s ELF header. The first 240 bytes of the header remain unused after the binary is loaded, and consist an ideal candidate. The same technique is also used by virus writers injecting parasitic code [3]. An alternative would be to place the shellcode in any location within the target process after backing up its contents using *ptrace()*, and restoring the original contents after the shellcode is finished executing. An even more complex solution is to exploit the space available between functions within an ELF. Modern compilers align functions on boundaries powers of two, leaving some unused bytes between functions. By segmenting our shellcode into smaller pieces we can place it within those gaps [1].

In the remainder of this section, we will go into the details of the possession and release operations.

4.2 Process Possession

4.2.1 Setting Up The Switch

The possession operation can be logically split in two phases. The first phase is shown in Figure 1 and consists of the following steps. Each of the steps will be explained in more detail below.

1. attach to target process;
2. discover the necessary emulator library symbols in target;
3. modify shellcode using the located symbol addresses from step 2;
4. read target’s state (registers, FP and MMX state);
5. inject activation shellcode.

To possess a process we first attach to it, and wait until the target is effectively stopped by the OS. Subsequently, we look up the target’s memory mappings to find out the location of the emulator dynamic shared library in its address space. We accomplish this by looking up `/proc/[pid]/maps`, where `[pid]` is the target PID. This is a file under the special *proc* filesystem, and contains a description of the memory mappings used by each process Figure 2 shows the contents of such a file. Every line of this file corresponds to a memory

```

08048000-08049000 r-xp 00000000 03:04 4450978 loop
08049000-0804a000 rw-p 00000000 03:04 4450978 loop
40000000-40016000 r-xp 00000000 03:04 719528 /lib/ld-2.3.6.so
40016000-40018000 rw-p 00015000 03:04 719528 /lib/ld-2.3.6.so
40018000-40019000 r-xp 40018000 00:00 0 [vdso]
40019000-4001a000 rw-p 40019000 00:00 0
40034000-400c1000 r-xp 00000000 03:04 3140602 libargos.so.0.2
400c1000-400c9000 rw-p 0008c000 03:04 3140602 libargos.so.0.2
400c9000-42118000 rw-p 400c9000 00:00 0
42118000-42240000 r-xp 00000000 03:04 719531 /lib/libc-2.3.6.so
42240000-42241000 r--p 00127000 03:04 719531 /lib/libc-2.3.6.so
42241000-42244000 rw-p 00128000 03:04 719531 /lib/libc-2.3.6.so
42244000-42246000 rw-p 42244000 00:00 0
42246000-42267000 r-xp 00000000 03:04 719535 /lib/libm-2.3.6.so
42267000-42269000 rw-p 00020000 03:04 719535 /lib/libm-2.3.6.so
bfa87000-bfa9d000 rw-p bfa87000 00:00 0 [stack]

```

Figure 2: Contents of a `/proc/[pid]/maps` file - note the presence of `libargos`

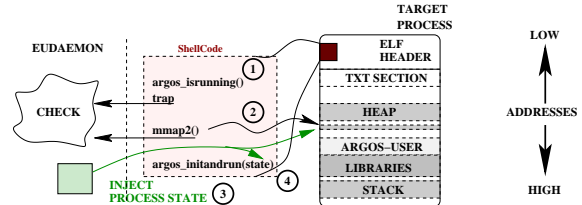


Figure 3: Process possession: phase 2

mapping and provides information on its address range, protection bits, size and source filename, if applicable. We are thus able to locate the address where the emulator library was loaded in the target, as well as in *Eudaemon* itself. Observe that `libargos` is listed twice in the file. The reason for this is that `bss` is also listed.

With this information in hand, we can at runtime look up any emulator symbol in the target. We accomplish this by also loading the emulator dynamic shared library in *Eudaemon*, using dynamic loading and linking, and calculating the offset of the symbol from the beginning of the dynamic shared library. The offset of the symbol remains the same in the target, so we can therefore calculate the address of the symbol in the target process. Interesting symbols at this point are: the function that returns whether the emulator is already running (`argos_isrunning()`), and the one starting the emulator (`argos_initandrun()`). We use their addresses to setup the emulator activation shellcode before injecting it in the target.

At this point we read the target process’s state that we need to pass to the emulator. It consists of the values of general purpose and floating point registers, as well as state used by MMX instructions. Finally, before proceeding to the next phase we inject the activation shellcode, as we have described earlier.

4.2.2 Activation Shellcode Execution

The second phase of possession starts by redirecting the target’s execution flow to the beginning of the injected shellcode. The actions performed collectively by *Eudaemon* and the shellcode are shown in Figure 3, and can be summarised into the following:

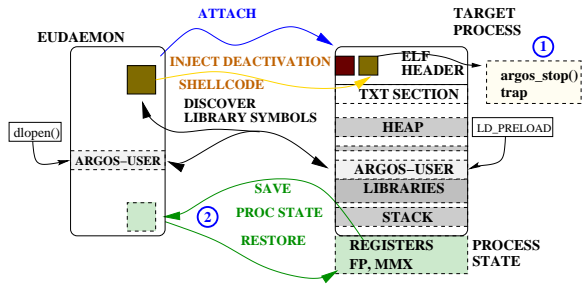


Figure 4: Process release: phase 1

1. check whether target is already possessed;
2. allocate memory to act as a stack for the emulator;
3. push the previously saved process's state in the stack allocated in step 2;
4. call the emulator initialisation and execution routine;
5. detach from process.

To avoid starting a possession procedure for an already possessed process, we first perform a call into the library (`argos_isrunning()`) to discover whether it is already running. The return value of the call is placed within the `eax` register. To retrieve the result, we place a trap instruction right after the call that returns control back to *Eudaemon*, where we can actually check whether we should proceed with the possession, or fallback reinstating the saved process state and detach.

Assuming that the process is not already possessed, execution resumes, and we attempt to allocate a memory area that will be used as a stack for the execution of the emulator. A new stack is necessary, since sharing the active stack between the emulator and the emulated code would lead to error. We use `mmap()` to request a new memory area from the OS, and verify its successful completion by using `ptrace()` semantics to receive control in *Eudaemon* right after the return of the system call.

Taking over the control after the return of `mmap()` is also necessary to supply required arguments to the emulator. These are the process state that we read during the first phase of the possession, and is the exact state where native execution stopped. We inject the data into the newly allocated stack, while also reducing its length by the size of data being stored.

Placing the process state in the emulator stack is the last action performed by *Eudaemon*, which then detaches and exits. The shellcode within the target process performs the last step, and calls the emulator main routine, which initialises itself and starts the emulation.

4.3 Process Release

4.3.1 Stopping The Emulator

Releasing a process is also partitioned in two phases with the first being similar to possession. An overview is shown

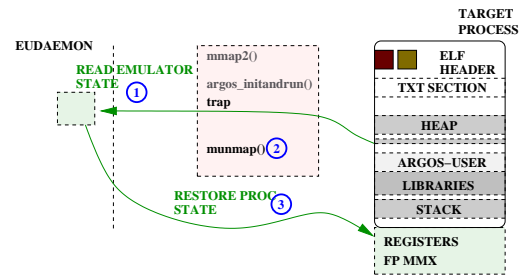


Figure 5: Process release: phase 2

in fig. 4, and the additional steps in respect to possession are the following:

1. call emulator stop routine and check whether the emulator was running;
2. reinstate saved process state and resume execution.

Just like in possession, *Eudaemon* also attaches to the target process, looks up the required library symbols in the target, sets up the shellcode, and injects it. The additional assembly code introduced in the process does not overlap with the shellcode injected during possession, and is quite small in size. It is simply performing a call within the emulator (`argos_stop()`), requesting it to exit. The same function also checks that the emulator is running, so there is no need to perform an additional call to retrieve its state beforehand.

If the process was possessed, `argos_stop()` initiates an exit from the emulator and reports success, while otherwise returns error. We receive control back in *Eudaemon*, by inserting a trap instruction right after the call. We proceed to read its return value to determine whether the release request was valid, in which case *Eudaemon* waits for the emulator to exit. In any other case, it restores the saved process state allowing it to resume execution uninterrupted.

4.3.2 Resuming Native Execution

When the emulator exits, execution returns to the original shellcode planted during possession. The remainder of that code in conjunction with *Eudaemon* is responsible for switching a process's execution back to normal. Figure 5 shows an overview of this procedure, which in brief is:

1. recover emulator state from its stack;
2. release memory space allocated for stack;
3. restore state read in step 1, as process's native state;
4. detach from process;

As soon as the emulator exits, a trap instruction is executed to notify *Eudaemon* of the event. We then re-read the target's state to discover the address of the stack being used, and consequently the location of the emulator state that corresponds to the real process state we need to reinstate for release to be carried out. After recovering the state, the

Application	Native Execution	Emulated Execution	Slowdown
bunzip2	27.99 sec	242.24 sec	856.45%
wget	10.73 MB/s	10.64 MB/s	8.46%
konqueror	29.4ms	463.4ms	1576.19%

Table 1: Emulation overhead

target is resumed and the stack we allocated is freed using `munmap()`. Once again, we use `ptrace()` semantics to receive control when this system call returns, to finally reinstate process state. Finally, we detach from the process effectively completing the release of the process.

5. EVALUATION

We evaluate *Eudaemon* in two aspects: performance and security. The first involves calculating the overhead induced on an application by Argos, as well as the cost of possessing and releasing a process. The latter attempts to evaluate the effectiveness of Argos in capturing certain types of attacks.

5.1 Performance

5.1.1 Argos User-Space Emulator

To evaluate the overhead imposed on an application when emulated by Argos, we measured the performance of a set of UNIX programs when run natively and when emulated. The programs we have chosen are: the *bunzip2* and *wget* utilities, as well as the *konqueror* web browser. *bunzip2* is a very CPU intensive decompression utility, *wget* is a non-interactive network downloader, while *konqueror* amongst other uses is the official web browser and file manager for the K Desktop Environment (KDE). We believe that these three applications represent typical use cases of *Eudaemon* and provide a clear indication on expected performance.

The experiments were conducted on a dual IntelTM Xeon at 2.80 GHz with 2 MB of L2 cache and 4 GB of RAM. The system was running SlackWare Linux 10.2 with kernel 2.6.15.4. The versions of the utilities used were *bzip2 v1.0.3* and *GNU Wget v1.10.2*.

In the first experiment we used *bzip2* to decompress the Linux kernel 2.6.18 tar archive which amounts to about 40 MB of data. For *wget*, we fetched the same file over a dedicated HTTP server located over a 100 Mb/s LAN. Table 1 shows the results of our experiments. In the first case, we used the UNIX utility *time* to measure the execution time of the decompression, while in the latter *wget* calculates the average transfer rate itself. Finally, we measured the time needed by *konqueror* to load and draw an HTML and along with a stylesheet. We used the script found at <http://nontroppo.org/test/Op7/loadtime.html> to conduct the measurement, but had it loaded locally to avoid incorporating variable network latencies in the experiment.

We observe that *bunzip2* under Argos requires about 8.5 times more time to complete. The figure is large, but not unexpected. The overhead is mainly attributed to the dynamic translator and the additional instrumentation instructions. Nevertheless, it is much lower than the performance penalty suffered when using the Argos *system* emulator (i.e., if we

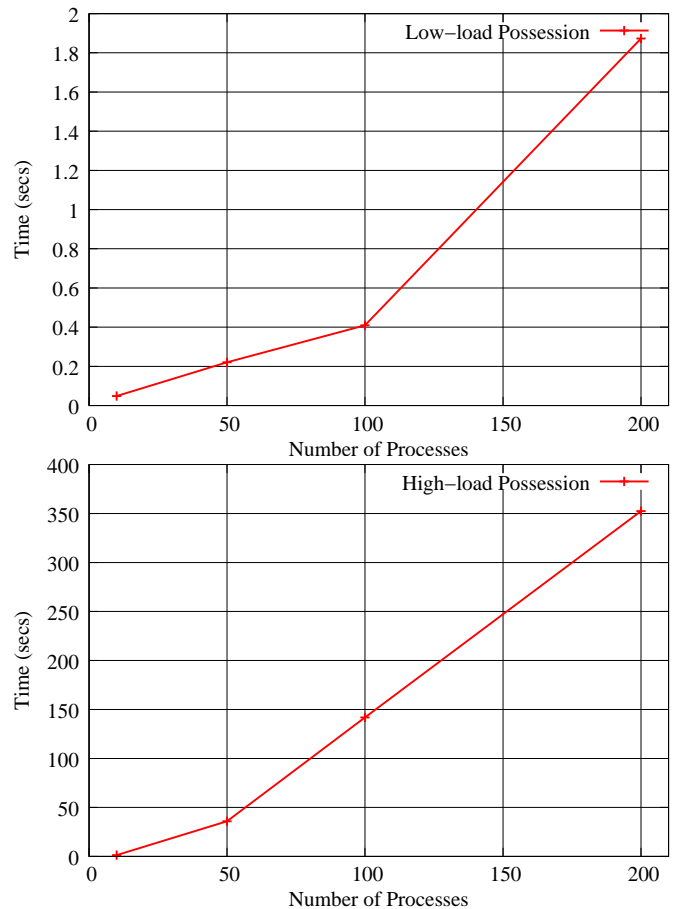


Figure 6: Multiple process possession under low- and high-load

run the entire OS on Argos), which compared to a native system runs at least 16 to 20 times slower. Furthermore, using *Eudaemon* we have the ability to choose when to employ emulation, reducing user inconvenience caused by performance slowdown to a minimum.

The results from *wget* are quite different. The network transfer of a file was subject to insignificant performance loss. Looking into the way *wget* operates, we discover that no data processing is performed, and the sole overhead is imposed by the instrumentation of *read* and *write* calls. The results are encouraging enough to allow for the possibility of running I/O dominated services such as FTP and file sharing entirely in emulation mode.

Finally, *konqueror* offered the worst results for *Eudaemon*. We ascribe this to the fact that rendering the GUI, as well as content, makes use of a lot of instructions that incur very high overhead in emulation mode. Examples of such instructions are floating point operations used for anti-aliasing of text, as well as MMX operations.

5.1.2 Eudaemon

An important performance metric for *Eudaemon* is the time it takes to possess and consequently release a process. We examine these two operations from two different aspects.

Eudaemon Action	Possession	Release
1st phase	1.195	0.159
Waiting time	<i>not applicable</i>	2782.617
2nd phase	0.095	0.106
Total	1.290	2782.882

Table 2: Eudaemon micro-timings (msec)

First we measure the time needed to possess and release a single process, by calculating the time spent on each of the two phases of the operations. Second we measure how process possession scales with an increasing number of target processes.

Table 2 shows the total time needed for the possession and release of a single process, as well as how this time is distributed amongst the different phases as they were presented in section 4. Possession of a single process takes very little time to complete. Release spends even less time performing the two phases, but it is delayed due to waiting for the emulator to exit gracefully. We have to clarify that even though the overall time taken by release is significant, it does not generate overhead for the target process. This is because, while *Eudaemon* is waiting for the emulator to exit, the target process is still executing for the majority of the time in emulation mode.

To measure the performance of *Eudaemon* when multiple process are possessed, we created an increasing number of processes, which we proceed to possess. Figure 6 plots the time needed to possess against to the number of processes. The results also include the time needed to retrieve the PIDs of processes using *ps*, as well as to *fork()* a separate *Eudaemon* process to perform the possession for each target. The two graphs shown represent two different scenarios. In the left graph we possess idle processes that at the time of possession are within *sleep()*, while in the right graph we possess CPU intensive processes with 100% host CPU utilisation. Even though performance is lower in the latter, in both cases *Eudaemon* scales linearly. We believe that this experiment supports our claim that *Eudaemon*'s performance is suitable for the idle-time honeypots and honey-on-demand scenarios as presented in Section 1.

5.2 Security

Dynamic taint analysis [22] and the Argos system emulator [24] have been thoroughly tested against various exploits targeting Windows and Linux operating systems. As a result, we have captured many exploits, including: Apache chunked encoding overflow, WebDav ntdll.dll overflow, IIS ISAPI .printer host header overflow, RPC DCOM Interface overflow, LSASS Overflow, nbSMTTP remote format string exploit, NetApi exploit, WMF exploit.

In fact, we captured many more. However, we do not intend to repeat the evaluation in this paper and refer to [24] for the exploits that were captured by Argos. The methodology employed is able to detect attacks that overwrite critical system values such as function addresses and return address, jump targets, and program instructions. Exploits that use different techniques are not captured by dynamic taint analysis and constitute false negatives. Such techniques manage to

```
char s[30];

gets(s);
printf(s);
....
```

Figure 7: Code vulnerable to stack overflow

```
char *s1, *s2;

s1 = malloc(32);
s2 = malloc(32);
strcpy(s2, "/usr/bin/ls");
gets(s1);
execl(s2, s2, NULL);
....
```

Figure 8: Code vulnerable to heap overflow

influence certain program values without them being tainted or checked for validity. For example, an attacker overwriting a value that is used in a critical branch within a program accomplishes to alter its execution, since such a use of tainted data is not checked or considered invalid. Using tainted values as array indexes can also potentially constitute a problem, since the attacker can implicitly control input without actually tainting it.

We will re-assert the effectiveness of dynamic taint analysis and Argos against other attacks than the ones described directly above, by using two synthetic exploits. We ran two programs containing vulnerable code using the Argos user-space emulator. The attempted exploits were detected, the memory footprint of the victim process logged and then terminated. A description of the exploits and the corresponding vulnerable code follows.

Figure 7 shows some code vulnerable to a stack overflow attack that can overwrite the function's return address. The unsafe *gets()* reads a string from standard input until the end of a line and places it within the character buffer *s*. It does not check though that the input overflows the buffer. As a result we can exploit the code shown, by supplying input longer than 30 characters. The exploit is captured, since the return address is "tainted".

Code susceptible to a heap overflow that can overwrite an argument to the critical *exec()* system call is shown in fig. 8. Two buffers are allocated in the heap, with the first used to read data from standard input using the unsafe *gets()* function. The second is filled with the string path of the legitimate executable *ls*. We exploit the program by supplying a large input that overflows buffer *s1*, with excess data overwriting *s2*. We craft input in such a way that the path to another binary is passed to *exec()*. This exploit is also detected, since the first argument to the *exec()* system call is "tainted".

6. CONCLUSIONS

We have described *Eudaemon*, a technique that allows us to grab a running process and continue its execution in safe mode in an emulator. The emulator provides extensive instrumentation in the form of taint analysis to protect the application. It allows us to turn a machine into a

honeypot in idle hours, or to protect applications that are about to perform actions that are potentially harmful. We have shown that the performance overhead of *Eudaemon* on Linux is reasonable for most practical use cases. To the best of our knowledge, this is the first system that allows one to force fully native applications to switch to emulation in mid-processing. We believe it provides an interesting instrument to increase the security of production machines.

7. ACKNOWLEDGEMENTS

This research is partly funded by the Dutch STW Sentinels DeWorm project and the EU FP6 NoAH project.

8. REFERENCES

- [1] Infecting elf-files using function padding for linux. <http://vx.netlux.org/lib/vhe00.html>.
- [2] Runtime process infection. <http://www.phrack.org/archives/59/p59-0x08.txt>.
- [3] Writing parasitic code in c. <http://ares.x25zine.org/ES/txt/C-parasites.txt>.
- [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] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, E. M. K. Xinidis, and A. D. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proc. of the 14th USENIX Security Symposium*, Baltimore, MD, August 2005.
- [6] F. Bellard. QEMU, a fast and portable dynamic translator. In *In Proc. of the USENIX Annual Technical Conference*, pages 41–46, April 2005.
- [7] S. Bhatkar, D. D. Varney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proc. of the 12th USENIX Security Symposium*, pages 105–120, August 2003.
- [8] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.
- [9] 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.
- [10] 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.
- [11] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, PerryWagle and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Symposium*, 2002.
- [12] 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.
- [13] W. de Bruijn, A. Slowinska, K. van Reeuwijk, T. Hruby, L. Xu, and H. Bos. Safecard: a gigabit ips on the network card. In *Proceedings of 9th International Symposium on Recent Advances in Intrusion Detection (RAID'06)*, pages 311–330, Hamburg, Germany, September 2006.
- [14] W. W. Hsu and A. J. Smith. Characteristics of i/o traffic in personal computer and server workloads. *IBM Systems Journal*, 42(2), 2003.
- [15] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. of the ACM Computer and Communications Security (CCS)*, pages 272–280, October 2003.
- [16] N. Krawetz. Anti-honeypot technology. *IEEE Security and Privacy*, 2(1):76–79, 2004.
- [17] B. Lampson. Accountability and freedom. In *Cambridge Computer Seminar*, Cambridge, UK, October 2005.
- [18] C. Leita, M. Dacier, and F. Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with scriptgen based honeypots. In *Proceedings of RAID'06*, pages 185–205, Hamburg, Germany, September 2006.
- [19] M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Application communities: Using monoculture for dependability. In *Proceedings of the 1st Workshop on Hot Topics in System Dependability (HotDep)*, pages 288 – 292, Yokohama, Japan, June 2005.
- [20] 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.
- [21] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware in the web. In *NDSS*. The Internet Society, 2006.
- [22] 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.
- [23] G. Portokalidis and H. Bos. Sweetbait: Zero-hour worm detection and containment using low- and high-interaction honeypots. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 51(5):1256–1274, April 2007.

- [24] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proc. of the 1st ACM SIGOPS EUROSYS*, Leuven, Belgium, April 2006.
- [25] J. Richter. Load your 32-bit dll into another process's address space using injlib. *Microsoft Systems Journal (MSJ)*, January 1996.
- [26] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proc. of LISA '99: 13th Systems Administration Conference*, 1999.
- [27] A. Slowinska and H. Bos. Prospector: Accurate analysis of heap and stack overflows by means of age stamps. Technical Report IR-CS-031 (a modified version of this report is currently under submission at USENIX Security), Vrije Universiteit Amsterdam, January 2007.
- [28] P. Szor and P. Ferrie. Hunting for metamorphic. In *Virus Bulletin Conference*, pages 123–144, Abingdon, Oxfordshire, England, September 2001.
- [29] J. Tucek, S. Lu, C. Luang, S. Xanthos, Y. Zhou, J. Newsome, D. Brunmley, and D. Song. Sweeper: a light-weight end-to-end system for defending against fast worms. In *Proceedings of Eurosys 2007*, Lisbon, Portugal, April 2007.
- [30] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proc. Network and Distributed System Security (NDSS)*, February 2006.
- [31] S. M. B. William R. Cheswick, Aviel D. Rubin. *Firewalls and Internet Security: repelling the wily hacker (2nd ed.* Addison-Wesley, ISBN 020163466X, 2003.
- [32] C. C. Zou and R. Cunningham. Honeypot-aware advanced botnet construction and maintenance. In *The International Conference on Dependable Systems and Networks (DSN-2006)*, Philadelphia, PA, USA, June 2006.