Eudaemon: Involuntary and On-Demand Emulation Against Zero-Day Exploits

Georgios Portokalidis
porto@few.vu.nl

Herbert Bos
herbertb@cs.vu.nl

Department of Computer Science, Faculteit der Exacte Wetenschappen
Vrije Universiteit Amsterdam
De Boelelaan 1081, 1081 HV Amsterdam, Netherlands

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 control flow, or to inject malicious code will be detected and averted. When desired *Eudaemon* can reattach itself to the emulated process, and return execution to the native binary. Selective emulation has been investigated before as a mean to heal an attacked program or to generate a vaccine after an attack is detected, by applying intensive instrumentation to the vulnerable region of the program. *Eudaemon* can move an application between protected and native mode at will, e.g., when spare cycles are available, when a system policy ordains it, or when it is explicitly requested. The transition is performed transparently and in very little time, thus incurring minimal disturbance to an actively used system. Systems offering constant protection against similar attacks have also been proposed, but require access to source code or explicit operating system support, and often induce significant performance penalties. We believe that *Eudaemon* offers a flexible mechanism to detect a series of attacks in end-user systems with acceptable overhead. Moreover, we require no modification to the running system and/or installation of a hypervisor, with an eye on putting taint analysis within reach of the average user.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*; C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*

General Terms

Security, Management

EuroSys'08, April 1–4, 2008, Glasgow, Scotland, UK.

'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 [28], Minos [11], and Vigilante [24] all use dynamic taint analysis, as pioneered by Dening [15] and evolved by Newsome and Song to capture zero-day attacks in TaintCheck [27]. In essence, data originating from suspicious origins (e.g., network data) are tagged "tainted", and an alert is raised when they are used to divert control flow, or when they are executed. In practice, taint analysis is performed using emulators or binary re-writing tools and incurs an overhead ranging from one to several orders of magnitude. As such, honeypots are ill-suited for full-time deployment on end-user production systems. Additionally, current honeypots have several fundamental disadvantages that severely limit their usefulness [19, 39]:

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, plugins, 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, by sitting in dark IP space and waiting for attackers to contact them. Unfortunately, the trend is for attackers to move away from the servers in favour of client-side attacks [16, 30].

Other intrusion detection methods that do not rely on taint analysis and perform better than it do exist, but suffer from other problems. Measures like StackGuard [12], and address/instruction set randomisation [8, 18] are cheap, but can be sometimes overcome [31] and do not enable the generation of any type of "vaccine" for the exploited fault. Replaying identified attacks against high-interaction honeypots has been suggested [24, 35] to address the latter issue, but successful replaying remains a subject of research in the presence of challenge/response authentication [13, 21]. Moreover, heavily instrumented applications or machines that serve as replay targets for many alerts do not scale easily.

To solve the honeypot problems mentioned above, without sacrificing the generation of valuable data about the attack, we propose to turn end-user hosts into heavily instrumented honeypots. This can be achieved by transparently switching any application between native and intensely instrumented execution, whenever desired and in a timely manner. Previous work in this area proposed selective protection of a particular *segment* of an application [32, 23]. Running the segment in instrumented mode provides the means to generate patches that 'fix' the faults. However, these solutions are dependent on a detector that will initially identify the attacks. We therefore claim that they are complementary to *Eudaemon*.

As an alternative, Ho et al. investigated ways to speed up taint analysis so as to make it deployable as a full-time solution on production machines [4]. Their solution is based on a virtual machine (VM), that transfers execution to a modified Qemu [6] emulator, whenever tainted data are read into the system or processed. They achieve much better performance than other systems providing system-wide protection, but the slow-down is still significant (a factor 2 on average). In addition, they require the installation of a modified Xen hypervisor on the machine which in practice hinders its deployment on the majority of home users' PCs. Finally, while full-system protection is attractive as it also catches attacks on the kernel, the downside is that it becomes harder to provide fine-grained analysis of the actual program under attack.

Ideally, one would make every host operate under heavyweight instrumentation constantly so as to provide full-time safety. Unfortunately, as we have seen, doing so is impractical (at least in the foreseeable future) due to the associated overhead which would likely result in a reduction in user productivity. On the other hand, we propose that it may be possible to explicitly switch to heavily instrumented 'honeypot mode' under certain conditions, provided the conditions are such that they strike a a balance between increased protection and performance. In the remainder of this section, we sketch two such scenarios: idle-time honeypots, and honey-on-demand.

Idle-time honeypots

Studies suggest that PCs tend to be idle more than 85% of the time [17]. This refers to both idleness due to lack of user interaction (idle desktop), and idleness in terms of processing (idle CPU). Client machines display both types, but the former presents an interesting opportunity, and can serve as the condition that triggers the switch to honeypot mode. Much like a screen-saver protecting the screen from damage while the user is away, turning a machine to a honeypot protects running processes (e.g., instant messengers, p2p programs, system services etc) from attacks such as buffer overflows, code injection etc. While acting as a honeypot the machine behaves exactly as it did before, with the sole difference being a reduction in processing speed.

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, and problems 1-4 are resolved.

Honey-on-demand

An alternative application of *Eudaemon* is sometimes popularly referred to 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.

As it may often seem ill-advised to depend on the user for making such decisions (on the other hand, similar principles are used extensively in a modern OS like Windows Vista^{TM}), we stress that the 'big red button' is a metaphor. It represents a generic 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 classified as spam, or from an unknown sender. Also, faster but less accurate intrusion detection systems or access control systems could trigger a switch to honeypot mode in the event of an anomaly. Alternatively, other mechanisms, such as whitelists of network addresses could be used to decide whether a web browser should switch to emulated execution.

Using *Eudaemon* in the above manner helps us tackle the last and potentially most important issue with current honeypots. Since 2003, client-side attacks are increasingly common. Hackers take over client machines and group 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. Such attacks are not caught by most current honeypots and client honeypots are much less common, and for the few that do exist (e.g., [37, 25]), the other problems remain.

Finally, *Eudaemon* 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 applied. Such usage escapes the honeypot and client-side exploits domain, and enters the area of intrusion prevention.

Contribution: Eudaemon

In this paper, we present the design, implementation and evaluation of *Eudaemon*, a 'good spirit' capable of temporarily possessing unmodified applications at runtime to protect them from evil. The contribution of this paper is a novel idea for applying honeypots, with a wide range of possible applications. Our focus is primarily on the *techniques* for possession, protection, and release, rather than on the applications that may make use of them. In addition we explain in detail how such a switch to and from honeypot mode works in a modern operating system.

In a nutshell, when *Eudaemon* receives the order to possess a process, it attaches to the process in such a way that we can observe and control the execution of the target process, and examine and change its core image and registers. Most modern OSes have functionality for doing so (for instance, UNIX provides the `ptrace` system call for this purpose, while Windows XP allows for the creation of threads in target processes). 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 an open source 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 the emulator 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 presents an overview of the system's design. Implementation details are given in section 4. Section 5 evaluates performance, and conclusions are drawn in Section 6.

2. RELATED WORK

Taint analysis is used by many projects such as TaintCheck [27], Minos [11], Vigilante [24], and Argos [28]. Most of the existing work assumes deployment on dedicated honeypots. This is mainly due to performance reasons. Likewise, client-side honeypots tend to be dedicated machines also [37, 25]. As a result, these techniques suffer from most of the problems identified in Section 1.

An interesting exception includes the work on speeding up taint analysis by switching between a fast VM and a heavily instrumented emulator by Ho et al. discussed earlier [4]. One drawback of the method (besides an overhead that is still considerable compared to native execution) is that it can only be installed by, say, home users willing to completely reconfigure their systems to run on a hypervisor.

In contrast, 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 is known as application communities [22]: assuming a software monoculture, each node running a particular application executes part of it in emulated mode. In other words, applications are sliced up in the spatial domain and a distributed detector is needed to cover the full application. *Eudaemon* directly benefits individual installations without relying on a monoculture. In practice, the OS used by communities tends to be uniform, but variation exists in applications, due to plug-ins, customisations and other extensions.

In a later paper, the same groups employs selective emulation to provide self-healing software by means of error virtualisation [23]. Again slicing is mostly in the spatial domain. As far as we are aware, neither of these projects supports taint analysis. Indeed, it seems that for meaningful taint analysis, the tainted data must be tracked through all functions and, thus, *selective* emulation may be more problematic. At any rate, as we mentioned earlier, the *Eudaemon* technique is complementary to [23] and could be used as an attack detector.

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]. A fast method is used to crudely classify certain traffic as suspect with few false negative but some false positives. Such traffic is then handled by a slow honeypot. Tuning the classifier is delicate as false positives may overload the server. In addition, shadow honeypots suffer from the problems of configuration and management overhead identified in Section 1.

Rather than incurring a slow-down at the end users' machine, many projects have investigated means of protection for *services* running on user machines by way of signatures and filters. Such projects include advanced signature generators (e.g., Vigilante, VSEF, and Prospector [24, 9, 33]), firewalls [38], and intrusion prevention systems on the network card [14].

For instance, Brumley et al. propose vulnerability-based signatures [9] that match a set of inputs (strings) satisfying a vulnerability condition (a specification of a particular type of program bug) in the program. When furnished with the program, the exploit string, a vulnerability condition, and the execution trace, the analysis creates the vulnerability signature for different representations, Turing machine, symbolic constraint, and regular expression signatures.

Packet Vaccine [36] detects anomalous payloads, e.g., a byte sequence resembling a jump address, and randomises it. Thus, exploits trigger an exception in a vulnerable program. Next, it finds out information about the attack (e.g., the corrupted pointer and its location in the packet), and generates a signature, which for instance can be based on determination of the roles played by individual bytes. To determine this, Packet Vaccine scrambles individual bytes in an effort to identify the essential inputs.

Vigilante relies on the possibility to replay attacks in order to generate Self-Certifying Alerts (SCAs), an extremely powerful concept that allows the recipient of an alert to check whether the service is indeed at risk [24]. If so, it may automatically generate a signature. The false positives rate seems to be zero.

As mentioned earlier, the signature generators for the above projects may be capable of handling zero-day attacks, but they produce them by means of dedicated server-based honeypots. Hence, they do not cater to zero-day attacks on client applications. To some extent the problem of zero-day attacks also holds for virus scanners [34], 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.

We should mention that we currently have a few signature generators for our emulator (known as Sweetbait [28] and Prospector [33], respectively). As part of our future work, we intend to use different types of signature generator (some with light-weight instrumentation, and other with heavy-weight analysis), so that different users may apply different forms of analysis to the same attack, hopefully yielding a more complete picture of the attack.

Protection mechanisms such as StackGuard [12], PointGuard [10], and address space and instruction set randomisation [8, 18] 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 such fast detection methods like address space randomisation and perform more detailed instrumentation on a different host by replaying the attack [35]. In our opinion, replaying is still difficult due to challenge/response authentication, although promising results have been attained [13, 26]. More importantly, the heavily instrumented machines that perform the analysis may become a bottleneck if many attacks are detected. *Eudaemon* inherently scales because it employs the users' machines.

Process hijacking is a common technique in the black-hat community [2, 29]. By injecting code into live processes, such attacks are hard to detect, as no separate process is created and no attack can be found at the file-system level. Also, Nirvana, as described by Bhansali et al in [7], is an engine for the Windows OS that permits detailed instruction level tracing by means of simulation, and holds the ability to transparently attach on a running process.

To conclude this section, in 2005 Butler Lampson proposed to partition the world into two zones: green (safe) and red (unaccountable) [20] and use a VM to isolate the two parts. While more work is clearly needed in this area, we believe *Eudaemon* might be a step toward having the two zones while maintaining an integrated view on the applications.

3. DESIGN

Eudaemon has been partially inspired by techniques used by hackers and debuggers alike to attach to running applications, instead of requiring them to be loaded from within the controlled debugger context. We use similar techniques to hijack or *possess* a process transparently with the goal of heavily instrumenting unmodified binaries to protect them against remote exploit attempts.

Eudaemon has been designed to run as a system service, where requests to possess or release applications can be made. The terms *possession* and *release* will be used to describe the act of switching a process to emulated and native execution respectively. A high level overview of the system is shown in Figure 1. Requests to possess or release an application can be issued based on any criterion, such as an explicit user request, or as a result of persisting inactivity at the host, as mentioned in the introduction.

After receiving a request (①), *Eudaemon* immediately attempts to attach to the target process to force it to run in an emulator (②). The complexity of the procedure varies depending on the platform implementation, but most modern operating systems do support (system) calls that implement the desired functionality (e.g. Linux, BSD, and Windows XP). Attaching to a process can be performed using its process identifier (PID) alone. When threading is used, the thread identifier (TID) can be used instead.

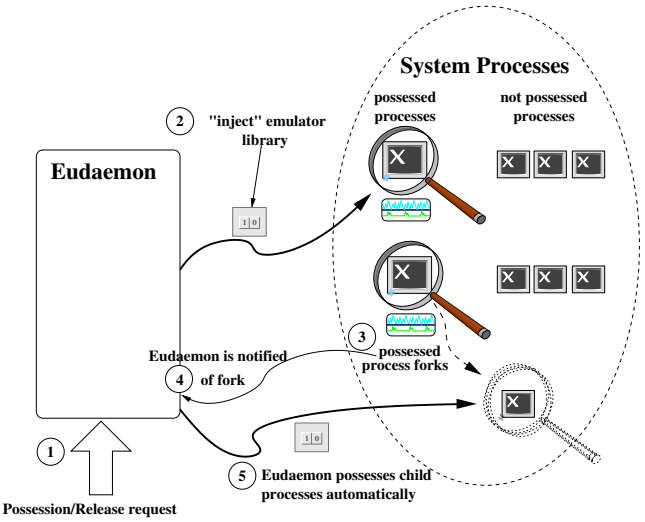

Figure 1: Eudaemon Overview

For safety, the operating system ensures that only a program running under the same or super-user credentials is able to attach to a given process. This scheme guarantees that users cannot possess or release processes they do not own.

When an emulated process spawns new processes (③), we can also request the automatic possession of its children to enable *Eudaemon* to protect an application consisting of multiple processes (⑤). We emphasise that even in this case a program need not be *Eudaemon*-enabled in any way. The emulator library which manages execution makes sure to notify the system on the creation of new processes (④). Threads can be handled internally, since all of them share the same "possessed" address space, and the emulator can proxy new thread requests.

3.1 Process Possession

Switching to emulated execution (possession), is accomplished by *injecting* code to perform this task within the target process space. For threaded applications it is sufficient to inject the code once, since all threads lie in the same address space. Nevertheless, the amount of code required to perform such a complex job can be significant. What this implies is that the costs of copying or injecting the emulator code within a process, could compromise the transparency of the system.

We overcome such an eventuality by making the emulator a dynamic shared library (known as DSO or DLL, depending on the platform). Libraries impose some restrictions on the included code, but on the other hand make code reusing simple and efficient. When a DLL is loaded by multiple processes, it is actually loaded once in system memory, and only mapped in each process's address space. As operating systems allow libraries to be loaded either at runtime, or *a priori* for every process, we have some freedom in how we inject the emulator code efficiently and in a way that will scale even when multiple targets are possessed. For more details on loading the emulator in a process, we refer the reader to Section 4.

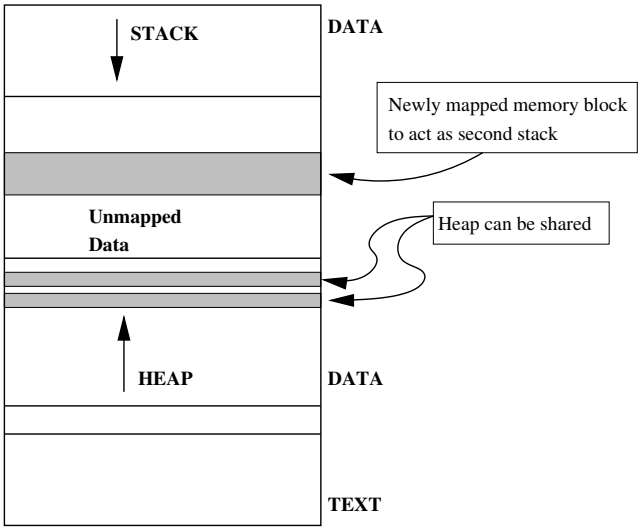

Figure 2: Process Memory Layout

After loading the necessary code in the target process, we still need to activate it, and supply the required state so that execution can resume virtually undisturbed. Acquiring a target's execution state is commonly performed by debuggers and we use the same technique. As we will explain in detail shortly, supplying such state to the newly injected code in the target, is more involved and requires that we protect the integrity of the target. Phrased differently, our code shares the target process's address space, and we need to isolate its memory.

Figure 2 shows the typical layout of a process's memory. Application code is loaded in what is called the *text* segment, and it is protected by being marked read-only. Loaded libraries, even though not shown in the figure, also have their corresponding code segments protected by being read-only, and as such our code is implicitly protected. Process data are stored mainly in two areas: the heap and the stack. Heap size is dynamic, and grows towards larger addresses, while stack is usually fixed, and is used as a LIFO queue. The stack grows towards lower addresses.

Every thread of execution has its own stack, which in architectures like the x86 is addressed using special CPU registers and implicitly updated by special instructions. As a result, executing our code using the same stack as the process code would lead to severe inconsistencies in the stack. In contrast, heap memory is larger and allocated objects are referenced explicitly by holding memory pointers. This permits us to share the heap for any objects we need to allocate. In both cases, however, we cannot rule out the possibility that the program will access data owned by the library either as a result of an error, or as part of a malicious attempt to thwart its proper operation. We handle data protection through the emulator, which we describe in more detail in Section 3.3.

To call safely the code we have already injected, we first map a memory segment in the target process that will serve as a stack for the emulator. This way we ensure that both our code and the process's code can be run in parallel without interfering with each other. The way this is accomplished depends on the underlying system. For example, some systems allow a process to reserve a memory segment in another process, while on others we are forced to inject a piece of short lived code to perform the allocation.

In the latter case, we need to choose carefully the location where to place the short lived *activation* code, so as to not compromise the target's integrity. One possible solution is to choose an area in the target's text memory space, save it, and then overwrite it with the activation code. When the activation code completes, the original code can be restored. However, this process requires pausing all threads in order to guarantee that the location will not be used while the activation code resides there.

Ideally, we would like to avoid such overhead and prefer to inject the code in a location that we know is no longer in use. In practice, the binary object's header that resides in the text segment is often a feasible location. Certain bytes in the header are used only when the executable is loaded by the system in memory. Usage of executable header memory to run code has been demonstrated before by virus writers to inject parasitic code in running programs [3]. A more extreme solution is to use the space left by compilers between the functions of an application for performance reasons [1], but we have not explored such a course in our implementation.

To activate the library, we use a part of the newly allocated stack to store the state we extracted when we attached to the target, and detach from the process. Finally, the activation code calls a function in our library which takes over the execution of the process.

3.2 Process Release

To return a process back to native execution the emulator needs to be notified to clean up and export the state of the process, as it would have been if the process has been running natively all the time. Delivering such a notification could be performed by various means, but to preserve semantics similar to those of possession we chose once more to inject *deactivation* code into the process. The code simply performs a call within the library to deliver the notification.

If the call succeeds then *Eudaemon* needs to wait until the emulator exits, and control is returned to the activation code that was injected during possession. The remainder of that code will notify *Eudaemon* that the process can be switched back to native execution, and if necessary also release the allocated second stack. To complete the switch, *Eudaemon* reads the state of the process as exported by the library, and reinstates it as the process's native state.

3.3 Emulator Library

The emulator library is decoupled from *Eudaemon*, so that it can be transparently replaced without affecting the system's operation. As long as the library adheres to *Eudaemon*'s predefined emulator interface, and the library itself does not compromise the process, any implementation that shields the process against attacks can be used. We now describe at a high level the required interface for a library to be used in *Eudaemon*, and also present the criteria that need to be obeyed in the remainder of this section. The exact library calls will be listed in Section 4. From a high-level perspective, the desired interface consists of three functions:

- We need a function to check that the library is not already in control of the target process in order to handle requests to possess a process that is already possessed.

To avoid possible conflicts the state of the library (active/inactive) is exported via such a call.

- A function is needed to initialise and give control over the process to the library. The function represents the entry point of the library, where control is redirected after setting up memory and process state. It should neither fail nor return, unless there is an error in the program itself or the library was notified to exit.
- The final function we require is one that signals the emulator library to relinquish control of the process, and return to the caller. This call need not be synchronous, in the sense that the library does not need to terminate immediately. *Eudaemon* will wait for the process to complete the switch to native execution.

A more important aspect of the library is that it should protect itself from unintentional or malevolent access of critical data. As we briefly mentioned earlier, a program could access library data in stack or heap, and in that way compromise the mechanism that is supposed to be protecting the application. To guard against such a possibility we adopt a memory protection method very similar to the one used in the x86 CPU architecture (see Section 4.1.4 for details).

4. IMPLEMENTATION

We completed an implementation of *Eudaemon* on Linux. We also completed most of the possession and release functionality for Windows, while work on the required library-based emulator is in early stages. Due to size restrictions, in the remainder of this section, we only discuss in detail the Linux implementation of the main components of our design: (i) a process emulator that implements taint analysis, and (ii) *Eudaemon* possession and release.

4.1 SEAL: A Secure Emulator Library

SEAL is a secure, x86-based user-space process emulator implemented as a library. It is based on Argos and employs the same dynamic taint analysis [28]. In a nutshell, Argos is a whole-system emulator and consists of a virtual CPU, NICs, video card, etc. It marks all data arriving on its virtual NICs as tainted and tracks them throughout their lifetime in the system. Argos raises an alert when tainted data is used in illegal ways (e.g., when it is executed).

We modified Argos in the following ways. First, we do not desire whole-system emulation, so we ported the dynamic taint analysis functionality to a user-space emulator. As Argos shares its code base with Qemu [6], which includes a user-space emulator, doing so was straightforward. Second, a user-space emulator has no notion of virtual NICs, so we had to modify the tagging mechanism. For instance, SEAL tags bytes when they are read from sockets (and certain other descriptors). Third, as the original process and SEAL share the same address space, we had to protect data used by SEAL from being clobbered by the process. Fourth, we packaged SEAL as a library with a succinct interface. We now discuss these issues and the general operation of SEAL in detail.

4.1.1 Tagging Data

Processes read data by means of the `read` system call which is used for sockets, files and pipes. To distinguish suspect data from harmless input, we introduce a 64KB bitmap (a bit for each one of the possible 2^{16} descriptors in Linux) that marks certain descriptors as tainted. Calls to `read` result in data tagging only if a tainted descriptor was used. We now describe how we monitor system calls to taint descriptors. First, `socket()` and `accept()` both create descriptors for network communication. As network data is suspect, the descriptors are marked tainted. Second, `open()` returns a descriptor for file access. Normally, we ignore this call, but users are allowed to mark certain directories as *unsafe* to capture exploits in files. Consider a malicious image in an attachment that triggers a vulnerability in the viewer. SEAL scans the path name provided to `open`, and taints the descriptor if the file is in a directory marked *unsafe* (e.g., `/tmp`, or `/home/...`). Third, the *pipe()* call creates a pair of descriptors for inter-process communication. SEAL considers input from another process as unsafe, since it is of unknown origin, and taints both descriptors. Finally, *dup()* and *dup2()* create a copy of a descriptor. If the original descriptor is tainted, we also taint the copy.

Besides the *read()* system call, programs can access input by means of message passing and memory sharing. Messages can be exchanged either over a network socket, or a message queue. In both cases, we can trivially monitor the message receiving system calls to taint incoming data. Handling shared memory is more difficult. Programs may either map files into their address spaces, or share memory pages with other programs. Simply tainting the memory is not sufficient, because it would miss updates made by other processes. We therefore included a sticky flag for every tainted page. Asserting this flag ensures that the page will be always considered tainted ignoring all writes performed by the process, until it is unmapped or not shared anymore.

4.1.2 Tracking Tainted Data

Data items tagged as *tainted* are tracked during execution. Tracking is achieved by instrumenting the guest's instructions to propagate tags. For example, arithmetic operations like ADD and SUB, 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. 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. As a result, every byte that depends on network data can be monitored.

Tags in SEAL are accessed through a one-level page table. We partition memory space in pages, and only when data belonging to a memory page are tainted, tag space for that page is allocated and the corresponding tags asserted. The page table contains pointers to structures actually containing the tags for each page, where a tag can either be a single bit, or a byte. While it would have been faster to use a one-dimensional tag array, we wanted to keep the memory footprint of the emulator as small as possible, especially since SEAL and user application share the same address space. In addition, by aligning the dynamically allocated blocks of tags on addresses that are multiples of four, the least significant bits of page tables entries are unused, and can be used to track inexpensively the sticky page flag mentioned above.

When a typical Linux process is running SEAL using single bit tags, the amount of memory X (in MB) that can be used by the process can be expressed as: $X + (X/8) + 4 <$

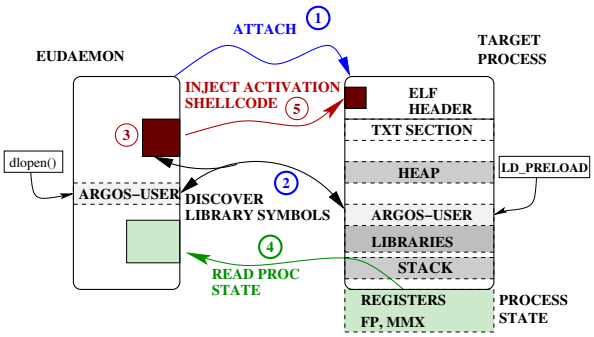

Figure 3: Process possession: phase 1

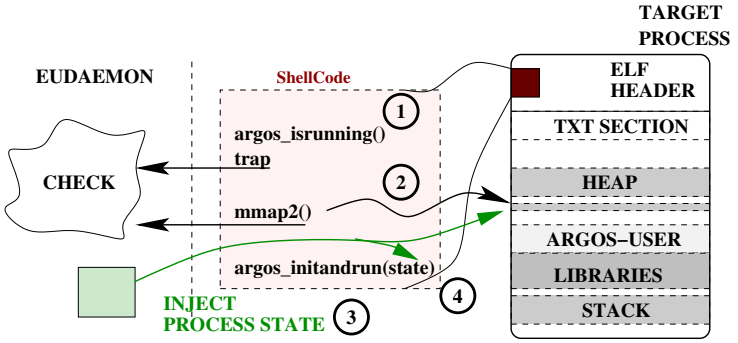

Figure 4: Process possession: phase 2

```
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 libseal.so.0.2
400c1000-400c9000 rw-p 0008c000 03:04 3140602 libseal.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 5: Contents of a /proc/[pid]/maps file - note the presence of `libseal`

3072 (the maximum addressable virtual memory being 3 GB or 3072 MB, the page table taking up to 3 MB, and 1 MB taken by library code and statically allocated data). So, a process under SEAL can use up to 2727 MB of the virtual address space, reducing its maximum available memory by about 9.64%. At runtime, the actual memory footprint of the library depends on application behaviour, and the amount of tainted data. We can calculate a 12.5% upper boundary for the memory overhead imposed by the library, if we assume all process data are tainted and a single bit is used for each byte.

4.1.3 Attack Detection

Most exploits attempt to redirect control to shellcode provided by an attacker, or to code that is already available (libc). They do this either by loading an attacker supplied value on the instruction pointer (EIP), or by injecting instructions within a program's control flow. On x86 CPUs EIP is manipulated using one of the *call*, *jmp*, and *ret* instructions. SEAL 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). In other words, SEAL is able to detect most overwriting and code injecting exploits.

After an attack is detected, SEAL generates an alert and logs the state of the emulator to persistent storage. It 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. It also collects information (like pid, name, and DLLs) about the victim application. The logs are subsequently used by signature generators to create anti-measures. Signature generation in Argos/SEAL is beyond the scope of this paper. Interested readers are referred to Sweetbait [28] and Prospector [33].

4.1.4 Protecting SEAL Data

As application and SEAL reside in the same address space, we need to protect emulator data against malicious or accidental accesses by the application. As mentioned earlier, our solution resembles memory protection in x86 architectures. The x86 CPU contains a hardware memory management unit (MMU) that partitions the linear physical memory space into pages of virtual memory space. The MMU is responsible both for translating a virtual address to a physical one, as well as enforcing a page protection mechanism. This way every process is assigned each own virtual address space isolating it from other processes, and protecting kernel space from the processes.

We adopt the same principle by using a virtual MMU that enforces page level protection. SEAL instruments all memory accesses in the application's code to go through the virtual MMU, where they are validated to make sure that library owned memory is not accessed. Every page allocated by the library is marked with a flag that allows the virtual MMU to perform the validation. The structure that these flags are stored in is of small importance; a reasonable choice in our case was to use one of the extra bits in the page table described in Section 4.1.2.

Keeping track of protected memory pages is straightforward. It only requires that on allocation and release of heap memory the library updates the corresponding bits. The virtual MMU can also be used to protect the stack, global library data, and library read-only data to defend against information leakage that could be exploited by attackers. Obviously, it protects its own bitmap and data, while the code is protected in the same way as all other code.

4.1.5 Checking System Calls

Monitoring the use of tainted data in critical operations is the same as in the whole-system emulator. However, being in user-space offers us the chance to expand operations that are monitored to include certain system calls. In theory, we could apply policies concerning tainted arguments to all system calls, but in practice it makes sense primarily for the *exec()* system call which executes a file by replacing the image of the current process with the one in the file. It has been frequently exploited by overwriting the arguments

to load arbitrary programs. By checking the arguments for tainted tags, SEAL shields programs against such attacks.

4.1.6 Signal Handling

SEAL handles signals transparently to the application. Upon receiving control of a process, original signal handlers are replaced with the emulator's handler. This single signal reception point queues arriving signals that will be processed at a later time. System calls used to update signal handlers and masks, are also intercepted to keep track of the process's signal related behaviour.

Such an approach is necessary to ensure that native code is never called directly, but to allow also switching to emulation mode while executing a signal handler at the target. To clarify this point, we will briefly describe how the Linux kernel handles signals. Upon signal delivery, a new temporary execution context is created by the kernel for the handler to execute, and the previous context is saved in user-space. Before relinquishing control to the signal handler, which runs in user-space, a call to *sigreturn()* is injected in the temporary handler context. This system call serves the sole purpose of returning control to the kernel, so that the original execution context can be restored. When SEAL is in place, it imitates the kernel. As a result, if the emulator receives control while a signal handler is executing, it is still able to switch to the process's original execution context in emulation mode by intercepting *sigreturn()*.

4.1.7 Eudaemon Support

The SEAL user-space emulator as described so far, can be used to run applications securely, but cannot be used with *Eudaemon* yet. To enable the transition of a process from native to emulated execution we need further extensions. Primarily, SEAL needs to be in a form which can be dynamically included in any process. Dynamic shared libraries or DLLs provide exactly that. Compiling SEAL as a dynamic shared library is trivial, but it requires a simple interface to interconnect with *Eudaemon*. We use the following as a basic interface for interconnection with *Eudaemon*:

- *bool seal_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.
- *void seal_initandrun(struct cpu_state *st)*. This is the library's main entry point. It initialises the emulator with the snatched process state such as register values, MMX, and floating-point state, and commences emulation.
- *bool seal_stop()*; this function requests that the emulator stops, and consequently that *seal_initandrun()* returns. It returns *true* on success and *false* if SEAL is not actually running. Calling this function does not cause the emulator to exit immediately. Instead it waits until the virtual CPU reaches a state that is safe to return.

Finally, the *exec()* system call also requires modification. Compiling SEAL as a library means that if the current process image is replaced with a different executable by *exec()*, we have to re-attach and switch it to emulation mode, or let the newly called binary execute natively. By default we use the latter option. To support the former, we permit *exec()* to signal *Eudaemon* of the event, so that the new process can be forced into emulation mode once again.

4.2 Possession And Release

Process possession and release are two distinct operations that are independent 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 present in the target process's address space.

The finer details of injecting the library in the target process, as well as activating and deactivating it are in some cases very dependent on the underlying OS platform. In the remainder of this section, we elaborate on the implementation details of *Eudaemon* on Linux.

We use the shared library pre-loading mechanism in Linux to transparently load the emulator library in the address space of every process. In detail, Linux and other Unix based systems support 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 a configuration file (like */etc/ld.so.preload*).

Eudaemon employs the Unix system call *ptrace()*, which was originally intended mainly for debugging purposes. Much like a debugger, 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.

4.2.1 Process Possession

The possession operation can be logically split in two phases. The first phase is shown in Figure 3 and consists of the following steps: (1) attach to target process; (2) discover necessary emulator library symbols in the target; (3) modify activation shellcode using the symbol addresses acquired during step 2. Each of these steps will be explained in more detail below.

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 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 5 shows the contents of such a file. Every line of this file corresponds to a memory 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 `libseal` is listed twice in the file. The reason for this is that `bss` is also listed.

With this information, we can at runtime look up any

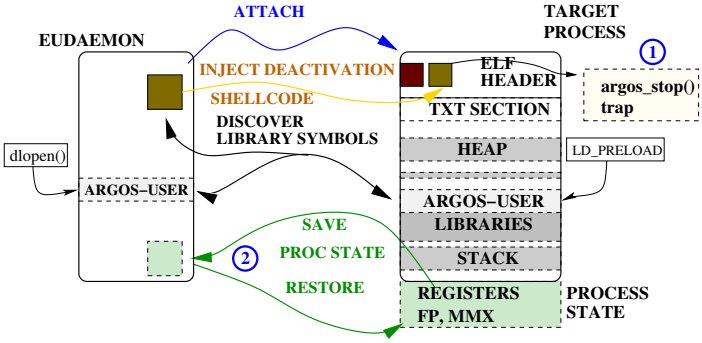

Figure 6: Process release: phase 1

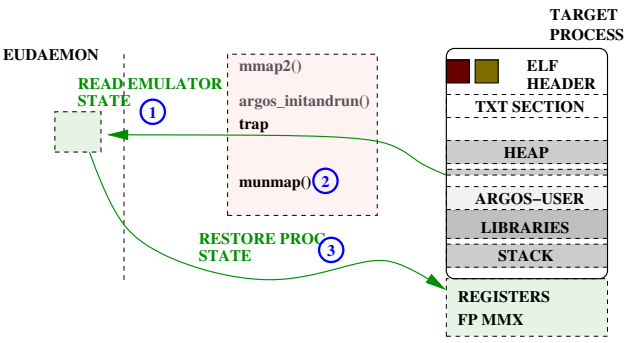

Figure 7: Process release: phase 2

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 (*seal_isrunning()*), and the one starting the emulator (*seal_initandrun()*). Using their addresses we setup the SEAL 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, in the ELF header of the executable which contains 240 bytes that remain unused after loading the binary into memory.

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 4, and can be summarised into the following: (1) check that the target is not already possessed, (2) allocate a memory block to be used as stack by the emulator library; (3) store the process state saved during the first phase in the memory block obtained in step 2; (4) call the initialisation and execution function of the emulator; (5) detach from the target process.

To avoid starting a possession procedure for an already possessed process, we first perform a call into the library 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.

Assuming control after the return of *mmap()* is also necessary to supply the required arguments to the emulator. The arguments comprise of the process state that we read during the first phase of the possession, which 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 the 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.

4.2.2 *Process Release*

Releasing a process is also partitioned in two phases with the first being similar to possession. An overview is shown in Figure 6, and the *additional* steps in respect to possession (listed in Section 4.2.1) are the following: (1) call the emulator's stop routine, and at the same time discover whether it was running; (2) reinstate the saved process saved state, and allow it to 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 simply calls the *seal_stop()* function in the emulator, 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, *seal_stop()* initiates an exit from the emulator and reports success, while otherwise it 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.

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 7 shows an overview of this procedure, which in brief is: (1) recover the emulated process's state, stored in the emulator stack; (2) release the memory block that is used as stack; (3) restore the state read in step (1) as native process state.

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 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 how *Eudaemon* performs in two aspects: the overhead induced on an application when executing under the emulator, and the cost of possessing and releasing.

5.1 SEAL

To evaluate the overhead imposed on an application when emulated by SEAL, we measured the performance of a set of UNIX programs when run natively and when emulated by SEAL. We also compare against the Argos full-system emulator. Our benchmark consists of one CPU-intensive application with little I/O (*bunzip2*), non-interactive network downloader with little CPU utilisation (*wget*), a network downloader with encryption (*sftp*), and one interactive graphical browser that performs both downloading and rendering (*konqueror*). Konqueror is the official web browser and file manager for KDE. With this mix of applications, we have covered the spectrum of use cases for *Eudaemon* fairly well so that the results represent a faithful indication of expected performance in general.

The experiments were conducted on a dual Intel™ 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*, *GNU wget v1.10.2*, and *konqueror 3.5.4*.

We used *bzip2* to decompress the Linux kernel 2.6.18 tar archive which amounts to about 40 MB of data. We used the UNIX utility *time* to measure the execution time of the decompression. For *wget*, and *sftp* we fetched the same file from a dedicated HTTP server over a 100 Mb/s LAN. In the experiment we used *wget* and *wget*'s own calculation of the average transfer rate as performance measure. Finally, we measured the time needed by *konqueror* to load and draw an HTML page along with a stylesheet. We used the `loadtime` browser benchmarking utility available from `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. Because of clock skew, a well-known problem with Qemu, we could not measure this test reliably on Argos. Table 1 shows the results.

We observe that compared to native execution *bunzip2* under SEAL requires about 8.5 times more time to complete. The overhead is fairly large, but this was expected and can be mainly attributed to the dynamic translator and the additional instrumentation. Nevertheless, it is much lower than the performance penalty suffered when using the Argos *system* emulator (i.e., if we run the entire OS on Argos), which compared to a native system was reported to run at least 16 to 20 times slower [28]. Furthermore, using *Eudaemon* we can choose *when* to employ emulation, reducing user inconvenience caused by the slowdown to a minimum.

The results from *wget* are quite different. The network transfer of a file was subject to insignificant performance loss. *Wget* performs no data processing, 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.

sftp incurs a slowdown of a factor 6.3. In our opinion, this is surprisingly good considering all the operations on tainted data involved in *ssh*. In other work, the reported overhead is more than two orders of magnitude [4]. We suspect that the difference is caused by the fact that *Eudaemon* attaches on the application after a shared secret key has already been established, and therefore does not suffer the initial expensive connection set up that uses asymmetric encryption.

Konqueror yields the worst results. We ascribe this to the fact that the GUI, as well as rendering the content, uses many instructions that incur much overhead in emulation, including floating point operations as well as MMX operations.

5.2 Eudaemon

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

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. To clarify this point we present the main execution loop of SEAL in fig.9. After the completion of the second release stage, *Eudaemon* is blocked waiting for the current block of emulated instructions to conclude, and the emulator to exit its main loop. As a result the target process is not blocked during this time, and the observed delay is small.

To measure the performance of *Eudaemon* when multiple process are possessed, we created an increasing number of processes, which we proceed to possess. Figure 8 plots the time needed to switch a number of processes from native to emulated execution. 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 reasonably well. 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.

Regarding security, the emulator used in SEAL was tested against many types of exploit, including: Apache chunked encoding overflow, WebDav ntdll.dll overflow, IIS ISAPI .printer host header overflow, RPC DCOM Interface overflow, LSASS Overflow, nbSMTP remote format string ex-

	bunzip2	wget	sftp	konqueror
Native Execution	27.99s	10.97MB/s	14.3MB/s	29.4ms
SEAL (1 byte tags)	242.24s	10.92MB/s	2.3MB/s	463.4ms
Slowdown (*factor*)	×8.6	×1	×6.3	×15.6
Argos (1 byte tags)	508.66s	0.90MB/s	0.55MB/s	n/a
Slowdown (*factor*)	×18.2	×12.2	26	n/a
SEAL (1 bit tags)	248.78s	10.93MB/s	2.3	725ms
Slowdown (*factor*)	×8.9	×1	×6.3	×24.5
Argos (1 bit tags)	635.15s	0.49MB/s	0.47MB/s	n/a
Slowdown (*factor*)	×22.7	×22.4	26	n/a

Table 1: Emulation overhead

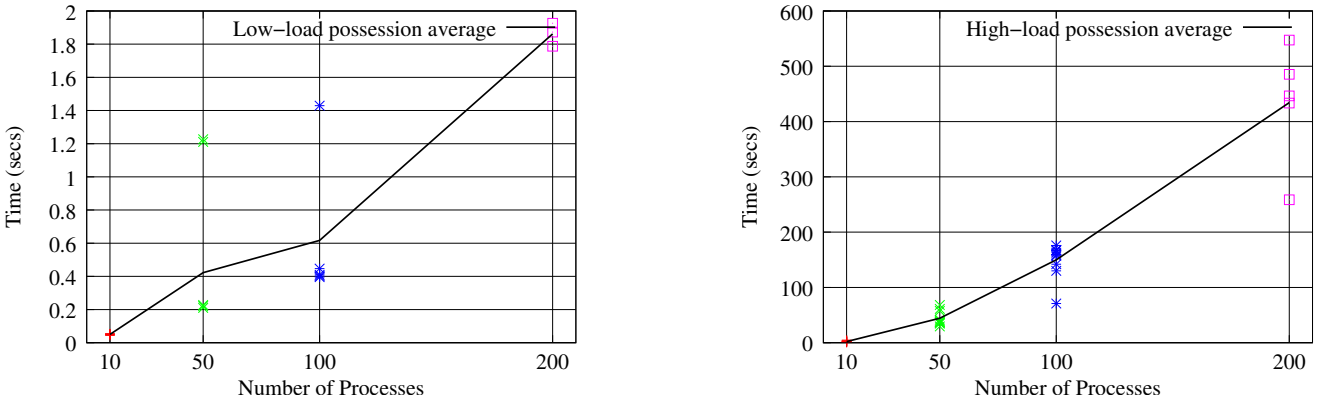

Figure 8: Multiple process possession under low- and high-load

Eudaemon Action	Possession	Release
1^{st} phase	1.195	0.159
Waiting time	*not applicable*	2782.617
2^{nd} phase	0.095	0.106
Total	1.290	2782.882

Table 2: Eudaemon micro-timings (msec)

```
while (honeypot_mode == true) {
    run_instruction_block();
    handle_system_call();
    handle_signals();
}
```

Figure 9: SEAL Execution Loop

ploit, NetApi exploit, WMF exploit, and many others. Interested readers are referred to the original Argos paper [28].

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 security 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. The authors would also like to thank Kostas Anagnostakis and Willem De Bruijn for the constructive criticism and brainstorming sessions.

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] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *EuroSys, Leuven, Belgium*, April 2006.

[5] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, E. M. K. Xinidis, and A. D. Keromytis. Detecting targeted attacks using shadow honeypots. In *Usenix Security*, Baltimore, MD, August 2005.

[6] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of USENIX ATC*, pages 41–46, Anaheim,CA, April 2005.

[7] S. Bhansali, W.-K. Chen, S. D. Jong, A. Edwards, and M. Drinic. Framework for instruction-level tracing and analysis of programs. In *Proceedings of the 2nd International Conference on Virtual Execution*

Environments (VEE '06), pages 154–163, Ottawa, Canada, June 2006.

[8] 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 USENIX Security*, pages 105–120, August 2003.

[9] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Security and Privacy*, Oakland, CA, May 2006.

[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] 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, Portland, Oregon, 2004.

[12] 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*, San Francisco, CA, 2002.

[13] 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.

[14] 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.

[15] D. Denning. A lattice model of secure information flow. *ACM Transactions on Communications*, 19(5):236–243, 1976.

[16] eEye. eeye industry newsletter. `http://www.eeye.com/html/resources/newsletters/versa/VE20070516.html#techtal%k`, May 2007.

[17] W. W. Hsu and A. J. Smith. Characteristics of i/o traffic in personal computer and server workloads. *IBM Systems Journal*, 42(2), 2003.

[18] 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, Washingtion, DC, October 2003.

[19] N. Krawetz. Anti-honeypot technology. *IEEE Security and Privacy*, 2(1):76–79, January 2004.

[20] B. Lampson. Accountability and freedom. In *Cambridge Computer Seminar*, Cambridge, UK, October 2005.

[21] 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.

[22] 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.

[23] M. E. Locasto, A. Stavrou, G. F. Cretu, and A. D. Keromytis. From stem to sead: Speculative execution for automated defense. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 219–232, June 2007.

[24] 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.

[25] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware in the web. In *Proc. of NDSS'06*, February 2006.

[26] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: automatic protocol replay by binary analysis. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 311–321, New York, NY, USA, 2006. ACM Press.

[27] 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.

[28] 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.

[29] J. Richter. Load your 32-bit dll into another process's address space using injlib. *Microsoft Systems Journal (MSJ)*, January 1996.

[30] SANS. Sans institute press update. `http://www.sans.org/top20/2006/press_release.pdf`, 2006.

[31] H. Shacham, M. Page, B. Pfaff, E. Goh, and N. Modadugu. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[32] S. Sidiroglou, M. Locasto, S. Boyd, and A. Keromytis. Building a reactive immune system for software services. In *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[33] A. Slowinska and H. Bos. The age of data: pinpointing guilty bytes in polymorphic buffer overflows on heap or stack. In *23rd Annual Computer Security Applications Conference (ACSAC'07)*, Miami, FLA, December 2007.

[34] P. Szor and P. Ferrie. Hunting for metamorphic. In *Virus Bulletin Conference*, pages 123–144, Abingdon, Oxfordshire, England, September 2001.

[35] 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.

[36] X. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. Choi. Packet vaccine: black-box exploit detection and

signature generatio n. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 37–46, New York, NY, USA, 2006. ACM Press.

[37] 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)*, San Diego, CA, February 2006.

[38] S. M. B. William R. Cheswick, Aviel D. Rubin. *Firewalls and Internet Security: repelling the wily hacker (2nd ed.)*. Addison-Wesley, ISBN 020163466X, 2003.

[39] 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.