Using Virtualisation to Protect Against Zero-Day Attacks

VRIJE UNIVERSITEIT

Using Virtualisation to Protect Against Zero-Day Attacks

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op donderdag 25 februari 2010 om 13.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Georgios Portokalidis

geboren te Alexandoupoli, Griekenland

promotor:	prof.dr.ir. H.E. Bal
copromotor:	dr.ir. H.J. Bos

Contents

List of Figures

List of Tables

Acknowledgements

I would like to start by thanking my adviser, Herbert Bos, for guiding and supporting me all these years, as well as for all the philosophic discussions we had on occasion. I would also like to thank my promoter, Henri Bal, who has been very supportive, and even seemed more excited than I was when this thesis was approved.

I am also very grateful for having some of the leading researchers in the field in the committee reviewing this thesis. Thanks to Manuel Costa, Marc Dacier, Sandro Etalle, Engin Kirda, and Andrew Tanenbaum.

Last but not least, I want to thank my family for supporting me throughout this endeavour.

Chapter 1

Introduction

Operating systems and software in general, continuously grow in size and complexity. As a result, software contains programming errors that frequently allow attackers to gain illegitimate access, and even fully control systems. In the past, we witnessed large scale infections from worms such as CodeRed [39], Blaster [9], and Sasser [142] that managed to infect hundreds of thousands of hosts, while the Slammer [40] worm exhibited phenomenal speed in infecting almost every vulnerable server in minutes. More recently, we saw attackers exploiting bugs in popular applications, such as web browsers, to take control and organise compromised systems into large collections of networks that are used for sending spam, carrying out distributed denial of service (DDoS) attacks, and extracting personal information (credit card numbers, passwords, etc).

Current solutions have been able to alleviate the problem partially, but in practice have proven inadequate in detecting attacks and generating countermeasures in a timely manner.

This dissertation addresses the problem of automatically and reliably detecting previously unknown attacks, and generating vaccines that can deter new infections in their early stages. We present three novel ways of using virtualisation to detect zero-day attacks, and automatically generate countermeasures. Our solutions are based on a technique called dynamic taint-analysis, used to capture the most prominent self-propagating attacks. Most importantly, they apply to legacy hardware and software, and generate no false positives.

First, we describe the design and implementation of *Argos*. *Argos* is a secure x86 emulator able to identify unknown attacks, and automatically extract information after the detection of an attack to generate countermeasures. Second, we introduce a new technique dubbed *Eudaemon* that blurs the borders between protected and unprotected applications on desktop sys-

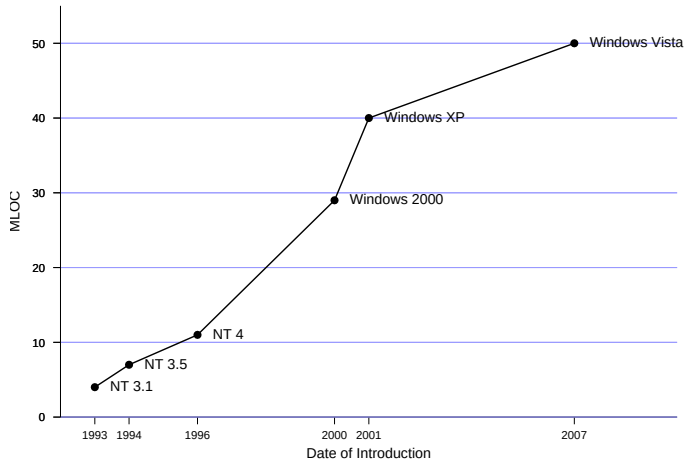


Figure 1.1: Code size of Windows operating systems

tems using on-demand emulation. Third, we address the problem of providing a similar level of protection to lightweight devices such as smartphones, overcoming their limited resources.

1.1 The Problem

Computer systems have evolved dramatically through time. CPU, memory, and storage have improved significantly. Software has admittedly not harnessed the full potential of available hardware, but it has undoubtedly grown in size and complexity, as shown in Fig. 1.1. Producing highly complex software is difficult. Hence, software frequently contains *programming errors* that exhibit themselves as crashes or unexpected behaviour. Fault distribution studies show that there is a correlation between the number of lines of code and the number of faults [88, 11, 114, 115]. To quantify this, it is approximated that code contains 6-16 bugs per 1000 lines of executable code. Attackers are often able to exploit certain types of program faults to circumvent security measures introduced by design to protect a system. Reports by organisations such as SANS[1], and various CERTs[2] show that there is large number of such *vulnerabilities*.

[1] `http://isc.sans.org` The Internet Storm Center was created by SANS in 2001 as a response to an increasing number of malicious attacks.

[2] Computer Emergency Response or Readiness Teams have been formed by governments and non-profit organisations around the world to analyse the state of Internet security and to provide emergency response support.

The security implications of software faults are evident. In 2002 and 2003, Internet worms such as CodeRed and Slammer compromised large numbers of server systems in record times. Later on, a plethora of vulnerabilities [122] in desktop applications were discovered and exploited to install malicious software (malware) on millions of desktop systems. Recently, another worm (Conficker [31, 51]) has been identified, leading a new wave of worm attacks [144, 97]. Many critical networks have been compromised by the Conficker worm [87, 60], while it is speculated that millions systems have been infected in total.

Eliminating the errors that enable such attacks has proven extremely challenging. Static code analysis tools have greatly improved code quality by identifying many errors before software is distributed, but are unable to exhaust the entire space of possible execution paths [160], and thus are unable to discover all errors. Certainly, using more secure programming languages would significantly improve security [67]. In practice though, most system software is written in C and C++ for reasons such as code reuse, backward compatibility, and performance.

Current security solutions that attempt to address the problems mentioned above can be classified in two major categories: (a) network security, and (b) host security. The first focuses on deploying countermeasures in the network, in an attempt to identify and filter (when possible) malicious network flows. The latter, aims to identify and block exploit attempts at the host level, where the attack actually takes place. Both approaches have had positive results, but also display many weakness that we will briefly attempt to sketch. Extensive discussion of current security approaches is presented in Chapter 2.

Security in the network is based on observing the data being trafficked. The observations can be either behavioural, or content based. The increasing use of encryption on the network, as well as attackers intentionally encrypting or encoding their attack payloads greatly subverts network intrusion detection systems (NIDS). Additionally, using behavioural traits of attacks for identification is also becoming harder, as worm propagation can be disguised as peer-to-peer traffic, and attackers switch to slower and stealthier spreading policies. *In practice, network solutions are rarely used for filtering malicious data from the network, because of their low accuracy that frequently results in falsely classifying benevolent traffic as malicious (false-positives), and vice-versa (false-negatives).* Instead, they are limited in *passively* monitoring network traffic, and logging potential threats, which need to be evaluated manually at a later time before taking any action.

Host security solutions come in much greater variety. The most commonly found solution is an anti-virus system, where file system and memory data are

scanned for known patterns or signatures. Such systems face the same problems as NIDS, but are more accurate especially with regard to false positives. Hardening software security by recompiling software using various types of security extensions, attempts to harden security of already available source code, and keep the performance overhead low. Unfortunately, depending on the strength of the extension such approaches have been either defeated (e.g., stack smashing compilers extensions [21, 72, 150] can be circumvented [19]), or have been incompatible with large parts of available software (e.g., they are unable to work with dynamic libraries not compiled with the same extension [1]). The lack of source code for proprietary software is also greatly restricting the applicability of compiler based solutions.

Runtime (host) solutions seek to take advantage of the wealth of information that is available when a system is actually running to protect against attacks on program errors. The majority of these solutions are very accurate in detecting certain types of attacks. Virtualisation is often used to deploy them, since it provides the needed isolation and transparency. Their main disadvantage is the significant performance overhead they impose, which makes them inappropriate for production systems. For instance, dynamic taint analysis as proposed by Dening et al [42] and later implemented in TaintCheck [107] is able to accurately detect and stop the exploitation of the majority of memory corruption vulnerabilities such as stack and heap overflows (see Section 2.1.1), but can cause the protected application to run up to 25 times slower. Efforts have been made to mitigate the overhead of such solutions, but they usually require specialised hardware or software.

Solutions in both categories that are based on signatures to detect attacks are also confronted by another problem: the timely generation of signatures for new and constantly evolving known attacks. Currently, signatures are produced manually by experts, after the identification of a new virus, worm, or other type of malware. History has shown that humans are not fast enough to counter certain attacks. For instance, the Slammer worm was able to infect the entire population of accessible and vulnerable hosts in 30 minutes, while researchers have argued that the creation of a "flash" worm that can accomplish the same feat in even less time is possible [139].

1.2 Goals

The main goal of this thesis is to investigate virtualisation-based solutions to protect commodity systems against zero-day attacks. Particularly, we focus on attacks that divert the control flow of a program, or inject arbitrary instructions in its flow, or both, after exploiting memory access errors in software (explained in Section 2.1). All the worms we have referred to thus

far have exploited such vulnerabilities on their victims.

The ability to automatically identify and analyse new attacks is critical for new solutions, as manual analysis is *time consuming*, and *does not scale*. Past worm outbreaks have shown that human intervention can be overly slow, while the constantly increasing number of malware collected by anti-virus vendors [117] shows that the cost of manual analysis may be too high in the future (if not already prohibitive). The increasing number of varying attacks seen by vendors can be attributed to attackers using code obfuscation techniques (such as polymorphism discussed in Section 2.2.3) to evade detection, which only intensifies the problem.

The unsupervised detection of zero-day attacks, requires an accurate detection technique, especially one with few or no false-positives. Dynamic taint analysis (explained in detail in Section 2.3.4) is such a technique. It is based on tracking the data flow of possibly dangerous data to detect attacks such as control-flow diversion that we mentioned earlier. An implementation of the technique in software requires a virtualisation layer, such as an emulator or a dynamic binary translation framework, and usually incurs a massive performance overhead in the range of 1000%-2000%. This thesis will attempt to take on the challenges involved with applying dynamic taint analysis on existing systems.

Computing systems can vary significantly in hardware, software, and in the ways they are used. For instance, server and desktop systems follow very different usage patterns, while smartphone hardware and software can be entirely dissimilar. Furthermore, we cannot assume that access to software source code is possible, as frequently proprietary or legacy software is deployed on these systems. These differences pose extra challenges when designing an out-of-the-box solution for immediate use on these systems.

The goals of this thesis can be summarised in the following research questions:

Question 1. *Can we find solutions for detecting zero-day attacks, by means of dynamic data-flow tracking, in unmodified software, and without requiring access to source code or specialised hardware?*

Question 2. *Can we mitigate the performance overhead imposed by dynamic data-flow tracking to scale our solutions to varying computing systems, such as servers, desktops, and smartphones?*

1.3 Contributions

The concrete contributions of this thesis can be summarised into the following:

- We created a platform for the next generation high-interaction honeypots that automates the procedure of capturing zero-day attacks, and generates a simple "vaccine" for deployment on NIDS (Chapter 3).
- We developed a technique that transparently enables desktop systems to act as honeypots. Our technique is able to overcome the issues of honeypot avoidance, and can detect client-side exploits, to protect desktop applications from attacks and generate signatures for NIDS (Chapter 4).
- We address the problem of protecting light-weight devices such as smartphones by delegating security checks to a loosely synchronised replica. By outsourcing security checks we enable the application of heavy-weight security checks such as the ones used on honeypots, and at the same time transparently offer backup functionality (Chapter 5).

Our honeypot platform has also found its way in multiple intrusion detection systems, maintained both by research and industrial institutions. Such systems are SURFids [141], SGNET [85], and Honey@Home [7].

1.4 Thesis Organisation

The rest of this book is organised as follows, In Chapter 2, we offer some background information on attack techniques, and current intrusion and prevention systems. There, we also review previous work on the detection of unknown attacks, and the automatic generation of "vaccines". In Chapter 3, we present *Argos* a secure emulator for use in high-interaction honeypots, and show how it can be used for the automatic generation of signatures for NIDS. Following, we present *Eudaemon* in Chapter 4. *Eudaemon* is able to transparently transform desktop systems to honeypots by employing on demand protection to alleviate performance costs. In Chapter 5, we focus on smartphones and propose a new design that applies security checks on a loosely synchronised replica of the device overcoming the resource limitations of these devices. Finally, we conclude in Chapter 6.

Chapter 2

Background

This chapter elaborates on three subjects that will assist the reader to better comprehend the remainder of this thesis: the software errors responsible for the majority of attacks, the most frequently encountered attack types, and finally the solutions developed as responses to these attacks. We will also discuss related work throughput this chapter.

2.1 Software Errors

Software errors, commonly referred to as bugs, have been the primary cause of most security vulnerabilities in recent times. They mainly arise from mistakes made by developers when coding programs, or can be the result of faulty designs. Less frequently, bugs can be introduced by a compiler that produces incorrect binary code even when given sound source code.

Software bugs can be extremely hard to detect, and do not always present themselves when running a program, as they are often only triggered by specific input. It comes as no surprise that software vendors subject their software to extensive testing procedures, in an attempt to discover and eliminate them. While testing has improved significantly, not all vendors can afford long and expensive debugging cycles for their software. Additionally, the debugging process itself is imperfect. As any average computer user can attest to, bugs always find their way on every system.

Unsolved bugs eventually cause programs to exhibit unexpected behaviour. This frequently results in a program crashing or freezing (i.e., it no longer functions but appears running). At other times, the program behaviour is altered without terminating (e.g., corrupted data are written to a file), keeping the error concealed.

There are various types of programming bugs, for instance mathematical bugs (i.e., division by zero), logic bugs (i.e., infinite recursion), and concur-

```
function(const char *srcstr)
{
 int authorized;
 char dststr[1024];
 ....
 authorized = 0;
 ....
 strcpy(dststr, srcstr);
 ....
 return authorized;
}
```

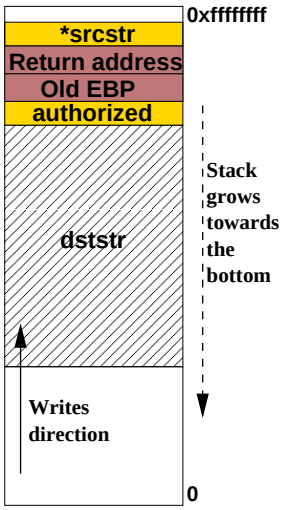


Figure 2.1: Example of stack overflow

rency bugs (i.e., race conditions). The most severe bugs from a security perspective are memory errors.

Memory errors are caused when the "wrong" memory location is accessed. That is a different location from the one originally intended by the programmer. When such an error occurs, the program will most probably terminate, because the accessed memory page does not exist (e.g., a null pointer dereference), or the program does not have the appropriate rights to access an existing page. On the other hand, if the location is valid, the program will not terminate, and the data on that location will be used or erroneously overwritten. Such errors can occur with both read and write operations, but are more severe in the latter case.

2.1.1 Buffer Overflows

A frequently encountered memory access error that results in storing data in a different location than the one intended by the programmer is a *buffer overflow*. Such errors usually occur when copying data between buffers without checking their size. Consider the function shown in Fig. 2.1 that attempts to copy string *srcstr* into *dststr*. The programmer assumes that *srcstr* contains a legitimate string that fits in *dststr*. If that is not the case, the standard copy function *strcpy* will keep writing beyond the end of *dststr*, *overflowing* into variable *authorized* and even beyond. This happens because of the way function local variables are stored in the stack. As the stack grows from top to bottom addresses, and most buffers are written in the opposite direction,

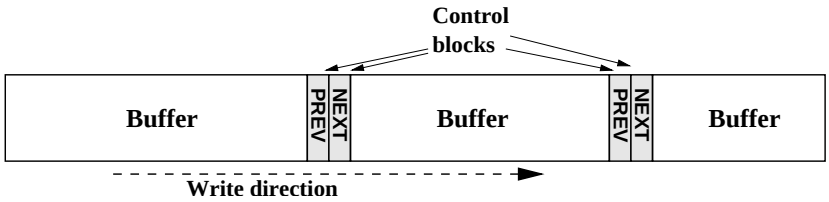


Figure 2.2: Typical heap structure

overflowing a buffer will overwrite the values following the buffer.

Buffer overflows are also possible in the heap. Fig. 2.2 shows the typical structure of a process's heap. Heap objects are allocated successively in the process's data segment, along with small prefix or suffix control blocks (e.g., pointers to the previous and next used or free heap object) used for the management of the heap. Thus, an overflow of a heap buffer will result in overwriting the control blocks. When a memory management function such as $free()$ is called, the heap lists containing used and free buffers are updated using the control blocks (e.g., $heapobj \rightarrow next \rightarrow prev = heapobj \rightarrow prev$), resulting in an illegal overwrite of an arbitrary location.

2.1.2 Format String Errors

A less common memory error can occur when an invalid format string is used with the $printf(format, ...)$ family of functions. These functions produce string output that can be printed in standard output, or written to a string buffer or a file. The output is created according to the string in the $format$ argument. The function accepts a variable number of arguments, which are stored in the stack. As the format string is processed, the arguments are retrieved from the stack to produce the output. Fig. 2.3 shows a simple invocation that will print the contents of string *str* followed by a new-line character.

A memory error occurs when the number of arguments supplied to one of these functions, does not satisfy the format string. For instance, if $fmt =$ "$\%s - \%s$" in the example in Fig. 2.3, the function will attempt to retrieve two pointers to strings and print them to standard output. As only a single argument is provided, $printf$ will print whatever the bytes stored in the stack after *str* point to.

Incorrect format strings can also lead to invalid writes when the flag $\%n$ is included. This flag causes $printf$ to store the number of characters that have been produced into the next argument in stack, which is interpreted as a pointer to an integer. Such errors can result to writes to arbitrary locations in memory.

```
fmt = "%s\n";

printf(fmt, str);
```

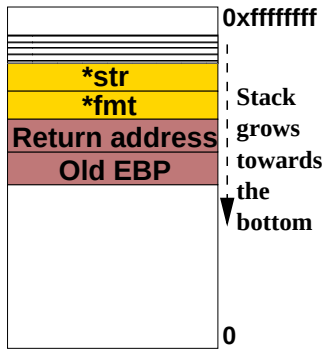


Figure 2.3: Example call of *printf* function

2.2 Attacks

Attacks against computer systems do not always rely on faulty software for their success. Attackers often rely on gullible users to open seemingly innocent attachments sent by email, which are in fact malicious executables. Furthermore, malware frequently masquerades as useful programs, offered freely over the web or peer-to-peer (P2P) file sharing systems to entice users looking for pirated software to download and execute them. Such approaches have long been used by virus writers, and proved to be very successful.

Nevertheless, such attacks are made possible because usability is more important than safety on the inflicted systems. For instance, on a highly-secure "hardened" system it might be impossible for users to download and run any other program besides the ones pre-installed on their system. This thesis does not focus on such attacks, so this section will only focus on attacks exploiting software bugs such as the ones we described earlier in Section 2.1.

A bug consists a vulnerability, if it can be triggered by means of supplying certain input. As bugs frequently manifest themselves as crashes, a malicious user would then be able to crash the application or service. Depending on the error at hand, more carefully crafted input could enable an attacker to take complete control of a program, install malicious programs, extract confidential information, or perform some other unintended action.

Attacks vary depending on the type of the bug being exploited, and exploitation can be performed locally or remotely. Remote exploitation is possible when the input that triggers a bug originates in the network. In the past, such remote exploitable vulnerabilities have provided the launching platform for numerous self-propagating malware attacks (most commonly referred to as *worms*).

2.2.1 Attack Types

Control-flow Manipulation

One of the most powerful attacks is performed by altering or taking control of the flow of a vulnerable program. The power of this attack lies in the following:

1. compromising a process with administrator privileges offers full control over the host
2. in the case of a remote attack, its orchestrator gains access to the vulnerable host

Such attacks usually exploit buffer overflows that overwrite certain critical values in the vulnerable program to divert control. The altered values may belong in two categories: *control data*, and (less frequently) *non control data*. The first refers to data that directly controls the flow of a program such as function addresses, non immediate jump targets, and return addresses. The latter includes program variables that hold a critical role in the program's logic.

Revisiting the stack overflow in Fig. 2.1, we can see how that error can be exploited by overwriting both control and non control data. Let us assume that the function shown determines whether a remote shell request is authorised. In this case, variable *authorized* plays a critical role, as it determines whether an incoming connection is authorised. By causing a buffer overflow of *dststr* an attacker would thus be able to control the value of *authorized*, and illegally gain access.

Alternatively, a larger overflow of the buffer would overwrite the control data located in the stack. Function calls use the stack to store the address (*Return address*) where execution should return to after the called function completes. The calling function's base pointer (EBP) is also stored. Overwriting a function's return address, allows attackers to take control of the program when the function returns. The hijacked process is usually pointed to an attacker controlled buffer such as *dststr* where assembly code is placed to perform some action. When no such appropriate buffer exists, attackers also resort to redirecting execution to existing code (e.g., code that launches a shell).

Arbitrary Code Execution

The ability of an attacker to execute arbitrary code consists probably the most powerful type of exploit. Arbitrary code execution usually goes hand-in-hand with control-data manipulation, as in most cases the attacker needs to hijack

the program's control flow to get the code executed. Frequently, the attacked buffer is used as a placeholder for the attacker-supplied code, but this does not need be the case. Instances of attacks where multiple buffers were used, or the code was injected at an earlier stage have also been observed.

2.2.2 Self-propagating Malware

A particularly malicious threat against computer systems is that of self-propagating malware or worms. Internet worms such as CodeRed, Blaster, and Sasser have created havoc in the past, while recently the Conficker worm has also made the news on various occasions by infecting various high-profile targets [87, 60]. Worms are malicious code that use various infection techniques to compromise systems, and are able to self-replicate by locating and compromising new targets without the user taking any action.

One of the most important properties of every worm is the way new targets are discovered and attacked. The choice of a target can be a conscious one, or can be strictly driven by opportunity and the desire to maximise the impact of an attack. In the case of cyber-warfare the systems of the "enemy" are intentionally targeted, while in the past most computer worms targeted Windows systems because an opportunity was present (in the form of vulnerable system services), and the popularity of the OS guaranteed wide impact.

From a technical point of view, target discovery deals with the ways worms discover and attack the largest number of vulnerable systems. Some means to achieve this are:

- *Scanning* was until recently the most prominent way of searching for vulnerable targets. The simplicity of the method is probably the reason behind its broad use. A random number generator is used to generate IP addresses that are consequently probed and attacked. The task of scanning the entire IP address space is not a trivial one, and it is frequently performed in a distributed fashion by having already compromised hosts also scan for new targets. The advent of IPv6, and the extended use of network address translation (NAT) has somewhat limited the effectiveness of scanning.

- *Hit-lists* of possibly vulnerable targets could allow worms to overcome the issues involved with scanning, but obtaining such a list requires more effort from attackers. It can be automatically generated using freely available services that offer meta-information on systems running a particular service. For instance, search engines have been used to discover web sites running particular software [109], while J. Kannan et

al [77] showed how the distributed hash table (DHT) of a peer-to-peer (P2P) network can be used to acquire targets when attacking such a network.

- *Malicious or infected servers* have been broadly used, in recent years, to identify and infect vulnerable systems. Attackers set up their own servers, and then attempt to lure possible victims to access their servers. Alternatively, vulnerabilities on popular servers are exploited to inject attacks. Consequently, users accessing these servers face the danger of being infected.

2.2.3 Payload

The program that implements the desired functionality of any malware, besides the infection of the target, is the *payload*. The payload of an attack is also called *shellcode* for historical reasons, as it was frequently used by attackers to acquire a remote shell on the compromised system. The terms *payload* and *shellcode* are used interchangeably in this thesis.

The first malware were relatively mild with nonexistent payloads, or payloads that only intended to annoy users. Not all malware has been that innocent though. Some of the most malignant viruses of the past have made entire systems inoperable by damaging hardware, or destroying critical system files. In general payloads are very flexible, and are only limited by the creativity and ability of their authors.

In recent years, stealing valuable user information has been one of the primary targets of attackers. Programs known as *keyloggers* or *trojans* are frequently installed on compromised systems to capture every user key stroke. The goal is to capture user passwords, and other sensitive information such as PIN, social security, and credit card numbers. The acquired data are directly used by attackers, or traded in what has come to be a flourishing underground economy [145].

Payloads are frequently obfuscated to conceal their presence on the infected system, and avoid detection from antivirus engines. Various stealth mechanisms are employed to achieve concealment:

- *Encryption* using a variable key can obfuscate most of the payload. Only a small part of the payload, which contains a decryption module, remains immutable amongst instances.

- *Polymorphism* improves on encryption by also modifying the small decoder module between instances. This is achieved by using a polymorphic engine that produces different decoding modules with the same effect.

- *Metamorphism* involves modifying the entire payload on each infection without using encryption. A metamorphic engine produces different payloads that achieve the same goal. Payloads that employ metamorphism are larger, as they also include the engine responsible for rewriting themselves.

2.3 Defences

In response to the threats we have described in the above sections, a multitude of defences have been proposed and many times implemented. The proposed solutions vary, focusing on different aspects of computer systems such as programming languages, networking, operating systems, etc. In this section, we will attempt to present (according to our position) the most significant defences developed by the security community, analysing their demonstrated advantages and shortcomings.

2.3.1 Safe Programming Languages

Safe languages attempt to strengthen security by addressing the route cause of most attacks, which as we have already explained comes from programming errors. Unsafe programming languages such as C and C++, albeit powerful and flexible, depend on the developer to make the right decisions and follow proper practices to write error free code. Experience has shown that programmers do and will make errors [11], and as such new programming languages that eliminate errors such as buffer overflows are needed.

Consequently, safer languages such as Cyclone [75], Java [59] and C# [96] were developed. Even though different approaches are taken by these languages, all of them offer protection against memory errors that would otherwise be possible with C or C++. The immunity to such errors emerges mainly from two facts: most (if not all) safe languages are strongly typed [24], and perform bounds checking when accessing arrays. The checks are usually performed by code that is introduced by the compiler when producing the binary, and in the case of Java they can also be applied at runtime by the VM.

Java and C# have seen broad adoption in certain areas, such as the scientific community and the World Wide Web (JSP and ASP powered web services respectively). Unfortunately, little penetration has been seen in areas that have suffered much from attacks, like popular desktop applications (browsers, e-mail clients, etc), operating systems (kernel drivers) and services (background daemons), or high-performance servers (web, database, etc). Initially, performance was considered the greatest disadvantage of using safe languages, but as they matured and processing power increased other

obstacles surfaced. As the larger part of existing software is written in C and C++, code reuse and backwards compatibility forms another barrier in the mass adoption of safer programming alternatives. This argument seems to be validated when looking at the latest releases of popular browser applications such as Internet Explorer, Google Chrome, and Mozilla Firefox, where the bulk of the code is written in C++.

2.3.2 Compiler Extensions

Security-oriented compiler extensions attempt to "patch" unsafe programming languages against certain types of attacks. The programming language is not altered in any way, instead security mechanisms are augmented in produced code. Most extensions do not alter the behaviour of the resulting executable, which is usually larger in size. Compiler extensions usually incur low overheads and enable code reuse, but as the "patched" programming language is not immune to these faults, programs compiled with such extensions usually terminate whenever an attack is detected.

Some of the first compiler extensions were developed to protect against stack overflows, which until recent years were the most frequently exploited vulnerabilities. Stackguard [21], ProPolice [72], and StackShield [150] are three such extensions that use similar techniques based on inserting a special value, called a *canary*, in the stack between control data and local variables. The code generated by the extension checks for altered canary values to detect overflows. These or similar extensions have found their way in popular developer toolchains like GCC and Microsoft Visual Studio. However, they can be overcome [19]. Similar extensions [22, 147] have been also developed to address format string attacks, by replacing the $printf()$ family of functions with other safe versions that perform explicit checks on the number of arguments defined in the format string, and the one actually passed to the function.

CRED [129] takes a different approach by checking the correctness of memory accesses on strings. It is able to detect both stack and heap overflows, but it is limited to string buffers, while also incurring a performance overhead of about 26%. CCured [53] identifies possible unsafe pointer usage in C source code, and retrofits it with runtime checks to identify illegal memory accesses. It incurs an overhead between of 0%-150%, while detected attacks cause target programs to terminate.

A more generic solution against all types of buffer overflows is offered by DFI [25]. It employs the Phoenix compiler infrastructure [98] to generate a static data-flow graph of the program being compiled, and instruments the code to check the integrity of the program at runtime. WIT [1] extends DFI

by introducing special values between buffers to detect overflows. It also performs somewhat different static analysis by classifying memory areas into colours and identifying the instructions that are associated with accessing them. The main advantage of DFI and WIT is low performance, which is less then 5% for the latter. Nevertheless, they both face problems when using libraries not compiled in the same fashion.

Finally, PointGuard [23] attempts to offer broader protection against buffer overflows by safe guarding all pointers (including stored addresses). It accomplishes this by encrypting pointers when they are stored in memory, and decrypting them only when they are about to be loaded on a register. It has been incorporated in certain Windows operating systems, but in a limited non automated way (as a developer API) to reduce performance overheads.

2.3.3 Static Analysis

Static analysis tools [76, 10, 151, 103] are similar to compiler extensions as they operate on existing source to detect errors. The detection can take many forms such as the generation of warnings and errors that need to be corrected during the developing phase of software [32, 20]. Static analysis solutions exhibit relatively low runtime performance overheads, or in the case of tools such as coverity they are applied only during the development and testing phase of software. Still, while software integrity undoubtedly improves, methods depending solely on static analysis of source code are not foolproof [161].

2.3.4 Dynamic Analysis

The security techniques presented in the above sections deliver increased security with relatively low performance costs. However, they require that source code is available for recompilation. In practice, source code is not always accessible due to copyright issues, and in some rare situations (older legacy applications and libraries) it might not be available at all. As a consequence, a lot of the software in use is still vulnerable to the attacks we described in the beginning of this chapter.

Dynamic analysis techniques focus on protecting applications and systems without requiring any changes to the target applications and/or OS being protected. They achieve this by transparently modifying the runtime environment (frequently using virtualisation). Many of these techniques are also frequently OS agnostic i.e., they are not bound to a specific OS. As such, they can be applied to different systems with little effort. On the other hand, performance costs are high, making them less applicable for production systems.

An interesting approach against all code injection attacks, proposes the randomisation of the CPU instruction set using emulation and a well designed random generator to eliminate the possibility of an attacker discovering the instruction set used at any given point [48, 78, 71]. Instruction set randomisation constitutes mostly a proof of concept solution, as the incurred overhead is prohibitive for practical use. The overhead could be alleviated if the solution is implemented on hardware, a challenging and costly endeavour. Additionally, this approach is unable to counter attacks that redirect program flow to existing code (e.g., return-to-libc attacks) instead of injecting code.

Other solutions [42] are based on the observation that to successfully compromise a system, an attacker needs to manipulate certain critical control values as shown in Section 2.2. According to this rationale and assuming that the attackers do not have local access on the targeted system, detection of an attack is performed by tracking network data as soon as they enter a system and scanning for their use as control data (function pointers, return addresses, branches, etc). To accurately track network data, which in this context are considered *tainted*, per instruction instrumentation is required along with extra memory storage to hold the appropriate *taint flags*. Systems implementing what is most frequently called as *dynamic taint analysis* employ emulation to perform the instruction instrumentation, and as such also incur relatively high overheads.

One of the first practical systems using taint tracking is Minos [33], which proposes the tracking of data in hardware, and implements a proof of concept solution on the Bochs [83] emulator. It optimises for an actual hardware implementation, focusing on minimising the number of additional circuits and memory required, and imposing little processing overhead. Its implementation on Bochs on the contrary, incurs a significant slowdown.

A slightly different approach is taken by TaintCheck [107] and Vigilante [91]. They both use process binary instrumentation frameworks that allow the application of taint tracking per process (Valgrind [105] for Linux and Nirvana [95] for Windows respectively). Vigilante does not stop at simply detecting and stopping an attack, but also attempts to generate a signature of the attack in the form of a self-certifying alert (SCA). SCAs were designed for collaboration between non-trusting systems, and enable the receiving party to certify its validity (i.e., that it actually describes a real attack) in a safe manner. Both solutions incur significant slowdowns ($\times 20$ or more), and they are bound to a certain OS, limiting their practical use (maybe more for TaintCheck than Vigilante, strictly because of the popularity of Windows OSes). Finally, while SCAs are a significant contribution, they are prone to false negatives during verification. False negatives can be caused when the message including the attack is subjected to some type of decoding (e.g.,

ASCII to binary), or when replaying the message is not sufficient for exploiting the program (e.g., because of a challenge-response protocol mechanism).

Chen et al [27] also use taint tracking in their approach, which tracks dereferences of tainted pointers to detect attacks. By tracking the usage of tainted pointers, they are able to broaden the field of captured attacks to also include non control-data being overwritten. They also propose a hardware implementation of their system to achieve good performance, but as later research demonstrated [137] such methods can only be applied in a controlled environment for a short period of time, before false positives (erroneous alerts) are generated.

2.3.5 Honeypots

Honeypots [38, 81] are idle systems that act as bait to attract attackers. Their goal is mainly to collect data on the methods used by attackers to compromise a system, while less frequently they are also used to delay attacks (tarpit). *High-interaction* honeypots consist of a real OS and applications running on hardware or more commonly under a VM for easier management and increased security. On the other hand, *low-interaction* honeypots expose virtual OSes and services to attackers. Multiple hosts can be simulated by a single low-interaction honeypot using fake network stacks to simulate different OSes, and scripts that perform simple protocol handling for simulated services. Low-interaction honeypots are deployed more frequently, because they are easier to manage and can expose multiple architectures to attackers, but provide less information than high-interaction ones. Common practice involves deploying honeypots to handle all or part of the dark (unused) IP address space in the network.

Honeypots are extremely useful security tools, able to provide early warning for many types of attacks. However, they have certain disadvantages: all network traffic received by a honeypot is considered by definition to be suspicious, as the system has an idle role and its existence is not advertised. Unfortunately, even idle connected systems receive plenty of noise traffic, which makes it harder for administrators to identify malicious from innocuous traffic. To overcome this issue, dynamic analysis systems have been brought into play to host high-interaction honeypots [34, 91], but have seen little practical use due to performance or manageability reasons.

Another weakness of honeypots is that by design they favour attacks that perform target discovery through network scans. As technologies like IPv6 and network address translation (NAT) become more popular, scanning has become less efficient, and attackers have turned to other means to discover targets. For instance, search engines have been used to get valid web server

IPs, while at the same time a major switch to attacking client applications has occurred. The latter implies that victims are lured to access malicious servers, rendering traditional honeypots useless. As a response, we have witnessed the development of client-side honeypots [155, 101], which by continuously connecting to remote servers (mostly web servers admittedly) attempt to discover malicious ones. Deployed instances of such honeypots are increasing, but their success is strongly bound to our ability to identify and collect a complete list of potentially malicious servers.

2.3.6 Network Intrusion Detection & Prevention Systems

Applying security in the network has been an attractive alternative to host security for many reasons. Most importantly, because it enables administrators to secure the entire network by performing filtering at the gateway(s) of their local LAN or WAN, constituting a less obtrusive solution. Most such systems operate by scanning network traffic for known patterns or *signatures* that identify attacks, while others aim to identify *anomalies* in network traffic.

Signature Based Detection

Snort [127] and Bro [119] are the main representatives of systems using a database of signatures to detect potential intrusions. Both of these systems, and especially Snort, are being used extensively, but suffer from a multitude of problems. First, as network traffic and the size of their signature database keep increasing, throughput becomes a bottleneck. Second, the detection of attacks is based mostly on known signatures, making such systems inappropriate for handling zero-day attacks. Finally, the significant number of false positives generated limits their use for prevention (in practice, they are used solely for network monitoring).

Various attempts have been made to address the issues listed above. High-performance network intrusion detection has been the subject of research in FFPF [16], SafeCard [41], and Gnort [149] using specialised hardware such as network processors and graphics processors. Scanning of multi gigabit networks links is achieved, but certain limitations on the type signatures supported are introduced. On the other hand, systems like Earlybird [130] and Autograph [68] attempt to address zero-day outbreaks by automatically generating signatures for signature based systems. While they have had some success in automatically generating signatures, they incur a fairly large false positives ratio that makes their use impractical.

Anomaly Detection

Anomaly detection systems can be separated in payload- and behaviour-based systems. Payload anomaly detection systems [92, 153, 15] identify anomalies in network packet payloads. This is accomplished by comparing network traffic with a statistical model of "correct" or innocuous traffic. Anomaly detection systems incur larger overheads than signature based systems, but are able to identify zero-day attacks. They also exhibit significant false positives that limit their application to early warning systems.

Packet Vaccine [154] addresses the increased overhead of anomaly detection, by generating a "vaccine" or a signature for detected attacks that could be applied on a more lightweight signature based system.

On the other hand, HP's Virus Throttle [158] aims to slowdown the propagation of worms by targetting anomalous behaviour. This is achieved by limiting the number of connections that can be performed by any given host in a period of time. This approach can be very effective against fast propagating worms, but cannot defend against stealthier worms or client application attacks. Furthermore, it can disrupt the operation of legitimate P2P applications.

Others

Solutions using neither signatures, nor statistical models for the detection of attacks in the network also exist. For instance, Nemu [120] performs network-level emulation that aims to identify self-modifying binary code in the network. In Section 2.2.3, we saw that attackers employ various techniques to obfuscate their shellcode. Consequently, techniques like polymorphism lead to increased false negatives when using network intrusion detection solutions. Nemu treats network streams as instructions and emulates their execution, scanning for self-modifying code that could be part of such obfuscated payloads. Network-level emulation incurs small numbers of false positives, but cannot be performed on gigabit network links, or encrypted traffic.

2.3.7 Operating Systems

Operating system extensions and improvements have been also employed to improve security, and increase the difficulty factor for compromising a system. Address space layout randomisation (ASLR) is one of them. ASLR attempts to shuffle certain program components in every process's address space. The starting address of a program's heap, stack, and dynamic linked libraries are different every time it is started. ASLR increases the difficulty for attackers to inject code, and properly use the system's APIs, but it is not foolproof [134].

PaX [118] and OpenWall [43] are extensions for Linux that implement non executable stacks (NX stacks). Consequently, attackers are not able to use the stack to inject code in vulnerable software. These extensions do not offer any protection against heap overflows, or return-to-libc attacks. Furthermore, non executable stacks are not compatible with OS mechanisms that require code to be executed from the stack, such as signal trampolines[1].

ASLR and NX stacks are lightweight solutions that increase security, as a result they have found a place in many OSes. On the downside, they only harden a system against certain attacks, while in certain cases they might not be backwards compatible.

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

Chapter 3

Argos Secure Emulator

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

3.1 Introduction

Self-propagating malware, such as worms, have prompted a wealth of research in automated response systems. We have already encountered worms that spread across the Internet in as little as ten minutes, and researchers claim that even faster worms can be realised [139]. For such outbreaks human intervention is too slow and automated response systems are needed. Important criteria for such systems in practice are: (a) reliable *detection* of a wide variety of zero-day attacks, (b) reliable generation of *signatures* that can be used to stop the attacks, and (c) *cost-effective* deployment.

Existing automated response systems tend to incur a fairly large ratio of false positives in attack detection and use of signatures [158, 130, 68, 38, 81]. A large share of false positives violates the first two criteria. Although these systems may play an important role in intrusion detection systems (IDS), they are not suitable for fully automated response systems.

An approach that attempts to avoid false positives altogether is known as dynamic taint analysis. As discussed in Chapter 2, untrusted data from the network are tagged and an alert is generated (only) if and when an exploit takes place (e.g., when data from the network are executed). This technique proves to be reliable and to generate few, if any, false positives. It is used in current projects that can be categorised as (i) hardware-oriented full-system protection, and (ii) OS- and process-specific solutions in software. These are

two rather different approaches, and each approach has important implications. For our purposes, the two most important representatives of these approaches are Minos [33] and Vigilante [91], respectively.

Minos does not generate signatures at all and for cost-effective deployment relies on implementation in hardware. Moreover, by looking at physical addresses only, it may *detect* certain exploits, such as a register spring attacks [138], but requires an awkward hack to determine where the attack originated [34]. Also, it cannot directly handle physical to virtual address translation at all.

In contrast, Vigilante represents a per-process solution that works with virtual addresses. Again, this is a design decision that limits its flexibility, as it is not able to handle DMA or memory mapping. Also, the issue of cost-effectiveness arises as Vigilante must instrument individual services and does not protect the OS kernel at all. Unfortunately, kernel attacks have become a reality and are expected to be more common in the future [73]. For signature generation it relies on replaying the attack which is often not possible due to challenge/response interaction with cryptographic nonces, random numbers, etc.

We believe that both hardware-oriented full-system solutions, and OS- and process-specific software solutions are too limited in all three aspects mentioned in the beginning of this section. It is our intention to present a third approach that combines the best of both worlds and meets all of the criteria.

In this chapter, we describe *Argos* which explores another extreme in the design space for automated response systems. First, like Minos we offer whole-system protection in software by way of a modified x86 emulator which runs our own version of dynamic taint analysis [107] that is able to protect any (unmodified) OS including all its processes, device drivers, etc. Second, *Argos* takes into account complex memory operations, such as memory mapping and DMA that are commonly ignored by other projects. At the same time it is quite capable of handling complex exploits, such as register springs. This is to a large extent due to our ability to handle both virtual and physical addresses. Third, buffer overflow and format string / code injection exploits trigger alerts that result in the automatic generation of signatures based on the correlation of the exploit's memory footprint and its network trace. Fourth, while the system is OS- and application-neutral, when an attack is detected, we inject OS-specific forensics shellcode to extract additional information on the exploited code. Fifth, by comparing signatures from multiple sites, we refine *Argos*' signatures automatically. Sixth, signatures are auto-distributed to remote intrusion detection and prevention systems (IDS and IPS).

We focus on attacks that are orchestrated remotely (like worms) and do

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

Argos was initially designed to serve as a platform for *"advertised" high-interaction honeypots*, i.e., honeypots that in addition to running real services do not need to be "hidden" from the Internet. In the contrary, by actively providing links to the honeypot (e.g. using DNS, search engines, etc) we hope that it will gain visibility with attackers employing hit lists rather than scanning to discover targets. In such a set up, the assumption that only malicious, or at least suspicious traffic is received by the honeypot is no longer valid, and as such *Argos* is responsible for correctly identifying malicious traffic. In terms of performance, honeypots do not have the same requirements as desktop systems, offering us a certain freedom to use accurate but slower detection techniques. Nevertheless, *Argos* needs to be fast enough to run real services while offering a reasonable response time.

The remainder of this chapter is organised as follows. While related work is discussed mainly throughout the text, we summarise various approaches in Section 3.2. In Section 3.3 we describe the design of *Argos*. Implementation details are discussed in Section 3.4. The system is evaluated in Section 3.5. Conclusions are in Section 3.7

3.2 Related Work

For an attacker to compromise a host, it is necessary to divert its conventional control flow to execute his own instructions, or replace elements of the host's control flow with his own. As we discussed in Ch. 2, an attacker can accomplish this by overwriting values such as jump targets, function addresses and function return addresses. Alternatively, he can also overwrite a function's arguments or even its instructions. Such attacks have been prominent the last years and can be classified to the following major categories:

- *Stack smashing attacks* [3] involve the exploitation of bugs that allow an attacker to overflow a stack buffer to overwrite a function's return address, so that when the function returns, arbitrary code can be executed;

- *Heap Corruption attacks* [126, 30] exploit heap overflows that allow an attacker to overwrite an arbitrary memory location, and as a result execute arbitrary code;

- *Format string attacks* [57] are the most versatile type of attack. They exploit a feature in the `printf()` family of functions, which allows the number of characters printed to be stored in a location in memory. When a user supplied string is used as a format string, an attacker can manipulate the string to overwrite any location in memory with arbitrary values. These attacks offer more options to the orchestrator, including overwriting function arguments, such as the file to be executed of the `execve()` system call;

Such attack methods have been the subject of research by the security community for years, and a variety of defences have been developed as seen in Sec. 2.3. In the rest of this section, we will take a more in depth look at solutions more strictly related with *Argos* that make use of information flow analysis.

Suh et al [54] use dynamic information flow analysis to guard against overflows and some format string attacks, but their mechanism is not widely available for the most commonly used processor/OS combinations and indeed, to the best of our knowledge it has not progressed beyond simulation. Instead of real machines Dunlap and Garfinkel suggest virtual machines [55, 56] for the analysis of intrusions. Also, similar work to ours is presented in [148] which uses a modified version of the Dynamo dynamic optimiser. While *Argos* is different from these projects in many respects, we do follow a similar approach in that we employ an *emulator* of the x86 architecture.

Most closely related to our work are Minos [33] and Vigilante [91]. Like *Argos* both employ taint analysis to discover illegitimate use of 'foreign' data [107]. The differences with *Argos*, however, are manifold. Briefly, Minos is a hardware project that in the current software implementation on Bochs can only be deployed at a great cost in performance (up to several orders of magnitude slowdown[1]). Once misbehaviour is detected, Minos also makes no attempt to generate signatures. One of the reasons for this is that by aiming

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

at a hardware solution, Minos has had to sacrifice flexibility for (potential) performance, as the amount of information available at the hardware level is very limited. For instance, since the hardware sees physical addresses it is difficult to deal with complex attacks requiring virtual addresses such as register spring attacks. Moreover, it seems that generating signatures akin to the self-certifying alerts (SCAs) in Vigilante would be all but impossible for Minos. In contrast, while *Argos* works with physical addresses also, we explicitly target emulation in software to provide us with full access to physical-to-virtual address mapping, registers, etc.

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

3.3 Design

An overview of the *Argos* architecture is shown in Fig. 3.1. The full execution path consists of six main steps, indicated by the numbers in the figure which correspond to the circled numbers in this section. Incoming traffic is both logged in a trace database, and fed to the unmodified application/OS running on our emulator ①. In the emulator we employ dynamic taint analysis to detect when a vulnerability is exploited to alter an application's control flow ②. This is achieved by identifying illegal uses of possibly unsafe data such as the data received from the network [107]. There are three steps to accomplish this:

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

In other words, data originating from the network are marked as tainted, whenever they are copied to memory or registers, the new location is tainted also, and whenever it is used, say, as a jump target, we raise an alarm. Thus

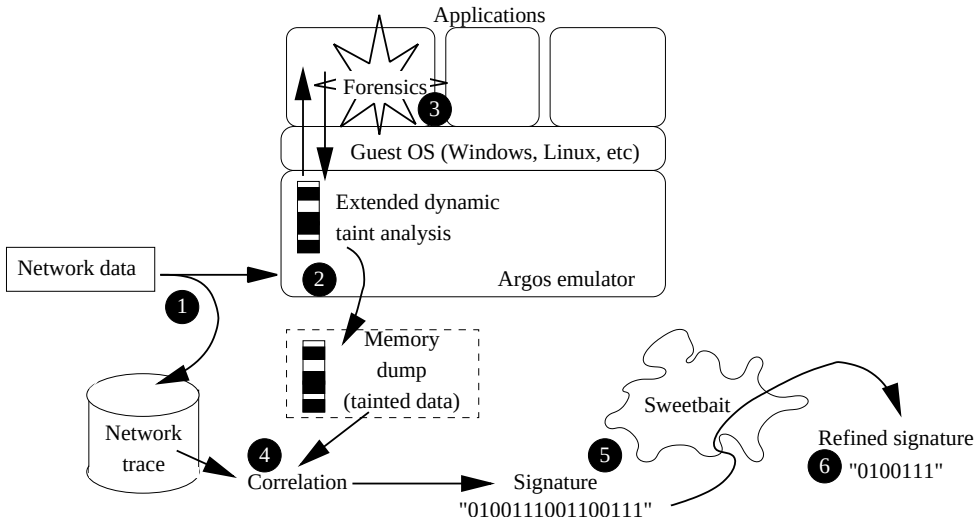


Figure 3.1: Argos: high-level overview

far this is similar to approaches like [91] and [107]. As mentioned earlier, *Argos* differs from most existing projects in that we trace physical addresses rather than virtual addresses. As a result, the memory mapping problem disappears, because all virtual address space mappings of a certain page, refer to the same physical address.

When a violation is detected, an alarm is raised which leads to a signature generation phase ③-⑥. To aid signature generation, *Argos* first dumps all tainted blocks and some additional information to file, with markers specifying the address that triggered the violation, the memory area it was pointing to, etc. Since we have full access to the machine, its registers and all its mappings, we are able to translate between physical and virtual addresses as needed. The dump therefore contains registers, physical memory blocks and specific virtual address, as explained later, and in fact contains enough information not just for signature generation, but for, say, manual analysis as well.

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

attack with our own shellcode.

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

The dump of the memory blocks (tainted data, registers, etc.) plus the additional information obtained by our shellcode is then used for correlation with the network traces in the trace database ④. In case of TCP connections, we reconstruct flows prior to correlation. The result of the correlation phase is a real signature that is, in principle, ready to be used for filtering. However, we do not consider the signature optimal and therefore try to refine it. For this purpose, *Argos* submits the signature to a subsystem known as *SweetBait*, which correlates signatures from different sites, and refines signatures based on similarity [121]. For instance, a signature consisting of the exploit plus the IP address of the infected host, would look slightly different at different sites. *SweetBait* notices the resemblance between two such signatures, and generates a shorter more specialised signature that it is then used in subsequent filtering.

The final step is the automated use of the signature ⑥. Attached to *SweetBait* are intrusion detection and prevention systems (IDS and IPS), that *SweetBait* provides with signatures of traffic to block or track. As IDS we use sensors based on the well-known open source network IDS *snort* [128] and for this purpose, *SweetBait* generates rules in *snort* rule format. The IPS is a relatively simple homegrown solution that employs the Aho-Corasick pattern matching algorithm to match network signatures. Although not very sophisticated, we have implemented it as a Linux kernel module that can be used directly with *SweetBait*. *SweetBait* is intelligent in the sense that it distinguishes between virulent attacks (e.g., many incidence reports) and rare events, and circulates the signatures accordingly. This is analogous to the way in which the police puts out bulletins for dangerous criminals rather than for, say, pickpockets.

The focus of this chapter is primarily on steps ①-④ and we will limit ourselves to summarising the *SweetBait* implementation. Interested readers are referred to [121] for details.

3.4 Implementation

Argos extends the *Qemu* [12] emulator by providing it with the means to taint and track memory, and generate memory footprints in case of a detected violation. *Qemu* is a portable dynamic translator that emulates multiple architectures such as x86, x86_64, POWER-PC64, etc. Unlike other emulators such as Bochs, *Qemu* is not an interpreter. Rather, entire blocks of instructions are translated and cached so the process is not repeated if the same instructions are executed again. Furthermore, instead of providing the software equivalent of a hardware system, *Qemu* employs various optimisations to improve performance. As a result, *Qemu* is significantly faster than most emulators.

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

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

3.4.1 Extended Dynamic Taint Analysis

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

Tagging

An important implementation decision in taint analysis concerns the granularity of the tagging. In principle, one could tag data blocks as small as a single bit, up to chunks of 4KB or larger. We opted for variable granularity; per byte tagging of physical memory, while at the same time using a single tag for each CPU register. Per byte tagging of memory incurs insignificant additional computational costs i.e., over per double word tagging, and provides higher accuracy. On the other hand, per byte tagging of registers would introduce increased complexity in register operations, which is unacceptable. It is worth noting that altering *Argos* to employ a different granularity is trivial. For reasons of performance and to facilitate the process of forensics at a later stage, the nature of the memory and register tags is also different.

Register tagging There are eight general purpose registers in the x86 architecture [69], and we allocate a tag for each of them. The tag is used to store the physical memory address from where the contents of the register originate. Segment registers and the instruction pointer register (`EIP`) are not tagged and are always considered *untainted*. Since they can only be altered implicitly and because of their role, they belong to the protected elements of the system. The `EFLAGS` register is also not tagged and is considered *untainted*, because it is frequently affected by operations involving untrusted data, and tagging it would make it impossible to differentiate between malicious and benevolent sources. FPU registers are also not tracked to reduce complexity and for the sake of performance, although *Argos* is *able* to tag them if required. MMX registers are frequently involved in memory copying operations, and are tracked by *Argos*. Due to the way they are represented internally by *Qemu*, MMX registers are tagged in the same way as memory (read below).

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

A *bitmap* is a large array, where every byte corresponds to 8 bytes in memory. The index idx of any physical memory address $paddr$ in the bitmap can be calculated by first shifting the address right by 3 ($idx = paddr \gg 3$) to locate the byte containing the bit flag ($map[idx]$). The individual bit flag is retrieved by using the lower 3 bits of $paddr$:

$$b = map[idx] \bigoplus (0x01 \ll (paddr \bigoplus 0x07))$$

The $size$ of the bitmap is an eighth of the guest's total addressable physical memory $RAMSZ$ ($size = \frac{RAMSZ}{8}$), i.e., the bitmap for a guest system of 512 MB would be 64 MB.

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

A *page directory* is a structure used by most modern computers to hold the virtual to physical memory address translations when paging is enabled [70]. We adopted a stripped-down version of this structure for tagging memory. For every memory page that contains a single *tainted* byte, a *submap* of type *bitmap* or *bytemap* is allocated for that page. The *submap* is placed in the page directory, so it can be quickly retrieved by using the 20 most significant bits of *paddr*. If a *submap* is not present then the entire page is treated as *untainted*, otherwise the status of an address is retrieved by using the 12 least significant bits of *paddr* ($idx = (paddr \oplus 0xfff) \gg 2$). A *page directory* is the slowest of the memory tagging schemes, as it adds another level for accessing a memory tag. On the other hand, it can reduce memory consumption significantly as most physical pages will never contain *tainted* data.

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

Tracking

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

- *2 or 3 operand* `ALU` *operations;* these are the most common operations and include `ADD`, `SUB`, `AND`, `XOR`, etc. The result is tainted, if one of the source operands is also *tainted*.

- *Data move operations;* these operations move data from register to register, copying the source's tag to the destination's tag.

- *Single register operations;* shift and rotate ops belong to this category. The tag of the register is preserved as it is.

- *Memory related operations;* all `LOAD`, `STORE`, `PUSH` and `POP` operations belong here. These operations retrieve or store the tags from or to memory respectively.

- *MMX operations;* are tracked by *Argos*. A *bytemap* stores the tags for these registers, which are internally represented as a byte array.

- *FPU and SSE operations;* are ignored, as explained above, unless their result is stored in one of the registers we track or to memory. In these cases, the destination is cleared. More advanced instructions such as SSE2 and 3DNow! are not supported by *Qemu*.

- *Operations that do not directly alter registers or memory;* some of these ops are `NOP`, `JMP`, etc. For most of these we do not have to add any instrumentation code for tracking data, but for identifying their illegal use instead, as we describe in the following section.

- *Sanitising operations;* certain fairly complex instructions result in always cleaning the tags of their destination operands. This was introduced to marginalise the possibility of false positives. Such instructions are BCD and SSE instructions, as well as double precision shifts.

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

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

Preventing Invalid Uses of Tainted Data

Most of the observed attacks today gain control over a host by redirecting control to instructions supplied by the attacker (e.g., shellcode), or to already available code by carefully manipulating arguments (return-to-libc attacks). For these attacks to succeed the instruction pointer of the host must be

loaded with a value supplied by the attacker. In the x86 architecture, the instruction pointer register `EIP` is loaded by the following instructions: `call`, `ret` and `jmp`. By instrumenting these instructions to make sure that a *tainted* value is not loaded in `EIP`, we manage to identify all attacks employing such methods. Optionally, we can also check whether a *tainted* value is being loaded on model specific registers (MSR) or segment registers, but so far we have not encountered such attacks and we are not aware of their existence.

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

Finally, to address attacks that are based solely on altering arguments of critical functions such as system calls, we have instrumented *Qemu* to check when arguments supplied to system calls like `execve()` are *tainted*. To reliably implement this functionality we need to know which OS is run on *Argos*, since OSes use different system calls. The current version of *Argos* supports this feature solely for the Linux OS.

Network Tracking: Mapping Tainted Data to Network Traffic

Argos tags mainly serve as binary flags indicating whether the associated data are *tainted*, with the exception of register tags that also point to the physical memory address where their contents were loaded from. As a result, anyone attempting to analyse a captured attack, would be presented with the challenge of mapping the contents of a *tainted* memory buffer, back to the original network packets that carried the data.

Argos provides this mapping by extending tags to track the network origin of each tainted byte. Every byte of data received by the virtual NIC is assigned a 32-bit *id* and logged to disk. The *id* is calculated by simply counting the number of bytes received by the NIC, and it stops being unique after the reception of 2^{32} bytes. In practice, honeypots are rather short-lived (i.e., they are periodically restarted to increase stability and security), so we do not expect that the *id* counter will wrap-around in real scenarios. Ultimately, an analyst can still map most data, even in the case the *id* does wrap-around.

Register and memory tags have to be extended to accommodate this *id*. This is straightforward for register tags, but introduces significant space overhead for memory tags. When network tracking *id*s are used, we enforce the

use of a page directory to hold memory tags, sacrificing some performance for reduced memory consumption. Performance is also burdened by the overhead of propagating the *id* during execution, making this mode of operation of *Argos* the slowest one. *Network tracking* is not enabled by default, but consists an invaluable tool for security analysts.

3.4.2 Signature Generation

In this section we explain how we extract useful information once an attack is detected, how signatures are generated, and how they are specialised by correlating memory and network traces. In addition we show how we refine signatures with an eye on obtaining small signatures containing an exploit's nucleus. Also, unlike related projects like [91], we intentionally investigated signature generation methods that do not require attacks to be replayed. Replaying attacks is difficult, e.g., because challenge/response authentication may insert nonces in the interaction. While we know of one attempt to implement replay in the face of cookies and nonces [36], we do not believe current approaches are able to handle most complex protocols.

We emphasise that the signature generation methods described in this section only server to demonstrate how the wealth of information generated by *Argos* can be exploited by suitable back-ends. Interested readers are referred to Prospector [136] for more information on generating accurate signatures with *Argos*.

Extracting Data

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

Additionally, every process is sharing its virtual address space with the kernel. OSes accomplish this by splitting the address space. In the case of Linux a 3:1 split is used, meaning that three quarters of the virtual address space are given to the process while one quarter is assigned to the kernel. Windows on the other hand uses a 2:2 split. The user/kernel space split is predefined in most OS configurations, so we are able to use static values as

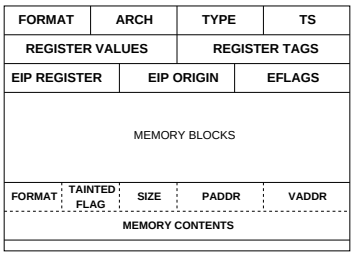


Figure 3.2: Memory dump format

long as we know which OS is being run. We take advantage of this information to dump only relevant data.

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

By locating all the physical pages accessible to the process / kernel, and making sure that we do not cross the user / kernel space split, we dump all *tainted* memory areas as well as the physical page pointed to by `EIP` regardless of its tags state. The structure of the dumped data is shown in Fig. 3.2. For each detected attack the following information is exported: the log's format (`FORMAT`), the guest architecture (`ARCH` could be `i386` or `x86_64`), the type of the attack (`TYPE`), the timestamp (`TS`), register contents and tags (including EIP and its origin), the `EFLAGS` register, and finally memory contents in blocks. Each memory block is preceded by the following header: the block's format (`FORMAT`), a *tainted* flag, the size of the block in bytes, and the physical (`PADDR`) and virtual (`VADDR`) address of the block. The actual contents of the memory block are written next. When network tracking is employed, the network *id*s for each byte in the memory block are also dumped. When all blocks have been written, the end of the dump is indicated by a memory block header containing only zeroes.

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

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

of both physical and virtual addresses in the log, a security specialist can effectively reverse engineer most, if not all, attacks.

Advanced Forensics

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

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

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

The forensics process retrieves information about the attacked process by running *netstat*, or if that is not available *OpenPorts* [45]. The above tools offer both the name of the process, as well as all the associated ports. The set of ports can be used to restrict our search in network traces (as discussed in Sec. 3.4.2) by discarding traffic destined to other ports. Currently, forensics are available for both Linux and Win32 systems. In the future, we envision extracting the same or more information without employing a third process at the guest.

Information Correlation

The memory fingerprint collected from the guest, along with the information extracted using advanced forensics are subsequently correlated with the network trace of data exchanged between the guest and the attacker. We capture traffic using `tcpdump` and store it directly in a trace database that is periodically garbage collected to weed out the old traffic streams.

The collected network traces are first preprocessed by re-assembling TCP streams to formulate a continuous picture of the data sent to the guest. For stream reassembly we build on the open source `Wireshark` library [113]. This enables us to detect attacks that are split over multiple packets either intentionally, or as part of TCP fragmentation.

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

(i) LCS is a popular and fast algorithm for detecting patterns between multiple strings also used by other automatic signature generation projects [81]. The algorithm's name is self-explanatory: it finds the longest substring that is common to memory and traffic trace. Along with the attacked port number and protocol we then generate a Snort signature. While this method appears promising, it did not work so well in our setup, as the common substring between the trace and memory is (obviously) huge. While we are still improving the LCS signature generation, we achieved the best results so far with CREST.

(ii) CREST is a novel but simple algorithm. The incentive behind its development was the fact that the output of *Argos* offers vital insight about the internal workings of attacks. The dumped information allows us to generate signatures targeting the string that triggers the *exploit*, and that may therefore be very accurate and immune to techniques such as polymorphism. Using the physical memory origin (O_{EIP}), value (V_{EIP}), and network *id* (N_{EIP}) of `EIP` (if available) we can pin-point the memory location that acts as the attacker's foothold to take control of the guest. The advantage of CREST is that it captures the very first part of an attack, which is less mutable.

Its current implementation is fairly simple. Essentially, we use N_{EIP} to locate V_{EIP} in the network trace, and proceed to match up the bytes surrounding it in the network trace, with the bytes surrounding V_{EIP} at O_{EIP}

in the memory dump. If N_{EIP} is not available, we locate V_{EIP} by scanning a subset of the network trace, based on the attacked port and protocol. We stop when we encounter bytes that are different, and use the resulting byte sequence combined with the port number and protocol to generate a signature in snort rule format. Signatures generated in this way were generally of reasonable size, a few hundred bytes, which makes them immediately usable. Moreover, as we show in Section 3.4.2, the signatures are later refined to make them even smaller.

Note that although we currently use only a small amount of it, for signature generation we are able to work with a wealth of information. In practice, *Argos* produces significantly more information than other projects [33, 91], because we have full access to physical and virtual memory addresses, registers, EIP, etc. So even though it proves to be very effective even in its current form, CREST should be considered as a proof-of-concept solution. For instance, CREST would fail to generate a signature for a format string attack, or any attack that manages to point EIP to a tainted memory area without tainting the register itself.

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

SweetBait

SweetBait is an automated signature generation and deployment system [121]. It collects snort-like signatures from multiple sources such as honeypots and processes them to detect similarities. Even though *Argos* is its main input for this project, we have also connected *SweetBait* to low-interaction honeypots based on *honeyd* [124] and honeycomb [81]. It should be mentioned that to handle signatures of different nature, *SweetBait* types them to avoid confusion. The *SweetBait* subsystem is illustrated in Fig. 3.3.

The brain of the *SweetBait* subsystem is formed by the control centre (CC). CC maintains a database of attack signatures that is constantly updated and it pushes the signatures of the most virulent attacks to a set of IDS and IPS sensors according to their signature budgets, as explained later in this section. In addition to the IDS/IPS sensors we also associate a set

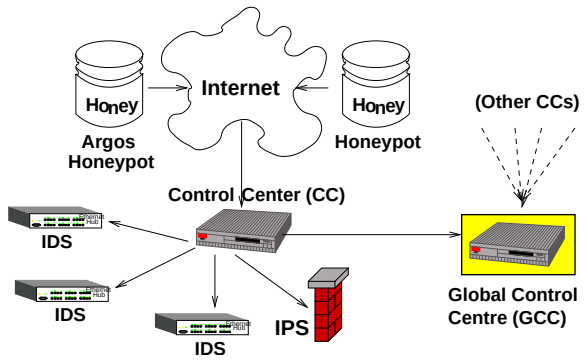


Figure 3.3: Architecture of the SweetBait subsystem

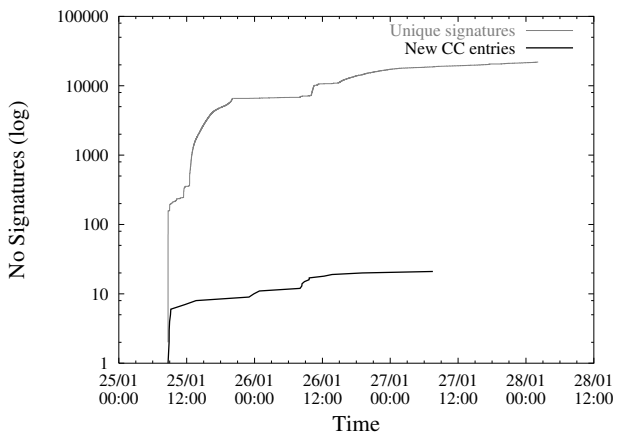


Figure 3.4: SweetBait signature specialisation results

of *Argos* honeypots with each CC. Honeypots send their signatures to their CC over SSL-protected channels. The signatures are gathered by the CC and compared against known signatures. In essence, it uses LCS to find the amount of overlap between signatures. If two signatures are sufficiently alike, it concludes that the LCS represents a more specialised signature for the attack and installs a new signature version that deprecates the older one. In this way, we attempt to locate the immutable part of signatures and remove the parts that vary, such as target IPs, host names, attack payloads, etc. Doing so minimises the number of collected signatures to a manageable size. For example, we employed *SweetBait* with low-interaction (honed/honeycomb) honeypots (since we had a much larger set of signatures for these honeypots than for *Argos*) and were able to reduce the thousands of signatures generated during the period of three days to less than 30 (Fig. 3.4).

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

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

SweetBait deploys the final versions of signatures to network IDSs and IPSs. To warrant increased performance levels of the connected IDSs and IPSs and deal with heterogeneous capacities of IDSs and IPSs, a signature budget in number of bytes can be specified, so that the number of signatures pushed to the sensors does not exceed a certain level.

To determine the signatures that will be pushed to the sensors, SweetBait uses network IDS sensors to approximate the virulence of the corresponding attacks. The density of generated alerts by the IDS is used as an indicator of aggression, which in turn determines whether a signature should be pushed to the prevention system or still be monitored[2]. In other words, a signature

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

that is reported frequently by many sites will have a higher virulence estimation than one that is reported less frequently and by a smaller number of honeypots, and is therefore more likely to be pushed to the IDS/IPS sensors. Additionally, signatures can be manually tagged as valid, or invalid to increase the level of certainty. Whether IPS sensors automatically block traffic based on signatures that are not manually tagged as valid is a configurable parameter. Details about the IPS sensors are beyond the scope of this thesis and can be found in [121] and [62].

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

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

3.5 Evaluation

We evaluate *Argos* along two dimensions: performance and effectiveness. While performance is not critical for a honeypot, it should perform well enough to allow the emulated system to run smoothly, and it should generate signatures in a timely fashion.

3.5.1 Performance

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

	Native	Qemu	Argos-B	Argos-B-CI
Served Requests/sec	499.9	23.3	18.7	18.3

Table 3.1: Apache throughput

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

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

In addition to the above, we used BYTE magazine's UNIX benchmark. This benchmark, `nbench` for brevity, executes various CPU intensive tests to produce three indexes. Each index corresponds to the CPU's integer, float and memory operations and represents how it compares with an AMD K6^{TM} at 233 MHz.

Fig. 3.5 shows the results of the evaluation. We tested the benchmark applications at the host, at guests running over the original *Qemu*, and at different configurations of *Argos*: using a bytemap, and using a bytemap with code-injection detection enabled. These are indicated in the figure as Vanilla QEMU, Argos-B, and ARGOS-B-CI respectively. The y-axis represents how many times slower a test was, compared with the same test without emulation. The x-axis shows the 2 applications tested along with the 3 indexes reported by `nbench`. Each bar in the graph is a configuration tested, which from top to bottom are: unmodified *Qemu*, *Argos* using a bytemap for memory tagging, and the same with code-injection detection enabled. `Apache` throughput in requests served per second is also displayed in Tab. 3.1.

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

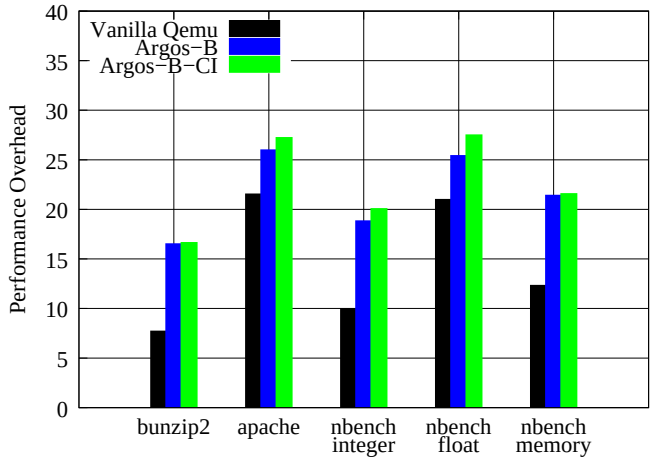


Figure 3.5: Performance Benchmarks

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

Moreover, even though the performance penalty is large, personal experience with *Argos* has shown us that it is tolerable. Even when executing graphics-intensive tasks, the machine offers decent usability to human operators who use it as a desktop machine. Moreover, we should bear in mind that *Argos* was not designed for use in desktop systems, but as a platform for hosting advertised *honeypots*. Performance is not our main concern. Still, we have plans to introduce novelties that will further improve performance in future versions of *Argos*. A related project that takes a similar approach, but focuses on performance with an eye on protecting desktops is described in [64].

3.5.2 Effectiveness

To determine how effective *Argos* is in capturing attacks, we launched multiple exploits against both Windows and Linux operating systems running on top of it. For the Windows 2000 OS, we used the Metasploit framework [94],

which provides ready-to-use exploits, along with a convenient way to launch them. We tested *all* exploits for which we were able to obtain the software. In particular, all the attacks were performed against vulnerabilities in software available with the professional version of the OS, with the exception of the War-FTPD ftp server which is third-party software. While we have also successfully run other OSes on *Argos* (e.g., Windows XP), we do not present their evaluation here for brevity's sake. For the Linux OS, we crafted two applications containing a stack and a heap buffer overflow respectively and also used `nbSMTP`, an SMTP client that contains a remote format string vulnerability that we attacked using a publicly available exploit.

A list of the tested exploits along with the underlying OS and their associated worms is shown in table 3.2. For Windows, we have only listed fairly well-known exploits. All exploits were successfully captured by *Argos* and the attacked processes were consequently stopped to prevent the exploit payloads from executing. In addition, our forensics shellcode executed successfully, providing us with process names, IDs, and open port numbers at the time of the attack.

Finally, we should mention that during the performance evaluation, as well as the preparation of attacks, *Argos* did not generate any false alarms about an attack. A low number of false positives is crucial for automated response systems. Even though the results do not undeniably prove that *Argos* will *never* generate false positives, considering the large number of exploits tested, it may serve as an indication that *Argos* is fairly reliable. For this reason, we decided for the time being to use the signatures as is, rather than generating self-certifying alerts (SCAs [91]). However, in case we incur false positives in the future, *Argos* is quite suitable for generating SCAs.

3.5.3 Signatures

The final part of the evaluation involves signature generation. To illustrate the process, we explain in some detail the signature that is generated by *Argos* for the Windows RPC DCOM vulnerability listed in Tab. 3.2.

We use the Metasploit framework to launch three attacks with different payloads using the same exploit mentioned above, against 3 distinct instances of *Argos* hosting guests with different IPs. The motivation for doing so is to force *Argos* to generate varying signatures for the same exploit. In this experiment, we employ the CREST algorithm (Sec. 3.4.2) to generate the signatures, and consequently submit them to the *SweetBait* subsystem.

During the correlation, CREST searches through the network trace and reconstructs the byte streams of relevant TCP flows. Note that the logs that are considered by the signature generator are generally fairly short, because

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

Table 3.2: Exploits captured by *Argos*

we are able to store separate flows in separate files by using the home-grown FFPF framework [16]. As a result, CREST may ignore flows that finished a long time ago and flows to ports other than the one(s) reported by forensics. The signature generation times including TCP reassembly for logs of various sizes is shown in Fig. 3.6.

SweetBait was configured to perform aggressive signature specialisation as explained in Sec. 3.4.2. Examining its database after the reception of all signatures, we discovered that it successfully classified them as being part of the same attack and generated a single specialised signature based on their similarities. The size of the signatures was effectively reduced from approximately 180 bytes to only 16 as it is shown in Fig. 3.7. The figure shows the payload part of the original signatures, generated by *Argos* without the *SweetBait* subsystem, as well as *SweetBait*'s specialisation. The signatures are represented in the way content fields are represented in snort rules, i.e., se-

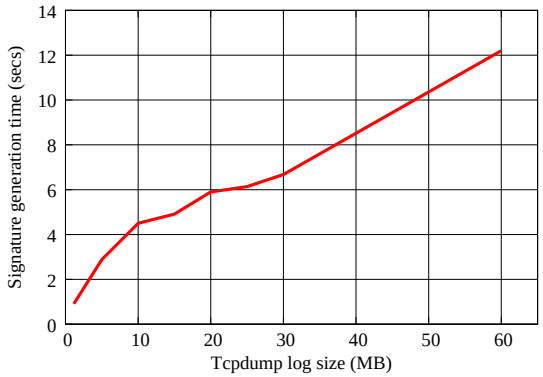


Figure 3.6: Signature generation

ries of printable characters are shown as strings, while series of non-printable bytes are enclosed on the left and right by the character '|' and represented by their hexadecimal values. Observe that the specialised signature generated by *SweetBait* is found in each of the original signatures, as shown by the boxes in Fig. 3.7.

Furthermore, we used the specialised signature to scan a benevolent network trace for the possibility of it generating false positives. Besides homegrown traces, we used the RootFu DEFCON[3] competition network traces that are publicly available for research purposes. We first verified that the exploit was not present in the traces, by scanning the trace with open source community snort rules, using rules obtained from bleeding snort[4]. Next, we scanned it with the signature generated by *Argos*. Again, there were no (false) alerts.

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

3.6 Systems Using Argos

SURFids is an open-source NIDS run by SURFnet, the network provider of the educational network in the Netherlands. SURFids collects data from sensors deployed on their client systems that act as low-interaction honeypots based on Nepenthes [8]. *Argos* has been integrated as a back-end of SURFids for handling unknown traffic that Nepenthes cannot handle.

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

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

Honey@Home [7] is a similar project, which forwards all unused ports on the installed system to servers that act as low-interaction honeypots. After filtering un-interesting traffic, Honey@Home forwards traffic to *Argos* honeypots to identify zero-day attacks.

SGNET [85] is a distributed honeypot deployment. It consists of lightweight sensors in the edges, which can identify existing attacks using Nepenthes [8]. Unknown traffic is forwarded to *Argos* and ScirptGen [86] technology is used to automatically generate Nepenthes modules based on the extracted data. Afterwards, the new modules are distributed to the lightweight sensors.

3.7 Conclusion

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

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

Original signatures:

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

```
|98 99|?B|99|H|99 99|I|96 93|J|F8 D6 F5 90|NKAJ|FC 97
90 91 D6|OA|27 F5 F9 92EB 10 EB 19 9F| u18 00|#7| F3
| w| EB E0 FD 7F|@N|9F 27 96|JH|FC|N|FC|F|90 D6 90
90 F9 97 9B|J7KO|91 D6|KKG|93 F9 9B 92 92|?KGF|FC|N|
93|F|9F 90 F9 98 92 98 96|A7C|97 99|J|FC|HI7|27|G|98
99|?F|D6 F9 98|@@|9F D6 98|@A|F8 92 93|IB|F8|BFH|98|
NHC|96 90 EB 04 FF FF FF FF|J|98 F8|J|92 9B 90|A|EB
04 EB 04|JKH|91 EB 04 FF FF FF FF F8 FD|J|FD|IH|96|?
?|93 91|C|D6|@NIHI|9F|@|F8 F5|G|D6 0A|
```

```
F?F|9B|C?|F5 98|F|27|IO|F9|?|FD|BB?|90 9B F5|?|FC|A
|9B|F|D6 97|CH|F5EB 10 EB 19 9F| u| 18 00| #7| F3| w
| EB E0 FD 7F9B|N|9F 27|?GC|F9|JH|F8|B@FICN|99 F9
97|B|9F 90 90 92|?|99 D6|JAB|90|ACO|93 27|N|FD|C|90|
O|96 F5 F9 90|H|98 90|?|93|A|99 93 FC 91 F8|O|9F 93
9B F9|I|D6 92|K@NH|F9 91|F|91|J7A?I|9B 98 93|N7A?|92
27|N|EB 04 FF FF FF FF|HIN7|99|N|98|G|EB 04 EB 04 99|
K|FC D6 EB 04 FF FF FF FF|AACK|98 90|@|92|77|93|?C
|9B|BF|9F 90 F5|A|FD 90 9B 9B 0A 0A|
```

SweetBait specialised signature:

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

Figure 3.7: Signature Specialisation (snort format)

Chapter 4

Eudaemon: On-demand Protection of Production Systems

'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]

4.1 Introduction

Sophisticated high-interaction honeypots like *Argos* (Chapter 3), Minos [33], and Vigilante [91] all use dynamic taint analysis. 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 [80, 162]:

1. Honeypot avoidance: an attacker may create a hit list containing all hosts that are not honeypots (e.g., using meta-information services) 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 [49, 131].

As we have seen in Section 2.3, other intrusion detection methods that do not rely on taint analysis and perform better than it do exist, but suffer from other problems 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 [91, 146] to address the latter issue, but successful replaying remains a subject of research in the presence of challenge/response authentication [37, 86]. 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 [135, 90]. 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 [64]. Their solution is based on a virtual machine (VM), that transfers execution

to a modified Qemu [12] 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 heavy-weight 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 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 [65]. 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 screensaver 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 they 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 and 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 [52]. 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., [155, 101]), 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 chapter, 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. Our contribution 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 chapter is organised as follows. In section 4.2 we place our work in the context of related work. Section 4.3 presents an overview of the system's design. Implementation details are given in Section 4.4. Section 4.5 evaluates performance, and conclusions are drawn in Section 4.6.

4.2 Related Work

Taint analysis is used by many projects such as TaintCheck [107], Minos [33], Vigilante [91], and *Argos* (Chapter 3). 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 [155, 101]. As a result, these techniques suffer from most of the problems identified in Section 4.1.

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 [64]. 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 [89]: 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 [90]. 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 [90] 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 4.1.

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 [91, 18, 136]), firewalls [157], and intrusion prevention systems on the network card [41].

For instance, Brumley et al. propose vulnerability-based signatures [18] 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.

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 [143], 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 [35], PointGuard [23], and address space and instruction set randomisation [14, 78] 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 based on compiler extensions and OS extensions like ASLR, and combine them with detailed instrumentation performed on a different host after replaying the attack [146]. In our opinion, replaying is still difficult due to challenge/response authentication, although promising results have been attained [37, 106]. 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 user machines.

Process hijacking is a common technique in the black-hat community [6, 125]. 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 [13], 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) [82] 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.

4.3 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

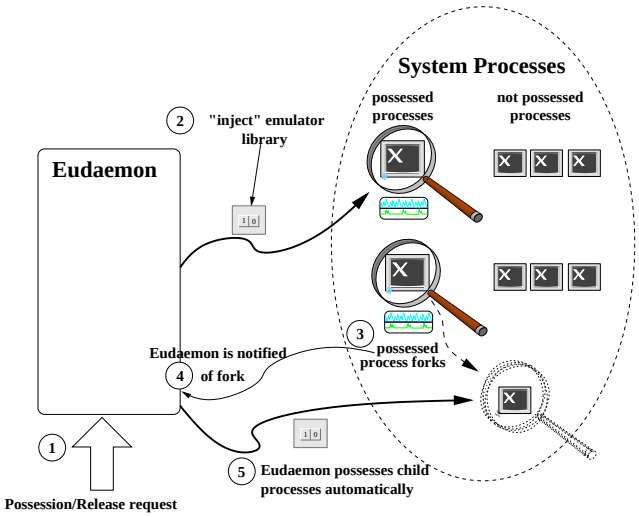


Figure 4.1: Eudaemon overview

shown in Fig. 4.1. 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.

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.

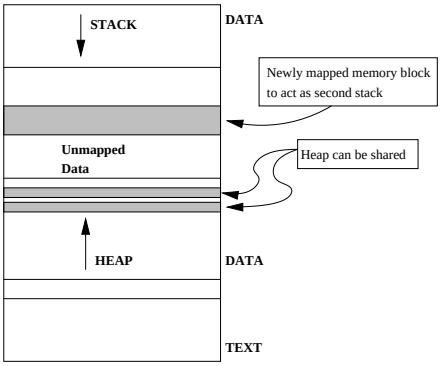


Figure 4.2: Process memory layout

4.3.1 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 they 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 4.4.

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.

Fig. 4.2 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 Fig., 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 4.3.3.

To safely call 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 [17]. A more extreme solution is to use the space left by compilers between the functions of an application for performance reasons [63], 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.

4.3.2 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, *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.

4.3.3 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 4.4. From a high-level perspective, the desired interface consists of three functions:

- 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 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 required 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 the 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 4.4.1 for details).

4.4 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 a prototype phase. 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.

4.4.1 SEAL: A Secure Emulator Library

SEAL is a secure, x86-based user-space process emulator implemented as a library. It is based on *Argos*, as described in Chapter 3, and employs the same dynamic taint analysis.

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 [12], 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 compact interface. We now discuss these issues and the general operation of SEAL in detail.

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.

Tracking Tainted Data

Data items tagged as *tainted* are tracked during execution. Tracking is achieved by instrumenting the guest's instructions to propagate tags, the same way as in *Argos*.

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: $X + (X/8) + 4 < 3072$ (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.

Attack Detection

Attack detection is also performed as in *Argos*, by checking instructions *call*, *jmp*, and *ret*. 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. 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 discussed in Section 3.4.2.

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 4.4.1.

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.

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.

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()*.

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.

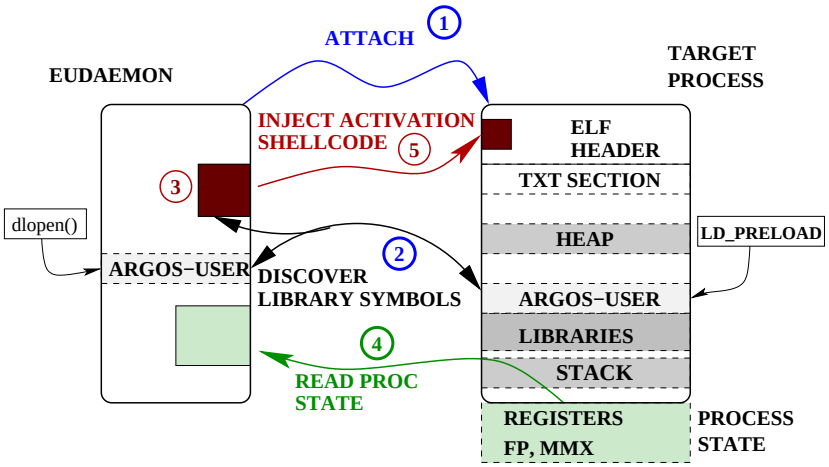


Figure 4.3: Process possession: phase 1

4.4.2 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 receiv-

```
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 4.4: Contents of a /proc/[pid]/maps file

ing 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.

Process Possession

The possession operation can be logically split in two phases. The first phase is shown in Fig. 4.3 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 runtime 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. Fig. 4.4 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 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

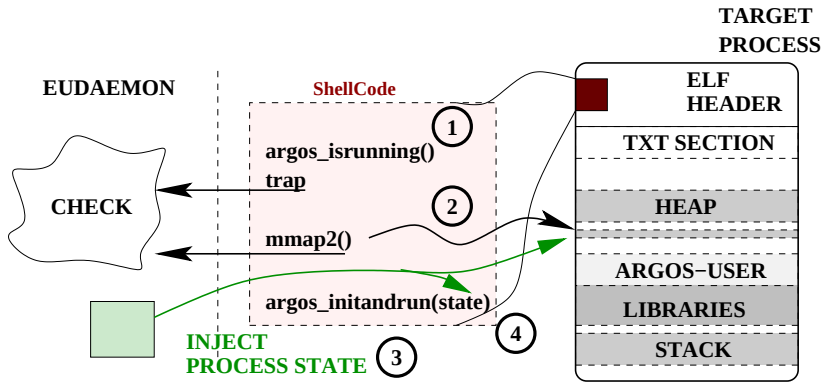


Figure 4.5: Process possession: phase 2

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 Fig. 4.5, 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

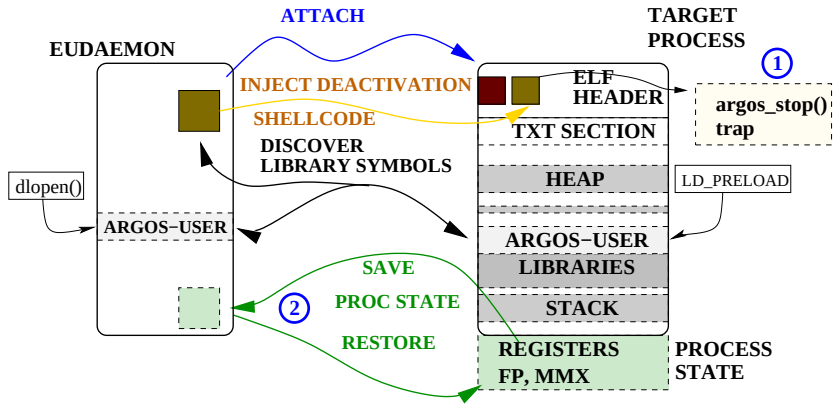


Figure 4.6: Process release: phase 1

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.

Process Release

Releasing a process is also partitioned in two phases with the first being similar to possession. An overview is shown in Fig. 4.6, and the *additional* steps in respect to possession (listed in Section 4.4.2) 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

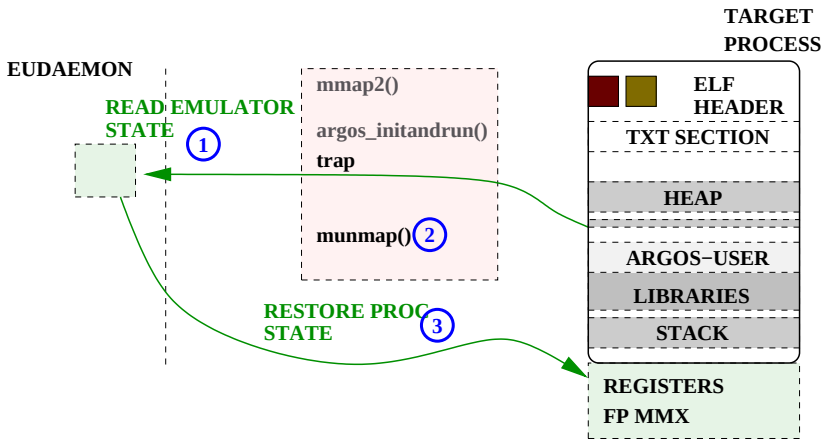


Figure 4.7: Process release: phase 2

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. Fig. 4.7 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

	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 4.1: Emulation overhead

release of the process.

4.5 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.

4.5.1 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 Slack-Ware 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 mea-

sure 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 style sheet. 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 4.1 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. 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 [64]. 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.

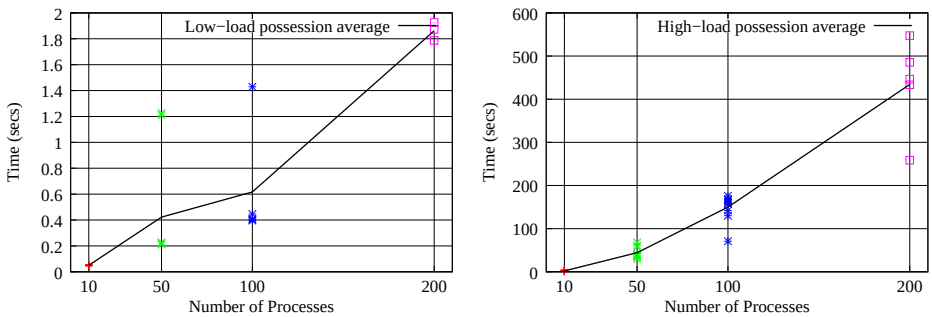
Konqueror yields the worst results. We ascribe this to the fact that the GUI, as well as rendering the content, uses many instructions that incur much overhead in emulation, including floating point operations as well as MMX operations.

4.5.2 Eudaemon

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

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 4.2: Eudaemon micro-timings (msec)



(a) Low CPU load (b) High CPU load

Figure 4.8: Scaling of process possession

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 4.2 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 4.4. 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:

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

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. Fig. 4.8 shows the time needed to switch a number of processes from native to emulated execution, under low and high CPU load. 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. 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 4.1.

For a security evaluation of SEAL/Argos the reader is referred to Section 3.5.2.

4.6 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.

Chapter 5

Decoupled Security for Smartphones

"Nothing is impossible if you can delegate." - Unknown

5.1 Introduction

Smartphones have come to resemble general-purpose computers: in addition to traditional telephony stacks, calendars, games and address books, we now use them for browsing the web, reading email, watching online videos, and many other activities that we used to perform on PCs. Additionally, a plethora of new applications, such as navigation and location-sensitive information services, are becoming increasingly popular.

The Problem As software complexity increases, so does the number of bugs and exploitable vulnerabilities [61, 116, 84, 112]. Vulnerabilities in the past have allowed attackers to use Bluetooth to completely take over mobile phones of various vendors, such as the Nokia 6310, the Sony Ericsson T68, and the Motorola v80. The process, known as bluebugging, exploited a bug in Bluetooth implementations. While these are older phones, more recent models, such as the Apple iPhone have also shown to be susceptible to remote exploits [108, 100].

Moreover, as phones are used more and more for commercial transactions, there is a growing incentive for attackers to target them. Credit card numbers and passwords are entered in phone-based browsers, while Apple, Google, Microsoft and other companies operate online stores selling applications, music, and videos. Furthermore, payment for goods and services via mobile phones is provided by Upaid Systems and Black Lab Mobile. Companies like Ver-

rus Mobile Technologies, RingGo, Easy Park, NOW! Innovations, Park-Line, mPark and ParkMagic all use phones to pay for parking, and yet others focus on mass-transit, such as for instance, HKL/HST in Finland and mPay/City Handlowy in Poland that both allow travellers to pay for public transport by mobile phone over GSM.

We see that both opportunity *and* incentive for attacking smartphones are on the rise. What about protective measures? On the surface, one might think this is a familiar problem: if phones are becoming more like computers, there is existing technology and ongoing research in fighting off attacks to PCs and servers. Unfortunately, it may not be feasible to apply the same security mechanisms in their current form.

While smartphones are like small PCs in terms of processing capacity, range of applications, and vulnerability to attacks, there are significant differences in other respects, most notably power and physical location. These two aspects matter when it comes to security. First, unlike normal PCs, smartphones run on battery power, which is an extremely scarce resource. For instance, one of the main points of criticism against Apple's iPhone 3G concerned its short battery life [102]. Vendors work extremely hard to produce efficient code for such devices, because every cycle consumes power, and every Joule is precious.

As a consequence, many of the security solutions that work for PCs may not be directly portable to smartphones. Anti-virus file scanners [99], reliable intrusion detection techniques [91], and other well-known techniques all consume battery power. While the occasional file scanning may be relatively cheap, more thorough security checks in light of the increasing software complexity and the threat of code injection attacks are pushing the likely security overhead upwards. Furthermore, for many organisations, such as law enforcement, banks, governments, and the military, the use of phones is both critical and sensitive, and cannot be subjected to the same aggressive security restrictions at the policy level that are common for their office intranets[1].

For high-grade security, it is desirable to run a host of attack detection methods simultaneously to increase coverage and accuracy. However, doing so exacerbates the power problem and may even incur some unacceptable slowdowns. Indeed, some of the most reliable security measures (like dynamic taint analysis) are so heavyweight that they can probably *never* be used on battery-powered mobile devices, unless we make significant changes to the

[1] A high-profile case in point is U.S. president Barack Obama's struggle to keep his Blackberry smartphone, after he was told this was not possible due to security concerns. Eventually, it was decided that he could keep an extra-secure smartphone and months of speculation followed about which phone, its additional security measures, and the tasks for which he is permitted to use it.

hardware. Battery life sells phones, and consumers hate recharging [102]. The likely result is that both vendors and consumers will trade security for battery life.

Second, phones are required to operate in unprotected or even hostile environments. Unlike traditional computers, phones go everywhere we go, and attacks may come from sources that are extremely local. A person with a laptop or another smartphone in the same room could be the source of a Bluetooth or WiFi spoofing attack [2]. That means that traditional perimeter security in general is insufficient, and even mobile phone security solutions that are based exclusively on "upstream" scanning of network traffic [28] will never even see the traffic involved in attacking the phone.

Worse, phones are small devices, and we do not always keep an eye on them. We may leave them on the beach when we go for a swim, slip them in a coat or shopping bag, forget them on our desks, etc. Theft of a phone is much easier than theft of a desktop PC or even a laptop. Attackers could 'borrow' the phone, copy data from it, open it physically, install back-doors, etc. For instance, after the bluebugging vulnerability mentioned above was fixed, phones could still be compromised as long as the attacker was able to physically access the device [84]. This is an important difference with the PCs we have sitting on our desks.

In summary, the trends are not favourable. On the one hand, mobile phones are an increasingly attractive target for attackers. On the other hand, because of power limitations and increased exposure to hostile environments, phones are inherently more difficult to protect than traditional computers.

Our Approach: Attack Detection on Remote Replicas At a high-level, we envision that security (in terms of detecting attacks) will be just another service to be devolved from the mobile device and hosted in a separate server, much like storage can be hosted in a separate file server, email in a mail server, and so on. Whether or not this is feasible at the granularity necessary for thwarting today's attacks has been an open research question, which we attempt to answer in this chapter.

More specifically, we explore the feasibility of decoupling the security checks from the phone itself, and performing all security checks on a synchronised copy of the phone that runs on a dedicated security server. This server does not have the tight resource constraints of a phone, allowing us to perform security checks that would otherwise be too expensive to run on the phone itself. To achieve this, we record a minimal trace of the phone's execution (enough to allow the security server to replay the attack and no more), and subsequently transmit the trace to the server for further inspection. The implementation of the full system is known as *Marvin*. It is illustrated in

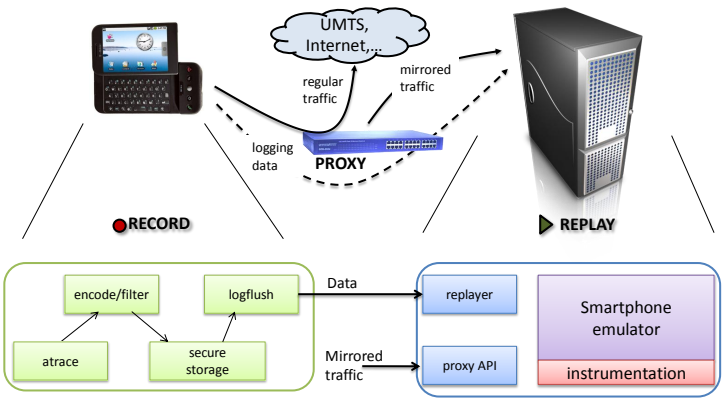


Figure 5.1: *Marvin* architecture

Fig. 5.1.

Our approach is consistent with the current trend to host activities in centralised servers, possibly consolidated in a *cloud*, including security-related functions. For instance, Oberheide *et al.* have explored antivirus file scanning in the cloud [110], and have more recently highlighted the opportunity for doing the same for mobile devices [111].

Although we subscribe to this trend at a high-level, we take a more aggressive approach to protection, especially considering the underlying threat landscape. Software on smartphones itself has frequently shown to be vulnerable to attacks for which the file scanning model is insufficient [84, 50, 112]. We therefore aim to prevent attacks on the phone software itself, focusing on exploits against client applications (as we have done on Chapter 4, and also covering misbehaving code installed by the user, and even cross-site scripting attacks. This is much more involved than scanning files for signatures. Nevertheless, our design still enables all the security checks to be pushed to an external security server.

Our solution builds on work on virtual machine (VM) recording and replaying [46, 159, 104, 29, 47]. Similar to these approaches, we record the execution of software running on a mobile phone and replay the exact trace on the security server. Rather than recording and replaying at VM level, we record the trace of a set of processes. We tailor the solution to smartphones, and compress and transmit the trace in a way that minimises computational and battery overhead. In addition to the replaying technique itself and the steps we take towards minimising the trace size, an important contribution presented in this chapter is therefore the new application domain for replaying: resource-constrained devices that cannot provide comprehensive security

measures themselves.

With the recorded trace, we can apply any security measure we want (including very expensive ones) and we can run as many detection techniques as we desire. By allowing heavyweight attack detection solutions, we are able to detect attacks that could not possibly be detected otherwise. Not only that, but we make it possible to study the attack in detail. We can replay attacks arbitrarily, possibly with more detailed instrumentation. And as not all phones are active at the same time, it is highly likely that replicas of multiple phones can share one physical machine.

An additional advantage is that loss or theft of a phone does not mean the loss of the data on it. All data, up to and including the last bytes transmitted by the phone, are still safely stored on the replica.

Contribution To the best of our knowledge, we are the first to use decoupled replaying to provide security for resource constrained mobile devices. More broadly, we claim that this is the first architecture capable of offering comprehensive security checks for devices that are increasingly important for accessing the network, often store or use sensitive information, exhibit a growing number of vulnerabilities (thus forming attractive targets), and cannot reasonably afford the security measures developed for less constrained systems.

Furthermore, we have fully implemented the security architecture on a popular smartphone (the HTC Dream/Android G1 [66]). The implementation demonstrates that the approach is feasible, while our experimental analysis suggests that the tracing and synchronisation cost is reasonable when compared to the kind of security offered on the server side. While the implementation is tied to Android, the architecture is not, as our dependencies on a specific phone or even operating system are very limited, and the *Marvin* design applies to other models also.

Implementing a project on the scale of *Marvin* involves several person years in programming effort, much of which is spent on solving low-level engineering problems. Rather than trying to cram these details into this chapter, we limit ourselves to the most interesting aspects of the architecture and implementation, and only discuss details when they are essential for understanding the bigger picture. The remainder of this chapter is organised as follows. Section 5.2 presents a brief overview of the threat model and the likely configuration in terms of function placement. Section 5.3 outlines the proposed system architecture and the key design decisions and trade-offs. Section 5.4 provides details on the tracing techniques and how they can be made efficient to minimise synchronisation overhead, and Section 5.5 outlines the server-side environment for replicating phone state and performing

security checks. Our experimental analysis of the Android-based implementation is presented in Section 5.6. Section 5.7 discusses related work that has influenced our design, and Section 5.8 summarises our research findings.

5.2 Threat Model and Example Configuration

We assume that all software on the phone, including the kernel, can be taken over completely by attackers. In practice, a compromise of the kernel takes place via a compromised user-space process. We do not care about the attack vector itself. We expect that attackers will be able to compromise the applications on the phone by means of a variety of exploits (including buffer overflows, format string attacks, double frees, integer overflows, etc.). Nor do we care about the medium: attacks may arrive over WIFI, 3G, Bluetooth, infrared, or USB.

In the absence of exploits, an attacker may also persuade users to install malicious software themselves by means of social engineering. Typical examples include trojans disguised as useful software.

Depending on the attack detection solutions that we provide on the security server, *Marvin* allows us to detect any and all of these types of attacks. To illustrate the power of the design, we implemented a security server that implements detection of code injection attacks by way of dynamic taint analysis [42, 107, 91].

As we have seen in previous chapters, dynamic taint analysis is a very powerful intrusion detection technique that is able to detect exploits (buffer overflows, format string attacks, double frees, and so on) that change the control flow of the program. However, it is also extremely expensive. For this reason, it is unlikely that taint analysis can *ever* be applied on the phone itself.

5.3 Architecture

A high-level overview of the *Marvin* architecture is illustrated in Fig. 5.1. We sketch the basic idea first and zoom in on various optimisations (such as the proxy and secure storage) in later sections. A *tracer* on the phone intercepts system calls of, and signals to, the set of processes that need protection. This set comprises all processes on the phone that may be attacked. It is typically a large set that includes the browser, media players, system processes, and so on). A *replayer* on the security server subsequently replays the execution trace, exactly as it occurred on the phone, while subjecting the execution to additional instrumentation. The transmission of the trace

is over an encrypted channel.

5.3.1 A Naive Implementation: Sketching the Basic Idea

A naive implementation would intercept and record *all* signals, all system calls, all the system calls' results, and all reads from and writes to shared memory. As soon as it records any of these events, it would transmit it *immediately* to the security server. The security server executes exactly the same processes on an exact replica of the system. Like the tracer, the replayer also intercepts all system calls and signals. Whenever it encounters a system call, it looks in the trace for the same call. At that point, it will not really execute the system call, but instead return the results that it finds in the trace.

Signals need special treatment. Because of their asynchronous delivery, they introduce nondeterminism. More precisely, since we do not know the exact moment of delivery on the phone and on the replica, they may cause race conditions. To ensure that signals are delivered at the same point both in the phone and in the mirrored execution on the security server, we do not deliver signals until the target process performs a system call. When the system call returns, we post the signal for immediate delivery. As both sides handle signals in exactly the same way, we synchronise signals delivery on the phone and the security server.

This way, we are *almost* able to replay the execution faithfully. The remaining issue concerns thread scheduling. As the kernel-level schedulers in phone and replica operate independently, it may be that threads on the replica are scheduled in an order different from that on the phone. For unrelated processes this is not a problem, but for threads that share memory (e.g., multithreaded applications), it is important that the scheduling order is preserved.

The simplest and fastest way to solve this problem is to modify the kernel scheduler. Of course, doing so limits portability and makes it difficult to apply our architecture to closed source systems. For now, we will assume that we have a kernel scheduler that schedules the threads exactly the same in both the phone and the security server. In Section 5.4, we will show an alternative way that enforces a schedule on the threads *over* the schedule generated by the kernel. Either method works as long as it satisfies the following two requirements: (1) two memory-sharing threads should never run at the same time, and (2) scheduling should be deterministic.

For now, we want to point out the flexibility that we have in terms of security measures. Given the trace, we can replay the execution as often as we like and employ any security measure we want, either one after another, or in

parallel. For instance, we can look for anomalies in system call patterns [123, 58], while at the same time applying dynamic taint analysis [42, 107, 91], and n-version virus scanning [111, 110].

A possible drawback is that there is a lag between the attack and its discovery (and possibly analysis). However, if the alternative is that the attack would not be detected at all, detecting an attack a few seconds after it infected the device still seems quite valuable [29].

As long as we can keep the cost of recording and transmitting the execution trace within reasonable bounds, the design above yields a powerful model to detect, stop and analyse attacks. We will shortly discuss various techniques to bring down the costs. There are three more issues that we need to discuss first: (i) where to place the security server, (ii) when to transmit the trace data, and (iii) how to warn the user when an attack is detected.

5.3.2 Location of the Security Server

Where to host the security server is a policy decision beyond the scope of this thesis. Rather than prescribing the right policy, we discuss three possible models. While the first of these models is the simplest and allows for most optimisation, we do not preclude the others. In practice, the optimal location of the security server is a trade-off between costs, privacy concerns, reliability, and performance.

In the most straightforward model, the security server is a service offered by the provider. The provider can use its security service to differentiate itself from other providers and to generate income by charging for the service. In addition, it is ideally suited for offering the service. Much of the data to and from the mobile phone is routed via the network provider. Routing the traffic via a security server is easy and cheap. While there may be concerns about privacy, we observe that even today many applications and data already reside in various 'clouds' and that much of the private data is already passing through the equipment of the providers. We trust the providers to respect the privacy of their clients.

However, alternative models are also possible. In a business environment where phones are provided by a company, the company may host its own security server. Sensitive business data will be stored only on company computers and the organisation can decide for itself what security measures to apply. The model requires that all phone communication is routed through the security servers in the company's server room. An extreme case would allow end users to run the replicas on their home machines. Doing so gives users full control over their data, at the cost of paying the provider to reroute the traffic plus the cost of the security server itself.

5.3.3 When to Transmit Trace Data

So far, we have assumed that the phone transmits the trace data immediately. In practice, however, this is probably not necessary. Transmitting data immediately implies that one of the device's network adaptors (e.g., 3G or WiFi) must be turned on continuously, which would use up power quickly. Furthermore, transmission cost per byte in power is lower if multiple events can be batched. Consequently, batching data before transmitting would save energy. The key insight is that we only need to transmit if there is a chance that the phone is compromised [132]. This is the case when the phone receives data from the network (be it over 3G, WIFI, USB, or Bluetooth), but not when processes on the phone exchange messages or when users update their calendars. In other words, we may be able to batch the trace data until we receive data that could potentially lead to a compromise.

Moreover, if the phone is fitted with secure storage that provides evidence of tampering even if the phone is completely under the control of the attacker, we may batch data arbitrarily. In that case, we save all the trace data in secure storage and sync with the security server at a convenient time (for instance, every hour, or once a day). In the extreme case, we switch to offline checks, where the phone only synchronises when it is recharging and battery life is thus no longer an issue.

Secure storage means that the attacker may falsify the events sent to the trace since the attack, but not any of the events that lead to the attack [156, 133]. Keeping the trace in storage for longer, also means that the attack can be active for a longer time. However, we will eventually discover it. Moreover, it means minimal overhead in battery consumption.

We shall see in Section 5.4.3 that *Marvin* includes a secure storage implementation that can guarantee the authenticity of all messages that lead to the attack without need for specialised hardware or VM isolation.

5.3.4 Notifying the User of an Attack

When *Marvin* detects an attack, it needs to warn the user, so that the user can start recovery procedures. This is not trivial. Sending an SMS or email message to say that the phone has been compromised may not work, as the phone is under the control of an attacker and the attacker may block such messages.

We have implemented a mechanism that allows us to download a clean image of the system including kernel, system files, and user data (as they were before they attack occurred), which is installed when restarting the device bringing the system to a "safe" state. This is an optimistic approach, as an adept hacker could disable the entire mechanism. A really reliable solution

requires hardware support that would enable us to securely restore system state under all circumstances. For instance, we could use what is known as a 'kill pill' on Blackberry phones: hardware that allows administrators to trash all data on a stolen or lost phone via a remote connection.

An alternative method (assuming lack of hardware support) could display a warning message on the display, and render the phone incommunicado on GSM, GPRS, UMTS, and other networks under control of the operator. This would inform the user that something is wrong, and force them to take action by, for instance, plugging the device to a computer, and restoring it to an acceptable state.

5.4 Recording in Practice

To make our architecture practical, we argued in Section 5.3 that the overhead of recording and transmitting the execution should be kept small both in computation and size. In this section, we will discuss how we achieved this in practice by means of various optimisations of the design. The implementation is known as *Marvin* and runs on an HTC Dream / Android G1, one of the latest 3G smartphones based on the open source Android software platform and operating system. The G1 in its default configuration comes with a host of applications, including a browser, mail client, media player, and different types of messaging applications.

The security server hosts replicas of multiple phones, running a replica of each on a Qemu-based Android emulator, which is part of the official SDK. For each replica one or more detection methods are applied, ranging from n-version virus scanning, to dynamic taint analysis.

5.4.1 Tracing on Android

For our implementation of Marvin on Android, we adopted a user-space approach based on the *ptrace* system call, which allows us to attach to arbitrary processes, and intercept both system calls and signals. By using *ptrace* we are able to track a system's processes, and receive event notifications each time they interact with the kernel. Events received include system call entry and exit, creation and termination of threads, signal delivery, etc.

Tracing from the Start

Marvin ensures that the tracing of relevant programs starts from the first instruction by means of a clean, two-step procedure that is illustrated in Fig. 5.2. In UNIX tradition, Android uses the `init` process to start all

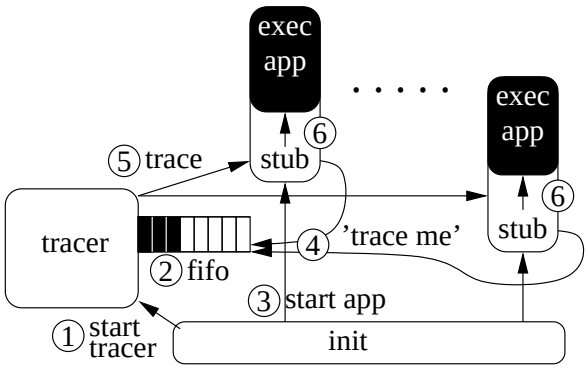


Figure 5.2: Tracing the processes from init

other processes, including the client applications such as a browser and media players, but also the JVM, and so on). `Init` in *Marvin* brings up the tracer process first. The tracer initialises a FIFO to allow processes that need tracing to contact it. Next, `init` starts the other processes. Rather than starting them directly, we add a level of indirection, which we call the *exec stub*. So, instead of forking a new thread and using the `exec` system call directly to start the new binary, we fork and run a short stub. The stub writes its process identifier (pid) to the tracer's FIFO (effectively requesting the tracer to trace it) and then pauses. Upon reading the pid, the tracer attaches to the process to trace it. Finally, the tracer resumes the paused process. The stub then executes the appropriate binary with the corresponding parameters.

Issues

Several complicating factors demand further attention.

SIGKILL *ptrace* cannot delay the delivery of SIGKILL and hence will not let the tracer intercept and defer it. To overcome this we intercept the signal at the source instead of the destination. Whenever a process sends a `SIGKILL` to another process, the tracer circumvents the kernel, and takes over the task of delivering the signal to the target, as well as recording the event.

Userspace scheduling A user-space implementation poses a challenge for the two scheduling requirements discussed in the previous section: (1) two threads that share memory should never run at the same time, and (2) scheduling should be deterministic. Rather than assuming that we can modify the kernel scheduler (or even receive scheduling events from the kernel), we opt

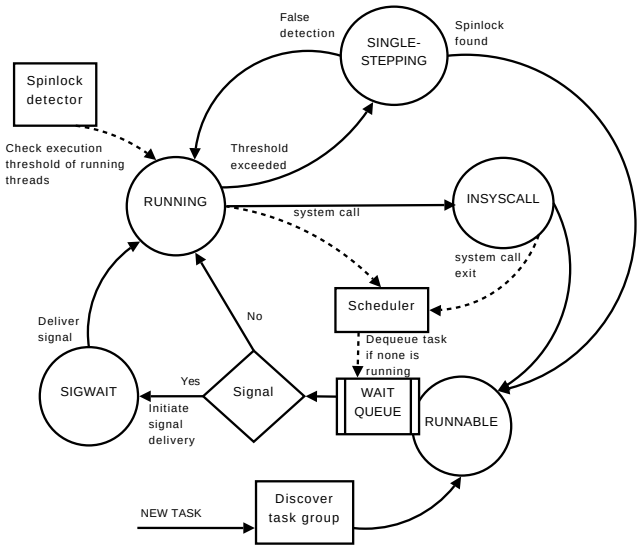


Figure 5.3: Scheduler FSM

for a user-space-only solution to allow our system to be ported to less open systems than Android.

Recall that the tracer intercepts both system call entry and exit, and signal delivery events. When the tracer receives such an event, it can decide to resume the thread, or delay its execution (e.g., by placing it on a waiting queue for later resumption). In other words, we have the functionality to determine which thread is run when. The aim is to enforce our own scheduling *over* the scheduling by the kernel.

We organise threads that share memory in task groups and maintain a run queue of runnable threads that are waiting to be run. Each of the threads can be in one of five states (see Fig. 5.3 for the finite state machine that controls the transitions):

1. RUNNING: the thread is currently running;
2. RUNNABLE: the thread is ready to run but waiting in the run queue;
3. INSYSCALL: the thread has entered a syscall;
4. SIGWAIT: the thread is waiting for a signal;
5. SINGLESTEPPING: a special state that is used to detect spinlocked threads (discussed later).

To satisfy requirement (1) we ensure that only one thread per group is in the RUNNING or SIGWAIT state. These states comprise all threads that

are 'active'. From the perspective of our scheduler, threads that enter the kernel on a system call are not active. They are not on the run queue either, since they are not ready to run. A thread waiting for a signal to be delivered (SIGWAIT) is 'pre-running' and makes a transition to the RUNNING state immediately upon signal delivery. A thread exiting from a system call is not resumed immediately, but is instead appended in RUNNABLE state to the back of the group's queue. The scheduler will decide on the scheduling and ensures that a thread can only make a transition to SIGWAIT or RUNNING if no other thread of the group is in either of these states. To satisfy requirement (2) we set the scheduler to run right after a deterministic event. Since system calls are deterministic, the scheduler is called after entry to or exit from a system call. When that happens it schedules the task at the head of the queue, and any pending signal is also delivered.

Sometimes processes share memory using mechanisms in the kernel. As in multithreaded applications, such sharing memory may introduce nondeterminism in the system. To cater to this issue, we have extended the scheduler to merge the run queue of processes that share a memory segment. Doing so ensures that all the threads sharing some memory are scheduled exclusively.

Spinlocks The scheduling solution above works well in practice. However, the scheduling that we force upon threads *may* lead to deadlocks when user-space threads use spinlocks. Spinlocks are considered to be bad programming practice for mobile device applications, because in terms of CPU cycles they are a wasteful way to perform locking. We have not encountered such deadlocks in Android, where other "sleeping" methods such as futexes are preferred (futexes perform a system call in case of contention). Nevertheless, we dealt with this potential issue by means of a spinlock detector which we tested on synthetic examples.

Marvin periodically activates a spinlock detector to look for tasks that are potentially within a spinlock. The detector marks tasks as 'possibly spinlocked' if they are in the RUNNING state (see Fig. 5.3) and the time since their last system call exceeds a threshold. As the situation is so rare, we optimistically set the threshold to a few seconds so that spinlock detection creates minimal overhead. A possibly spinlocked thread is moved to the SINGLESTEPPING state. We then single-step the thread to check whether it really is within a spinlock (e.g., we check whether it is running in a tight loop and we may test other properties also). When the thread is not spinlocked, it returns to RUNNING. If it is, however, *Marvin* sets the thread's state to RUNNABLE, appends it to the back of the run queue and schedules another thread. As the thread that is holding the lock will eventually run, the deadlock is removed.

Memory mapped by hardware Some memory, like the frame buffer, is mapped by hardware. This is not a problem in practice, as the memory access is essentially write only (e.g., the hardware writes bits to the framebuffer, which are not read by applications). However, it could be a problem in the future in a different hardware/software combination, if processes were to read the values produced by the hardware. In that case, we will probably have to record all reads to this memory area (to reproduce the same values at the replica). For instance, by mapping the area inaccessible to the reader, we could intercept all read attempts to this memory as page faults, but doing so would be expensive. Fortunately, we have had no need for this in our implementation.

I/O control Finally, I/O control, usually performed using the `ioctl` system call, is part of the interface between user and kernel space. Programs typically use `ioctl`s to allow user-land code to communicate with the kernel through device drivers. The interface is very flexible and it allows the exchange of data of variable length with few restrictions. Each `ioctl` request uses a command number which identifies the operation to be performed and in certain cases the receiver. Attempts have been made to apply a formula on this number that would indicate the direction of an operation, as well as the size of the data being transferred [26]. Unfortunately, due to backward compatibility issues and programmer errors actual `ioctl` numbers do not always follow the convention. As a result, the tracer needs knowledge of each command number, so that it is able to identify and log the data being read into user-space. Obtaining this metadata is a tedious procedure, since it requires referring to the kernel's source code. Luckily, a lot of metadata for common `ioctl` commands are available in various user-space emulators which saved us a lot of time.

5.4.2 Pruning Redundant Data: Trimming the Trace

The design above allows us to trace any process and replay it to detect attacks at the mirrored execution on the security server. Using *ptrace* is attractive, as it allows us to implement the entire architecture in user-space, with the sole exception of secure storage (see Section 5.4.3 for details about secure storage). A user-space implementation facilitates portability and allows the *Marvin* architecture to be applied to other phones even if the software on these phones is closed by nature, as long as they provide a tracing facility comparable to *ptrace*. However, from a performance point of view, system call interception in user-space is not the most optimal solution, as context switching is computationally costly. Most of the overhead can be removed by

implementing system call interception in the kernel.

The main challenge in either case is to minimise transmission costs. All aspects of the execution that can be reconstructed at the security server should not be sent. In the next few sections, we will discuss how we were able to trim the execution trace significantly. Each time, we will introduce a guiding principle by means of an example and then formulate the optimisation in a general rule.

Assuming that the phone and the replica are in sync, we are only interested in events that (a) introduce nondeterminism, and (b) are not yet available on the replica. In principle, replica and phone will execute the same instruction stream, so there is no need to record a system call to open a file or socket, or to get the process identifier, as these calls do not change the stream of instructions being executed. Phrased differently, they do not introduce nondeterminism. We summarise the above in a guiding principle:

Rule 1. *Record only system calls that introduce nondeterminism.*

Similarly, even though the results of many system calls introduce nondeterminism in principle, they still can be pruned from the trace, because the results are also available on the replica. For instance, the bytes returned by a `read` that reads from local storage probably influence the subsequent execution of the program, but since local storage on the security server is the same as on the phone, we do not record the data. Instead, we simply execute the system call on the replica. The same holds for local IPC between processes that we trace. There is no need to transmit this data as the mirror processes at the security server will generate the same data. As data in IPCs and data returned by file system `read`s constitute a large share of the trace in the naive implementation, we save a lot by leaving them out of the trace. Summarising this design decision:

Rule 2. *Record only data that is not available at the security server.*

In some cases, we can even prune data that is not immediately available at the security server. Data on network connections is not directly seen by the replica. However, it would be a serious waste to send data first from the network (e.g., a web server) to the phone, and then from the phone back to the network to make it available to the security server. Instead, we opted for a transparent proxy that logs all Internet traffic towards the phone and makes it available to the security server upon request (see also Fig. 5.1). As a result, whenever the replica encounters a read from a network socket, it will obtain the corresponding data from the proxy, rather than from the phone. In general, we apply the following rule:

Rule 3. *Do not send the same data over the network more than once. Use a proxy for network traffic.*

Besides deciding *what* to record, we can further trim the trace by changing *how* we record it. By encoding the produced data to eliminate frequently repeating values, we can greatly reduce its size. An out of the box solution we employed was stream compression using the standard DEFLATE algorithm [44] which is also used by the tool gzip. Compression significantly reduces the size of the trace, but being a general purpose solution leaves room for improvement. We can further shrink the size of the trace by applying delta encoding on frequently occurring events of which successive samples exhibit only small change between adjacent values. We found an example of such behaviour when analysing the execution trace after applying guidelines 1-3. System calls such as `clock_gettime` and `gettimeofday` are called very frequently, and usually return monotonically increasing values. By logging only the difference between the values returned by two consecutive calls we can substantially cut down the volume of data they produce. Special provisions need to be made for `clock_gettime`, since the kernel frequently creates a separate virtual clock for each process. As a consequence we must calculate the delta amongst calls of the same process alone for higher reduction.

In theory, delta encoding could be applied to all time related system calls with similar behaviour. However, doing so does not always reduce the trace size. For instance, we applied the technique to reads from `/proc/[pid]/stat` files, which also generated significant amounts of data. `/proc/[pid]/stat` files are files in Linux' `procfs` pseudo file system that are used to track process information (such as the identifier of the parent, group identifier, priority, nice values, and start time, but also the current value of the stack pointer and program counter). Typically, the entire file is read, but only a small fraction of the file actually changes between reads. As we will show in Section 5.6 the manual delta encoding of such reads may even lead to an increase in log size. The reason is manual encoding may replace high frequency data with less efficient encoding. In the final prototype, we therefore dropped this 'optimisation'.

We use related, but slightly different optimisations when items in the trace are picked from a set of possible values, where some values are more likely to occur than others. Examples include system call numbers and return values, file descriptors, process identifiers, and so on. In that case we prefer Huffman encoding. For instance, we use a single bit to indicate whether the result of a system call is zero, and a couple of bits to specify whether one or two bytes are sufficient for a syscall's return value, instead of the standard four. We summarise the principle in the following rule:

Rule 4. *Use delta encoding for frequent events that exhibit small variation between samples and Huffman encoding when values are picked from a set of possible values with varying popularity. Check whether the encoding yields real savings in practice.*

5.4.3 Secure Storage

We argued in Section 5.3.3 that transmitting trace information to the security server as soon as it is generated uses a lot of power, reducing battery life. It also requires a continuous connection to the network, which is unlikely. Instead, data are batched on the mobile device until we receive data over the network (which could potentially lead to an intrusion).

Batching introduces two security concerns [132]: firstly, the attacker must be prevented from tampering with the trace information to hide the evidence of an attack; and secondly, the attacker must be prevented from erasing the trace data.

We solve the problem of modified trace data by using digital signatures based on a keyed hash message authentication code (HMAC) [79]. HMAC enables the authentication of messages, and is based on a secret key and a cryptographic hash function (e.g., MD5, SHA-1, SHA-2, etc.). The secret key is established between the phone and the security server, when the systems starts up and is used to digitally sign the first message. After signing each message, the key is hashed to produce a new key of equal size, which replaces the previous key. The old key is completely overwritten, so that an attacker compromising the phone cannot tamper with messages and re-sign them to avoid detection from the security server. These mechanisms satisfies the requirements we have set earlier for secure storage by *preventing the tampering of all messages leading to an attack that can fully compromise a device.*

The steps taken to make storage tamper-evident can be summarised to the following:

1. $message' = message + HMAC(key, message)$

2. $key = HASH(key)$

Secure storage alleviates the problem of having to transmit data constantly. Current smartphones like the Apple iPhone 3G and the Android G1 already support microSDHC cards with 16GB of storage. We shall see in Section 5.6 that with this amount of storage we are able to store an entire day's worth of activity locally and only synchronise with the security server when we recharge the phone at the end of the day.

5.4.4 Local Data Generation

While we can save on data that is already available 'in the network' (at the security server or the proxy), no such optimisations hold for data that is generated locally. Examples include key presses, speech, downloads over Bluetooth (and other local connections), and pictures and videos taken with the built-in camera. Keystroke data is typically limited in size. Speech is not very bulky either, but generates a constant stream. We will show in Section 5.6 that *Marvin* is able to cope with such data quite well.

Downloads over Bluetooth and other local connections fall into two categories: (a) bulk downloads (e.g., a play list of music files), typically from a user's PC, and (b) incremental downloads (exchange of smaller files, such as ringtones, often from other mobile devices). Incremental downloads are relatively easy to handle. For bulk downloads, we can save on transmitting the data if we duplicate the transmission from the PC such that it mirrors the data on the replica. However, this is an optimisation that we have not used in our project.

Pictures and videos incur significant overhead in transmission. In application domains where such activities are common, users will probably switch to offline checks, storing the data in secure storage and synchronising only when recharging the phone. Video conferencing is not possible at all on most smartphones, including the Android and Apple iPhone models, as the cameras are mounted on the back. Furthermore, it becomes more common that content generated on mobile devices, such as pictures and videos, is uploaded on social networking web sites (e.g, Facebook, Twitter, MySpace, etc). In the future, we could exploit this fact, and proxy the uploaded data to be retrieved by the security server when required.

5.5 The Security Server

The security server decrypts, decompresses, and validates the trace it receives from the phone, and writes it to a file. The replica runs exact mirrors of the execution of the code on the phone in the Android emulator. The emulator is a QEMU-based application that provides a virtual ARM mobile device on which one can run real Android applications. It provides a full Android system stack, down to the kernel level. On top, we run the exact same set of applications as on the phone. The replica implements one or more security checks and uses the trace file to remove potential nondeterminism in the execution (as described previously). Initial execution starts with the same processor, memory and file system state.

A simple security measure is to scan files in the replica's file system us-

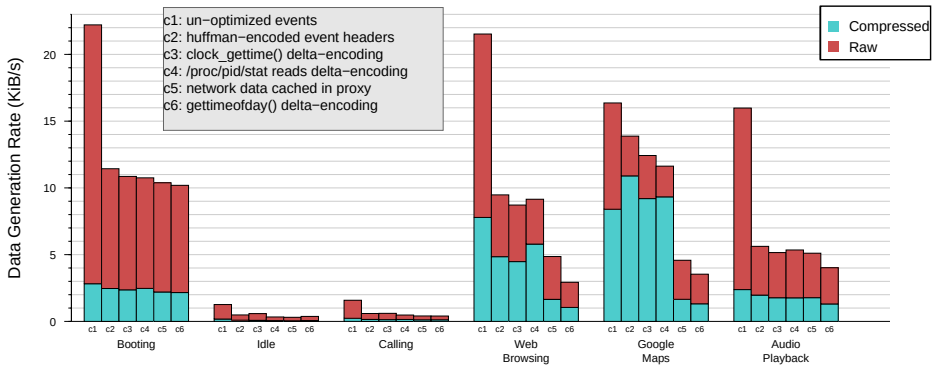


Figure 5.4: Data generation rate

ing traditional virus scanners. To increase accuracy and coverage we may employ multiple scanners at the same time, as suggested in the CloudAV project [111]. Even more interesting is the application of more heavyweight protection measures that could not realistically be applied on the phone itself.

To illustrate the sort of heavyweight security checks that are possible with *Marvin*, we modified the Android emulator to include dynamic taint analysis [42, 107, 91], similarly to Argos in Chapter 3. As we have seen, taint analysis is a powerful, but expensive method to detect intrusions, which would otherwise be impractical to apply on mobile devices.

5.6 Results

In this section we evaluate our user-space implementation of *Marvin*. The recording side (tracer) was ran on an Android G1 developer phone, while the security server side (replica) ran on the Qemu-based Android emulator, which is part of the official SDK. We will attempt to quantify various aspects of the overhead imposed by the tracer on the device, and also evaluate the various optimisations we described in earlier sections.

5.6.1 Data Generation Rate

We have frequently mentioned that data transmission is costly in terms of energy consumption and consequently battery life. As such, the amount of data that our implementation generates and transmits to the replica consists a critical overhead metric. We calculated the rate the tracer generates data under different usage scenarios. Fig. 5.4 shows the average rate measured in KiB/s. The tasks evaluated are from (left to right): booting the device,

idle operation, performing and receiving a call, browsing the WWW using randomly selected URLs from a list of popular links [4], browsing random locations using the Google Maps application, and finally audio playback.

We also evaluate the effectiveness of the optimisations described in Section 5.4.2. Six different configurations (c1-c6) were tested in total. Each configuration introduces another data trimming optimisation, starting from *c1* were no optimisations are used. *C2* adds Huffman-like encoding for event headers. Event headers comprise of common information logged for all system calls. Such information include the call number, the *pid* of the sender, return value, etc. *C3* and *c4* add delta encoding for `clock_gettime`, and reads from `/proc/pid/stat` respectively. Finally, *c5* uses a proxy to cache network data, and *c6* performs delta encoding for `gettimeofday`.

Compression was tested with all configurations, since it significantly reduces data volume. Fig. 5.4 shows that DEFLATE is the most efficient step in finding and eliminating repetition than any of our optimisations when network data are not involved, but the other optimisation also reduce the already small trace even more. Network access scenarios show that caching data using a proxy is necessary to keep overhead reasonable. Also, as mentioned earlier, the `/proc/pid/stat` delta encoding is counter-productive, since the encoding substitutes high frequency data with less compressible data. We do not use it in the prototype,

A mobile device usually spends most of its time idle, or it is used for voice communication. Fig. 5.4 shows that the data generation for these two scenarios is really negligible, with an average of just 64B/s and 121B/s for idle and calling respectively. These rates also show that employing secure storage (Section 5.4.3) to store even an entire days of execution trace locally is feasible using devices such as the G1 (or the IPhone).

5.6.2 Battery Consumption

Transmission and reception of network data, along with the CPU, and the display are the largest energy consumers on mobile devices. *Marvin* directly affects two of these components, since it requires transmitting generated data and uses CPU cycles to operate. We evaluated the effect of the tracer on battery consumption, by using the device to browse the web in a similar fashion as earlier. Additionally, SSL encryption was employed to protect the data being transmitted. Encryption itself is probably detrimental to battery life, but it is necessary. In the future, specialised hardware performing encryption cheaply could be included in mobile devices, allowing for broader adoption of encryption.

Fig. 5.5 shows how battery levels drop in time while browsing. As ex-

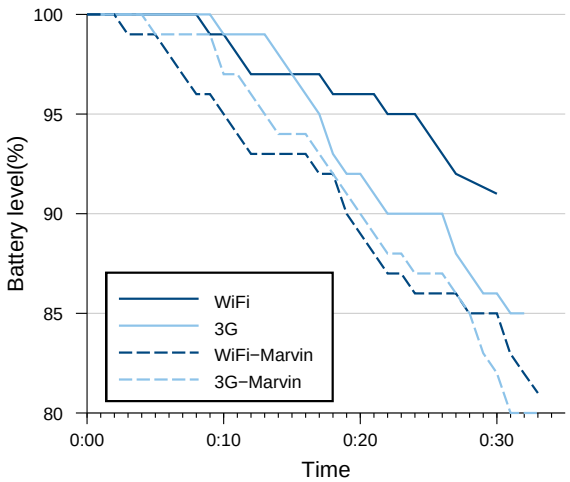


Figure 5.5: Battery consumption

pected battery levels drop faster when using *Marvin* than running Android natively. We also used both 3G and WiFi to evaluate their impact on battery life as well. When not using the tracer it is clear that WiFi is more conservative energy-wise. On the other hand, when using *Marvin* it is unclear if one is better than the other. On the positive side, our implementation incurs only a minor overhead on battery life, which, even for a costly operation such as browsing, does not exceed 7%.

5.6.3 Performance

The tracer also incurs a performance overhead. Fig. 5.6 shows the mean CPU load average during the experiment described in Section 5.6.2. In both cases CPU load was higher when using the tracer. Using profiling tools we analysed the tracer to identify bottlenecks. Table 5.1 shows the top calls where time was spent in the tracer.

A bit more than 65% is spent in system calls that are responsible for controlling or waiting for events concerning the traced processes. On the other hand, DEFLATE only takes up 7.62%. A more optimised, and platform dependent kernel-space implementation could shed most of the overhead we see here by eliminating context switching and notification costs, as well as data copying between address spaces.

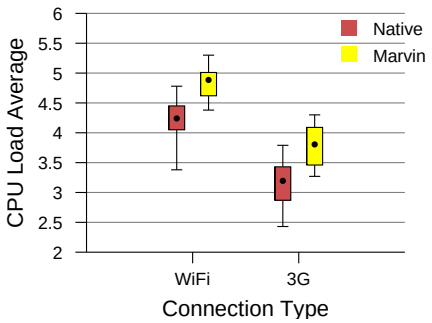


Figure 5.6: CPU load average

Function	Time Spent %
ptrace()	%33.63
waitpid()	%32.68
deflate_slow()	% 7.62
pread64()	%6.78
mcount_interval()	%2.84
event_handler_run()	%2.15

Table 5.1: Time spent in various parts of the tracer

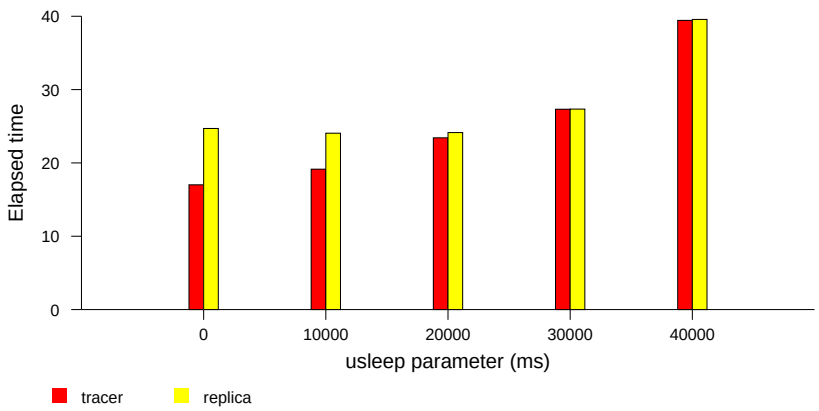


Figure 5.7: Security server lag

5.6.4 Security Server Lag

For particularly CPU intensive analysis, we expect that the security server will sometimes lag behind the tracer. Fig. 5.7 shows how varying the average load will affect the ability of the replayer to keep up with the tracer. This micro-experiment consists of running a test program that repeatedly compresses and decompresses 64KB worth of data. Between the compression and decompressing step there is a *usleep* call, with the parameter shown on the x-axis of the graph. The y-axis shows the elapsed time result of running the compression/usleep/decompression test 500 times. The tracer runs on the Android G1 device, and the replayer runs on the Android emulator on an AMD Athlon64 3200+.

The replayer cannot keep up with the real hardware when it comes to compression and decompression. However, the replayer does not have to execute the blocking system call (usleep). Therefore, increasing the time spend in blocking system calls (e.g., when the device is idle) reduces the lag of the replayer. The results for 0, 10000, and 20000 microseconds delay show that replayer is essentially unaffected by the delays and takes constant time. With the delays of 30000 and 40000 microseconds the replayer has to wait for results from the tracer and will therefore run at the same speed as the tracer.

5.7 Related Work

The idea of decoupling security from execution has been explored previously in a different context. Malkhi and Reiter [93] explored the execution of Java applets on a remote server as a way to protect end hosts. The code is executed at the remote server instead of the end host, and the design focuses on transparently linking the end host browser to the remotely-executing applet. Although similar at the conceptual level, one major difference is that *Marvin* is *replicating* rather than *moving* the actual execution, and the interaction with the operating environment is more intense and requires significant additional engineering.

The Safe Execution Environment (SEE) [140] allows users to deploy and test untrusted software without fear of damaging their system. This is done by creating a virtual environment where the software has read access to the real data; all writes are local to this virtual environment. The user can inspect these changes and decide whether to commit them or not.

The application of the decoupling principle to the smartphone domain was first explored in SmartSiren [28], albeit with a more traditional anti-virus file-scanning security model in mind. As such, synchronisation and replay is less of an issue compared to *Marvin*. However, as smartphones are likely to be targeted through more advanced vectors compared to viruses that rely mostly on social engineering, we argue that simple file scanning is not sufficient, and a deeper instrumentation approach, as demonstrated in *Marvin*, is necessary for protecting current and next generation smartphones. Oberheide *et al.* [111] explore a design that is similar to SmartSiren, focusing more on the scale and complexity of the cloud back-end for supporting mobile phone file scanning, and sketching out some of the design challenges in terms of synchronisation. Some of these challenges are common in the design of *Marvin*, and we show that such a design is feasible and useful.

The *Marvin* architecture bears similarities to BugNet [104] which consists of a memory-backed FIFO queue effectively decoupled from the monitored

applications, but with data periodically flushed to the replica rather than to disk. We store significantly less information than BugNet, as the identical replica contains most of the necessary state.

Schneier and Kelsey show how to provide secure logging given a trusted component much like our secure storage component [132, 133]. Besides guaranteeing the logs to be tamper free, their work also focuses on making it unreadable to attackers. We can achieve similar privacy if the secure storage encrypts the log entries. Currently, we encrypt trace data only when we transmit it to the security server.

Shadow honeypots [5] selectively trigger replicated execution when first-level anomaly detection techniques indicate that an action is potentially harmful and warrants further inspection. Some of the challenges of state transfer and replay are common with *Marvin*, but the focus of this work is on Web servers which are less constrained in terms of synchronisation cost.

Related to the high-level idea of centralising security services, in addition to the CloudAV work [110] which is most directly related to ours, other efforts include Collapsar, a system that provides a design for forwarding honeypot traffic for centralised analysis [74], and Potemkin, which provides a scalable framework for hosting large honeyfarms [152].

5.8 Conclusion

In this chapter, we have discussed a new model for protecting mobile phones. These devices are increasingly complex, increasingly, vulnerable, and increasingly attractive targets for attackers because of their broad application domain, and the need for strong protection is urgent, preferably using multiple different attack detection measures. Unfortunately, battery life and other resource constraints make it unlikely that these measures will be applied on the phone itself. Instead, we presented an architecture that performs attack detection on a remote security server where the execution of the software on the phone is mirrored in a virtual machine. In principle, there is no limit on the number of attack detection techniques that we can apply in parallel. Rather than running the security measures, the phone records a minimal execution trace. The trace is transmitted to the security server to allow it to replay the original execution in exactly the same way. The architecture is flexible and allows for different policies about placement of the security server and frequency of transmissions.

The evaluation of an implementation of the architecture in user-space, known as *Marvin*, shows that transmission overhead can be kept well below 2.5KiB/s after compression even during periods of high activity (browsing, audio playback) and to virtually nothing during idle periods. Battery life is

reduced by 7%. We conclude that the architecture is suitable for protection of mobile phones. Moreover, it allows for much more comprehensive security measures than possible with alternative models.

Chapter 6

Conclusion

The main goal of this thesis has been to investigate protection mechanisms against attacks exploiting memory access errors in software. Our work was initially motivated by the explosion in the number of computer worm attacks, and their profound effect on networks. For that reason, our focus has always been on techniques that can identify unknown attacks with a high-degree of certainty, such as dynamic taint-analysis. Another primary concern of ours has been to provide solutions that can be immediately applied on a variety of existing systems, such as servers, desktops, and smartphones. This chapter summarises our results, and offers a glance at possible future work directions.

6.1 Results

We can summarise the results of this thesis in the following points:

1. The *Argos* secure emulator (Chapter 3)
    - *Argos* is able to detect attacks without any previous knowledge or training, while at the same time not producing any false positives.
    - It consists a heavyweight security solution, but it balances performance with versatility, as it can be used to host next generation high-interaction honeypots running unmodified OSs (Linux, Windows 2000, Windows XP, BSD, etc) and applications.
    - It analyses captured attacks, and is able to automatically generate simple signatures for deployment on NIDS
    - The generated signatures can be used with existing NIDSs such as Snort, as well as our intrusion and detection system *SweetBait*.
2. *Eudaemon* (Chapter 3)

- We show that *Eudaemon* can almost instantly transform idle desktop machines to honeypots, thus overcoming the issue of honeypot avoidance, and greatly increasing attack detection range.
- Dynamic taint analysis has been mainly used in non-production environments such as honeypots. With *Eudaemon*, we show that it can be also applied on production desktop systems, protecting against client-side exploits, even if just for a short time.
- *Eudaemon* is able to automatically generate signatures for NIDSs, similarly to *Argos*. When employed in a significant number of nodes, the timely generation of signatures can protect entire communities of systems.

3. Decoupled security for smartphones (Chapter 5)

   - We are able to reproduce the execution of an Android-based smartphone on an Android emulator running on a remote server, while reducing battery life no more than 7%.
   - Replicating the execution of a smartphone, enables us to apply a novel and versatile model of security by running diverse security checks on the replica instead of the device.
   - Replicating execution, transparently backs up a device's settings and content.
   - Decoupling security checks from the smartphone, enables us to constantly apply heavyweight protection methods, such as dynamic taint analysis, despite the resource constraints (battery, CPU) of the device.

Additionally, *Argos* has become an important part of other network intrusion detection systems, i.e., SURFids, Honey@HOME, and SGNET.

6.2 Limitations and Future Work

Throughout this thesis we have used dynamic taint analysis for detecting attacks. The method has its limitations on the type of attacks detected. Consequently, none of the solutions we have implemented can protect against memory access errors that instead of directly altering the control flow of a program, modify a critical variable of the program (e.g., variable *authorized* in Fig. 2.1).

The signature generators we have presented in Chapter 3, served the purpose of demonstrating how *Argos* can be used at the heart of an automated

response system. The wealth of information made available by the virtualisation layer of *Argos*, grants us the ability to generate accurate signatures that focus on the exploit vulnerability, instead of particular malware instances. More research is needed in the generation of even more accurate and easily deployable signatures, such as self-certifying alerts (SCAs). In particular, for a system such as *Eudaemon* where untrusted nodes can generate signatures, such a mechanism is essential for their safe distribution and deployment.

Bibliography

[1] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 263–277, Washington, DC, USA, 2008. IEEE Computer Society.

[2] P. Akritidis, W. Y. Chin, V. T. Lam, S. Sidiroglou, and K. G. Anagnostakis. Proximity breeds danger: emerging threats in metro-area wireless networks. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–16, Berkeley, CA, USA, 2007. USENIX Association.

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

[4] Alexa The Web Information Company. The top 500 sites on the web. `http://www.alexa.com/topsites`.

[5] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 14th conference on USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2005. USENIX Association.

[6] anonymous. Runtime process infection. `http://artofhacking.com/files/phrack/phrack59/live/aoh_p59-0x08.htm`, July 2002.

[7] S. Antonatos, K. Anagnostakis, and E. Markatos. Honey@home: a new approach to large-scale threat monitoring. In *WORM '07: Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 38–45, New York, NY, USA, 2007. ACM.

[8] P. Baecher, M. Koetter, M. Dornseif, and F. Freiling. The nepenthes platform: An efficient approach to collect malware. In *In Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 165–184. Springer, 2006.

[9] M. Bailey, E. Cooke, F. Jahanian, D. Watson, and J. Nazario. The blaster worm: Then and now. *IEEE Security and Privacy*, 3(4):26–31, 2005.

[10] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, 2006.

[11] V. R. Basili and B. T. Perricone. Software errors and complexity: an empirical investigation0. *Commun. ACM*, 27(1):42–52, 1984.

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

[13] 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.

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

[15] D. Bolzoni, E. Zambon, S. Etalle, P. Hartel, and mmanuele Zambon. Poseidon: a 2-tier anomaly-based network intrusion detection system. In *In Proceedings of the 4th IEEE International Workshop on Information Assurance (IWIA)*, April 2006.

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

[17] BrainStorm. Writing ELF parasitic code in C. `http://vx.netlux.org/lib/vbs00.html`.

[18] 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.

[19] bulba and Kil3r. Bypassing Stackguard and Stackshield. *Phrack Magazine*, 0xa(0x38), May 2000.

[20] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

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

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

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

[24] L. Cardelli. Typeful programming. Technical report, Digital Equipment Corporation, 1989.

[25] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association.

[26] M. E. Chastain. Ioctl numbers. Linux Kernel Documentation - `ioctl-number.txt`, October 1999.

[27] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and C. Verbowski. Defeating memory corruption attacks via pointer taintedness detection. In *DSN'05*, pages 378–387, June 2005.

[28] J. Cheng, S. H. Wong, H. Yang, and S. Lu. Smartsiren: virus detection and alert for smartphones. In *MobiSys '07*, pages 258–271, New York, NY, USA, 2007. ACM.

[29] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *ATC'08: USENIX 2008*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

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

[31] M. Corporation. Protect yourself from the conficker computer worm. `http://www.microsoft.com/protect/computer/viruses/worms/conficker.mspx`, Arpil 2009.

[32] Coverity. Coverity software integrity solutions. `http://www.coverity.com`.

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

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

[35] 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.

[36] 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.

[37] 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.

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

[39] D. Moore, C. Shannon and K. Claffy. Code-Red: A case study on the spread and victims of an internet worm. In *Proceedings of the 2nd ACM/SIGCOMM Workshop on Internet measurement*, 2002.

[40] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford and N. Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, July 2003.

[41] 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.

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

[43] S. Designer. Openwall project. `http://www.openwall.com/`.

[44] P. Deutsch. DEFLATE compressed data format specification version 1.3. RFC 1951, May 1996.

[45] DiamondCS. Openports: Easy port analysis. `http://diamondcs.com.au/consoletools/openports.php`.

[46] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI '02*, pages 211–224, New York, NY, USA, 2002. ACM.

[47] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *VEE '08*, pages 121–130, New York, NY, USA, 2008. ACM.

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

[49] eEye. eEye industry newsletter. `http://www.eeye.com/html/resources/newsletters/versa/VE20070516.html#techtalk`, May 2007.

[50] F-Secure. "sexy view" trojan on symbian s60 3rd edition. `http://www.f-secure.com/weblog/archives/00001609.html`, February 2008.

[51] F-Secure. How big is downadup? Very big. `http://www.f-secure.com/weblog/archives/00001579.html`, January 2009.

[52] C. Fetzer and M. Süßkraut. Switchblade: enforcing dynamic personalized system call models. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 273–286, New York, NY, USA, 2008. ACM.

[53] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *In Proceedings of the Principles of Programming Languages (PoPL)*, January 2002.

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

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

[56] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proceedings of the 10th ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, February 2003.

[57] gera and riq. Advances in format string exploitation. *Phrack Magazine*, 0x0b(0x3b), July 2002.

[58] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *11th NDSS*, San Diego, CA, February 2004.

[59] J. Gosling and H. McGilton. *The Java Language Environment*. SUN Microsystems Computer Company, 1995.

[60] T. Guardian. Conficker is a lesson for MPs - especially over ID cards. `http://www.guardian.co.uk/technology/2009/apr/02/conficker-parliament-security-charles-arthur`, April 2009.

[61] L. Hatton. Reexamining the fault density component size connection. *IEEE Software*, 14(2):89–97, 1997.

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

[63] herm1t. Infecting ELF-files using function padding for Linux. `http://vx.netlux.org/lib/vhe00.html`.

[64] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 29–41, New York, NY, USA, 2006. ACM.

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

[66] HTC. T-Mobile G1 - Technical Specification. `http://www.htc.com/www/product/g1/specification.html`, 2009.

[67] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the singularity project. Microsoft research msr-tr-2005-135, Microsoft Corporation, Redmond, Washington, Oct 2005.

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

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

[70] Intel Corporation. *System Programming Guide*, volume 3 of *Intel Architecture Software Developer's Manual*. Intel, 1997.

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

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

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

[74] X. Jiang and D. Xu. Collapsar: A VM-Based Architecture for Network Attack Detention Center. In 13^{th} *USENIX Security Sumposium*, pages 15–28, August 2004.

[75] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang2. Cyclone: A safe dialect of C. In *Proceedings of the USENIX 2002 Annual Technical Conference*, pages 275–288, June 2002.

[76] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). *Security and Privacy, IEEE Symposium on*, 0:258–263, 2006.

[77] J. Kannan and K. Lakshminarayanan. Implications of peer-to-peer networks on worm attacks and defenses. Technical report, University of California, Berkeley, 2003.

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

[79] H. Krawczyk, M. Bellare, and R. Canetti. *RFC2104 HMAC: Keyed-Hashing for Message Authentication*. Network Working Group, February 1997.

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

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

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

[83] K. P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux J.*, page 7, 1996.

[84] G. Legg. The bluejacking, bluesnarfing, bluebugging blues: Bluetooth faces perception of vulnerability. TechOnline `http://www.wirelessnetdesignline.com/showArticle.jhtml?articleID=192200279`, April 2005.

[85] C. Leita. *SGNET : automated protocol learning for the observation of malicious threats*. PhD thesis, Thesis, December 2008.

[86] 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.

[87] J. Leyden. London hospital recovers from Conficker outbreak. The Register, `http://www.theregister.co.uk/2009/08/24/nhs_hospital_conficker/`, August 2009.

[88] M. Lipow. Number of faults per line of code. *IEEE Transactions on Software Engineering*, SE-8(4):437–439, July 1982.

[89] 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.

[90] 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.

[91] M. Costa, J. Crowcroft, M. Castro, A Rowstron, L. Zhou, L. Zhang and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–68, Brighton, UK, October 2005. ACM.

[92] M. Mahoney, , M. V. Mahoney, and P. K. Chan. Phad: Packet header anomaly detection for identifying hostile network traffic. Technical report, Florida Institute of Technology, 2001.

[93] D. Malkhi and M. K. Reiter. Secure Execution of Java Applets Using a Remote Playground. *IEEE Trans. Softw. Eng.*, 26(12):1197–1209, 2000.

[94] D. Maynor and K. K. Mookhey. *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research.* Syngress, 2007.

[95] Microsoft. Binary technologies projects: Vulcan and nirvana. `http://www.microsoft.com/windows/cse/bit_projects.mspx`.

[96] Microsoft. The C# language. `http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx`.

[97] Microsoft. Microsoft security intelligence report (SIR) volume 7 January - June 2009.

[98] Microsoft. Phoenix compiler framework. `http://research.microsoft.com/phoenix/phoenixrdk.aspx`.

[99] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. Avfs: an on-access anti-virus file system. In *13th USENIX Security*, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.

[100] H. Moore. Cracking the iphone (part 1). Available at `http://blog.metasploit.com/2007/10/cracking-iphone-part-1.html`, October 2007.

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

[102] W. Mossberg. Newer, faster, cheaper iPhone 3G. Wall Street Journal, July 2008.

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

[104] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05*, pages 284–295, Washington, DC, USA, 2005. IEEE Computer Society.

[105] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[106] 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.

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

[108] Niacin and Dre. The iphone / itouch tif exploit is now officially released. Available at `http://toc2rta.com/?q=node/23`, October 2007.

[109] P. Nylokken. Automated defacement through search engines, February 2007.

[110] J. Oberheide, E. Cooke, and F. Jahanian. Cloudav: N-version antivirus in the network cloud. In *17th USENIX Security Symposium*, San Jose, CA, July 2008.

[111] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahania. Virtualized in-cloud security services for mobile devices. In *Proceedings of MobiVirt*, Breckenridge, CO, June 2008.

[112] oCERT. CVE-2009-0475: #2009-002 opencore insufficient boundary checking during mp3 decoding. `http://www.ocert.org/advisories/ocert-2009-002.html`, January 2009.

[113] A. Orebaugh, G. Ramirez, J. Burke, and J. Beale. *Wireshark & Ethereal network protocol analyzer toolkit*. Jay Beale's open source security series. Syngress, 2007.

[114] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 55–64, New York, NY, USA, 2002. ACM.

[115] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 86–96, New York, NY, USA, 2004. ACM.

[116] A. Ozment and S. E. Schechter. Milk or wine: Does software security improve with age? In *15th USENIX Security Symposium*, Vancouver, BC., July 2006.

[117] D. Pauli. Number of viruses to top 1 million by 2009. Computer World `http://www.networkworld.com/news/2008/040408-number-of-viruses-to-top.html`, May 2008.

[118] PaX Team. Pax. `http://pax.grsecurity.net/`.

[119] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, pages 2435–2463, 1998.

[120] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *RAID*, pages 87–106, 2007.

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

[122] T. W. Post. Web browser vulnerabilities calenda. `http://www.washingtonpost.com/wp-srv/technology/interactives/browsers/`, February 2006.

[123] N. Provos. Improving host security with system call policies. In *12th USENIX Security Symposium*, 2003.

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

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

[126] rix. Smashing C++ VPTRS. *Phrack Magazine*, 0xa(0x38), May 2000.

[127] M. Roesch. Snort - lightweight intrusion detection for networks. In *LISA '99: Proceedings of the 13th USENIX conference on System administration*, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

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

[129] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th NDSS*, pages 159–169, 2004.

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

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

[132] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *7th USENIX Security Symposium*, pages 4–4, Berkeley, CA, USA, 1998. USENIX Association.

[133] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM TISSEC*, 2(2):159–176, 1999.

[134] 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.

[135] 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.

[136] 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.

[137] A. Slowinska and H. Bos. Pointless tainting? evaluating the practicality of pointer tainting. In *Proceedings of EUROSYS 2009*, Nuremberg, Germany, March-April 2009.

[138] D. Spyrit. Win32 buffer overflows (location, exploitation, and prevention). *Phrack Magazine*, 9(55), September 1999.

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

[140] W. Sun, Z. Liang, R. Sekar, and V. N. Venkatakrishnan. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *SNDSS*, pages 265–278, February 2005.

[141] SURFnet. SURFids. `http://ids.surfnet.nl`.

[142] Symantec. W32.sasser.worm. `http://www.symantec.com/security_response/writeup.jsp?docid=2004-050116-1831-99`, 2004.

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

[144] The Register. Microsoft security report shows worms are returning. `http://www.theregister.co.uk/2009/11/02/microsoft_security_report/`, November 2009.

[145] N. Times. Black market in stolen credit card data thrives on internet. `http://www.nytimes.com/2005/06/21/technology/21data.html?_r=1`, June 2005.

[146] J. Tucek, S. Lu, C. Luang, S. Xanthos, Y. Zhou, J. Newsome, D. Brumley, 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.

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

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

[149] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 116–134, Berlin, Heidelberg, 2008. Springer-Verlag.

[150] Vendicator. StackShield. `http://www.angelfire.com/sk/stackshield`, January 2001.

[151] J. Viega, J. Bloch, Y. Kohno, and G. McGraw. ITS4: a static vulnerability scanner for C and C++ code. In *ACSAC '00. 16th Annual Conference*, pages 257–267, December 2000.

[152] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *SOSP'05*, pages 148–162, 2005.

[153] K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. In *In Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID*, pages 203–222, 2004.

[154] X. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. Choi. Packet vaccine: black-box exploit detection and signature generation. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 37–46, New York, NY, USA, 2006. ACM Press.

[155] 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 *Proceedings Network and Distributed System Security (NDSS)*, San Diego, CA, February 2006.

[156] B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters. Building an encrypted and searchable audit log. In *NDSS'04*, 2004.

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

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

[159] M. Xu, R. Bodik, and M. D. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *ISCA '03*, pages 122–135, New York, NY, USA, 2003. ACM.

[160] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 29(6):97–106, 2004.

[161] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 29(6):97–106, 2004.

[162] 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.

Publications

Parts of Chapter 3 have been published in the ACM SIGOPS EuroSys 2006[1].

Parts of Chapter 4 have been published in the ACM SIGOPS EuroSys 2008[2].

Parts of Chapter 5 are under submission for publication.

Sammenvatting

Titel: Bescherming Tegen Nieuwe Internet Aanvallen met Behulp van Virtualisatie

Besturingssystemen, en software in het algemeen, nemen voortdurend toe in omvang en complexiteit. Als gevolg hiervan bevat software programmeerfouten die er vaak toe leiden dat via een aanval illegaal toegang of zelfs volledige controle kan worden verkregen over systemen. In het verleden hebben we besmettingen op grote schaal gezien van wormen zoals CodeRed, Blaster, en Sasser [39, 9, 142], die honderdduizenden hosts hebben weten te infecteren. De Slammer [40] worm was fenomenaal snel en infecteerde bijna alle kwetsbare servers binnen enkele minuten. Recent hebben we gezien hoe aanvallers bugs in populaire applicaties zoals web browsers misbruiken en de controle overnemen. De gekraakte systemen werden gecombineerd tot grootschalige netwerken die gebruikt werden voor het verzenden van spam email, het uitvoeren van distributed denial of service (DDoS) aanvallen en het verkrijgen van persoonsinformatie zoals credit card gegevens en wachtwoorden.

De praktijk wijst uit dat bestaande oplossingen onvoldoende zijn om aanvallen te detecteren en op tijd tegenmaatregelen te treffen. Deze dissertatie adresseert het automatisch en betrouwbaar detecteren van voorheen onbekende aanvallen en het genereren van vaccins om nieuwe infecties tegen te gaan in het beginstadium. We presenteren drie nieuwe methodes om virtualisatie te gebruiken voor het detecteren van zero-day aanvallen en het automatisch nemen van tegenmaatregelen. Onze oplossingen zijn gebaseerd op een techniek genaamd dynamische smet analyse, of, in het engels, 'dynamic taint analysis'. Dynamic taint analysis volgt het pad van data door een computersysteem, door alle interacties van data met het systeem te merken. (Hoofdstuk 2.3.4 beschrijft de methode in detail). Zeer belangrijk is dat deze oplossingen kunnen worden toegepast op bestaande hardware en software, en dat geen valse meldingen worden gegenereerd. Deze methode gebruiken we om het pad van zelfverspreidende aanvallen te volgen en te onderscheppen.

Netwerkgegevens worden praktisch nooit gebruikt om de uitvoering van

een programma direct te beïnvloeden. Zo worden bijvoorbeeld netwerkwaarden niet gebruikt als pointers naar een functie. Aanvallers misbruiken vaak geheugencorruptiefouten om de uitvoering van programmas te beïnvloeden, bijvoorbeeld door middel van deze functiepointers. Dynamic taint analysis detecteert wanneer op deze manier gebruik wordt gemaakt van netwerkgegevens en kan zo pogingen tot misbruik identificeren. Het implementeren van deze techniek in de software vereist het gebruik van een virtualisatielaag, zoals een emulator of een dynamisch 'binary translation framework', wat vaak leidt tot een significante vertraging van 1000%-2000%. Deze dissertatie heeft als duel het toepassen van DTA mogelijk te maken op bestaande systemen. Onze doelstellingen kunnen worden samengevat in de volgende onderzoeksvragen:

- is het mogelijk om oplossingen te vinden voor het detecteren van zogenaamde zero-day nieuwe aanvallen, door middel van het dynamisch volgen van gegevensstromen, in onaangepaste software en zonder toegang tot broncode of gespecialiseerde hardware?
- kunnen we de performance overhead van het dynamisch volgen van gegevensstromen mitigeren, zodanig dat onze oplossingen schaalbaar zijn naar diverse computersystemen zoals servers, desktops en smartphones?

Hoofdstukken drie tot en met vijf bevatten de kern-contributies van dit proefschrift.

Argos Hoofdstuk 3 presenteert een veilige emulator genaamd *Argos*. *Argos* is een platform voor de volgende generatie van hoge-interactie honeypots die de procedure van het vangen van zero-day aanvallen automatiseren en een simpel vaccin genereren voor netwerk inbraak detectie systemen ('network intrusion detection systems', of NIDS). Het biedt bescherming van het gehele systeem in software door middel van een aangepaste x86 emulator die onze eigen versie van dynamic taint analysis uitvoert [107]. *Argos* kan elk (onaangepast) besturingssysteem beschermen, inclusief bijbehorende processen en device drivers. *Argos* houdt rekening met complexe geheugenoperaties, zoals memory mapping en DMA, welke meestal genegeerd worden bij vergelijkbare projecten. Het kan aanvallen zoals buffer overflow en format string / code injection exploits detecteren en waarschuwingen afgeven die resulteren in automatische generatie van virus definities gebaseerd op de correlatie van de geheugen afdruk van de aanval en haar netwerk log. Nadat een aanval is gedetecteerd, voegen we een besturingssysteem-specifiek forensisch stuk code in om additionele informatie te verzamelen over de code van de aanval. Tot slot, door het vergelijken van definities van meerdere sites,

verfijnen we de gegenereerde definities automatisch en distribueren deze naar netwerk inbraak-detectie en -preventie systemen (IDS en IPS).

Eudaemon In Hoofdstuk 4 hebben we een techniek ontwikkeld om op transparante en veilige wijze desktopsystemen te laten fungeren als honeypots. *Eudaemon* stelt zich tot doel om de grenzen tussen beschermde en onbeschermde applicaties te vervagen, en combineert honeypot-technologie met eindgebruiker inbraak-detectie en -preventie. Het kan zich hechten aan elk lopend proces en de uitvoering verplaatsen naar een user-space emulator die de applicatie emuleert en taint analysis uitvoert. Zo lang de doel applicatie wordt geemuleerd zullen alle pogingen om de uitvoering te beïnvloeden, of om kwaadwillende code in te voegen, gedetecteerd en gemitigeerd worden. Indien gewenst kan *Eudaemon* zichzelf opnieuw hechten aan het geemuleerde proces en de uitvoering teruggeven aan de oorspronkelijke niet-geemuleerde proces. Het kan elke applicatie schakelan van beschermde naar native modus, bijvoorbeeld wanneer vrije cycles beschikbaar zijn, wanneer beveiligingsbeleid dat vereist, of indien dit expliciet wordt gevraagd. De transitie wordt transparant uitgevoerd en in zeer korte tijd, waardoor minimale verstoring optreedt in een actief gebruikt systeem. Net als *Argos* heeft het geen toegang nodig tot broncode of expliciete ondersteuning van het besturingssysteem, en is het in staat om virusdefinities te genereren voor NIDS.

Marvin In Hoofdstuk 5 onderzoeken we hoe mobiele apparaten zoals smartphones beschermd kunnen worden, door middel van het delegeren van veiligheidscontroles naar een zwak gesynchroniseerde replica van de software op een krachtiger machine. Smartphones zijn meer en meer op PCs gaan lijken in softwarecomplexiteit, waarbij deze complexiteit weer heeft geleid tot bugs en veiligheidslekken. Bovendien worden deze apparaten steeds meer gebruikt voor financiële transacties en andere privacygevoelige taken, waardoor ze een aantrekkelijk doelwit worden voor aanvallers. Smartphones verschillen echter van PCs in termen van de beperkte resources die beschikbaar zijn voor het ontwerp van beschermingsmechanismen, aangezien de batterij duur schaars is. Hierdoor zijn beveiligingsoplossingen die voor PCs zijn ontwikkeld niet direct toepasbaar op smartphones. Het outsourcen van veiligheidscontroles maakt complexe veiligheidscontroles zoals dynamic taint-analyis haalbaar, en tegelijkertijd transparante backup functionaliteit mogelijk. We hebben een prototype geimplementeerd genaamd *Marvin* op de HTC Dream / Android G1 en laten zien dat de extra kosten in termen van rekentijd en stroomverbruik acceptabel zijn.