Scargos: Towards automatic vulnerability distribution of zero-day vulnerabilities

Florian Rhinow

A dissertation submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Master of Science (Computer Science)

August 2013

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

____

Florian Rhinow

Dated: August 29, 2013

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this dissertation upon request.

Florian Rhinow

Dated: August 29, 2013

Abstract

Information about vulnerabilities spread too slow and allow for a significant attack window during which applications are virtually unprotected. Zero-day attacks jeopardise the security of any IT-system due to the lack of an effective remedy. Recent work has suggested automated approaches to vulnerability distribution, but are limited to memory-corruption detection techniques and disallow custom vulnerability response processes. We present Scargos, a novel approach to automate the distribution and verification of vulnerabilities, while allowing for automatic, custom countermeasures without the need to trust a central authority. By leveraging collaborative detection, vulnerabilities can be contributed by anybody and are announced to an open network by using packet-based self-certifying alerts (SCA), which are a proof of existence of a vulnerability by capturing the original, unmodified attack.

We compare two ways to generate and verify an attack: brute-force replay and exact stream replay. After successful verification, SCAs allow for a custom vulnerability response process such as generating automatic malware analysis reports or IDS signatures.

We evaluate Scargos with 24 real-world attacks, and show that for all detected attacks, we can generate and verify packet-based SCAs inexpensively and accurately. Scargos performs better for bigger SCA file sizes than previously proposed mechanisms. We show that our approach allows for detection of previously unknown attacks, whereas an entire life cycle including distribution and verification is achieved on average in under 2 seconds. While vulnerability distribution is at present mainly done manually, often reaching end-users after several hours, Scargos reduces the available attack window of adversaries to a minimum.

Acknowledgements

First and foremost, I would like to express the deepest appreciation to Michael Clear whose enormous support and comprehensive feedback was invaluable for completing this work. He gave me the necessary orientation and focus to achieve results that would have been out of reach without his guidance and persistent help. He not only supported me with regular feedback sessions, but he was especially supportive in the critical phases of completing this work and helped to solve any issues along the way, for which I am deeply grateful.

Advice and comments given by Stephen Barret have been a great help as they made me aware of a limitation in one of my algorithms. After overcoming this limitation the results of my work were significantly improved.

I also want to thank Mark Kührer and Thorsten Holz, who initially pointed me to investigate honeypots a few years ago. This eventually led me into developing the ideas that were necessary for this work.

I would also like to express my special thanks to my family, girlfriend and friends for their moral support and warm encouragements.

Florian Rhinow
University of Dublin, Trinity College
August 2013

Contents

Abstract iii

Acknowledgements v

List of Tables ix

List of Figures x

Chapter 1 Introduction 1

1.1 Motivation 2
1.2 Research Question 3
1.3 Contribution 3

Chapter 2 Background 5

2.1 Virtual Machines and their Security 5
2.2 Dynamic Taint Analysis 6
2.2.1 Performance 7
2.2.2 Accuracy 8
2.3 Honeypots 10
2.3.1 Overview 10
2.3.2 Argos 11
2.4 Vulnerability Distribution 12
2.4.1 Vigilante and Self-Certifying Alerts 12
2.4.2 Noah 16
2.5 Malware Processing Tools 17
2.5.1 CWSandbox 17
2.5.2 Honeycomb 17

2.6 Vulnerability Management . . . 17
2.6.1 Common Vulnerabilities and Exposures Identifier . . . 17
2.6.2 Open Vulnerability Assessment Language . . . 18
2.6.3 Open Vulnerability Assessment System and Network Vulnerability Tests . . . 19

Chapter 3 Architecture **20**
3.1 Detection and SCA Publishing . . . 21
3.2 Packet-Based Self-Certifying Alerts . . . 22
3.3 Distribution: SCA Repository . . . 23
3.4 SCA Verification . . . 24
3.4.1 Common Configuration . . . 25
3.4.2 Placement of the SCA Verifier . . . 27
3.4.3 Security Considerations . . . 28
3.5 Vulnerability Response . . . 30
3.5.1 Semi-Automatic Vulnerability Response . . . 30
3.5.2 Full-Automatic Vulnerability Response . . . 30

Chapter 4 Replay Mechanisms **33**
4.1 Brute-Force Replay . . . 35
4.2 Exact Stream Replay . . . 36
4.3 Limitations . . . 37
4.4 Summary . . . 39

Chapter 5 Implementation **41**
5.1 SCA Repository . . . 42
5.1.1 Packet-Based Self-Certifying Alerts . . . 44
5.1.2 Push-Notification . . . 45
5.2 SCA Publisher . . . 47
5.2.1 Attack Detection and Logs . . . 47
5.2.2 Single Packet Extraction . . . 48
5.2.3 Stream Extraction . . . 48
5.3 SCA Verification . . . 52
5.3.1 Brute-Force Replay . . . 52
5.3.2 Exact Stream Replay . . . 53

Chapter 6 Experimental Evaluation 55
6.1 Accuracy 58
6.1.1 Argos 60
6.1.2 SCA Generation 62
6.1.3 SCA Verification 66
6.2 Performance 67
6.2.1 SCA Verification 67
6.2.2 SCA Generation 73
6.2.3 Comparison to Vigilante 74
6.2.4 Overall Performance 76
6.3 Discussion 79

Chapter 7 Conclusion 81
7.1 Future Work 82

Bibliography 83

Appendices 87

Chapter A Additional Experimental Results 1

List of Tables

3.1 Comparison of Vigilante's SCAs with Packet-Based SCAs. 23
3.2 Benefits and Risks of using either an IDS or IPS for fully-automatic vulnerability response. 32

5.1 Drawbacks of using different types of distribution methods for push-notifications. 46

6.1 Applications and versions we used during our experiments. 56
6.2 Protocols and operating systems of the applications that we have investigated experimentallly. 58
6.3 Accuracy results of detecting, generating and verifying vulnerabilities by using packet-based SCAs. 60
6.4 Number of runs each search algorithm needs to perform to find a substring that matches a packet from the session packet capture. 64
6.5 Combined overview of the accuracy results for SCA generation. 66
6.6 Performance of single packet extraction and brute-force replay in relation to number and size of layer-5 PDUs. 69
6.7 Performance of stream extraction and exact stream replay in relation to number and size of layer-5 PDUs. 70

A.1 Attack Vectors of vulnerable FTP applications. 1
A.2 Performance of Scargos for selected well-known applications. 1
A.3 Performance of Vigilante [9]. 2
A.4 Standard Deviation of 10 runs of all investigated applications and metrics. The total time of Scargos' lifecycle was being measured using Stream Extraction and Stream Replay. 2
A.5 Attacks and vulnerabilities used in our experiments. 3

List of Figures

2.1 An example of Vigilante's SCA. 13
2.2 SCA verification of Vigilante [9]. 14
2.3 Vigilante's performance for SCA signature generation 14

3.1 Scargos' life cycle. 21

4.1 Different states a FTP client transitions into according to the responses of an FTP server. 34
4.2 Verifying an SCA by using brute-force replay for the FTP protocol. 36

5.1 Depiction of the different processes that are required to enable exact stream replay and brute-force replay. 42
5.2 An example application entry with an assigned application ID by a SCA repository. 44
5.3 An example SCA of the application WFTPD Server. 45
5.4 Output of the honeypot Argos showing a manipulated memory sections. . 48

6.1 The number of available windows server attack per protocol per application. 57
6.2 Image of Argos being successfully compromised. 61
6.3 Occurrence of different FTP command attack vectors 68
6.4 Performance of SCA verification in relation to the number of layer-5 PDUs. 71
6.5 SCA verification performance of brute-force replay and exact stream replay. 72
6.6 SCA Generation: Performance of single packet extraction and stream extraction. 74
6.7 Comparing Vigilante's and Scargos' Performance. 76
6.8 Total time of to complete Scargos' life cycle. 78

Chapter 1

Introduction

> "*In the early days of the Internet, port scans were the background noise of attackers and detected by firewalls. A few years later, vulnerability scanners were in vogue. They were detected by intrusion detection systems. Today, we use honeypots to detect automatic tools exploiting well-known flaws. To capture more interesting activities we have to look ahead and develop the next generation of honeypots. As always the arms race continues, and we need to stay ahead of the game!*"
> Niels Provos and Thorsten Holz, *Virtual Honeypots – From Botnet Tracking to Intrusion Detection*, 2008 [34].

Niels Provos and Thorsten Holz neatly summarise the past years of fighting virtual attackers. The MITRE Corporation, the official issuer of the *Common Vulnerabilities and Exposures* (CVE) identifier, has recently announced the need for a syntax change to address more than the current maximum of 10,000 vulnerabilities per year. Taking this into account, we seem to be more in need for advances in computer security than ever, "*to stay ahead of the game!*" [41].

Although, in some areas significant advancements have been made in security, other areas have been neglected; vulnerability distribution is one of these areas. The security of companies and organisations relies heavily on its IT and IT-security departments. Often specialised incident response teams are employed to surveil the state of the network. These teams react to intruders or analyse past attacks. However they also take action if a new vulnerability is announced for one of their used applications. New vulnerabilities are conventionally first announced by the vendor of an application and are later redistributed by secondary sources such as a CVE. The time from a vulnerability announcement until it is known by an incident response team is critical, yet it often relies

on manual checking of websites or newsletters, or is heavily reliant on secondary sources. The situation becomes worse when considering full-disclosure announcements, which can be made by virtually everybody. As a consequence, the number of primary sources seems indefinite. Automated software exists, but likewise, relies on the publication by a primary source. Furthermore, vendors often know about a vulnerability much earlier, but delay its announcement in order to write an appropriate patch in time – time, in which applications are vulnerable in companies and organisations, without their knowledge. Informed attackers can intrude and steal valuable assets, while there is no effective way to ensure protection.

However, there are vulnerabilities which remain unknown for an even longer period: zero-day vulnerabilities. These vulnerabilities are not known to the public, but only to attackers. A recent study by Symantec [4] developed the worldwide intelligence network environment (WINE) to collect data from over 11 million hosts around the world. The data was analysed in retrospect from 2008 until 2011 to determine whether vulnerabilities were previously used by attackers prior to their official discovery. In total, 18 zero-day vulnerabilities were found and the data suggests that the attacks remained undetected for between 19 days to 30 months; on average 312 days [4].

We present Scargos, a framework for automatic vulnerability distribution. Scargos can detect zero-day attacks by using dynamic taint analysis in a honeypot. The information about the vulnerability is then decoded as a packet-based self-certifying alert and verified by all interested parties. As we will show in this work, the entire process can be accomplished in on average less than 2 seconds, while not requiring that the participants trust each other.

1.1 Motivation

The attack window a sophisticated attacker has to compromise innocuous machines is big because:

1. Zero-day attacks are not known to the public for on average 312 days
2. Vendors which know about a vulnerability, withhold publication to develop a remedy
3. It takes time until end users know about a vulnerability announcement

Furthermore, the time until end users know about a vulnerability is increased because an announcement can be made by n seemingly indefinite amount of different primary

sources, while popular secondary sources such as CVE ID publications or vulnerability scanners rely on the publication of a vulnerability by a primary source.

This works strives to provide a solution to greatly minimise the attack window of an attack, while not requiring trust to be placed in a central authority.

1.2 Research Question

Vigilante and *self-certifying alerts* (SCAs) [9] have been suggested previously as a way to distribute vulnerabilities without the need of trusting a central authority. However, such SCAs modify the content of the original attack and can only be used by memory-violation detection engines. In contrast, packet-based SCAs require that the original attack remain unchanged, which allows for a custom vulnerability response process by end users, and can be used with any attack detection engine.

In this work we want to investigate whether packet-based SCAs can be generated and verified, and furthermore, to investigate what performance and accuracy they can achieve. We formulate our primary research question as follows:

Can packet-based SCAs be efficiently generated and verified?

This question leads to several other questions that we would like to answer.
Accuracy:

- Can packet-based SCAs be successfully generated and verified for a variety of attacks?
- Do packet-based SCAs achieve the same accuracy as the state-of-the-art?

Performance:

- Can Scargos effectively reduce the available attack window of attackers?
- Do packet-based SCAs outperform the state-of-the-art for well-known attacks?

1.3 Contribution

We show in our work, that packet-based SCAs can be constructed by using specialised extraction algorithms on the logs of DTA honeypots and we introduce replay mechanisms to enable verification. Our evaluation of packet-based SCAs shows that the time from generation to verification including server-based distribution is on average less than 2 seconds. Furthermore, we compare packet-based SCAs to conventional SCAs. Our

results suggest that packet-based SCAs perform better for greater file sizes and seem to have significantly better worst-case performance.

Chapter 2

Background

Our architecture is based on well-known security concepts such as *honeypots*, *dynamic taint analysis* or *virtual machines*, which we want to discuss in more detail in the following sections. Furthermore, we want to outline solutions which are currently either used or proposed for vulnerability management and distribution.

2.1 Virtual Machines and their Security

While virtual machines (VM) are guest operating systems (OS) that are being operated within a certain host, VM monitors (VMMs) are the software layer between the host's OS and the guest's OS, which emulate and control a virtual environment for the guest OS. Using VMMs enables us to perform a variety of operations with a certain guest OS such as:

- Instantiation, termination, stopping and starting of VMs;
- Saving the state of a VM (*snapshot*);
- Reinstantiating a VM from a saved snapshot.

Virtualisation is one of the core concepts which power today's cloud computing infrastructure, as it facilitates running multiple machines on one host, and greatly improves administration. Additionally, VMs provide a layer of segregation between the host and guest OS. The goal of this isolation is to prevent the guest operating system from obtaining or altering any information located on the host OS. As one might suspect, this lends itself well to many security applications; VMs are often used to prevent malware and threats from spreading to the host OS. VMs are employed in the realization of concepts such as:

Honeypots: gather malware and attacks in the wild [34].

Malware Analysis: dissects malware and observes its behavior [48].

Self-Certifying Alerts: process potentially infected information [9].

Cloud Computing: Applications such as *eucalyptus* rely on the security of VMs to prevent consumers from intruding the host OS [28].

Unfortunately, there are also three well-known attacks that can be performed against VMs:

Detection of the VM: The attacker can identify the vendor of the virtual machine and possibly more information. Most VMs are not designed to be protected against their detection and there exists several ways to detect all well-known VMMs [12,34].

Denial-of-Service: The VM terminates and any network-facing services of the VM is no longer available [12].

VM escape: An attacker can *escape* from the guest OS and execute arbitrary code on the host OS [12].

The latter attack is the most interesting because it defeats the goal of isolation and was first demonstrated by the attack *Cloudburst*. Cloudburst [21] allows an attacker to escape from *VMware Workstation 6.5.1* and earlier, as well as *VMware Player 2.5.1* and other versions, by using a vulnerability in its 3D support component (CVE-2009-1244). While attacks remain relatively scarce, arbitrary code execution vulnerabilities also exists in some versions of other VMMs such as *QEMU 1.3.0-rc2* (CVE-2012-6075) or *Microsoft Virtual PC 2004* (CVE-2007-0948). Also, there are VMMs for which we have not found a reported arbitrary code execution vulnerability (*Oracle Virtual Box*).

2.2 Dynamic Taint Analysis

Dynamic taint analysis (DTA) also known as *dynamic dataflow analysis* or *dynamic information flow tracking* has been proposed independently by multiple groups [8,10,27,38]. The proposed systems tackle the same problem in a similar way. In the following chapters, we will mainly focus on the implementation of dynamic taint analysis because this is the technique that is used in the honeypot Argos, which we used as a basis to build our prototype.

Software often has vulnerabilities that allow an attacker to execute malicious code. Attackers use exploits which are a certain type of attack to utilise a vulnerability. Most often *overwrite attacks* are used as an exploit. This category of attack subsumes most common threats such as *buffer overflow attacks* or *format string attacks.* In general, overwrite attacks manipulate the execution flow by using return addresses, function pointers, or format strings. This enables an attacker to control the program's execution flow and eventually execute arbitrary code [7].

DTA detects the attacks described by tracking all external data that is received by the host. External data is often received as incoming traffic such as network-facing services, but can also include other data such as that from external drives. External data is marked and all variables that use marked variables as inputs are also tagged. If a marked variable is used to change the execution flow, than this means that some external data is trying to dictate the program flow, which by definition makes it malicious. As a result, the execution of the external data is stopped and an alarm may be thrown [27].

DTA has a variety of applications in security but is often used in relation to detecting attacks. The advantages of DTA compared to conventional security measures such as *intrusion detection systems* (IDS) or other solutions are as follows:

- attacks can be discovered which have previously been unknown, so called *zero-day exploits* [35];
- a high accuracy of the detection mechanism is provided coupled with a very low false-positive rate [35];
- does not require any source code or software manipulation;

2.2.1 Performance

DTA performs quite poor compared to other security solutions. The slowdown depends on the application and implementation that is being used.

Newsome et al. experienced a slowdown of 3x to 100x in their first prototype implementation *TaintCheck* compared to the original execution time of the original application [27]. The DTA framework *Dytan* reports a similar high slowdown of 40x [7]. However, there are multiple proposals to improve performance. *FlexiTaint* [44] is a hardware solution which is added to an existing processor. Experiments have shown that it achieves a slowdown of only 1%-9%. Lam et al. [23] and Xu et al. [49] present a solution which in only 1.5x to 3x slower, but both solutions require that the source code of the application be present. However, recently the emulator *Minemu* [5] has

been proposed. The emulation architecture is specifically designed for DTA. In the environment, applications are only 1.5x to 3x slower.

Although recent improvements seem promising, the slowdown is still too high to be practical for real world scenarios. If a production network deployed DTA in its systems, a system that is at least 3x faster than the previous system would be required to uphold previous performance. As a result, a much higher cost would be required to operate the production network, which often outweighs the benefits in the eyes of the stakeholders. As a result, DTA has mainly been deployed in systems where speed is not as important, such as honeypots. The honeypot *Argos* [32] is on average about 15x slower compared to the performance of the original operating system. Yet, given that a honeypot has to handle far fewer requests than a production system, this slowdown is often negligible.

2.2.2 Accuracy

DTA can only detect attacks in which the control flow of the application is manipulated. In so-called *overwrite attacks*, jump targets such as return addresses or function pointers are overwritten in the benign application. Often the control flow is either redirected to point to injected code (code injection attack) or to existing code (existing code attack). Jump targets are overwritten using attacks such as:

format string attacks: An attacker is able to modify the format string argument into functions like `printf()` and uses special format tokens to modify the control flow.

buffer overflow attacks: An attack is able to write data to a buffer that has no boundary checks in place to prevent an overflow, which can ultimately allow arbitrary cod execution.

double free attacks: A memory portion that was previously allocated is deallocated twice with functions such as `free()`. If data is later written to a doubly-allocated memory portion, it might be vulnerable to a buffer overflow attack.

It is important to note that overwrite attack are by far the most common attacks used in worms and viruses. Many major worms such as the Slammer worm or CodeRed worm are based on overwrite attacks [27].

However, DTA will not prevent many other types of attacks that can be used to attack a system such as cryptographic weaknesses in the system, remote command executions or SQL injections. Although there are systems that use a special form of DTA to protect web application from attacks such as *SQL injections* [16] or *cross-site scripting* [45].

2.2.2.1 False Positives

Experiments that have been conducted with DTA suggest that its false-positive rate is extremely low. Newsome et al. [27] and Clause et al. [7] independently conducted accuracy experiments, which both resulted in no false positives. However, Newsome et al. states the following two cases in which false positives could occur:

1. No attack took place, but a software vulnerability that could be used for an attack was used.

2. The program intentionally uses external data to influence the control flow.

The first case might not be as severe because it still signals a vulnerability in the software. The second case will always throw an alert because external data is used to directly manipulate the control flow. A software developer may perform such an operation after applying security checks or escaping on the provided data. Although, this may not be considered best practice because these checks can fail to ensure the security of the input and thus open a backdoor into the application.

2.2.2.2 False Negative

False negatives can occur as a result of *under-tainting*, which is a known limitation of DTA. Yet, in some cases they might also occur because of an incorrectly configured analysis.

2.2.2.2.1 Under-tainting Under-tainting is an error of DTA in which a value is not tainted although it is influenced by another tainted value. Under-tainting is often a result of *implicit flows*, which occur when a tainted value implicitly affects another program value. An example is a table index of a hash map, which is closely connected to the related value of the hash map index. *Dytan++* is a solution which attempts to mitigate under-tainting. While the solution is able to fix most under-tainting errors, there are still special situations where it can not be prevented [17].

2.2.2.2.2 Trusting of input that should not be trusted A taint-analysis software has a policy that states which data inputs are trusted and which data inputs may contain malicious input. If these parameters are not specified correctly, data may be not be tainted which is in fact malicious and false negatives occur. For instance, some applications save external data to the disk before processing it. In this case, DTA may not taint these values if all data input is trusted that is read from the disk. Thus, a

policy could be configured that states that all data from a certain file path should be regarded as tainted [27].

2.3 Honeypots

Honeypots are resources which only obtain value when accessed with malicious intentions. Because honeypots hold no production value, any network-facing service that they offer exists to lure attackers into accessing the honeypot. As a result, any access to a honeypot can generally be regarded as suspicious. The dummy data that honeypots provide acts as a decoy to convince attackers about the authenticity of the machine. False positives are very seldom encountered when honeypots report an access because no user should have intentions to access the machine [37].

2.3.1 Overview

Honeypots are most often used by researchers to gather information about attackers such as their geographical distribution or used attacks. However, honeypots can also be used to enhance security in production networks, for example as an early warning system.

Commonly, honeypots are categorised into two categories: low-interaction honeypots and high-interaction honeypots. Low-interaction honeypots provide only an emulation of services to make attackers believe that they are communicating with a real service. However, because emulations have limitations, low-interaction honeypots are best suited for investigating automated attacks [34].

High-interaction honeypots offer a full system and are usually virtual machines with specialised monitoring software. They bear a higher security risk because they execute real attacks directly on the virtual machine. While this allows for detailed forensic analysis, high-interaction honeypots are usually secured further by using reverse-firewalls. These firewalls only permit requests to the honeypot, but the honeypot itself is disallowed to establish any connections [34]. This way worms are contained within the host and cannot infect real machines. Low-interaction cannot become compromised as such, as they only provide emulations of services and gather incoming traffic accordingly. However, attempts could be made to specifically target honeypots for an attack, in which case a vulnerability within the emulation has to be found and exploited. However, to date there have not been any attacks specifically against honeypots which is most likely owed to their few users compared to other network-facing services [47].

2.3.2 Argos

Argos is a honeypot that attempts to mitigate many of the drawbacks that accompany conventional high-interaction honeypots. It decreases the risk and maintenance of high-interaction by accurately detecting attacks and stopping them before they compromise the system.

The development of Argos started in 2006 at Vrije University in Amsterdam. The current available version is 0.5.0 [46]. This high-interaction honeypot works differently to other solution tools. It works only in conjunction with the virtualisation software QEMU. On QEMU, all common operating systems can be installed; thus it can be used like practically any other virtualisation software for high-interaction honeypots. Yet, DTA is built into the virtualisation software. Thus Argos and the DTA lies in between QEMU and the virtual operating system [34]. This leads four distinct advantages:

Reduction of risk Argos accurately detects attacks by using DTA and stops them before any malicious code is executed.

Improved maintainability After detecting attacks, Argos logs the incident and extracts more useful data about the attack. As a result, less forensic analysis is needed compared to conventional analysis.

System-wide detection Argos offers an entire virtualisation environment which detects applications as well as kernel attacks.

Captures zero-day attacks Low-interaction honeypots can only detect attacks which fall into a certain pattern and are thus previously known. In contrast, DTA detects any exploit that uses an overwrite attack, and hence also detects zero-day attacks.

In essence, Argos taints all data that is received externally and throws an alert when this data is used to manipulate the execution flow of an application. When an attack is detected Argos extracts the exploit code and stops the attack. The honeypot also offers the option to continue the attack, while recording all external inputs to gather more information about the attacker. Additionally, Argos can generate a signature for *intrusion detection systems* (IDS) to protect systems in a real network from the attack.

Argos successfully utilises the power of DTA, albeit the DTA is extremely slow. This is not as relevant to honeypots because they usually are not attacked very often. However, the high accuracy of the analysis is very beneficial because it allows Argos to virtually detect any attack.

2.4 Vulnerability Distribution

There have been previous attempts to enable automatic vulnerability distribution. In the following sections we will outline two previously suggested approaches and discuss their drawbacks.

2.4.1 Vigilante and Self-Certifying Alerts

Vigilante [9] is a proposed system to stop worms and viruses from spreading in an infected network. Worms like the Slammer worm infected 90% of all vulnerable hosts in the Internet within 10 minutes [26], which calls for an automatic solution to contain worms in networks. In essence, vigilante detects an attack, generates a fingerprint of the attack and sends it to all the other hosts on the network. Because these fingerprints are self-certifying, hosts do not need to trust each other.

Vigilante uses so called *self-certifying alerts* (SCAs) as mean to transfer information about an software vulnerability. SCAs are unique because they contain some parts of the exploit code itself. SCAs are generated when a host detects an attack by using DTA or *non-executable pages*. Other hosts then verify SCAs by executing modified parts of the attack code in virtual machine. If the exploit succeeds, the host sets a custom filter onto its network interfaces to protect against the worm. Costa et al. suggested that every host runs a mechanism to detect attacks. Thus, each host can detect a worm and in turn notify all other hosts about the worm by sending its exploit code [9].

Figure 2.1 shows the composition of an SCA. It specifies parameters such as the service name, alert type or verification information. Vigilante defines three alert types which cover most vulnerabilities:

Arbitrary Execution Control The exploit redirects the execution to a different point in the service's address space.

Arbitrary Code Execution The exploit injects code into the vulnerable service that is to be executed.

Arbitrary Function Argument The vulnerable service is instructed to call a system critical function, such as *exec* with a supplied argument.

The verification information of an SCA specifies where an SCA verifier has to modify the message to trigger the calling of the *verified function*, as described in Section 2.4.1.1. Consequently, it specifies the offset in a message, where the exploit executes either a system-critical function or its payload.

Service: Microsoft SQL Server 8.00.194
Alert type: Arbitrary Execution Control
Verification Information: Address offset 97 of message 0
Number messages: 1
Message: 0 to endpoint UDP:1434
Message data: 04,41,41,41,41,42,42,42,42,43,43,43,43,44,44,44,44,45,45,45,
45,46,46,46,46,47,47,47,47,48,48,48,48,49,49,49,49,4A,4A,4A,4A,4B,4B,4B,4B,
4C,4C,4C,4C,4D,4D,4D,4D,4E,4E,4E,4E,4F,4F,4F,4F,50,50,50,50,51,51,51,51,
52,52,52,52,53,53,53,53,54,54,54,54,55,55,55,55,56,56,56,56,57,57,57,57,58,58,
58,58,0A,10,11,61,EB,0E,41,42,43,44,45,46,01,70,AE,42,01,70,AE,42,......

Fig. 2.1: An example of Vigilante's SCA. The message data has been truncated and is originally 376-bytes long [9].

2.4.1.1 Verification

As depicted in Figure 2.2, Vigilante uses three components to verify an SCA: a *SCA Verifier*, a *Verification Manager* and a virtual machine. Using a binary rewriting tool, inside the virtual machine, all network-facing services are instrumented to load a new *verified* function into their address space. System critical functions, which are functions that are often used by attackers in their exploits, such as *exec*, trigger the pre-loaded *verified* function. The *verification manager*, a service inside the virtual machine, surveils all network-facing services and detects calls to the *verified* function.

When a new SCA is to be verified, the SCA is sent to the verification manager by the SCA verifier. Depending on the alert type, the manager then modifies the byte string at the offset of the message to call either the verified function or a system critical function. The message is then sent to the vulnerable service. If the verified function is called, the SCA passes; otherwise it fails after a certain timeout. The SCA verification manager informs the SCA verifier accordingly.

2.4.1.2 Generation

Vigilante provides two methods to detect attacks: non-executable pages and DTA. The latter is described in Section 2.2. While non-executable pages have a high false negative rate, DTA is much more accurate but also much slower. Given that almost no worms and viruses should remain undetected, DTA has to be chosen as a detection scheme. The trade-off between accuracy and performance is shown in Figure 2.3. DTA is at least 9x slower than non-executable pages in generating exploits.

In Vigilante DTA is implemented by using a binary rewriting tool at load time. When an attack is detected, an appropriate SCA is generated. Due to the memory information

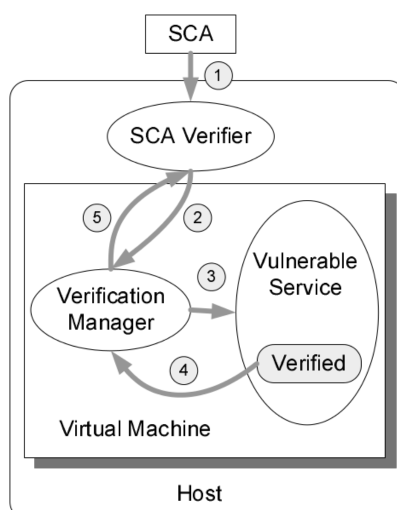

Fig. 2.2: SCA verification of Vigilante [9].

that are constantly maintained by DTA, an SCA candidate can be generated fast. The SCA generator then tries to verify the SCA candidate. It will pass for single-message attacks, but fail for multi-message requests. If the verification fails, Vigilante reassembles the multiple message of the attack and includes an increasingly long message in the SCA Candidate until it verifies.

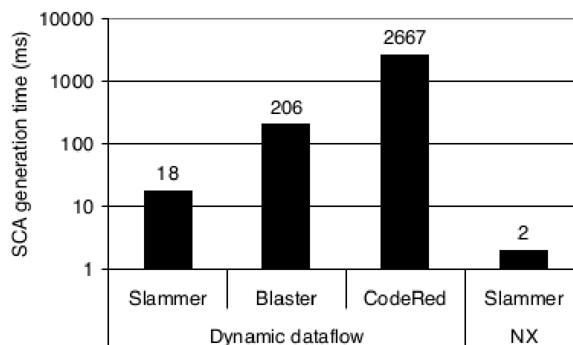

Fig. 2.3: Performance of the SCA signature generation; left: dynamic taint analysis; right: non-executable pages [9].

2.4.1.3 Performance

Although Costa *et. al* [9] documented the performance of SCA generation and verification, they did not measure the performance overhead that instrumented network-

facing services experienced while using DTA. As described in Section 2.2.1, DTA causes a significant overhead on the used services. The best performing system, to our knowledge, *Minemu* [5] still has an 1.5x-3x performance overhead compared to the performance of the original application. Taking into account that Minemu was developed almost five years after Vigilante, the performance overhead per service can only be speculated. Given that the information to generate an SCA can be gathered about as fast as for non-executable pages, Figure 2.3 hints that Vigilante incurs a performance overhead of 9x the original service.

2.4.1.4 Drawbacks

Vigilante was developed to contain worms and viruses in production network and as such is not suited for global distribution of SCAs. However, even with modifications of the system it has fundamental disadvantages:

Non Packet-Based SCAs – Vigilante modifies the original attack before decoding it as a SCA and verifies it with further modifications. Using these non-packet-based SCAs leads to a variety of disadvantages:

- Non-packet-based SCAs cannot be processed by other services because information about the original attack might be missing. As a consequence, custom vulnerability response process are not possible.
- Vigilante's verification VM can be compromised easily because messages are executed directly on the machine. Before executing the code of an SCA on the VM, Vigilante replaces arbitrary code with its own detection code. However, the position where the replacement is to be made (*offset*) is part of the SCA and can easily be replaced with a bogus offset. This increases the chance that Vigilante may be a target of a successful VM escape. Scargos can pre-filter uninteresting SCAs with an IDS and is further protected by using full-system DTA, which increases the required attack complexity.

Binary Instrumentation – Because Vigilante uses non-packet-based SCAs, it relies on binary instrumentation to verify and generate SCAs, which leads to multiple disadvantages:

- Vigilante relies on binary instrumentation of its network facing services for the usage of DTA. The drawback of this is that calls to other programs or kernel vulnerabilities cannot be detected because only the binaries of the network-

facing service are surveilled. In contrast, using DTA honeypots allows full-system protection.

- Binary instrumentation is OS-dependent, which requires that every Vigilante implementation would need to be modified for every possible operating system to be universal. On the contrary, DTA honeypots are based on VMs, which naturally support a vast number of operating systems.
- By using binary instrumentation, systems which intend to verify and generate SCAs have to run the network-facing service twice: once in the verification manager VM and once as a network-facing service. In contrast, Scargos' verification and generation of SCAs is done with the same resources.

Limited to Memory-Based Detection Techniques: Any detection mechanisms that Vigilante uses must be translated prior to use into binary instrumentation algorithms. This essentially limits the choices for Vigilante's detection engines to memory violation-detection techniques. In contrast, Scargos' usage of packet-based SCAs allow for virtually any detection engine. We decided to use DTA because it currently provides the best accuracy. However, packet-based SCAs could also be generated using technologies such as highly-accurate anomaly-based detection systems. This is not possible for Vigilante because network analysis tools can hardly be translated into binary instrumentation code.

2.4.2 Noah

The *network of affined honeypots* (NoAH) is a network of honeypots which occupies unused IP-addresses on the internet. Automated attacks which scan portions of the Internet's IP-addresses are detected by using Argos as a DTA honeypot. The results from Argos are then converted into IDS signatures and are intended to be used by third parties [19, 20]. NoAH is an interesting approach because it allows for detection of zero-day attacks by using DTA honeypots. However, the system has a significant drawback: IDS signatures cannot be verified and discard the original content of the attack. This disallows any alternative vulnerability response process. Furthermore, malicious IDS signatures could be injected into the system by compromising Argos, which can have serious ramifications as we discuss in Section 3.4.3. The possibility of such an attack is fair, taking into account (1) that we have found real-world attacks that have remained undetected by Argos, as shown in Section 6.1.1, and (2) that there have been vulnerabilities that have allowed a VM escape from QEMU, the VM used by Argos, as described in Section 2.1,

2.5 Malware Processing Tools

In the following sections, we discuss the two tools *CWSandbox* and *Honeycomb* that can process malware automatically for different purposes.

2.5.1 CWSandbox

CWSandbox is a tool that is developed for the Win32 platform to automatically generate malware analysis reports. The tool uses Windows application programmers' interface (API) hooking to monitor all system-level behavior of Windows. Although, malware can circumvent the Windows API, this is often unusual because attacks strive to use a very small payload. CWSandbox is operated as a virtual machine and after it has received malicious traffic, it generates a human-readable report about the intentions of the malware. As such CWSandbox notes in its reports among other events the creation, interaction and modification of files, windows registry entries, dynamic link libraris (DLLs), processes and network connections [34, 48].

2.5.2 Honeycomb

Honeycomb is a plug-in for the low-interaction honeypot *Honeyd* [33]. Using Honeyd, Honeycomb collects malicious traffic to automatically create IDS signatures of the received data. IDS signatures are created for *intrusion detection systems* (IDSs) such as *Snort* [36] by using protocol analysis and pattern detection algorithms on the received data. Connection tracking is an additional feature of Honeycomb that offers to aggregate signatures. In the algorithm, new malicious packets are compared to previously stored attack to generate compressed signatures instead of new ones [22].

2.6 Vulnerability Management

The following sections outline the most common ways to exchange information about vulnerabilities. This information exchange is essential for automatic vulnerability management because systems need to know, which vulnerabilities exists and what platforms and versions are affected.

2.6.1 Common Vulnerabilities and Exposures Identifier

Common Vulnerabilities and Exposures Identifier (CVEs) is a dictionary of publicly known software vulnerabilities. Before CVEs existed, different security tools and vendors

used different names and identifiers for the same or multiple vulnerabilities. This led to a global inconsistency and it became increasingly difficult to determine which tools cover which vulnerability. CVEs attempt to solve this problem by giving each distinct vulnerability a unique identifier. As a result, security vendors can refer to the CVE in their own software and this allows for interoperability between security products. Additionally, end-users can compare their security products based on their CVE coverage, and hence choose their most appropriate product [40].

2.6.2 Open Vulnerability Assessment Language

The *Open Vulnerability Assessment Language* (OVAL) provides a method to perform vulnerability testing on information systems. The language consists of multiple parts. There is the OVAL XML language, which describes a specific configuration state of a system; the OVAL Schema, which collects and reports configurations; and the OVAL content tests for vulnerabilities or configuration issues. However, the OVAL definitions are most relevant to vulnerability management because they are involved in each step of the process. Generally, vulnerability management is roughly divided into three stages:

1. A software vulnerability is discovered and published by a software vendor.
2. The existence of vulnerability is verified by end-users in their systems.
3. The vulnerability is patched by the end-user.

OVAL definitions can act as a central point during this process. An OVAL definition is created when a new vulnerability is discovered. This new OVAL Definition is usually created by the affected software vendor after checks have been made to check which software versions and platforms are affected. The definition itself usually contains the following items:

CVE A CVE is included as a global reference.

Description A description states which security concerns the vulnerability raises and in some cases the cause of the vulnerability.

Affected Software It is stated what software products and platforms are affected by the vulnerability.

Vulnerability Tests A series of tests are included to verify the existence of the vulnerability. Usually, these tests check whether the affected software version is installed.

With the release of an OVAL definition, end-users can easily verify whether their machines are affected. If so, they can leverage patch management utilities to update their software or deactivate the component if the involved risks are too high [42].

The OVAL offers a great tool to facilitate software vendors and end-users in terms of security vulnerabilities. However, the process of publishing an OVAL is often too long to effectively protect systems. Zero-day exploits can spread in just a few minutes, such as the Slammer worm which infected 90% of all vulnerable hosts in the Internet within 10 minutes [26]. On the other hand, it can take weeks until an appropriate OVAL definition is published. Consequently, systems are in an unprotected state without end-users knowing about any threat.

2.6.3 Open Vulnerability Assessment System and Network Vulnerability Tests

The *Open Vulnerability Assessment System* (OpenVAS) is one of the most used open-source vulnerability scanners [43]. In a similar way to the OVAL system, the OpenVAS system checks for vulnerabilities by checking whether vulnerable software versions are installed on the system. The checks that are used to verify this are called *Network Vulnerability Tests* (NVTs). NVTs are licensed under the GNU GPLv2+ license which allows free distribution and usage. Unfortunately, the process of publishing a NVT is not well documented. Although, the developers of OpenVAS state that vulnerabilities can be submitted without the existence of an CVE, it remains unclear how bogus or even malicious submissions are prevented [29, 30].

Chapter 3

Architecture

The Scargos architecture is an approach to detect, distribute and verify the existence of previously unknown network-based attacks, conventionally called *zero-day attacks*. By combining dynamic taint analysis (DTA) and honeypots with traffic replaying, zero-day attacks can be distributed and verified much faster than non-automatic approaches. Figure 3.1 shows the relationship between all of Scargos' components. A prerequisite for Scargos' process is the detection of an attack; the process continues with SCA publishing, distribution and verification. Vulnerability response is optional and depends on the need of the end user.

By using DTA, described in Section 2.3.2, the detection component detects previously unknown network-based attacks. The goal of the detection component is not necessarily to gather as many threats as possible but rather to gather the most recently developed attacks. After a threat has been detected, all attack information is decoded into a custom format: *packet-based self-certifying alerts*.

New SCAs are submitted and aggregated in SCA repositories. The SCA repository stores self-certifying alerts and distributes them. User groups, which are interested in receiving the newest attacks can subscribe to SCA repositories. For every unique payload that the repository receives, subscribers receive a notification.

Subscribers will usually verify a SCA before it is further processed to initiate an individual vulnerability response process, given the SCA is valid. SCA verification is done by leveraging the same methods that were used for SCA generation: DTA. What happens after a SCA has been verified depends on the needs of the user group. There are different user groups which may benefit from using Scargos:

Researchers: can study new threats to complement their vulnerability research for academia, security vendors, ISPs, institutes or corporations.

Incident Response Teams: need to be aware of the newest attacks in order to take appropriate action.

IT-security Processes: Scargos can be operated in a fully-automatic fashion. As such, Scargos can be part of a larger IT-Process to automate handling of new vulnerabilities in a predefined manner.

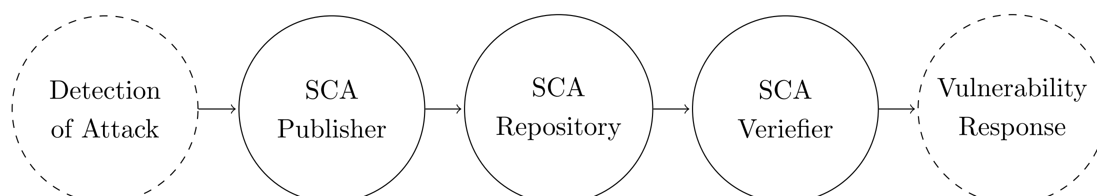

Fig. 3.1: Scargos' life cycle starts with the detection of an attack and is followed by SCA publishing, distribution, verification and optionally ends with vulnerability response. Detection is usually handled by a DTA honeypot and handling of vulnerability response depends on the needs of the end user.

3.1 Detection and SCA Publishing

The goal of SCA publishers is to automatically publish preferably zero-day exploits of attacks on network-facing services. This assumes that a large and likely volatile pool of SCA publishers exists. Scargos is designed to encourage a high number of participants by allowing anonymous submission of SCAs.

Our system makes no assumptions as to how SCAs are gathered and published, but we propose the use of honeypots as described in Section 2.3. To automate the process of instantly publishing new attacks, DTA honeypots can be leveraged, as described in Section 2.3.2, as this allows for the automation of SCA generation. As soon as the DTA engine of a DTA honeypot detects an attack, an alert is thrown and detailed information is supplied. The SCA publisher can use this information to craft a unique SCA for submission to a SCA Repository.

Our solution encourages usage in corporations because honeypots which utilise the production environment posses more value than pure research honeypots that are used in the wild. Ultimately, in production networks the likelihood of detecting zero-day exploits in a timely manner is increased by receiving more:

Traffic: A percentage of traffic received by production networks is naturally malicious. Increasing traffic for research honeypots often requires further advertising such as the usage of search engines [6, 24].

Sophisticated Attacks: A production system naturally holds information assets that it needs to protect from disclosure, which makes it more attractive for targeted attacks, and thus leads to more sophisticated attacks.

We propose different ways to integrate honeypots into production systems by using:

Unused Resources: Unoccupied IP-addresses could be replaced by redirections to honeypots. This concept is well established and has been realised by honeypots such as Honeyd or LaBrea [25, 33, 34]. Likewise, unoccupied ports of production servers could be used to forward requests on these ports to a special honeypot.

Malicious Packet Forwarding: Production networks which are protected by an anomaly-based IPS can forward all traffic that is to be discarded to the DTA honeypot. There is a high probability that this traffic is malicious. A similar approach in a different context has been proposed by shadow honeypots [1].

It should be noted that honeypots used in production environments should lie outside of the production network and should be encapsulated by a reverse firewall.

3.2 Packet-Based Self-Certifying Alerts

While Vigilante's SCAs [9], described in Section 2.4.1, use a custom format to generate and verify themselves by using binary instrumentation, we propose packet-based SCAs to preserve attacks in their original format.

Packet-based SCAs store the transport-protocol payload of all packets of an attack in binary format. Each transport-protocol payload that has been made during the conversation which lead to the compromise is separated as an element in a list. As such, packet-based SCAs lead to many simplifications when compared to conventional SCAs as shown in Table 3.1.

Packet-based SCAs allow Scargos to generate and verify SCAs with the same machine. This has many advantages such as the fast accumulation of all affected versions in an SCA repository and a greater number of possible contributors to vulnerability repositories.

Also, as described in Section 3.5.2.2, packet-based SCAs can be easily processed by many third-party applications such as for IDS signature generation because the exploit's content remains preserved. Likewise, as described in Section 3.5.1 and 3.5.2.1, researchers

or incident response teams can more easily analyse attacks as part of their vulnerability response.

	Vigilante's SCAs	*Packet-Based SCAs*
Generation	Each network-facing service of the surveilled system is binary-instrumented separately to allow verification or generation of SCAs.	The entire system is constantly surveilled using system-wide DTA for verification and generation of SCAs.
Verification	Vigilante modifies the original payloads that led to the compromise to trigger an injected verified function as a sign of validity.	Packet-based SCAs preserve original payloads of an attack and determine validity by replaying it to a verifier.
SCA Types	Verification of SCAs differs depending on which kind of SCA-type was generated (Arbitrary Execution Control, Arbitrary Code Execution, Arbitrary Function Argument).	All SCAs are generated and verified in the same manner. There is no need for different SCA types.
Parameters	SCAs, as shown in Figure 2.1, depend on a variety of parameters to verify an SCA, such as *offset* or *alert-type*.	Only the transport-protocol payload of the attack needs to be present to determine validity of a SCA.
File Format	SCAs are present in a custom format. They can only be processed by other custom applications.	SCAs can be converted easily into many common formats such as pcap-files, IDS signatures or malware analysis reports as described in Section 3.5.

Table 3.1: Comparison of Vigilante's SCAs (*left*) with Packet-Based SCAs (*right*).

3.3 Distribution: SCA Repository

Similar to software repositories which store software, SCA repositories store SCAs for different applications. These repositories have different channels. Each channel has a

unique ID which corresponds to a specific application. SCAs can be submitted and downloaded from the SCA repository, but each SCA is uniquely tied to an application.

A unique aspect of SCA repositories is that no assumptions are made about the validity of SCAs. While their might be an upper limit as to how many SCAs a user can submit and download before the connection is aborted or slowed down, the repository imposes no restriction on who can submit or download SCAs. It is to be expected that anonymous submissions invite attackers to submit malicious SCAs. However, given that the SCA verifiers are sufficiently isolated, SCAs can only pass or fail, and compromises should be sufficiently hard, as described in Section 3.4.3.

Because SCA verifiers may also be SCA publishers, this can create a feedback loop. We advise that SCA repositories detect duplicates of SCAs with the same payload and thus inform SCA verifiers only if a SCA with a unique payload was published. However, SCA repositories should aggregate other information about an SCA duplicate such as the version of the application: A newly published SCA specifies a particular version; resubmissions of the same SCA with a different specified version can thus be easily combined. Using this method, within an instant all affected versions can be populated. Although, this might include false positives. Given that the pool of SCA publishers and verifiers is large enough to cover all common versions of an application, very few false negatives should exists.

3.4 SCA Verification

Scargos can be used in different ways by different user groups. However, in most cases, it is only desirable to receive a subset of newly submitted SCAs for the applications that are of interest. After all, it is necessary that a DTA honeypot is used for verification, which can only run a limited number of network-facing services. While some user groups may arbitrarily select the applications that SCAs are wanted for, others may want to receive SCAs to protect used systems from being compromised by previously unknown vulnerabilities. We call hosts that are protected this way *Permanently Audited Machines* (PAMs).

SCA verification uses similar components as Vigilante [9]: a *SCA verifier*, a *Verification Manager* and a virtual host that we call a *PAM Mirror*.

PAM mirror: The PAM mirror is located inside a DTA honeypot, which effectively is a VMM that constantly surveils its VMs by using DTA, see Section 2.3.2. The PAM Mirror is configured to closely mirrors the configuration of the PAM with its installed services and operating system, which is described in more detail in

Section 3.4.1. Similar to the SCA Publisher, described in Section 3.1, the PAM mirror detects attacks by throwing an alert every time external input manipulates the execution flow of a program.

Verification Manager: The verification manager is an application installed on the host OS which runs the PAM mirror VM. It is responsible for interpreting alerts of the PAM mirror to decide about the validity of a SCA. For every verified SCA, the verification manager initiates a custom vulnerability response process, as described in Section 3.5.

SCA verifier: The purpose of the SCA verifier is to gather the most recent vulnerabilities and to replay them to the network-facing services of the PAM Mirror. The SCA verifier subscribes to at least one SCA Repository and maintains a *PAM List* which contains the IDs of all installed network-facing services on the PAM/PAM Mirror. Each SCA repository sends a notification containing an application ID to all subscribers as soon as a new SCA with a unique payload is submitted. The SCA verifier replays the payload to the PAM Mirror whenever the application ID matches one of the IDs in its PAM list.

In contrast to Vigilante, our architecture uses packet-based SCAs as described in Section 3.2 and thus we are not surveilling a particular network-facing service inside a virtual machine, but verify a specific SCA system-wide. We accomplish this by hosting the PAM Mirror inside a DTA honeypot.

After the PAM mirror has been initially set up, its state is being saved in a *default snapshot*. The PAM Mirror runs in a DTA virtual machine and as such it notifies the host OS (verification manager) as soon as the system becomes compromised. This way, the verification manager can verify SCAs directly by waiting for a notification, or the verification fails after a certain timeout. For every TCP stream that is received by the verification manager, the verification manager destroys the PAM Mirror and reinitiates its state from the default snapshot. Given that a SCA is verified, a vulnerability response process is initiated.

3.4.1 Common Configuration

Scargos is designed to replay attacks that were successful on one system on another similarly-configured system. In order to ensure that replaying is successful for the vast majority of attacks, it is advisable that the DTA honeypots on the SCA verifier and publisher are configured very similarly.

SCA publishers can take a number of steps to increase the likelihood that an SCA verifies. Although, the application's configuration can be set in a arbitrary fashion by SCA publishers, we advise that for any resource that an application uses, the default, preconfigured resource is to be used. We define application resources as any additional files or static content that a network-facing service serves, such as:

HTML Files The HTML files that a webserver serves clients which make HTTP requests. Web servers are often bundled with a set of default web pages.

Databases The databases a database server offers to query. Often, after a database installation, there are default databases preconfigured.

FTP Files FTP servers usually specify a default path in which the served files are located.

While the SCA verifier's PAM mirror should use the configuration of the PAM (or any other arbitrary one that is of interest to the verifier), we advise that the default application resources should be used instead of copying any resources from PAM. This has two advantages:

- When the PAM mirror is compromised, no production information is leaked, such as source code of websites or sensitive databases.
- The likelihood that an SCA verifies when the application is in fact vulnerable is increased, when the SCA publisher followed our guidelines.

Additionally, we advise both the SCA publisher and SCA verifier to run all applications on their official standard ports as specified by the *Internet Assigned Numbers Authority* (IANA). If the application does not fall in any of the IANA specified uses, the preconfigured default port should be used. Because some applications occupy multiple ports, this is necessary to identify the affected component of a network facing service. Moreover, in Section 5.3.1 we propose an alternative implementation for package replay which relies on the usage of standard ports.

Lastly, some services require that accounts or host names be set up. we advise that all SCA verifiers and publishers use a default username/hostname such as *anonymous* and a default password such as *An0nym0uS*, so that attacks which in some way process this data, can successfully be replayed.

3.4.2 Placement of the SCA Verifier

There are different possibilities as to how the individual verification components are relatively placed with varying benefits and drawbacks. While the PAM Mirror always is the guest OS of the verification manager, there exists three possibilities as to how the SCA verifier is placed:

Repository Placement: The SCA verifier is placed inside the SCA repository and the verification manager maintains the PAM list and performs *requests for attacks*. In this placement, the SCA repository performs the replaying of SCA payloads and the verification manager issues requests to be attacked. Because all network-facing services are contained in the PAM mirror, the attack is eventually received only by the PAM mirror, while the verification manager decides about the validity of the attack. Although, this placement allows for increased security of the verification components (since the SCA verifier can only be compromised on the SCA repository), it increases the risk for the repository and raises ethical questions. Specially crafted payloads could be stored on the SCA repository and misused to infiltrate innocent hosts or distribute malicious commands to malicious structures such as bot nets.

Network Placement: The SCA verifier is being placed between the SCA repository and the verification manager. A firewall is configured so that the SCA verifier is only allowed to connect to the SCA repository and PAM Mirror. This leads to high performance as the SCA verifier is an individual host. For increased security, the SCA verifier needs to be placed in a new VM and reinstantiated for every new SCA that is to be verified as described in Secion3.4.3.

PAM mirror Placement: The SCA verifier is placed inside the PAM Mirror. The contents of an SCA is replayed to the loopback interface of the PAM Mirror. All network-facing services of the PAM Mirror can be kept local or made accessible for the outside world, if the PAM mirror is also used as a honeypot/SCA publisher. Because the SCA verifier is located inside the PAM mirror, attacks against the SCA verifier are detectable. Yet performance may be crippled because DTA is constantly operating inside the PAM mirror.

In our implementation and experimental evaluation, we use the network placement.

3.4.3 Security Considerations

Every output from the SCA repository has to be regarded as malicious because an anonymous user can submit any data to the SCA repository. The main attack vectors involve targeting the DTA honeypot/PAM mirror or the SCA verifier.

The SCA verifier processes the SCA from the SCA repository and replays it. Although, this includes only a handful of functions likely to be implemented by using defensive programming, exploitation may not be impossible. However, given that a SCA verifier is being compromised that uses our described network placement, this only leads to a denial-of-service (DoS) of the SCA replaying because firewalls should be configured to only allow connections to SCA repositories or the DTA honeypot. Furthermore, a DoS can be prevented by running the SCA verifier in a VM and by reinstantiating it for every new SCA that is being processed. In a SCA verifier that uses our described PAM mirror (DTA honeypot) placement, this is already accomplished.

A second attack vector can be the DTA honeypot. A SCA might be carefully crafted such that the SCA verifier replays a special message to its DTA honeypot, which eventually leads to a VM escape. To be able to exploit the DTA honeypot, an exploit must successfully fulfill all of the following requirements:

1. One of the network-facing services must be successfully exploited in the DTA honeypot.
2. The exploit is not detected by the DTA engine.
3. The exploit can compromise the VMM and escape to its host OS.
4. All tasks have to be accomplished before the verification manager times out and terminates the VM.
5. The entire attack must be crafted into one SCA.

Taking into account that such a chain of attacks has not been demonstrated yet, the time such an attack may take remains unknown, but theoretically such an attack is possible. Yet, it should be noted, that VM escape vulnerabilities and attacks remain scarce as described in Section 2.1 and that many security applications depend on the security of VMs. Additionally, if the DTA honeypot is only operated as a research tool as part of semi-automatic vulnerability response as described in Section 3.5.1, it is unlikely that such an investment would be made by an attacker. Likewise, the consequences of a compromise may not be very harsh because when Scargos is used for semi-automatic processing, there should be no connection to any production environment.

However, we have to apply a higher notion of security when Scargos is operated in a fully-automatic fashion as described in Section 3.5.2. When Scargos operates in this mode, production systems may gather and process data created by the verification manager for verified SCAs, which increases the attractiveness of attacking the SCA verification process. Thus, the goal of Scargos has to bethat the effort required to successfully compromise the DTA honeypot should be higher than that to successfully exploit the actual, well-secured PAM.

We can increase the security by improving on its most likely attack vector. The DTA honeypot can be secured by having all versions of the network-facing services up-to-date and protected by the same IDS as the PAM. We also propose to run the DTA honeypot inside another VM from another vendor to add an additional layer of security. This intermediate VM could notify the host OS about the validity of an SCA through a covert channel to further increase security. Such a channel could be used by the intermediate VM to initialise an immediate shutdown or send a ping to the host OS. This can be easily detected by the verification manager. By using this set up, we further increase the effort an attacker has to make to compromise the verification manager:

1. The SCA must be a zero-day exploit/near zero-day exploit, as the network-facing services and IDS signatures are up-to-date.
2. The exploit is not being detected by the DTA engine.
3. The exploit compromises the VMM and escapes to the intermediate VM.
4. The exploit escapes from the intermediate VM.
5. All tasks are accomplished by one single exploit before the verification manager times out and terminates the intermediate VM.
6. The entire attack must be crafted into one SCA.

While an attack may still be theoretically possible, in most cases it should require more effort than to compromise the actual PAM because two consecutive VM escapes have neither been demonstrated and are highly unlikely, given that only a handful of suitable VM vulnerabilities have been found for mostly older VMMs.

3.5 Vulnerability Response

In this section, we want to outline which actions a user group might take after a SCA has been successfully verified. Vulnerability response falls into two categories: semi-automatic and full-automatic.

3.5.1 Semi-Automatic Vulnerability Response

Using a semi-automatic approach, new vulnerabilities are detected automatically, but further processing requires human-interaction. This is particularly interesting for researchers to investigate the cause of the vulnerability and for incident response teams in corporations in order to take the appropriate action manually.

A semi-automatic approach can be realised in many ways. The most straightforward approach is to let the SCA verification manager store new verified vulnerabilities in a log file which can then be investigated. Remote access could be realised using a log management tool or protocols such as SSH.

In most cases, faster access to the newest vulnerabilities is desirable. Either push-notification could be sent to incident response teams as soon as a new vulnerability has been discovered or automatic polling can be leveraged. By reusing the previously mentioned components, the SCA repository and SCA publisher from Section 3.1 and Section 3.4, the verification manager could republish a SCA to a locally running an SCA repository. Researchers or incident response teams could be notified from the SCA repository using its native push-server or constantly poll for changes.

3.5.2 Full-Automatic Vulnerability Response

Using a semi-automatic approach, new vulnerabilities are detected and processed automatically into desired formats such as IDS signatures or malware analysis reports. This enables faster response time as a semi-automatic approach, but in some cases bears a higher security risk. The fully-automatic approach uses the gathered data of a verified vulnerability and processes this data in the desired fashion. We propose two use cases: automatic malware analysis and IDS signature generation.

3.5.2.1 Automatic Malware Analysis

In Section 2.5.1 we presented a tool that enables automatic malware analysis. Because our proposed packet-based SCAs preserve the actual payloads of an exploit, a combination of Scargos with CWSandbox enables us to get a detailed analysis of the most

recent worms and viruses. Because Scargos uses packet-based SCAs, the communication between Scargos and CWSandbox works natively. Scargos' verification manager replays an SCA as it is implemented in the SCA verifier to a locally running CWSandbox. Subsequently, a detailed report about the malware is generated, which can then be further analysed.

3.5.2.2 IDS Signature Creation

Automatically generating an IDS signature for previously unknown attacks might be one of the most compelling attributes of Scargos for production networks. The tool Honeycomb 2.5.2 generates and aggregates IDS signatures when it receives malicious traffic. Verified packet-based SCAs can be authentically replayed to honeycomb to generate and aggregate IDS signatures for every new verified SCA. These signatures can then be signed and published by the verification manager. IDSs or IPSs can incorporate these signatures automatically by using polling or push notifications as described in Section 3.5.1.

Although this approach may be very useful for institutions and corporations (it significantly shortens the timeframe where a zero-day attack remains undetected), it also bears greater risks. Given that our proposed system can be compromised as described in Section 3.4.3, risks and benefits fall into two categories as shown in Table 3.2, depending on whether the IDS exclusively has detection capabilities or whether it can also be used as an IPS.

	Benefits	*Risks*
IDS	An IDS alert is thrown for every attack which has been verified as a SCA.	An attacker may be able to compromise the IDS given that a vulnerability in the IDS signature processing can be found. Additionally, an attacker may increase the amount of false positives by inserting bogus signatures.
IPS	Every attack which has been verified as an SCA is being prevented by similar packets.	All the risks of an IDS also apply for an IPS. Moreover, an IPS might enable an attacker to *drop* valid traffic which effectively would causes a denial-of-service.

Table 3.2: Benefits and Risks of using either an IDS or IPS for fully-automatic vulnerability response.

Chapter 4

Replay Mechanisms

In contrast to Vigilante, Scargos uses packet-based SCAs, which preserve the original attack. This requires us to use new approaches to verify an SCA. We propose that verification of an attack is done in the same manner as that in which the original attack compromised the system. Using this approach we allow for a custom vulnerability response process by end users and any attack detection engine can be used with Scargos.

Attacks on network-facing services are usually conducted by sending a series of malicious packets. We present two replay mechanisms as a way to imitate attackers: brute-force replay and exact stream replay. We base our approach of successful imitation on Assumption 1 and later show its validity in Section 6.1.

Assumption 1 *Given two identical machines are in the same internal state, sending the same input to both machines will transfer both machines into the same state.*

It seems assumption 1 is hardly practical because two computers can rarely be identical, when we for example take the many possible hardware configurations into account. Yet, more often, malicious attacks are automated and not developed for one specific machine; instead, attackers aim to increase the likelihood of a successful compromise; thus, attacks on network-facing services are more likely to be tailored to a particular application, version and operating system. Also, taking into account how pervasively worms and viruses have spread in the past, while using the same attack on every system, it seems that trivial replaying of malicious traffic is often enough to compromise a system that has the same vulnerability.

When we look at network services in an abstract manner as event-driven systems, we can categorise them into two types:

Stateless Protocols: In a stateless protocol, the server always remains in the same state during the conversation. On an abstract level, protocols using SSL/TLS connection only effect the transport of the protocol, and encrypted stateless protocols can for our purposes still be viewed as stateless.

Stateful Protocols: In a stateful protocol, the server remembers the state it is in during a conversation and processes client inputs accordingly.

We are particularly interested in stateful network-facing services as they expect different inputs depending on their state; likewise, clients have to remember the state of the session to issue valid requests. In the following discussion we call these states *input states* because it is the next input that the application expects; otherwise the input is discarded instead of being processed. Figure 4.1 shows the various input states of the FTP protocol. After a connection has been established, the protocol expects to receive a *USER* command with a valid username. Subsequently, the password is expected before the user is either logged in or further account information is requested. If for example a *LIST* command were issued by the client as the first command, the server would discard the input instead of processing it because a *LIST* command can only be issued after a successful login.

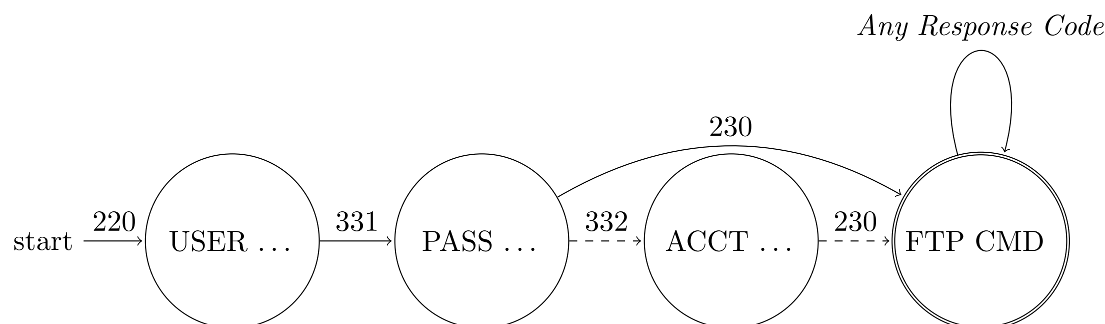

Fig. 4.1: Different states an FTP client transitions into (*circles*) according to the responses of a FTP server (*arrows*). The command *ACCT* is optional. *FTP CMD* represents any valid FTP command after successful login.

Taking this information into account is vital because, as a result, there exists to the best of our knowledge only two valid ways to replay an attack:

Brute-Force Replay: Given the single layer-5 PDU which led to the compromise of one host, a second host that has the same vulnerability can be compromised by replaying the single layer-5 PDU to each input state of the vulnerable service.

Exact Stream Replay: Given an entire layer-5 PDU conversation, which led to the compromise of one host, a second host that has the same vulnerability can be compromised by replaying the entire conversation as it was previously recorded.

Both of our approaches transfer the network service into the required state before sending the compromising packet. This is necessary because packets which arrive in the wrong input state are discarded.

4.1 Brute-Force Replay

In brute-force replay, a malicious layer-5 PDU is sent to all possible input states of an application. The basis of brute-force replay is formulated in Assumption 2.

Assumption 2 *When a system has multiple states and a particular input only succeeds in one particular state, then sending the input to all possible states ensures that it succeeds at least once.*

Brute-force replay requires that all input states and the actions that are required to transfer an application into a particular state are previously known. We call this set of information a *protocol handler*.

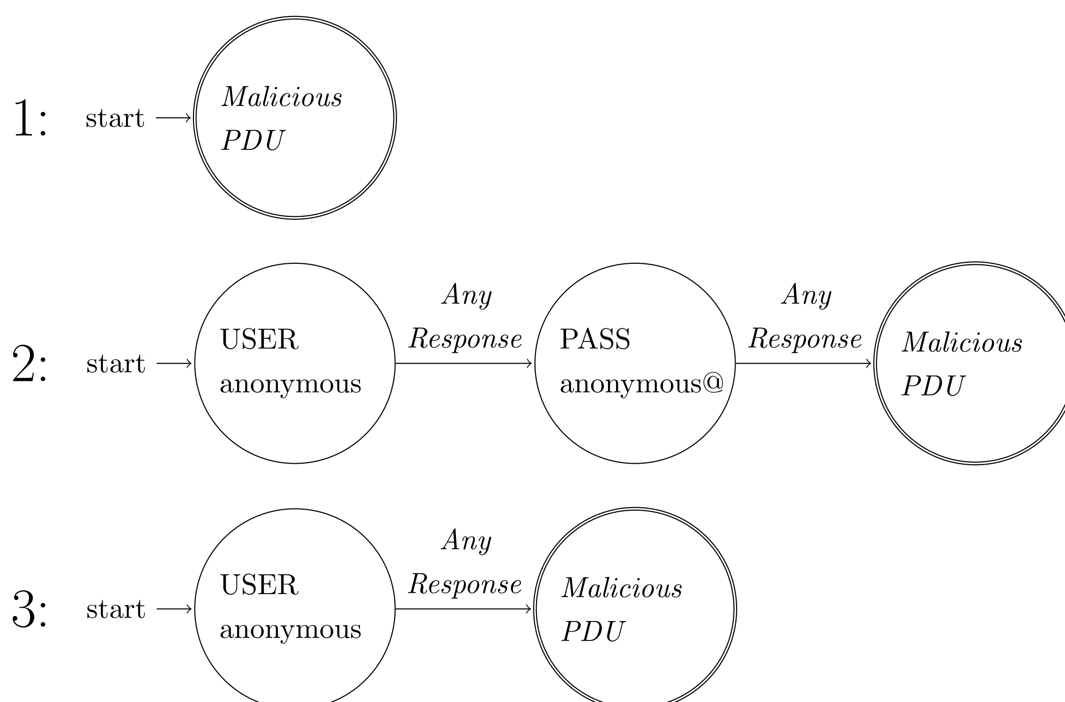

Fig. 4.2: Brute-force replays an attack to all input states of the FTP protocol in order of likelihood.

Figure 4.2 shows an example of a protocol handler for FTP. As we have shown before, the FTP protocol has four input states. By sending the malicious PDU as shown in Figure 4.2, we reach all four states and a vulnerable system should become compromised.

4.2 Exact Stream Replay

In contrast to brute-force replay, exact stream replay does not need to know which protocol is being used. Instead, the entire conversation which led to the compromise of a SCA publisher is simply replayed by SCA verifiers. However to ensure that the attack remains successful, we need to follow the guidelines mentioned in Section 3.4.1.

Exact stream replay is protocol-independent. One of the challenges this poses is that we do not know if or when a network-facing service will send an answer to a PDU that we have previously sent. An example of this is illustrated in Figure 4.1. A FTP service will usually greet a newly connected client with a `220` Response Code and a custom welcome message. The FTP service may discard any other commands sent by the client such as `USER`, up until this message is successfully delivered. On the other hand, other services

do not send a welcome message e.g. HTTP. Consequently, we developed Algorithm 1 to guarantee correct delivery of PDUs, while ensuring that all packets from the server are received.

Algorithm 1 shows that we wait for a welcome message and then proceed to wait for an answer for each layer-5 PDU that we send. However, regardless of whether we have received an answer or not, after 100ms we proceed with sending the next PDU. We felt that 100ms is a very generous timeframe because we expect that even slow services would respond within this time, especially taking into account the fact that the verifier and the vulnerable host can be arranged close enough to attain a very low latency.

Algorithm 1: Exact Stream Replay

Data: L as an ordered list layer-5 PDUs; $Host$ as the IP and port of a host
Result: Each packet from L is replayed to $Host$

```
    begin
1       socket = establish_connection(Host)
2       receive_data(socket)
3       foreach element p of L do
4           send_data(socket, p)
5           receive_data(socket)
6       close_connection(Host)
7       return

    Procedure receive_data(socket)
8       Set timer to 100ms
9       start to countdown timer
10      while timer ≠ 0 do  receive from socket
11      return
```

4.3 Limitations

There are limitations to our suggested replay mechanisms, in which attacks cannot be replayed or a valid SCA fails to succeed.

The most obvious limitation is when the SCA publisher uses a different operating system, application version or configuration than the SCA verifier. However, because the SCA verifier should use a DTA honeypot which closely mirrors a permanently audited

machine (PAM mirror), SCAs that do not verify should not jeopardise the system's security. Hence, the SCA only fails to verify when system is not vulnerable to the attack. This is the desired outcome.

Secondly, SCAs may not verify because the SCA publisher or verifier has not been configured as described in Section 3.4.1.

There also exists other types of possibly non-verifiable SCAs. Some of these can be addressed, while others cannot:

IP-address/URL: Each host is assigned an IP-address and optionally a URL. Some protocols such as HTTP require that a `HOST` field be specified, which contains the IP or URL of the server. This has privacy implications and could disrupt correct replaying because an IP or URL is unique to a host. However, given all other parameters are configured generically, as mentioned ins Section 3.4.1, the only remaining unique host-specific values are its network addresses: IP, URL and MAC.

We suggest that SCA publishers identify all network addresses of their honeypots in their unpublished SCAs. Every occurrence can then be replaced with a universally-known unique string. After the SCA has been published and is received by a SCA verifier, we search for the universally known unique string and replace it with the IP-address or URL of the PAM mirror/DTA honeypot.

Run-time generated values: Some applications may generate unique or random values for their application resources. An attack could be based on first extracting the unique value and then using this value as part of an attack. Although such an attack may exist in a proprietary protocol, we assume they are rare as we have not found such an attack.

Using previously created malicious resources: There are attacks which may involve two separate requests to the honeypot in order to launch an attack. In the first request, data is submitted to the honeypot to create a new resource. In a second request the new resource is then used to compromise the honeypot. Such a multistage attack could for example include first creating a user account before using it in a second request to compromise the system.

We can not effectively generate SCAs for these attacks as we only extract the stream which led to the compromise because an attacker can wait an arbitrary amount of time between the two requests and can easily change identities by using forged IP- and MAC-addresses. This is not a Scargos-specific problem, but rather

an open problem since Vigilante [9] is also not able to generate a universally-verifiable SCA for such an attack.

However, attacks that rely on creating a malicious resource beforehand and then using the resource to compromise the server are rare. If at all, they are most likely encountered as part of a targeted attack. In fact, none of the vulnerabilities that we have found, including all of the most recent high profile attacks such as the worms *Slammer*, *Blaster*, *Sasser* or *Lsass* use this technique.

This type of attack should not be confused with common staged attacks that utilise techniques such as *egg-hunting*. With this technique a service is compromised twice to place long arbitrary code into the machine's memory, which would not otherwise fit into one request. Egg-hunting is detected and can easily be verified because the execution flow is manipulated in both requests.

Apart from the mentioned limitations, brute-force replay may have further limitations, compared to exact stream replay. First, SCAs cannot be verified if no protocol handler is present. Second, SCAs cannot be verified if the number of input states of the used protocol is too high. Finally, attacks which consist of multiple layer-5 PDU cannot be verified except when instead of single packet extraction, stream extraction is used to generate the SCA, as described in Section 5.2.3. However, this would deteriorate many of the performance advantages that brute-force replay has compared to exact-stream replay.

4.4 Summary

Packet-based SCAs preserve the original content of an attack, which requires us to use new approaches for verification mechanisms. In this chapter, we presented two replay mechanisms to verify packet-based SCAs: brute-force replay and exact stream replay. Verification is achieved by replaying the attack to a DTA honeypot and thus imitates the original attack.

Brute-force replay requires that one packet be present, which eventually leads to the compromise of a service. This packet is then sent to every possible input state of the network-facing service. Thus, one of the input states that the packet is replayed to must be the state in which the original service was compromised.

Exact stream replay requires that the entire stream of packets that led to the compromise be present. Exact stream replay is protocol-independent and replays packets sequentially as they have been decoded in the SCA.

Both replay mechanisms have limitations. They both require that the host which created the SCA and the host to which the SCA is replayed on be configured in the same way. Furthermore, there are edge cases, in which SCAs can be created but not verified automatically, such as when a service uses randomly generated resources or when a service was compromised by a certain targeted attack.

Chapter 5

Implementation

In this chapter we present how we implemented the three components of Scargos: SCA publishing, SCA verification and the SCA repository. We presented two replay mechanisms: *brute-force replay* in Section 4.1 and *exact stream replay* in Section 4.2. Both approaches operate quite differently and expect different prerequisites. In order to investigate both approaches, a different implementation was required for each suggested approach as depicted in Figure 5.1. On the one hand, exact stream replay relies on successful SCA generation of stream extraction, which in turn relies on the successful outcome of either *jump target memory block search*, *compromised memory block search* or *exploit Ethernet frame search*. On the other hand, brute-force replay relies on single packet extraction which requires that an exploit Ethernet frame be successfully extracted from the logs. In the following, sections we will outline how we implemented each of the components.

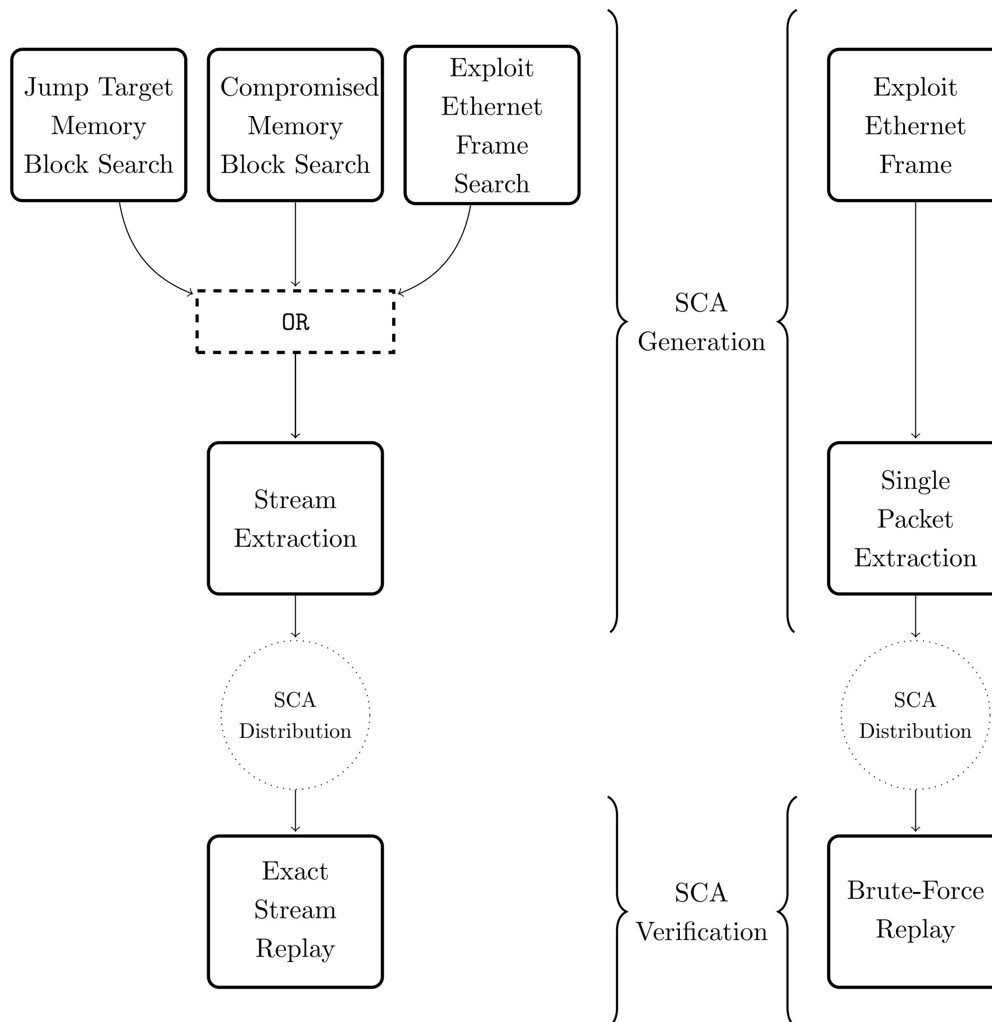

Fig. 5.1: Different processes are required to enable exact stream replay (*left*) and brute-force replay (*right*).

5.1 SCA Repository

An SCA repository is responsible for receiving, storing, presenting and notifying about SCAs. In our implementation we used *Django 1.5.2* with the extension *Django REST Framework 2.3.6*. By combining these programs we created a RESTful web service [13] that offers its data in the JSON-format. Web services are well suited for distributing and receiving data and also, RESTful web services provide high flexibility for further

processing, since virtually any programming language can process HTTP. Additionally, an SCA repository can be operated using HTTPS and thus protect SCA verifiers from being eavesdropped. By using JSON, JavaScript's native object notation, the data of a SCA repository is in a human-readable format and can easily be integrated into external JavaScript applications. This facilitates using our data for other purposes such as a super SCA repository which aggregates from all known SCA repositories. SCA repositories offer the following functions:

Receiving and submitting application entrys: The SCA respository assigns each application an ID. This ID has to be referenced in SCAs to refer to a specific applications. New application IDs can be assigned via a HTTP request immediately if SCA publishers or verifiers have applications installed, which do not yet have an application ID.

Receiving and submitting SCAs: The SCA repository receives and provides SCAs. Like application IDs, each SCA is assigned a unique ID. The process and composition of SCAs is discussed in more detail in Section 5.1.1.

Sending a push-notification for every unique SCA: The SCA repository sends a *push-notification* to all subscribers for every newly created SCA. Push-notifications are described in more detail in Section 5.1.2

Aggregating versions: SCA repositories need to be able to detect duplicates. A duplicate SCA is defined as an SCA which contains the same layer-5 PDU entry as another SCA for the same application. In case duplicates are received, we discard all of its content, except the specified version. The version is then appended to the original SCA that was being duplicated. Because SCA verifiers can also act as SCA publishers, many duplicates can be received once push-notifications have been sent to all interested SCA verifiers.

Filtering SCAs: We suggests that SCA repositories use filters on their SCA listings to improve look-up performance of SCA verifiers. Firstly, an *application filter* can be applied to only show SCAs for particular applications because many SCA verifiers may be mainly interested in their installed applications. Secondly, users can use a *SCA filter* to only receive SCAs which were created after a certain previously submitted SCA. We implemented this filter because users may not want to receive SCAs that they have already assessed. We suggest that SCA verifiers store the ID of their most recently checked SCA to benefit from using *SCA filters*. This filter could also be used for polling algorithms if receiving push-notifications is not of

interest to the SCA verifier. Also, both mentioned filters can be combined in one query. In our implementation, filtering is realised by using URL query strings.

5.1.1 Packet-Based Self-Certifying Alerts

Packet-based SCAs are an integral part of Scargos. All three components of Scargos use SCAs as a mean of communication:

- SCA publishers create and publish SCAs when an attack is detected;
- SCA repositories store and distribute SCAs;
- SCA verification manager decides about the authenticity of a SCA and takes appropriate actions.

SCAs always refer to a vulnerability of a specific network-facing service. By using unique application IDs, all mentioned components can address a specific application. The IDs are assigned by each repository individually and SCA publishers and subscribers need to be aware of existing IDs for their surveilled application or create a new entry, if one of the used application has not been registered yet. Figure 5.2 shows an application entry for *WFTPD Server*. To assign an application ID, only the official application name and the vendor needs to be specified. A unique application ID can be looked up for each submitted application entry in a SCA repository.

Application ID:	1
Name:	WFTPD Server
Vendor:	Texas Imperial Software

Fig. 5.2: An example application entry with an assigned application ID by a SCA repository.

SCAs use application IDs to refer to a vulnerability in an application and describe the affected software version, the transport protocol, the affected port and the layer-5 PDUs. Figure 5.3 shows an example SCA of the application *WFTPD Server* with the assigned application ID 1. Transport Protocol and port are given because an application can serve on multiple ports using different transport protocols. If there is a standard port specified by the IANA for the application's use, then this port is to be included in the SCA; otherwise the default port that the application is preconfigured with is used. We define a layer-5 PDU as the transport-protocol payload, which is conventionally

contained in a TCP segment/UDP datagram. Packet-based SCAs contain a list of layer-5 PDUs, where each list item corresponds to single layer-5 PDU. The layer-5 PDU list is ordered, where the first element corresponds to the first PDU that was sent, the second element to the secondly send PDU, and so on. Because requests often contain binary data, each PDU is encoded using `Base64`.

Application ID:	1
Version:	3.23
Port:	21
Transport Protocol:	TCP

Layer-5 PDUs:	`VVNFUiBhbm9ueW1vdXMNCg==\n`
	`UEFTUyBtb3ppbGxhQGV4YW1wbGUuY29tDQo=\n`
	`U0laRSAvHUc0JUIcs6kURvm4Q0sEtJE1tiywTgz4k3RPJ` ...

Fig. 5.3: An example SCA of the application WFTPD Server. The attack consists of three Layer-5 PDUs, which are encoded using Base64. The last PDU has been truncated and is 727-Bytes long.

5.1.2 Push-Notification

We term notifications that SCA verifiers receive for every newly submitted SCA *push-notifications*. In our implementation we use *Twisted 13.1.0* which is a highly-configurable Python library for event-driven server architectures. Push-notfications are usually short messages which contain the ID of an application for which a new SCA was submitted. On receipt of this while using the application and SCA filter discussed in Section 5.1, the SCA verifier would query the SCA repository for the all new SCAs of that application ID. The SCA filter then ensures that the verifier only receives all SCAs that it has not yet verified.

Push-notifications can be sent in two ways, either in *single-channel mode* in which a SCA repository sends all push notifications to only one channel, or in *multi-channel mode* in which the SCA repository offers a separate channel for each application ID. When using the single-channel mode, a SCA verifiers may receive push-notfications about applications that are of no interest to the verifier. However, when using multi-channel, sensitive information could be disclosed, given that the PAM mirror is configured in the same manner as a PAM in the internal network. This can be partially mitigated by

using encryption and adding random time delay for each push-notification sent to an individual host. The drawbacks of each approach are shown in Table 5.1.

Method to distribute push-notifications	*Drawbacks*
Single-Channel	All SCA verifiers receive the same information. Sensitive information is can only be disclosed by the SCA verifier itself.
Multi-Channel	Sensitive information is disclosed when the PAM mirror is configured similarly to hosts in the internal network.
Encrypted Multi-Channel	Although encryption prevents eavesdroppers seeing which push-notification was received, an attacker can listen on all channels and infer from the timing that a notification was sent and which applications was used. Furthermore, an overhead is introduced by using encryption.
Encrypted and Time-Delayed Multi-Channel	Delaying the time a push-notification was sent by a random amount can be semi-effective to prevent information disclosure. An attacker could still create multiple bogus SCAs and then decide which application is used based on the received traffic. Additionally, the receipt of a push-notification is delayed which may defeat the purpose of push-notifications.

Table 5.1: Drawbacks of using different types of distribution methods for push-notifications.

In our implementation, the SCA repository uses single-channel push notifications because of the following reasons:

- Push-notifications are small and usually contain only one message, which specifies the affected application. Hence, receiving push-notifications causes a minimal overhead.
- Sending every SCA verifier all push-notifications is the only way to ensure that no sensitive information is disclosed by SCA repositories. While the SCA verifier can

disclose which applications are used by querying the SCA repository right after a certain push-notification was received, this can be easily prevented by the SCA verifier itself. An attacker could for example not determine which specific FTP application is used by a PAM mirror, if the SCA verifier requests a SCA for every push-notification related to FTP-servers.

5.2 SCA Publisher

The SCA publisher is concerned with processing the logs of its DTA honeypot into a SCA for publishing. SCA publishing consists of three steps:

1. A network-facing service of the SCA publisher's honeypot is being attacked and the threat is being detected.
2. The attack logs are analysed and relevant information is extracted.
3. A valid SCA is being produced and published to a SCA repository.

5.2.1 Attack Detection and Logs

Generally, the SCA publisher uses an DTA honeypot to automatically detects attacks. In our implementation we use the Argos honeypot which is described in Section 2.3.2. Argos automatically detects new attacks and instantly generates two log files:

session packet capture: Contains all of Argos' network logs, which are all the packets Argos has received while running. The data is contained in the `argos.netlog` and can be exported as a *pcap-File* [14] using the Argos tool *raw-2-pcap*.

attack log: Contains the memory pages and other data that can be correlated with the detected attack. The data is contained in the `argos.csi` and can be extracted using Argos' *carlog* tool of the *cargos-lib*.

The attack log is particularly interesting. It essentially consists of three types of data:

Exploit Ethernet Frame: The Ethernet frame which contains the attack that led to the manipulation of the execution flow. It is extracted by Argos by looking at which tainted data manipulated the execution flow and from which memory block the data originated.

Compromised Memory Block: The memory block in which the execution flow was manipulated.

Jump Target Memory Block: The memory block in which the arbitrary code is contained that the manipulated execution flow was instructed to jump to.

5.2.2 Single Packet Extraction

For brute-force replay, described in Section 4.1, only the packet which led to the compromise of the host is necessary to replay the attack. Because Argos already exports the exploit Ethernet frame natively, the goal of the single packet extraction is to convert Argos' output into a valid layer-5 PDU.

```
 carlog -E argos.csi.660313445 argos.netlog

Sequence number(total 33): 25
Header(Index Length):
3751    603


52 54 00 12 34 56 C6 C5 D8 42 59 44 08 00 45 00 02 4D 50 FA 40 00 40 06 8F 23
AC 14 00 01 AC 14 00 64 94 3C 00 15 52 DB AA E1 1B B6 B0 58 80 18 00 80 AA 6D
00 00 01 01 08 0A 00 19 FA 73 00 00 0A 53 53 49 5A 45 20 2F 1D 47 34 25 42 1C
B3 A9 14 46 F9 B8 43 4B 04 B4 91 35 B6 2C B0 4E 0C F8 93 74 4F 27 B5 96 3F 04
7F 71 7A 24 93 3C 85 D5 08 D3 E0 7C 75 70 76 7D 05 A8 41 A9 80 FD BE 88 F8 9B
```

Fig. 5.4: Truncated output for the command `carlog -E argos.csi.660313445 argos.netlog` using the logs of an attack against WFTPD Server running in Argos with the ID `660313445`.

Figure 5.4 shows the shell output of Argos' carlog tool. By combining a variety of Linux shell commands this output can be converted into a pcap-file, which enables further analysis by tools like Scapy [18] or Wireshark [31]. Using Scapy we are able to dynamically extract the layer-5 PDU of the exported exploit Ethernet frame and information such as the destination port. The generated SCA is published to a SCA repository by correlating the extracted information with information from the SCA publisher's global configuration file.

5.2.3 Stream Extraction

Aggregating all layer-5 PDUs of the TCP conversation which let to a compromise requires more steps than single packet extraction. TCP stream extraction is necessary to create a packet-based SCA for *exact stream replay* described in Section 4.2. We mentioned in Section 5.2.1 the different outputs that Argos' carlog tool produces from it logs. It is

essential that these logs are present to perform stream extraction. We extract a TCP stream from these logs, by taking the following steps:

1. Extract a TCP package which is part of the larger conversation that led to the exploit
2. Get the ID of the TCP conversation
3. Extract the conversation as pcap
4. Extract and reassemble the individual layer-5 PDUs
5. Create and submit the SCA to the SCA repository

5.2.3.1 Finding the compromising packet in a packet capture

Finding the compromising packet seems trivial by directly using the *exploit Ethernet frame* in a similar way to single packet extraction, as described in Section 5.2.2. Unfortunately, Argos can not effectively extract the Ethernet frame for all cases. In Section 6.1.2 we discuss the likelihood that the correct data is found. Our experiments showed that if Argos was unable to extract the data, a random UDP packet was exported instead. We can only suspect that this is an implementation error or the false result of an operation. However, this shows that the generation of the exploit Ethernet frame is rather unreliable.

To circumvent the problem, we instead leverage other parts of the attack log. Both the compromised memory block and the jump target memory block contain data that must have been present in the original attack. The goal of Overwrite attacks is to manipulate the execution flow and thus always manipulate the data of some memory block. The two memory blocks that Argos extracts for us both are directly connected to the TCP conversation which led to the exploit. The jump target memory block contains the arbitrary code an attacker wants to execute on the machine. On the other hand, the compromised memory block is the memory block in which the execution was manipulated and thus contains external data which enabled the manipulation. Because the attacker injected the data in both memory blocks by means of exploiting a vulnerability, the data is contained somewhere in the packet capture. Taking this information into account we can formulate Assumption 3.

Assumption 3 *Given that the conversations received by a compromised host along with the manipulated memory block are available, the conversation which led to the compromise is most likely the conversation which shares the* Longest Common Substring *with the manipulated memory block.*

Using suffix trees the longest common substring problem can be solved in $\mathcal{O}(n)$ [2, 15], whereas n is the length of the longest text input. Because n can be quite large, as it usually contains all packets that Argos has encountered while running, the algorithm can significantly impact overall performance. In our experiments in Section 6.1.2 we show that the first couple of substrings of one of our memory blocks is often sufficient to find the compromising packet, which leads to a much better performance of $\mathcal{O}(1)$. Algorithm 2 shows the entire algorithm that we use and has a worst-case performance of $\mathcal{O}(n/m)$, where m is the minimum length that a string has to be. As a consequence, we us a weaker version of Assumption 3 depending on the type of memory block:

Jump Target Memory Block Search: The jump target memory block starts with the first byte of arbitrary code that an attacker injected. Thus by taking a long enough sequence of bytes, one packet should carry the same byte sequence, which is the packet that injected the code. We use Algorithm 2 for the search, but instead of continuously increasing the position for our search, we decrease the substring length to a minimum point because the jump target always includes the correct byte sequence which the attacker wanted to execute. In our implementation we use a minimum substring length of 50 Bytes.

Compromised Memory Block Search: The compromised memory block may also contain other data that is unrelated to the attack; thus we take a predefined byte sequence length interval, extract parts of the memory with a certain interval length and search the session packet capture for any matching packets with the same byte sequence. If unsuccessful we increase the offset by the interval length and take the next byte sequence. We use Algorithm 2 for our search with a minimum substring length of 15 Bytes.

Exploit Ethernet Frame Search: We extract the layer-5 PDU of the Ethernet frame and proceed with the same search algorithm as for the *compromised memory block search*

Similar to the exploit Ethernet frame which cannot always be extracted by Argos, the jump target memory block also fails to be exported in some instances. Thus, we

combine all three methods to extract a package that is part of the attacking TCP stream. We evaluated the likelihood of encountering a false packet and the overall accuracy is described in Section 6.1.2.

Algorithm 2: Compromising Packet Search Algorithm

Data: *pcap* as the session packet capture;
mem as the manipulated memory block or compromising Ethernet frame;
substringlength the length of the substring that is to be searched with;
Result: The compromising packet within the packet capture

```
  begin
1     position = 0
2     repeat
3         substring=get_substring(mem,substringlength, position)
4         result=get_string_position(pcap, substring)
5         position = position + substringlength
      until result ≠ ∅ or position = length(mem)
6     return result
```

5.2.3.2 Extracting TCP streams

In order to create a valid SCA for *exact stream replay*, we need to extract the entire stream of packets sent by the attacker by using the single packet that we already have extracted. Our implementation is simplified by using available tools to aid stream extraction. The tool Wireshark [31] offers a command-line version of its tool called *tshark*. Using tshark we can apply filters on packet capture files. One of the features tshark/Wireshark offers it that every TCP stream is assigned a unique ID, which we call *stream ID*. By using the payload of our single package, we filter in our packet capture file for the same packet, but output the stream ID. Subsequently, we use this ID to extract the entire stream.

5.2.3.3 Generating SCAs from TCP streams

Having the TCP stream, the next step to create a valid SCA is to extract all layer-5 PDUs. We discard packets that do not carry any payload such as SYN, ACK, FIN, RST as well as any retransmissions, and then use the tool scapy [18] to extract all layer-5 PDUs. Unfortunately, a layer-5 PDU might be segmented and reassembly becomes

necessary to create to an equivalent PDU. In our implementation, we do the reassembly ourselves by reassembling all TCP packets which acknowledge the same previously sent packet. The layer-5 PDUs are then used to create a SCA by combining it with any information that has been set in the SCA publisher's configuration such as version and application ID.

5.3 SCA Verification

SCA verification is initiated by the SCA verifier. The SCA verifier maintains a *PAM list*, which we have briefly mentioned in Section 3.4. This list contains all the application IDs of all installed services on the PAM mirror, which we implemented as part of the SCA verifier's configuration file. Simultaneously, the SCA verifier listens for new push-notifications, which are sent by the SCA repository. If a new push-notification is received, which contains one of the IDs of the PAM list, the SCA verifier issues a HTTP request to the SCA repository, to receive all SCAs that have not been verified yet using the SCA repository filters mentioned in Section 5.1. The received SCAs are then replayed one after the other using one of our proposed methods: brute-force replay or exact stream replay. If the verification manager detects an attack using its DTA honeypot, the corresponding SCA is valid; otherwise it is invalid after a timeout.

5.3.1 Brute-Force Replay

The mechanisms of brute-force replay have already been discussed in Section 4.1. Brute-force replay sends a malicious PDU to all possible input states and relies on the implementation of a protocol handler. The malicious PDU needs to be previously extracted, as we have described in Section 5.2.2.

Protocol handlers are necessary to be able to replay a packet to all input states. Thus, it is vital that the protocol used is publicly known. In our implementation, a new protocol handler inherits from an abstract base class. The handler has to specify the IANA standard port of the protocol and a function which takes as an argument the layer-5 PDU and a host IP to which the payload is replayed to. When the replay function of a protocol handler is being called it is desirable that the input states that most likely get compromised are attacked first. This is to shorten the average time frame that brute-force replaying takes. In Section 4.1 we showed Figure 4.1 to depict the process of brute-force replay. Figure 4.1 also depicts how we designed the protocol handler for the FTP protocol to achieve best performance. First, we send the malicious PDU when the `USER` command would be expected. Our experimental evaluation, discussed

in Section 6.2.1.1, shows that the USER command seems to be among the most common attack vectors. Also, sending only the USER command is fastest to execute because it consists of only one command. We choose to issue the second malicious PDU after the PASS command because our experiments have shown that it is the second most likely attack vector. Also, sending the PDU after the PASS command includes vulnerabilities in the ACCT command. Finally, an attack on the PASS command is sent last, since we have found no PASS attack vectors in our selection of FTP attacks, which indicates that this type of attack is rather unlikely. During brute-force replay, we ignore any response codes that the servers issues to us, as they might be a predictor as to whether a particular state transfer was successful. However, given that not much can be changed to re-ensure the state transfer, overall response codes are irrelevant for the success or failure of an attack.

5.3.2 Exact Stream Replay

While brute-force replay relies on the implementation of a protocol handler, exact stream replay is protocol-independent. Hence, we only need to dissect and decode each individual PDU from the layer-5 PDU list and then send it to the vulnerable service accordingly. Replaying is non-trivial because it is protocol-independent, and as a result we describe it further in Section 4.2. Although Algorithm 1 works fast for SCAs with few layer-5 PDUs, performance is significantly limited by the timeout value. If a process responds faster than within the set timeout value, it stays idle, while we could have sent the next layer-5 PDU.

As a result, we developed Algorithm 3 for our implementation for SCAs that include more than one layer-5 PDU. In this algorithm, the next layer-5 PDU is sent either when a timeout occurs, or directly after a consecutive data block was received by the client. We increased the timeout value to $150ms$ because the chances of taking longer to deliver a packet are higher with multiple packets and also we cannot fit in other computational tasks before we receive an answer as we did in Algorithm 3 by decoding all SCA packets.

While lower timeout values may also lead to better performance, an SCA may not verify if it is set too low. Defining the timeout inherently depends on the latency between SCA verifier and its DTA honeypot as well as the performance of the network-facing service. Given that both values can fluctuate quite a bit, the maximum timeout can only be estimated empirically. Our chosen timeout values of $100ms$ or $150ms$ respectively worked well for all of our tested attacks.

Algorithm 3: Exact Stream Replay for multiple layer-5 PDUs

Data: L as an ordered list layer-5 PDUs; $Host$ as the IP and port of a host

Result: Each packet from L is replayed to $Host$

```
    begin
1       socket = establish_connection(Host)
2       receive_data(socket)
3       foreach element p of L do
4           send_data(socket, p)
5           receive_data(socket)
6       close_connection(Host)
7       return

  Procedure receive_data(socket)
8       Set timer to 150ms
9       start to countdown timer
10      repeat
            receive data from socket
        until timer = 0 or data ≠ ∅
11      return
```

Chapter 6

Experimental Evaluation

This chapter evaluates Scargos by discussing a variety of experiments. We implemented Scargos as a prototype on a x86 machine running *Linux 3.5.0* and using *Argos 0.5.0* as DTA honeypot. Experiments were being done on a Intel(R) Core(TM) i3 CPU M 370 which is a 2.40GHz dual-core processor with 4GB RAM.

Application Name	*Version*	*Vendor*
3CDaemon	2.0.10	3Com Corp,
Dream FTP Server	1.02	BolinTech Inc.
FileCopa FTP Server	1.01	InterVations, Inc
FreeFloat FTP server	1.0.	FreeFloat
Sami FTP Server	2.0.1	KarjaSoft
WAR-FTPD	1.65	Jarle Aase
WFTPD	3.23	Texas Imperial Software
Savant Web Server	3.1	Michael Lamont
Win.: MS05-039 (PnP)	—	Microsoft
PCMan's FTP Server	2.0.	PCMan
freeSSHd	1.0.9	WeOnlyDo
freeFTPd	1.0.8	WeOnlyDo
Apache HTTP Server	1.3.24	Apache Software Foundation
BadBlue Enterprise	2.72b	Working Resources Inc.
Light HTTP Daemon	0.1	The LHTTPd Development Team
KNET	1.04b	Zero Point Five

Continued on next page

Table 6.1: (continued)

Application Name	*Version*	*Vendor*
Kolibri	2.0.	Fedja Stevanovic
Xitami	2.5c2	iMatix
Win.: MS03-026 (RPC)	—	Microsoft
Win.: MS04-011 (LSASS)	—	Microsoft
Mercury/32 Mail	4.51	NetWare Systems
IntraSrv Web Server	1.0.	Leigh Brasington
Mdeamon PRO	8.03	Alt-N Technologies
Win.: MS08-067 (NetAPI)	—	Microsoft

Table 6.1: Applications and versions we used during our experiments. Windows are abbreviated as *Win* and include the appropriate Microsoft security bulletin ID for the exploited vulnerability that we have chosen.

Figure 6.1 show the different applications and used versions that we attacked as part of our evaluation. Scargos is evaluated with real attacks, which we have listed in the Appendix in Figure A.5. We have included high profile attacks against Windows and Apache. The vulnerability *MS03-026 (RPC)* was for example used in the *Blaster Worm* [3] and infected 100,000 Microsoft Windows hosts within one week; some estimate that a total of 500,000 hosts had once been infected [9].

Likewise, the *Sasser* [39] worm uses vulnerability *MS04-011 (LSASS)* and is estimated to have infected over 250,000 Windows computers. More recently the worm *Conficker* [11] uses *MS08-067 (NetAPI)* to infect millions of hosts. The virus downloads arbitrary files and disables security related system settings. In the second quarter of 2011, Microsoft reported that still about 1.7 million hosts were infected by Conficker. While attacks against operating system services certainly offer a large pool of victims, it is important to also consider threats on other services. *Metasploit* is an open-source framework for launching automated attacks and offers insight into which services are most likely to be attacked by intruders.

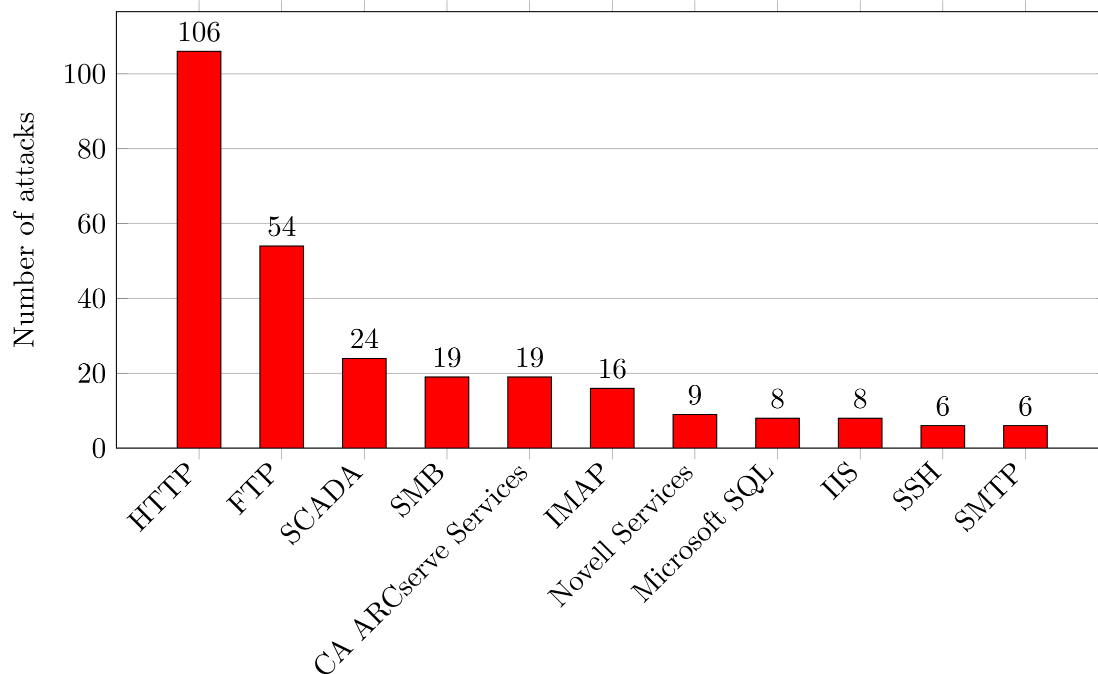

Fig. 6.1: The number of available windows server attack per protocol per application. The results were aggregated from the open-source attack framework Metasploit.

Figure 6.1 show the number of available attacks per protocol in the Metasploit framework. We can see that while there are a number of attacks on Windows SMB service (19 attacks), most threats exists for HTTP (106 attacks) and FTP (54 attacks). One of the reasons might be that there are many HTTP and FTP servers openly accessible on the Internet, while other protocols maybe often hidden behind firewalls or offer less of an attack surface. When comparing Windows to Linux in terms of number of attacks, we found there a total of 853 Windows attacks, while only 94 attacks for Linux exists.

Application Name	*Operating System*	*Protocol*
3CDaemon	Windows 2000 (SP0)	FTP (21)
Dream FTP Server	Windows 2000 (SP0)	FTP (21)
FileCopa FTP Server	Windows 2000 (SP0)	FTP (21)
FreeFloat FTP server	Windows 2000 (SP0)	FTP (21)
Sami FTP Server	Windows 2000 (SP0)	FTP (21)
WAR-FTPD	Windows 2000 (SP0)	FTP (21)

Continued on next page

Table 6.2: (continued)

Application Name	*Operating System*	*Protocol*
WFTPD	Windows 2000 (SP0)	FTP (21)
Savant Web Server	Windows 2000 (SP0)	HTTP (80)
Win.: MS05-039 (PnP)	Windows 2000 (SP0)	MS SMB (445)
PCMan's FTP Server	Windows XP (SP0)	SFTP (22)
freeSSHd	Windows XP (SP0)	SSH (22)
freeFTPd	Windows XP (SP0)	SFTP (22)
Apache HTTP Server	Windows XP (SP0)	HTTP (80)
BadBlue Enterprise	Windows XP (SP0)	HTTP (80)
Light HTTP Daemon	Windows XP (SP0)	HTTP (80)
KNET	Windows XP (SP0)	HTTP (80)
Kolibri	Windows XP (SP0)	HTTP (80)
Xitami	Windows XP (SP0)	HTTP (80)
Win.: MS03-026 (RPC)	Windows XP (SP0)	MS DCE/RPC (135)
Win.: MS04-011 (LSASS)	Windows XP (SP0)	MS SMB (445)
Mercury/32 Mail	Windows XP (SP2)	SMTP (25)
IntraSrv Web Server	Windows XP (SP2)	HTTP (80)
Mdeamon PRO	Windows XP (SP2)	IMAP (143)
Win.: MS08-067 (NetAPI)	Windows XP (SP2)	MS SMB (445)

Table 6.2: Overview of the protocol each application uses and the operating system we have conducted the experiments on.

In the light of this data, our selection of applications seems to be well-suited to the current threat level. Table 6.2 shows which protocols our selected applications use and which operating systems we used for our experiments. The majority of evaluated attacks are Win32 HTTP and FTP server applications as well as attacks on native windows services, which covers the group of applications that is most likely being attacked in the wild.

6.1 Accuracy

A high accuracy is vital to ensure Scargos' widespread use. We define accuracy of Scargos as having a high probability to generate and verify SCAs correctly. When evaluating

accuracy it is important to look at all parts which can effect the result. In Scargos, accuracy depends on three components:

Argos: We use Argos as our DTA honeypot and rely on the correct detection of attacks. SCA can not be created, if attacks remain undetected.

Packet Extraction: The SCA publisher relies on a robust algorithm to extract the needed packets. If packet extraction fails, no SCA can be published.

Packet Replay: The SCA verifier needs a replay algorithm that can successfully replay SCAs such that honeypots become compromised. If packets cannot be replayed or are replayed in a manner that the attack fails, false negatives occur because correct SCAs are not verified.

In our accuracy experiment, we tested all three components and subcomponents by conducting 10 runs for each of our selected applications shown in Table 6.1.

		Packet Extraction			Packet Replay	
Application Name	Argos DTA	Jump Target Memory Block	Compromised Memory Block	Exploit Ethernet Frame	Brute-Force	Exact Stream
Mercury/32 Mail	—	—	—	—	—	—
Mdeamon PRO	—	—	—	—	—	—
3CDaemon	✸	✸	—	—	—	✸
Dream FTP Server	✸	✸	—	—	—	✸
FileCopa FTP Server	✸	—	✸	✸	✸	✸
FreeFloat FTP server	✸	—	✸	✸	✸	✸
Sami FTP Server	✸	—	✸	✸	✸	✸
WAR-FTPD	✸	—	✸	✸	✸	✸
WFTPD	✸	—	✸	✸	✸	✸
Savant Web Server	✸	—	✸	✸	✸	✸
MS05-039 (PnP)	✸	✸	—	—	—	✸

Continued on next page

Table 6.3: (continued)

Application Name	Argos DTA	Packet Extraction: Jump Target Memory Block	Packet Extraction: Compromised Memory Block	Packet Extraction: Exploit Ethernet Frame	Packet Replay: Brute-Force	Packet Replay: Exact Stream
PCMan's FTP Server	✸	—	✸	✸	—	✸
freeSSHd	✸	—	✸	✸	—⋆	✸
freeFTPd	✸	—	✸	✸	—⋆	✸
Apache HTTP Server	✸	✸	✸	—	—	✸
BadBlue Enterprise	✸	✸	—	—	—	✸
Light HTTP Daemon	✸	—	✸	✸	✸	✸
KNET	✸	—	✸	✸	✸	✸
Kolibri	✸	—	✸	✸	✸	✸
Xitami	✸	—	✸	✸	✸	✸
MS03-026 (RPC)	✸	—	✸	✸	—⋆	✸
MS04-011 (LSASS)	✸	—	—	✸	—⋆	✸
IntraSrv Web Server	✸	✸	✸	—	—	✸
MS08-067 (NetAPI)	✸	—	✸	✸	—⋆	✸

Table 6.3: Success or Failure of each component/subcomponent to process its previous input correctly. The symbol ✸ signifies success, — signifies failure and —⋆ means we did not implement the protocol used by the application. Failure in packet extraction are either due to that no logs were generated successfully or no matching packet could be found in the session packet capture.

6.1.1 Argos

During our experiments we worked extensively with Argos, as it is was our selected approach to detect attacks. Argos uses DTA as described in Secion 2.2, which is known to have a very low false negative rate. However, the results from Table 6.3 show, that out of our 24 applications an attack against *Mercury Mail* and *Mdeamon Pro* was not

detected by Argos and successfully circumvented its DTA. We can not say with certainty why Argos did not detect the attack, but can only suspect that it is due to:

1. under-tainting as described in Section 2.2.2.2.1;
2. a bug in the implementation;
3. or Argos was trusting input, which should not be trusted as described in Section 2.2.2.2.2.

Due to the detection failure, we can not create a SCA and all the following steps fail accordingly.

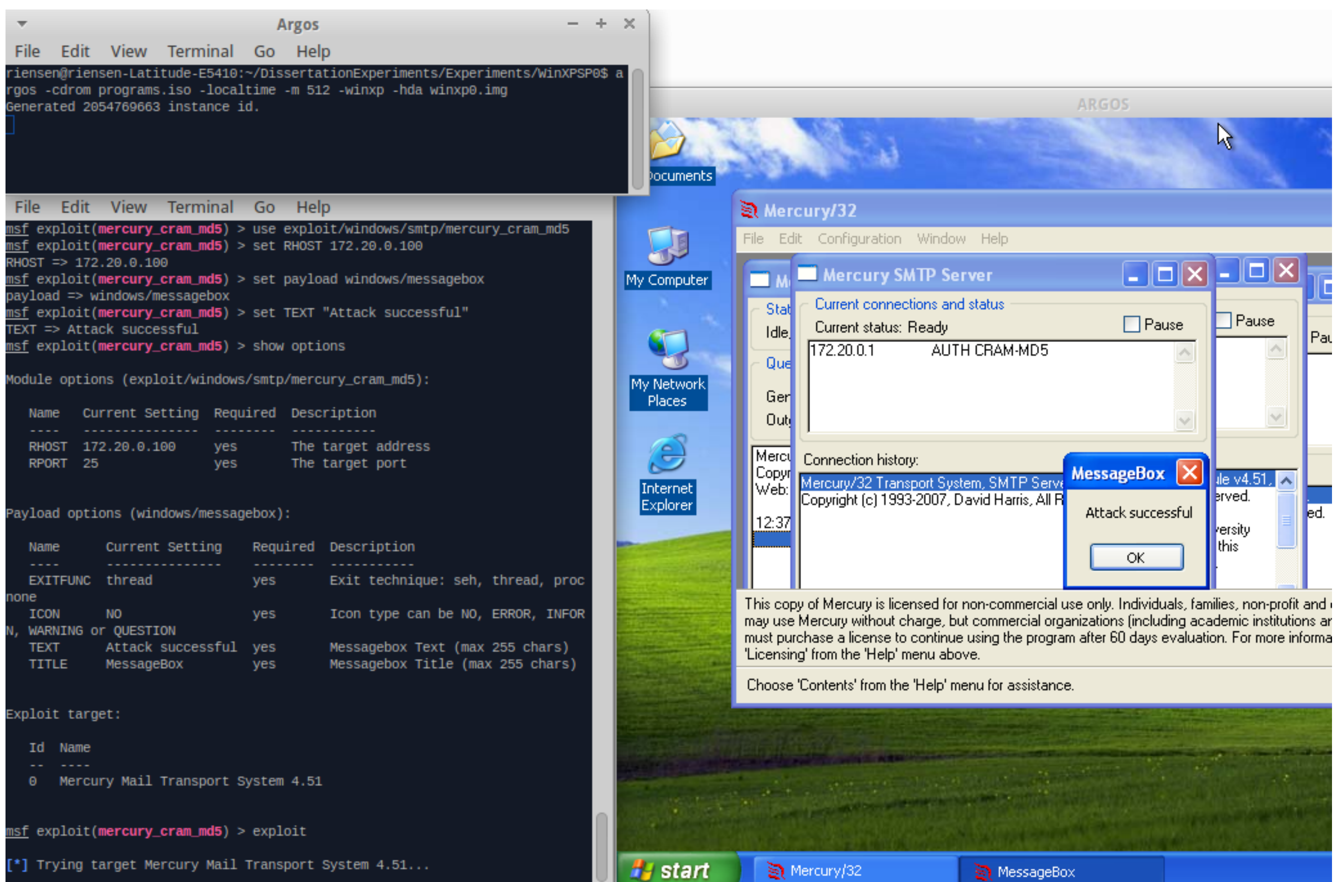

Fig. 6.2: An attack is successfully launched against Mercury/32 Mail 4.51 on Windows XP SP2 while circumventing Argos DTA detection. We use the Metasploit Framework to exploit a weakness in the SMTP CRAM-MD5 handling and inject a payload to open a message box on the client saying “Attack successful”.

6.1.2 SCA Generation

In order for SCA generation to succeed, it is of vital importance that we find the correct stream/packet which led to the compromise of the host. We described two ways to accomplish packet extraction: *single packet extraction*, described in Section 5.2.2 and *stream extraction*, further described in Section 5.2.3. Both ways rely on the success of particular algorithms. While single packet extraction relies on the successful generation of the exploit Ethernet frame, stream extraction relies on the success of either of the following:

Jump Target Memory Block Search: Substrings of the *jump target memory block* are used to search for a matching packet in the *session packet capture*.

Compromised Memory Block Search: Substrings of the *compromised target memory block* are used to search for a matching packet in the *session packet capture*.

Exploit Ethernet Frame Search: Substrings of the exploit Ethernet frame are used to search for a matching packet in the *session packet capture*.

In contrast to single packet extraction, in stream extraction just one of the described search algorithms must be successful to ensure the success of SCA generation. However, if the correct exploit Ethernet frame is available, both methods will succeed because we are always able to find the same packet in the session packet capture. Unfortunately, generating the exploit Ethernet frame from Argos' logs is also an unreliable approach because if no packet could be found a random packet is returned.

Table 6.3 shows the results of our experiments: packet extraction is inherently unreliable. The results show that there is no method which consistently extracts a packet/stream correctly. This is mainly due to the fact that Argos cannot always determine the right memory block and neither can it reliably trace back the compromising data to export the correct exploit Ethernet frame. Furthermore we do not use the longest common substring method because it is too slow; instead, our search algorithm simply uses substrings of the generated memory block results to find matching packets in the session packet capture. This may contribute to some false negatives in the results. The most reliable method is using the compromised memory block search, while the most unreliable is using the jump target memory block.

We gathered more detailed results about the SCA generation algorithms we used in Table 6.4. It shows that either our algorithm is successful in its first or second run, or it searches through the entire block before ending up with no results. That results are generated either in the first few runs or never helps us to adopt our solution

to improve performance, as further discussed in Section 6.2.2. However, one of the remaining questions is: why do the searches exhaust without any results? When our search exhausts, no matching packet is found in the *session packet capture* with different substrings of the memory block. We suggest two possibilities: firstly, a false memory block was exported, or secondly, no consecutive, at least 15 byte long byte sequences were pushed by an attacker into the appropriate memory block during the attack. Given that false packets could be exported at times and that there are maybe other packets with the same byte sequence as the one we use in our search, the question remains, how likely is it to find a packet in the *session packet capture* which in fact belongs to another unrelated non-compromising conversation. We try to mitigate the probability of such an event by requiring that a minimum amount of consecutive bytes be used as a substring to search for in the packet capture. The minimum substring length for jump target memory block search is 50 Bytes and for compromised memory block search it is 15 Bytes, which leads to a very low probability of finding an unrelated packet.

Application Name	Length of all layer-5 PDUs in Bytes	Number of Runs		
		Jump Target Memory Block Search	Compromised Memory Block Search	Exploit Ethernet Frame Search
3CDaemon	2055	1	—⋆	—⋆
Dream FTP Server	1049	1	—	—
FileCopa FTP Server	815	—	1	1
FreeFloat FTP server	571	—	1	1
Sami FTP Server	567	—⋆	2	1
WAR-FTPD	1031	—	1	1
WFTPD	579	—	1	1
Savant Web Server	401	—⋆	1	1
Win.: MS05-039 (PnP)	3266	1	—⋆	*19 (false packet)*
PCMan's FTP Server	5007	—	1	1
freeSSHd	20272	—	1	1
freeFTPd	20609	—	1	1
Apache HTTP Server	7145	1	1	—⋆

Continued on next page

Table 6.4: (continued)

Application Name	Length of all layer-5 PDUs in Bytes	Number of Runs: Jump Target Memory Block Search	Compromised Memory Block Search	Exploit Ethernet Frame Search
BadBlue Enterprise	4285	1	—	—⋆
Light HTTP Daemon	782	—	1	1
KNET	1039	—	1	1
Kolibri	1013	—	1	1
Xitami	857	—	1	1
Win.: MS03-026 (RPC)	2272	—⋆	1	1
Win.: MS04-011 (LSASS)	11287	—	—⋆	1
IntraSrv Web Server	4792	1	1	—⋆
Win.: MS08-067 (NetAPI)	7711	—⋆	2	1

Table 6.4: Number of runs each search algorithm needs to perform to find a substring that matches a packet from the *session packet capture*. — signifies that Argos was not able to export the needed memory block/Ethernet frame or the export was too short (below 90 Bytes) and —⋆ signifies that a log file was generated by Argos, but no matching packet could be found in the packet capture using our search algorithm. *false packet* indicates that the correct matching packet was found but by using a falsely exported Ethernet frame.

If Argos cannot export the correct Ethernet frame, it returns an arbitrary packet. In all our tested cases this was a small UDP packet. To reduce the chances of rediscovering the same falsely-exported packet in the *session packet capture*, we restrict our exploit Ethernet frame search to only look for TCP packets. This circumvention backlashes as we can see when looking at the results of vulnerability *MS05-039 (PnP)*. In this case, Argos exported a DHCP packet instead of a packet form the attacking stream. Interestingly, using our algorithm we still correctly retrieved the right packet. The chances of discovering the correct packet instead of an unrelated one are higher because:

1. Argos might not return a completely random packet, when it cannot find the correct packet, but rather a packet which may be similar byte-wise;

2. when two packets are discovered at once during our search, we return the newest packet.

However, this example shows that there is a high risk involved when using the exploit Ethernet frame logs, as it can lead to generating a false SCA.

Our accuracy results for SCA generation are summarised in Table 6.5. When using the jump target memory block search, we have a mediocre chance of being successful to find a matching packet, while the risk of discovering an unrelated packet is very low. Using the compromised memory block seems to combine the best of both worlds: we have a very good chance of finding a matching packet and the risk of getting an unrelated packet is relatively low. Using the exploit Ethernet frame for searching can only be recommended as a last means, as the risk of detecting a false packet is comparatively high.

In conclusion, we can say that the best approach to finding a matching packet of the attackers' conversation is by combining all three methods in the following order: compromised memory block search, jump target memory block search and exploit Ethernet frame search. This way we only use the more risky exploit Ethernet frame search when the other methods have failed.

	Successful Packet Extractions	Probability to generate false results
Jump Target Memory Block Search	6	*Very low:* We start with a substring length of 600 Bytes and stop the search after using a substring length of 50 Bytes. In the worst case, the probability of a encountering a unrelated packet from another stream depends on the number of packets in the *session packet capture* and is: $\frac{numberofpackets}{400^2}$
Compromised Memory Block Search	17	*Low:* We use a substring length of 15 Bytes. The probability of encountering a different packet with the payload depends on the number of packets in the *session packet capture* and is: $\frac{numberofpackets}{120^2}$
Exploit Ethernet Frame Search	16	*High:* Argos returns a random packet if it was unsuccessful to generate the exploit Ethernet frame.

Table 6.5: Combined overview of the accuracy results for SCA generation.

6.1.3 SCA Verification

SCA verification can be done in two ways: brute-force replay or exact stream replay. Table 6.3 shows the results of our accuracy experiments. Brute-force replay depends on the success or failure of Argos' exporting the correct exploit Ethernet frame. Our results show that brute-force replay fails when either no packet is exported or an incorrect exploit Ethernet frame is extracted. Furthermore, brute-force replay requires that an appropriate protocol handler has been developed. For our experiments we implemented a handler for HTTP and FTP. Lastly, the table shows that a vulnerability in *PCMan's FTP Server* failed to be verified. The reason for this is that the exploit Ethernet frame only exports a single frame, while the the original attack against *PCMan's FTP Server* consisted of two constituent Ethernet Frames. Stream reassembly would be necessary to enable correct verification as we have implemented it in exact stream replay, but this would also significantly decrease performance of SCA generation for brute-force replay.

In our implementation, we have different search algorithms for stream extraction. If one search fails to find a matching packet, we use the next algorithm; thus only one

the algorithm needs to succeed to generate a correct SCA for exact stream replay. By combining compromised memory block search, jump target memory block search and exploit Ethernet frame search, it turns out that we always find the correct compromising packet in the *session packet capture* to generate a SCA. Furthermore, we have tested all generated SCA with exact stream replay and correctly verified every SCA each time.

6.2 Performance

In the following, we want to measure the performance of verifying and generating packet-based SCA as well as the overall performance of Scargos. We then compare our results with the results of Vigilante [9]. We define performance as the speed in which a process finishes successfully.

6.2.1 SCA Verification

SCA verification performance differs greatly depending on the used variant. We proposed exact stream replay and brute-force replay. While both protocols suffer from relying the performance of the attacked service, in general, exact stream replay has a lower performance, but is protocol independent.

When we measure SCA verification performance, we measure the time beginning from when the SCA verifier fully received a SCA from a SCA repository until the SCA verifier's DTA honeypot alerts the verification manager that a compromise occurred.

6.2.1.1 Brute-Force Replay

As we have discussed in Section 5.3.1, brute-force replay sends a malicious layer-5 PDU to every possible input state of an application to ensure that one of the input states is the correct one. Brute-force replay requires that a protocol handler be implemented for every protocol that is to be attacked. Hence, performance decreases with the number of input states. We have implemented two protocols for brute-force replay: HTTP and FTP. While HTTP is stateless and thus only one input state exists, as we depicted in Figure 4.1, FTP has four input states that can be reached by sending layer-5 PDUs at three different times: after the welcome message, after the user command and after being successfully logged in. We need to define the priority in which each input state is being tried because sending requests takes time. We analysed the attack vectors of our vulnerable FTP applications and compiled them in Figure 6.3. In our scenario, five FTP attacks targeted the `USER` command, which is usually issued after the welcome message,

and three attacks used commands which would usually be issued after being logged in. We gathered no attack which targeted the `PASS` command. We used this data to adapted the priorities of the FTP protocol handler accordingly in the following order:

1. We send the payload after the welcome message, which requires to issue no other layer-5 requests beforehand.

2. We send the payload after authenticating, which requires to issue two layer-5 requests (`USER` and `PASS`).

3. We send the payload after the `USER` command, thus one layer-5 requests needs to be issued beforehand.

We believe that this is a good order of priorities because item 1 can be done quickly and also is being used by many attacks. Thus, even though our pool of tested applications is small, putting attacks on the `USER` command first seems to be a good choice. The rest of the ordering results from the fact that attacks on the `PASS` command seem rare.

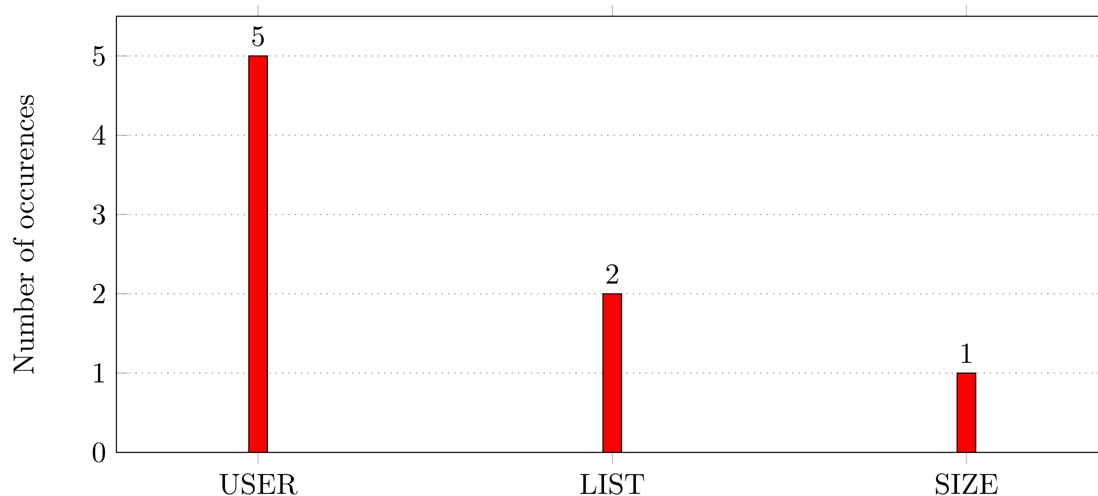

Fig. 6.3: Number of times each FTP command was used as an attack vector. Further details can be found in Table A.1

Application Name	Number of L5-PDUs	Length of all L5-PDUs (bytes)	SCA Generation: Single Packet Extraction (ms)	SCA Verification: Brute-Force Replay (ms)
FileCopa FTP Server	3	815	36.67	168.49
FreeFloat FTP server	1	571	26.97	49.74
Sami FTP Server	1	567	37.37	512.65
WAR-FTPD	1	1031	52.04	795.31
WFTPD	3	579	44.41	350.91
Light HTTP Daemon	1	782	32.03	37.33
KNET	1	1039	37.4	188.67
Kolibri	1	1013	31.29	5153.4
Xitami	1	857	31.64	260.3

Table 6.6: Performance of single packet extraction and brute-force replay in relation to number and size of layer-5 PDUs (L5-PDUs). The results are the average of 10 runs. Standard deviations are shown in Table A.4.

6.2.1.2 Exact Stream Replay

We compiled the results of evaluating exact stream reply and stream extraction in Table 6.7. When evaluating exact stream replay, the results seem at first glance quite mixed and a general pattern is not easily found. The application vulnerabilities that can be verified the fastest targets *FreeFloat FTP server*, while the slowest is targeting *Kolibri*. At first glance, there is not much difference between the two SCAs, as both need only one request to compromise the target. While *Kolibri's* SCA does carry twice as many bytes as *FreeFloat FTP server*, other attacks such as the attack against *freeFTPd* carry 20x more bytes and are still much faster than *Kolibri*. This implies that there is another factor that affects the performance of exact stream replay, which is the performance of the attacked applications and its time to process the attack.

Applications that should have a high performance and behave quite uniformly are system services offered by Windows itself. Yet, if we look at the performance of these applications, we see quite a fluctuation. The reason for this lies in our implementation. Exact stream replay waits after each request was sent $100ms$ for a reply from the server as explained in Section 5.3.2. This enables exact stream replay to be very accurate and protocol-independent, since the attacked applications transitions into its next input state automatically. However, the downside of a timeout is poorer performance.

Application Name	Number of L5-PDUs	Length of all L5-PDUs (bytes)	SCA Generation	SCA Verification
			Stream Extraction (ms)	Exact Stream Replay (ms)
3CDaemon	1	2055	927.12	171.08
Dream FTP Server	1	1049	488.5	1937.52
FileCopa FTP Server	3	815	475.34	238.68
FreeFloat FTP server	1	571	684.73	126.47
Sami FTP Server	1	567	696.28	822.31
WAR-FTPD	1	1031	697.91	890.91
WFTPD	3	579	471.96	727.71
Savant Web Server	1	401	487.75	164.26
Win.: MS05-039 (PnP)	13	3266	487.42	442.1
PCMan's FTP Server	1	5007	839.07	297.76
freeSSHd	1	20272	504.1	241.8
freeFTPd	1	20609	500.86	229.17
Apache HTTP Server	1	7145	851.3	498.38
BadBlue Enterprise	1	4285	537.43	185.54
Light HTTP Daemon	1	782	702.87	128.81
KNET	1	1039	716.1	242.09
Kolibri	1	1013	485.1	5103.76
Xitami	1	857	854.32	401.56
Win.: MS03-026 (RPC)	2	2272	488.12	207.06
Win.: MS04-011 (LSASS)	27	11287	980.18	494.18
IntraSrv Web Server	1	4792	505.89	180.28
Win.: MS08-067 (NetAPI)	36	7711	760.74	754.17

Table 6.7: Performance of stream extraction and exact stream replay in relation to number and size of layer-5 PDUs (L5-PDUs).

In Figure 6.4 we contrast the verification performance of Windows system services in relation to the layer-5 PDUs. We also added the average of all remaining applications, which have between one and three layer-5 PDU per attack. The graph clearly shows that the performance of exact stream replay depends directly on the number of layer-5 PDUs that the SCA contains.

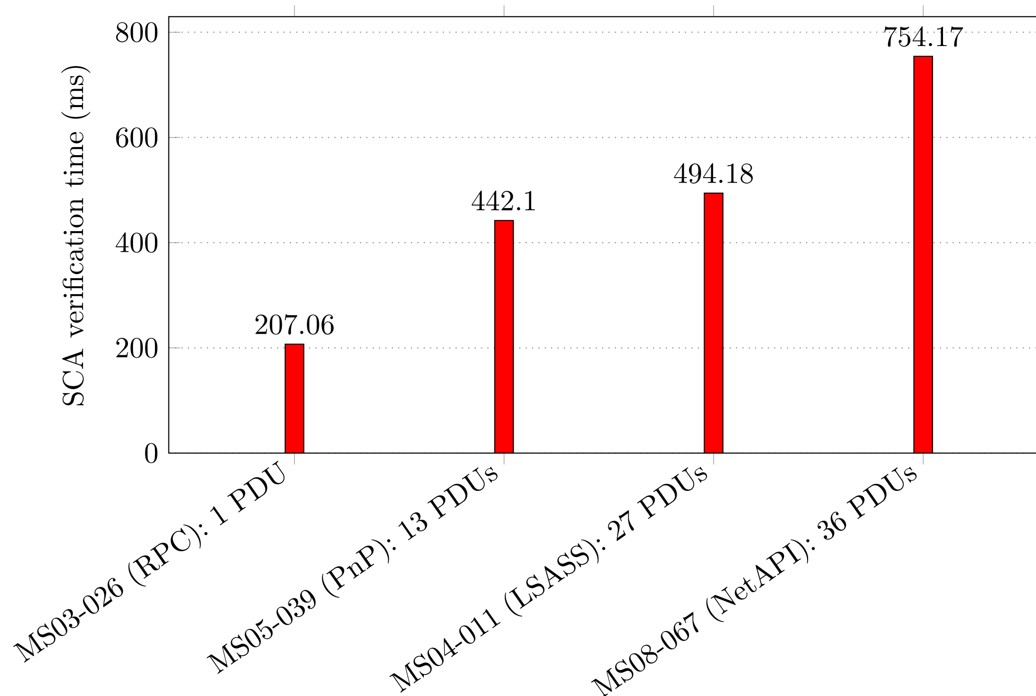

Fig. 6.4: Performance of SCA verification (exact stream replay) in relation to the number of layer-5 PDUs for system services. The label *Avg.* represents the average time of all applications which are not already listed as a label. The results are the average of 10 runs. Standard deviations are shown in Table A.4.

6.2.1.3 Comparison of Brute-Force Replay and Exact Stream Replay

In this section we want to compare the performance of brute-force replay and exact stream replay. In Figure 6.5 we can see that brute-force replay performs consistently better than exact stream replay. The average performance of exact stream replay is 884.66ms, while brute-force replay takes on average about 761.1ms. Brute-force replay can be up to 3 times faster as it is the case with *Light HTTP Daemon*, it is on average about 16% faster.

The reasons for this lie in the implementation. While brute-force replay possess a protocol handler and knows when the server has finished answering and when to e.g. expect a welcome message. exact stream replay is protocol-independent. Exact stream replay waits 100ms before each request to see if the server wants to send any new messages. As a result, exact stream replay always takes more than 100ms to verify an attack. The biggest difference can be seen in an attack against *Sami FTP Server*, in which exact stream replay takes more than 300ms longer to verify the SCA. The

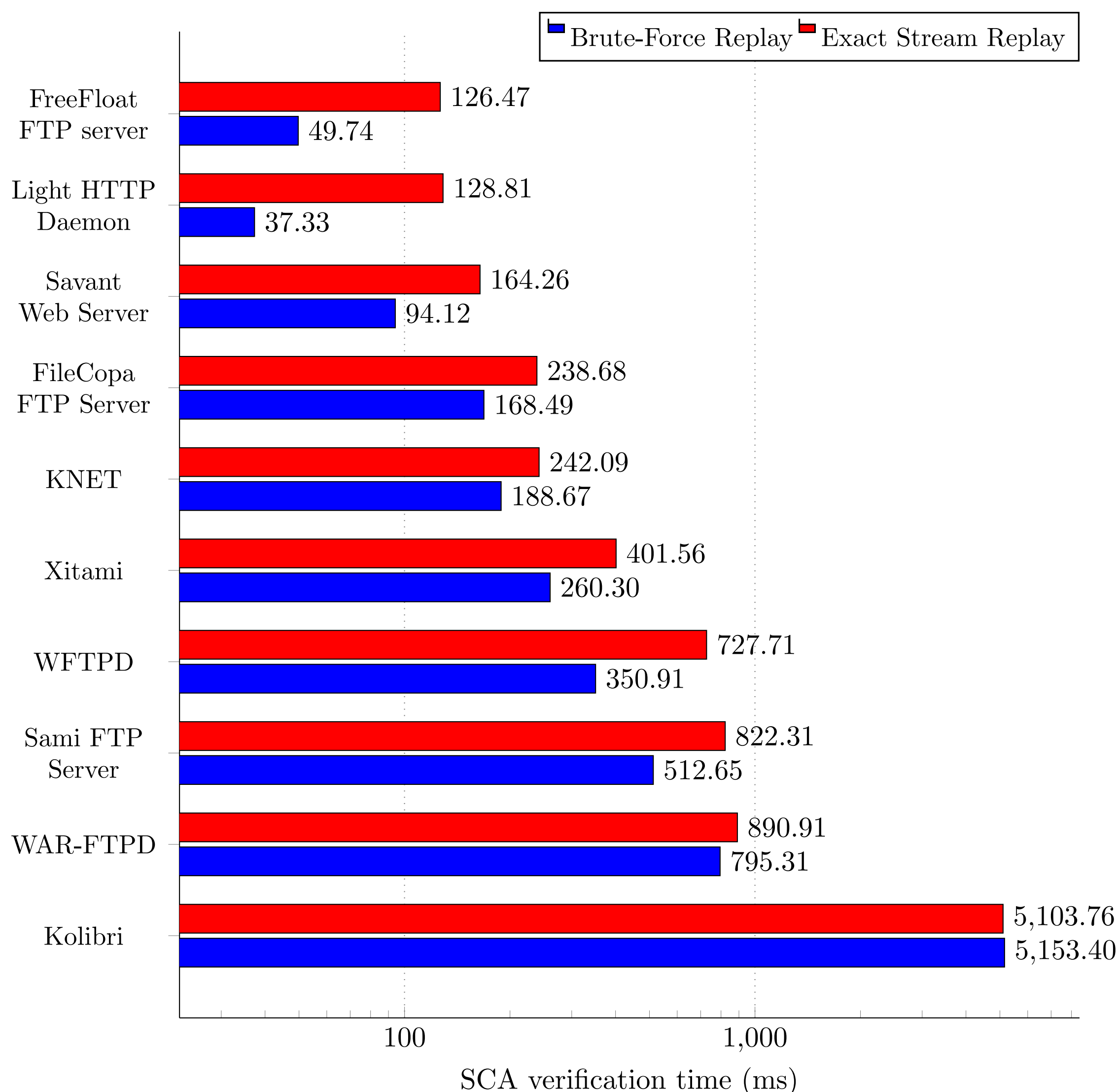

Fig. 6.5: SCA verification performance of brute-force replay and exact stream replay. The results are the average of 10 runs. Standard deviations are shown in Table A.4.

attack against *Sami FTP Server* targets the `LIST` command, which in order to bring the server into the right input state takes two requests before the the malicious PDU can be launched. Consequently, it takes exact stream replay at least 300ms to reach the state because the algorithm waits 100ms before each request. Also, as discussed in Section 6.2.1.2, services may take longer to process an attack, which is the reason for the long time to verify the attack against *Kolibri*.

Although, brute-force replay performs better time-wise, it has to be noted that its accuracy is worse and relies on the implementation of a protocol handler.

6.2.2 SCA Generation

Similar to SCA verification, SCA generation can be done in two ways, by using single packet extraction or stream extraction. Single packet extraction uses the Argos tool carlog directly to generate from the logs the exploit Ethernet frame, which contains the compromising packet of the attacker.

In contrast, stream extraction takes many more steps; the generated data from Argos' carlog tool is further processed to find the stream of packets which led to the compromise of the host. This search takes time and in some cases has to be repeated with different substrings from the logs, which we have partly investigated in Section 6.1.2 and show that our used algorithms are unreliable by itself. Thus the algorithms need to be combined to achieve best accuracy. This leads to a big performance decrease because if one search algorithm fails to return results the next one is tried and previous computations were useless. However, Table 6.4 of Section 6.1.2 show that our search algorithms generate results either in the first few runs or never. This helped us greatly improve performance because we alternate in the first few runs between all three algorithms to see if any of them can generate a result quickly, before doing the complete search. In detail, we do one run for a matching packet using compromised memory block search and start with an offset of one interval. If unsuccessful we try jump target memory block search for one run as well. Only if neither algorithm succeeds do we one run using the exploit Ethernet frame search, until we proceed with the normal order to do an exhaustive search with each.

In Figure 6.6 we see how much faster single packet extraction is in comparison to stream extraction. Because of the previously mentioned reasons, the performance difference is even greater that compared brute-force replay and exact stream replay from Section 6.2.1.3. On average, stream extraction takes $627.24ms$, while single packet extraction can be achieved in only $38.49ms$, which means it is on average 16 times faster. However, this performance increase comes with a price, as we have shown in Section 6.1.2, we can not blindly rely on the accuracy of single packet extraction and SCAs generated with single packet extraction can only be replayed using brute-force replay, which requires the implementation of a protocol handler.

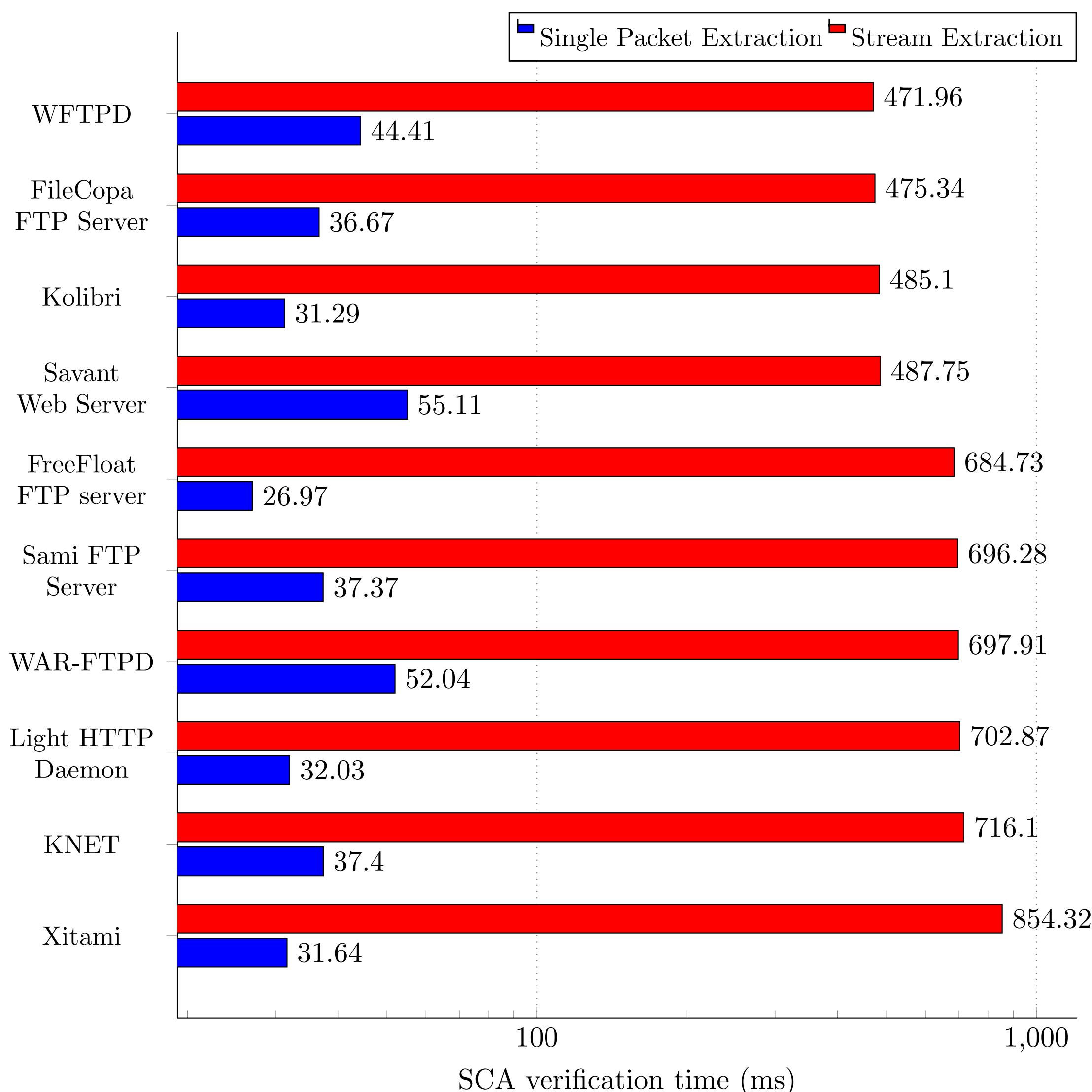

Fig. 6.6: SCA Generation: Performance of single packet extraction and stream extraction. The results are the average of 10 runs. Standard deviations are shown in Table A.4.

6.2.3 Comparison to Vigilante

In this section we want to compare our solution to Vigilante in terms of performance. Scargos and Vigilante are not easily comparable because Vigilante reduces the file size of SCAs by about 20% as it can be seen for the attack *MS05-039 (PnP)* in Table A.2 and Table A.3 in Chapter A. Furthermore, evaluated applications also depend on the

number of layer-5 PDUs and especially on the performance of the network-facing service to process the request.

In the original paper by Costa et. al [9], only three applications were evaluated. All three were either system services or well-known windows application: *Microsoft SQL Servers*, *Microsoft IIS Server* and *Microsoft Windows (MS03-026 (RPC))*. To enable a fair comparison we selected equally well-known applications and service vulnerabilities:*Apache HTTP Server*, *Microsoft Windows MS03-026 (RPC)*, *Microsoft Windows MS04-011 (LSASS)*, *Microsoft Windows MS05-039 (PnP)* and *Microsoft Windows MS08-067 (NetAPI)*. We compared the total time needed for SCA generation and verification in relation to SCA file size. The results are depicted in Figure 6.7.

We can see that Vigilante performs better for smaller SCA file sizes while Scargos performs better for bigger file sizes. Even taking into account that the SCA file size of Vigilante is truncated and in fact bigger than in the graph shown, Scargos outperforms Vigilante. If we look in more detail at Table A.2 and Table A.3 of Chapter A we see that Vigilante performs consistently better than Scargos in SCA verification, while SCA generation takes much more time in Vigilante than in Scargos.

Although both data sets are hard to compare, an observation can be made that in our data set, the worst-case performance in Scargos is about 45% better than that of Vigilante, while Vigilante has a better best-case performance.

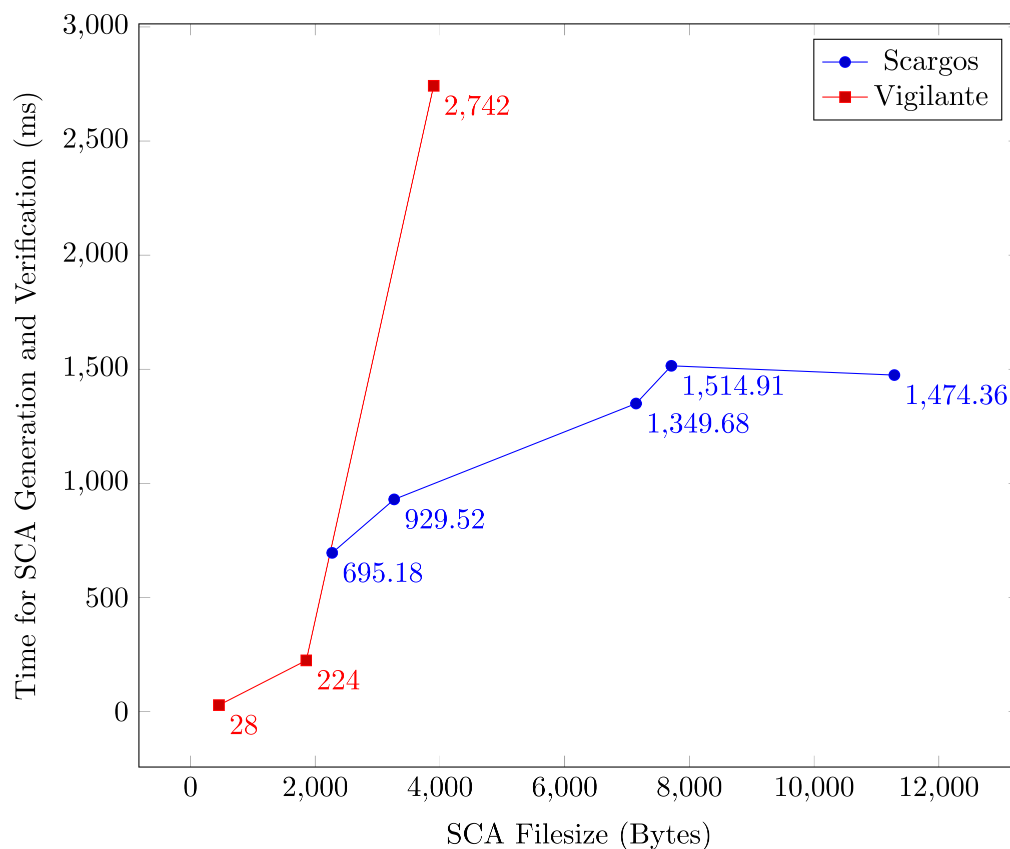

Fig. 6.7: Comparing Vigilante's and Scargos' performance for the combined times of SCA generation and verification excluding distribution.

6.2.4 Overall Performance

In this section we want to outline what the overall performance of Scargos by showing how much time it takes until a full life cycle of Scargos completes including distribution. Simply put, we measure the time form when a threat is detected and complete until it is verified by an verification manager and include each part of the process, which is specifically in the following order:

1. A threat is detected by a SCA publisher. *(start)*
2. An SCA is generated using stream extraction.
3. The SCA is published to an SCA repository.
4. The SCA repository notifies all interested SCA verifiers by using a push notification.

5. A SCA verifier downloads and replays (exact stream replay) the SCA to its DTA honeypot (PAM mirror).

6. The SCA's authenticity is verified. *(end)*

We decided to use exact stream replay and stream extraction as our algorithms because they allow higher accuracy and thus we were able to assess more applications. Our SCA repository was provided by an Amazon EC2 micro instance, which has 0.615 GiB memory and 1 virtual CPU. The average latency of 10 runs to this instance from our system was 6.86ms with a standard deviation of 0.11ms.

Figure 6.8 show the results of our experiments. It shows that Scargos operates fast. The majority of vulnerabilities of our applications could be verified and thus detected in under 2 seconds. On average, it takes 1993.88ms from the first detection of an attack by a honeypot until all interested SCA verifiers have verified the vulnerability on their systems. In the worst case, the process takes 6094.11ms (*Kolibri*), which should still give enough leeway to safely initiate a vulnerability response process.

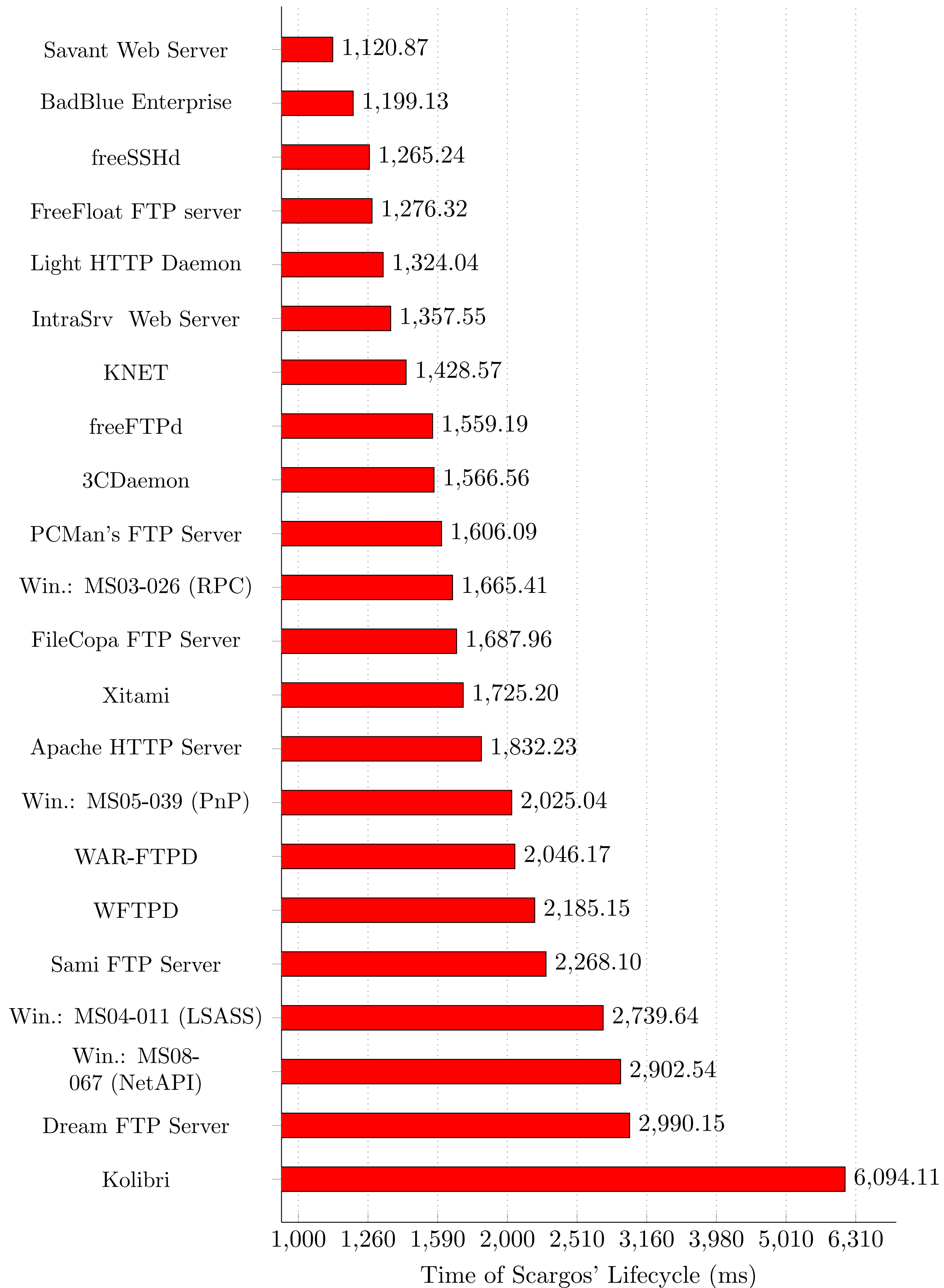

Fig. 6.8: Total time from SCA generation to successful verification including distribution. The results are the average of 10 runs. Standard deviations are shown in Table A.4.

6.3 Discussion

In this work, we want to answer the question: *Can packet-based SCAs be efficiently generated and verified?*

We show that packet-based SCAs are generated and verified correctly for all detected attacks of 24 applications. Furthermore, packet-based SCAs allow us to automate the distributing, receiving and initiating of appropriate actions for new vulnerabilities. A full life cycle to distribute a vulnerability can be achieved in under 2 seconds and thus can greatly accelerate current practices. Compared to the state-of-the-art non-packet-based SCAs our solution not only performs better for bigger SCA file sizes and thus more complex attacks, but also has a 45% better worst-case performance within our data set. We can thus confidentially answer the question with *yes*.

We also want to address all remaining questions that we have previously raised:

Can packet-based SCAs be successfully generated and verified for a variety of attacks?
We have show in Section 6.1 that using DTA honeypots all detected attacks can be successfully generated and verified using packet-based SCAs in conjunction with exact stream extraction and exact stream replay. Furthermore, it has to be noted that generation and verification can be done with only few limitations in a protocol-independent manner.

Do packet-based SCAs achieve the same accuracy as the state-of-the-art? Vigilante's SCA successfully generated and verified non-packet-based SCA for 3 out of 3 vulnerabilities. In this work, we evaluated packet-based SCAs with 24 different applications using many different protocols and operating systems. For all detected vulnerabilities we generated packet-based SCAs and correctly verified them, while using the same detection mechanisms as proposed by Vigilante. As a result, we can state that the accuracy of packet-based SCAs is at least the same as the state-of-the-art.

Can Scargos effectively reduce the available attack window of attackers? Currently, vulnerabilities are announced by primary sources which is usually the affected vendor (coordinated disclosure) or the group who discovered the vulnerability (full-disclosure). Due to the seemingly indefinite amount of possible primary sources, end users have to check manually for new vulnerabilities, and furthermore, rely often on secondary sources. As a result, the time until vulnerability information has reached end users is often hours and in the best case a couple of minutes. The

time before zero-day attacks are discovered is much higher and takes on average 312 days [4].

Our evaluation shows that an entire life cycle of Scargos is on average accomplished in under 2 seconds. Additionally, our implementation uses DTA honeypots, which can effectively detect zero-day attacks. Thus, Scargos can effectively reduce the attack window to a minuscule fraction of current best practices.

Do packet-based SCAs outperform the state-of-the-art for well-known attacks? In Section 6.2.3 we compare Vigilante's SCAs with packet-based SCAs. The results show that Scargos performs better for higher file sizes than Vigilante, while high profile attacks such as the very recent *Conficker* worm with an increasing amount of complexity lead to bigger SCA file sizes. We further show that packet-based SCAs outperform the worst-case performance of the state-of-the-art in our data set, although both solutions are hard to compare given the very different SCA file sizes.

Chapter 7

Conclusion

In this work, we presented Scargos, a framework to distribute vulnerabilities in seconds without needing a central authority. The goal of Scargos is it to shorten the attack window attackers have available to attack vulnerable applications. We propose packet-based SCAs, which differ from previously presented SCAs. Packet-based SCAs require that an attack be preserved in its original form, in packets; furthermore, they are independent from the detection engine used and allow for a custom vulnerability response process. We proposed and implemented an architecture for Scargos which uses the DTA honeypot Argos for detecting attacks. Packet-based SCAs are generated by using the logs of its detection engine.

We show in our work how packet-based SCAs can be used for a custom vulnerability response, which includes automatic generation of IDS signatures or malware analysis reports. We further outline the architectural components which are necessary to facilitate real-time vulnerability distribution, which are SCA repositories and push-notifications as well as the structure of packet-based SCAs.

Packet-based SCAs require different verification methods to previously known approaches. We present two replay mechanisms: brute-force replay and exact stream replay. While brute-force replay replays an attack to every possible input state of an application, exact stream replay replays an attack as it was originally received. Although, brute-force replay depends on the previous implementation of a protocol handler and is less accurate than exact stream replay, appropriate SCAs are generated 16 times faster and verification is completed on average 16% faster than when using exact stream replay.

Finally, we evaluated the overall performance of Scargos and packet-based SCAs. We showed that packet-based SCAs perform better for bigger SCA file sizes and seem to have a much better worst-case performance than conventional SCAs, while having

all previously mentioned advantages. Further, Scargos entire life cycle from the first detection until end-users verify an attack including distribution takes on average less than 2 seconds, which is a significant improvement compared to today's typically used vulnerability distribution methods, and which considerably shortens the attack window available to attackers.

Scargos can be the vaccine that helps globally contain malware, and it might be a solution that further slows down the continuing virtual arms race between attackers and defenders.

7.1 Future Work

In future work, improvements can be made to further increase accuracy, performance and usability of packet-based SCAs and Scargos. A specialised DTA honeypot could be developed or the honeypot Argos could be improved to reduce the amount of false negatives in attack detection that we have found. If the false negatives are due to under-tainting, new approaches to DTA could be used as suggested by Kang et. al [17]. Improvements of DTA honeypot could also be addressed to improve the accuracy of exporting the correct manipulated memory blocks and exploit Ethernet frames.

Research questions that could be addressed in future work with particular implications for Scargos include the following: how efficiently can a dual VM solution be utilised as a honeypot? Furthermore, an automatic way to estimate an optimal timeout value for our proposed exact stream replay algorithm would increase usability of our solution. Finally, we could investigate whether to combine the protocol independence of exact stream replay with the performance benefits of brute-force replay into a unified hybrid solution.

Bibliography

[1] Anagnostakis, K. G., Sidiroglou, S., Akritidis, P., Xinidis, K., Markatos, E., and Keromytis, A. D. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 14th USENIX security symposium* (2005), vol. 1.

[2] Arnold, M., and Ohlebusch, E. Linear time algorithms for generalizations of the longest common substring problem. *Algorithmica 60*, 4 (2011), 806–818.

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

[4] Bilge, L., and Dumitras, T. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 833–844.

[5] Bosman, E., Slowinska, A., and Bos, H. Minemu: The worlds fastest taint tracker. In *Recent Advances in Intrusion Detection* (2011), Springer, pp. 1–20.

[6] Chicago Honeynet Project. The Google Hack Honeypot, August 2013. `http://ghh.sourceforge.net/`.

[7] Clause, J., Li, W., and Orso, A. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis* (2007), ACM, pp. 196–206.

[8] Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Shannon, C., and Brown, J. Can we contain internet worms. In *Proceedings of the 3rd Workshop on Hot Topics in Networks (HotNets-III)* (2004), Citeseer.

[9] Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., and Barham, P. Vigilante: End-to-end containment of internet worms. In *ACM SIGOPS Operating Systems Review* (2005), vol. 39, ACM, pp. 133–147.

[10] Crandall, J. R., and Chong, F. T. Minos: Control data attack prevention orthogonal to memory model. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on* (2004), IEEE, pp. 221–232.

[11] Faulhaber, J., Lambert, J., Probert, D., Srinivasan, H., Felstead, D., Lauricella, M., Rains, T., and Stewart, H. Microsoft security intelligence report. Tech. Rep. 11, Microsoft Corporation, Redmond, WA 98052-6399, February 2011.

[12] Ferrie, P. Attacks on more virtual machine emulators. *Symantec Technology Exchange* (2007).

[13] Fielding, R. T. *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, 2000.

[14] Garcia, L. M. Programming with libpcap±sniffing the network from our own application. *Hakin9-Computer Security Magazine* (2008), 2–2008.

[15] Gusfield, D. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press, 1997.

[16] Halfond, W. G., Orso, A., and Manolios, P. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering* (2006), ACM, pp. 175–185.

[17] Kang, M. G., McCamant, S., Poosankam, P., and Song, D. Dta++: Dynamic taint analysis with targeted control-flow propagation. *Proc. of the 18th NDSS* (2011).

[18] Kobayashi, T. H., Batista, A. B., Brito, A., and Pires, P. S. M. Using a packet manipulation tool for security analysis of industrial network protocols. In *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on* (2007), IEEE, pp. 744–747.

[19] Kohlrausch, J. Experiences with the noah honeynet testbed to detect new internet worms. In *IT Security Incident Management and IT Forensics, 2009. IMF'09. Fifth International Conference on* (2009), IEEE, pp. 13–26.

[20] Kontaxis, G., Polakis, I., Antonatos, S., and Markatos, E. P. Experiences and observations from the noah infrastructure. In *Computer Network Defense (EC2ND), 2010 European Conference on* (2010), IEEE, pp. 11–18.

[21] Kortchinsky, K. Cloudburst - a vmware guest to host escape story. BlackHat USA 2009, Las Vegas, USA.

[22] Kreibich, C., and Crowcroft, J. Honeycomb: creating intrusion detection signatures using honeypots. *ACM SIGCOMM Computer Communication Review 34*, 1 (2004), 51–56.

[23] Lam, L. C., and Chiueh, T.-c. A general dynamic information flow tracking framework for security applications. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual* (2006), IEEE, pp. 463–472.

[24] Leyden, J. Scada honeypots attract swarm of international hackers. *The Register* (Mar. 26, 2013).

[25] Liston, T. Labrea: Sticky honeypot and ids, May 2009. http://labrea.sourceforge.net/labrea-info.html.

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

[27] Newsome, J., and Song, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software.

[28] Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., and Zagorodnov, D. The eucalyptus open-source cloud-computing system. In *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on* (2009), IEEE, pp. 124–131.

[29] OpenVAS project. Openvas - about openvas, May 2013. http://www.openvas.org/about.html.

[30] OpenVAS project. Openvas - nvt development, May 2013. http://www.openvas.org/nvt-dev.html.

[31] Orebaugh, A., Ramirez, G., and Beale, J. *Wireshark & Ethereal network protocol analyzer toolkit*. Syngress, 2006.

[32] Portokalidis, G., Slowinska, A., and Bos, H. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *ACM SIGOPS Operating Systems Review* (2006), vol. 40, ACM, pp. 15–27.

[33] Provos, N. Honeyd-a virtual honeypot daemon. In *10th DFN-CERT Workshop, Hamburg, Germany* (2003), vol. 2.

[34] Provos, N., and Holz, T. *Virtual honeypots: from botnet tracking to intrusion detection*, third ed. Addison-Wesley Professional, 2009.

[35] Schwartz, E. J., Avgerinos, T., and Brumley, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 317–331.

[36] Sourcefire, I. Snort, May 2012. http://www.snort.org/.

[37] Spitzner, L. Honeypots: Catching the insider threat. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual* (2003), IEEE, pp. 170–179.

[38] Suh, G. E., Lee, J. W., Zhang, D., and Devadas, S. Secure program execution via dynamic information flow tracking. In *ACM SIGPLAN Notices* (2004), vol. 39, ACM, pp. 85–96.

[39] Sullivan, B. Sasser infections begin to subside. *NBC News* (May 5, 2004). http://www.nbcnews.com/id/4890780/ns/technology_and_science-security/t/sasser-infections-begin-subside/#.UhANu3byrUI.

[40] The MITRE Corporation. Cve - about cve, May 2013. http://cve.mitre.org/about/index.html.

[41] The MITRE Corporation. Cve - cve-id syntax change, August 2013. http://cve.mitre.org/cve/identifiers/syntaxchange.html.

[42] The MITRE Corporation. Oval - oval use cases guide, May 2013. http://oval.mitre.org/adoption/usecasesguide.html.

[43] The Nmap Security Scanner Project. Vulnerability scanners sectools top network security tools, May 2013. http://sectools.org/tag/vuln-scanners/.

[44] Venkataramani, G., Doudalis, I., Solihin, Y., and Prvulovic, M. Flexitaint: A programmable accelerator for dynamic taint propagation. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on* (2008), IEEE, pp. 173–184.

[45] Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)* (2007), vol. 42.

[46] Vrije Universiteit Amsterdam. Argos - an emulator for capturing zero-day attacks, May 2012. http://www.few.vu.nl/argos/.

[47] Wicherski, G. Medium interaction honeypots. *German Honeynet Project (April 2006)* (2006).

[48] Willems, C., Holz, T., and Freiling, F. Toward automated dynamic malware analysis using cwsandbox. *Security & Privacy, IEEE 5*, 2 (2007), 32–39.

[49] Xu, W., Bhatkar, S., and Sekar, R. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium* (2006), pp. 121–136.

Appendices

Appendix A

Additional Experimental Results

Application Name	FTP Attack Vector
3CDaemon	User
Dream FTP Server	User
FileCopa FTP Server	List
FreeFloat FTP server	User
Sami FTP Server	List
WAR-FTPD	User
WFTPD	Size
PCMan's FTP Server	User

Table A.1: Attack Vectors of vulnerable FTP applications.

Application Name	SCA size in Bytes	SCA Generation (ms)	SCA verification (ms)	Total (ms)
Win.: MS03-026 (RPC)	2272	488.12	207.06	695.18
Win.: MS05-039 (PnP)	3266	487.42	442.1	929.52
Apache HTTP Server	7145	851.3	498.38	1349.68
Win.: MS08-067 (NetAPI)	7711	760.74	754.17	1514.91
Win.: MS04-011 (LSASS)	11287	980.18	494.18	1474.36

Table A.2: Performance of Scargos for selected well-known applications.

Application Name	SCA size in Bytes	SCA Generation (ms)	SCA verification (ms)	Total (ms)
Microsoft SQL Servers	457	18	10	28
Win.: MS03-026 (RPC)	1857	206	18	224
Microsoft IIS Server	3899	2667	75	2742

Table A.3: Performance of Vigilante [9].

Application Name	Exact Stream Extraction (ms)	Exact Stream Replay (ms)	Total Time of Scargos' Lifecycle (ms)	Single Packet Extraction (ms)	Brute-Force Replay (ms)
3CDaemon	19.08	6.15	28.08	—	—
Dream FTP Server	12.74	41.93	445.86	—	—
FileCopa FTP Server	6.4	31.12	37.19	19.57	6.9
FreeFloat FTP server	18.3	7.37	92.41	3.43	30.82
Sami FTP Server	13.16	268.75	865.55	25.23	257.09
WAR-FTPD	9.56	430.15	430.67	45.65	504.07
WFTPD	12.38	96.3	103.3	30.08	35.15
Savant Web Server	18.76	126.91	146.01	88.62	10.46
Win.: MS05-039 (PnP)	6.45	50.79	63.55	—	—
PCMan's FTP Server	18.38	52.35	57.39	—	—
freeSSHd	24.1	74.31	83.59	—	—
freeFTPd	10.49	41.81	979.9	—	—
Apache HTTP Server	5.04	14.13	37.66	—	—
BadBlue Enterprise	15.77	11.73	66.04	—	—
Light HTTP Daemon	8.64	22.28	48.58	1.2	7.83
KNET	10.8	7.26	27.79	17.66	8.53
Kolibri	7.35	90.06	133.71	1.2	10.41
Xitami	9.83	233.07	248.05	16.58	223.25
Win.: MS03-026 (RPC)	8.39	36.63	65.05	—	—
Win.: MS04-011 (LSASS)	8.94	41.03	52.06	—	—
IntraSrv Web Server	6.81	137.75	623.55	—	—
Win.: MS08-067 (NetAPI)	17.35	138.34	142.15	—	—

Table A.4: Standard Deviation of 10 runs of all investigated applications and metrics. The total time of Scargos' lifecycle was being measured using Stream Extraction and Stream Replay.

Application Name	*CVE or OSVDB*	*Attack*
3CDaemon	CVE-2005-0277	Metasploit: exploit/windows/ftp/3cdaemon_ftp_user
Dream FTP Server	CVE-2004-0277	Metasploit: exploit/windows/ftp/dreamftp_format
FileCopa FTP Server	CVE-2006-3726	Metasploit: exploit/windows/ftp/filecopa_list_overflow
FreeFloat FTP server	OSVDB-69621	Metasploit: exploit/windows/ftp/freefloatftp_user
Sami FTP Server	CVE-2008-5106	Metasploit: exploit/windows/ftp/sami_ftpd_list
WAR-FTPD	CVE-2007-1567	Metasploit: exploit/windows/ftp/warftpd_165_user
WFTPD	CVE-2006-4318	Metasploit: exploit/windows/ftp/wftpd_size
Savant Web Server	CVE-2002-1120	Metasploit: exploit/windows/http/savant_31_overflow
PCMan's FTP Server	CVE-2013-4730	Exploit-DB-ID: 26471
freeSSHd	CVE-2006-2407	Metasploit: exploit/windows/ssh/freesshd_key_exchange
freeFTPd	CVE-2006-2407	Metasploit: exploit/windows/ssh/freeftpd_key_exchange
Apache HTTP Server	CVE-2002-0392	Metasploit: exploit/windows/http/apache_chunked
BadBlue Enterprise Edition	CVE-2007-6377	Metasploit: exploit/windows/http/badblue_passthru
Light HTTP Daemon	CVE-2002-1549	Exploit-DB-ID: 24999
KNET	CVE-2005-0575	Exploit-DB-ID: 24897
Kolibri	CVE-2002-2268	Metasploit: exploit/windows/http/kolibri_http
Xitami	CVE-2007-5067	Metasploit: exploits / windows / http / xitami_if_mod_since
Windows: MS03-026 (RPC)	CVE-2003-0352	Metasploit: exploit/windows/dcerpc/ms03_026_dcom
Windows: MS04-011 (LSASS)	CVE-2003-0533	Metasploit: exploit/ windows/smb/ms04_011_lsass
Windows: MS05-039 (PnP)	CVE-2005-1983	Metasploit: exploit/windows/smb/ms05_039_pnp
Mercury/32 Mail	CVE-2007-4440	Metasploit: exploit/windows/smtp/mercury_cram_md5
IntraSrv - Simple Web Server	OSVDB-94097	Exploit-DB-ID: 25836
Mdeamon PRO	CVE-2004-1520	Metasploit: exploit/windows/imap/mdaemon_cram_md5
MS08-067 (NetAPI)	CVE-2008-4250	Metasploit: exploit/windows/smb/ms08_067_netapi

Table A.5: Attacks and vulnerabilities used in our experiments.