Operational Programme "Competitiveness"

R&D Cooperations with Organizations of non-European Countries

EAR: Early warning system for automatic detection of Internet-based cyberattacks
(Code G.S.R.T.: ΗΠΑ-022**)**

D2.1 "System Design"

Abstract: This document describes the design of the early warning system and presents the results of preliminary measurements.

Contractual Date of Delivery	12 January 2005
Actual Date of Delivery	12 January 2005
Deliverable Security Class	Public
Editor	Periklis Akritidis

The EAR Consortium consists of:

FORTH	Coordinator	Greece
GA Tech	Principal Contractor	USA
FORTHnet	Principal Contractor	Greece

Contents

List of Figures

List of Tables

Introduction

In this document, we describe the design of the EAR Early Warning System for Internet Epidemics satisfying the requirements layed out in the Specification document.

The rest of the document is organized as follows. In Chapter 1 we give an overview of the system. In Chapter 2 we describe the main module of the system, and in Chapter 3 the rest of the modules. In Chapter 5 we present preliminary measurements which helped in the design and calibration of the detection techniques. In Chapter 5, we present preliminary measurements, and in Chapter 7 we describe future enhancements and optimizations. In Chapter 8 we give a timeline for the implementation of the modules. In Chapter 9 we show how the design conforms to the requirements.

Chapter 1

Overview of the Architecture

In this Chapter, we describe the overall architecture of the system. We provide a system metaphor and an overview of the system's modules.

1.1 System Metaphor

The basic function of the system is to passively monitor network traffic and identify strings that belong to worms and can be used as signatures for filtering worms.

The main objectives of the system are (a) to detect previously unknown Internet worms, (b) to detect them in time, (c) to detect them without false positives, and (d) to detect them without human intervention.

1.2 Architecture

The system operates by detecting substrings in network stream contents that are sent to more than a number of destination hosts within a certain period of time. A fundamental condition for the system to detect a worm is that the attack must contain an invariant string that can be used as a signature. This functionality is provided by the main filter module, and is described in Chapter 2.

A number of additional heuristics employed to boost performance and reduce false positives are described in Chapter 3. They are briefly listed in Table 1.1. Depending on their computational cost and their parallelism, they can be placed either before, or after the main filter. Computationally cheap filters are used to preprocess traffic and reduce the strain on the main module. On the other hand, computationally expensive mechanisms can be applied on the results of the main filter to weed out false positives, or, if they are applicable on individual streams without the need for communication, they can be applied before the main filter on a cluster of processing nodes.

Some worm detection heuristics can be evaded by future worms, but using them could nevertheless boost the detection of worms that are still susceptible. Also sometimes evading such a heuristic comes with a cost to the attacker, such as

Filter	Placement
Sled Detection	Depends
Client Traffic	Pre
Address White-list	Pre
Content White-list	Inside Main
Scan Detection	Pre
ASCII nul	Pre
Replay	Post
Flow Limit	Pre

Table 1.1: Additional worm detection heuristics.

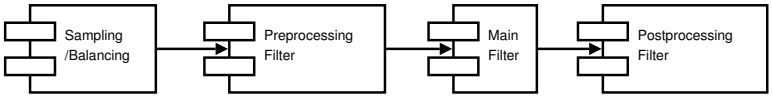

Figure 1.1: An overview of the system. Traffic is first processed by a flow sampling/balancing module, then by the preprocessing filters, then by the main worm detection module, and then by the postprocessing filters.

a slowdown in spreading speed. To take advantage of such heuristics without sacrificing detection of more elaborate worms we propose using a two-tier approach where instances of the worm detection system configured to higher sensitivity but using the above mentioned heuristics and therefore blind to elaborate attacks are used in combination with instances configured for less sensitivity but capable to detect elaborate attacks.

Traffic is first processed by a flow sampling and load balancing module that picks an appropriate portion of the flows and distributes them to nodes running preprocessing filters. These filters can be run separately and so we plan for the possibility of distributing the computation on many machines. The output of these filters is fed to the main filter, which searches for worm substrings. The generated alerts are then sent to a notification mechanism, a worm containment mechanism, or an alert verification mechanism. These mechanisms are beyond the scope of this system. This flow of information through the components of the system is presented in Figure 1.1.

The system will be implemented as plugin to the Snort NIDS. This allows the reuse of infrastructure provided by Snort such as connection tracking, stream reassembly, and reporting. Appropriate scripts will be provided for hiding the complexity introduced by the dependence on Snort.

Chapter 2

Main Module

2.1 Worm Detection Algorithm

In this section we present a worm detection method based on three observations which are commonly found in known worms:

- **Diversity of Destinations:** The network packets that belong to the same worm tend to have a very large number of destinations. Actually, this seems to be an inherent property of all the worms: worms tend to spread to as many victims as possible, and therefore, their network packets seem to have a large number of destinations.
- **Spread by Clients:** Most worms are usually spread by clients, i.e. by computers that initiate a (usually TCP) connection. This property, as well, seems to be an inherent property of the aggressive worms. Indeed, in order for a worm to spread fast, it needs to initiate connections to its potential victims, rather than to wait for the potential victims to connect to it.
- **Payload Repeatability:** Several of the network packets that belong to the same worm, tend to contain similar (if not identical) payloads. [1]

Well-known worms such as the CODE-RED, the Blaster/Welchia, the Slammer/Sapphire, and the Witty worm [13], depicted all the above three properties. Indeed, all CODE-Red, Blaster, and Slammer worms used identical packets to propagate to as many destinations as possible. Interestingly enough, the Witty worm used a naive form of polymorphism as it used random padding of its network packets. Besides having payload repeatability, all the above worms have also shown diversity of destinations as they propagated to many different destinations on packets that had lots of common substrings in their payload.

[1] It has been proposed that future worms will be polymorphic and will be able to change the payload of the network packets that carry the worm. The detection of such worms is outside the scope of this work.

```
for each reassembled packet in trace
  for each fingerprint in packet
    if fingerprint in queue
      if packet destination not recorded
        record destination
        increase count
        if count > threshold
          report
          reset counter
          clear destinations
        promote to front
    else
      if queue full
        evict last queue entry
      create new entry for fingerprint
      insert at front
```

Table 2.1: Pseudocode for worm fingerprinting algorithm without sampling.

Thus, to identify new worms, our algorithm identifies common substrings that appear in the payloads of several (client) packets, which are heading for lots of different destinations. To identify such packets we use an LRU queue of fixed size. Each node of the queue contains an encountered string[2] together with its occurrence count and a list of distinct destinations where it was sent to. The nodes are ordered in the queue by time of last occurrence and they are also indexed by string using a hash table.

Our algorithm operates as follows:

- For each encountered string, if a corresponding queue entry exists and its distinct destination list does not include the current destination, the entry is promoted to the front, the current destination is recorded in the entry's distinct destination list, and its distinct destination count incremented. The distinct destination count is then compared with a distinct destination threshold and an alert is issued when the threshold is reached.
- Otherwise, if the string is not already represented in the queue, a new entry is inserted in the front of the queue and the last entry of the queue is evicted.

The pseudocode can be found in table 2.1

2.1.1 Client Traffic

Given that rapidly-spreading worms spread mostly through clients, and not through servers, in our implementation we wil discard server replies and process only client

[2]In order to save space the queue contains a 32-bit fingerprint of the string.

requests.

Only traffic originated from clients is examined. The rationale is that the attack of an actively spreading worm (the kind of worm for which human mediated responses are too slow) will be contained in the part of a session sent by the client who initiated the connection to a victim server. The gain from this heuristic is twofold: (a) a huge reduction of the amount of traffic that has to be examined, because the bulk of the data is usually in a response and not in a request (consider typical http transfers), and (b) a reduction of false positives, since data contained in replies is sent to multiple destinations and also is of sufficient length, and would therefore easily trigger detection based on the criteria of multiple destinations and sufficient length. Also, actual content distributed through p2p networks is typically downloaded through connections initiated for that purpose, and is therefore prevented from causing a false positive.

We will rely on Snort's [11] session tracking to decide the direction of a packet.

2.1.2 Repetitive Packets vs. Repetitive Strings

Many worms are spread using identical packet payloads, and therefore can be easily detected by identifying repetitive packets seen in the network. However, sometimes entire packets may be too course-grained for worm detection. For example, the Witty worm [13], has actually implemented random padding of packets. Therefore, in our approach, to identify payload repeatability, we will consider packet substrings of a fixed length, instead of entire packets.

2.1.3 Substring Length

False positives due to peer-to-peer systems can be categorized either as protocol messages or as downloads of popular content. An important observation is that protocol messages have a relatively small length. which leads to a solution to the false positives problems, we can Requiring that the substrings have sufficient length can weed out such repetitive protocol messages. The same solution can be applied for false positives caused by common headers of requests.

The substrings considered are required to be sufficiently long. Statistically, shorter substrings are bound to occur more frequently. Moreover, protocol messages (such as those used by p2p networks) and requests (e.g. http requests) are typically short in length.

Many recent worms operate in two phases, first an attack phase injects code to the victim and then the injected code connects back to the infecting host to download the rest of the worm. The client-only filter will discard such downloads, and therefore the system must not rely on them, but instead rely on the attack part. Table 2.3 presents the lengths of whole worms and of their initial attack parts.

Using large strings has the added benefit that the resulting signatures are less probable to collide with legitimate traffic.

Table 2.2: Traffic types and detection heuristics. The combination of multiple criteria can separate worm traffic among other types that could cause false positives.

	Long	Sent to many	Sent from client
P2P messages	No	Yes	Yes
P2P downloads	Yes	Yes	No
Server requests	No	Yes	Yes
Server replies	Yes	Yes	No
Worm	Yes	Yes	Yes

2.1.4 Multiple Destinations vs. Source-Destination Pairs

Intuitively, worm traffic is best characterized by a large number of distinct source-destination pairs. However, to detect a worm we chose to consider the number of distinct destinations instead, because multiple source-destination pairs are also characteristic of traffic caused by client requests to popular (web) servers. Worms not only cause traffic with a large number of source-destination pairs, but in order to spread, also target a large number of destinations. In the early stages of worm spread, when re-infection attempts are less likely, the two numbers should be roughly equal. Note that the traffic caused by the server replies, which, like worm traffic, is also characterized by a large number of destinations, is not a concern, since we specifically filter out such traffic.

2.1.5 Stream Reassembly

Clever attackers may easily hide their attack into several different fragmented packets that may be sent out-of-order. To solve this problem, we plan to use the packet reassembly mechanisms provided by the Snort NIDS [11]. We will integrate our filters with Snort in the form of a Snort preprocessor plugin. This way we can take advantage of the existing stream4 preprocessor that comes with Snort and reassembles raw packets into larger ones. The reassembled packets are then fed to the worm fingerprinting algorithm.

2.1.6 Performance

Counting of Destinations

For each encountered substring, the system records all the destinations to which it has been sent. Fortunately, most substrings will only be sent to a single destination before they are evicted from the queue and therefore the space required for recording more than one destination does not have to be allocated for the majority of the encountered substrings.

Rabin Fingerprints

The system has to process substrings included in network streams. It considers all substrings of a certain length, and has to store them and be able to look them up. To perform these operations, it has to compute a hash for each substring, and use it as a key.

To efficiently compute the hash values of consecutive overlapping packet substrings we employ Rabin fingerprints [10]. The Rabin fingerprint f_α of an n-gram a is computed according to the formula $f_\alpha(a_0, a_1, \ldots, a_{n-1}) = \sum_{i=0}^{n-1} a_i \alpha^{n-i-1}$. Rabin fingerprints can be used to incrementally update the hash value of a sliding window over the packet payload, by considering the contribution to the hash of the next byte and removing the contribution to the hash of the last byte of the previous window according to property $f_\alpha(a_1, a_2, \ldots a_n, a_{n+1}) = \alpha(f_\alpha(a_0, a_1, \ldots, a_n) - \alpha^n a_0) + a_{n+1}$. In order to be able to perform the arithmetic modulo 2^{32}, α must not be a power of two.

However, even Rabin fingerprints employ a constant number of operations for each and every byte of network traffic. Given that modern networks deliver up to 10 Gigabytes of traffic per second, even Rabin fingerprints impose a substantial computational overhead to the system. Further more, having to account for each and every overlapping substring is undesirable. We use three mechanisms to reduce these overheads in our detection scheme:

- Flow limit
- Discard server replies
- Fingerprint sampling

Flow Limit

The worm attack is typically carried out in the first few kilobytes of connections created by the worm with the purpose of infecting other machines. Therefore, a very effective performance optimization is to discard traffic known not to belong to the initial traffic of flows.

Discard Server Replies

We have already mentioned in section 2.1.1 that we discard server replies and focus only on client requests as a measure to reduce false positives. The same mechanism is used to improve performance as well. Indeed, server replies are typically large contributors to Internet traffic. Focusing our detection algorithm in client requests, instead of server replies, we reduce the load of our detection mechanism by having to compute fewer Rabin fingerprints without reducing its accuracy.

Table 2.3: Total length, length of the attack portion, and protocols for various worms.

Worm	Total Length	Attack Length	Protocol
Witty	600 Bytes (+padding)	600 Bytes	UDP
Sapphire/Slammer	376 Bytes	376 Bytes	UDP
CodeRedII	3,8KBytes	3,8KBytes	TCP
Welchia	10KBytes	1,7KBytes	TCP

Fingerprint Sampling

Processing each substring that starts at each and every byte of the network traffic, may result in unnecessary overhead. For example, assume a network packet that is $packetsize$ characters long. Assume also that our algorithm searches for substrings which are n-bytes long. Then, for this specific example, we will end up processing $packetsize - n + 1$ substrings. Given that these substrings are highly *overlapping*, we might be able to reduce the overhead of our approach without reducing its accuracy. For example, instead of considering all Rabin fingerprints we may sample them based on their value [14]. The frequency of the sampling can be determined by the number of bits set in a sampling mask. The fingerprint it further processed only if the result of applying the mask to it is not zero. A string that matches the sampling criteria is always sampled, so no loss of sensitivity is incurred and no memory lookup is required to determine whether a string is sampled or not.

With this technique the number of strings contained in a worm that are visible to the system is reduced by a factor of 2^m, where m is the number of bits set in the sampling mask. Judging from the data in Table 2.3, reducing the number of strings by a factor of 4-16 is safe enough. The critical length is that of the attack, since the rest of the worm body may be downloaded from the infecting machine and thus be ignored based on the *sent by client* filter.

Note, that is is theoretically possible for an attacker to exploit this sampling mechanism. Indeed, if the attacker knows the exact value of the mask, (s)he will create worms whose content will never match the mask and therefore will never be sampled. On the other hand, the mask bits could be shuffled at regular intervals, so that it would be practically impossible for an attacker to avoid detection by carefully crafting the body of the worm to avoid sampling.

Chapter 3

Other Modules

In this chapter, we describe functionality that is not essential, but rather is an addition to the main functionality. This extra functionality takes the form of additional filters, which can be applied either before the main filter or after it, depending on their computational cost. Filters which are cheap to apply can offload the main filter and therefore should be applied before it. On the other hand, filters which introduce considerable overhead should only be applied on suspect strings.

3.1 Buffer Overflow Detection

Most worms rely on buffer overflow vulnerabilities for their spread, and several buffer overflow detection mechanisms have been proposed [15, 9]. These mechanisms, however, are not without false positives, but can help to improve the overall robustness of the system. In the context of the EAR project we have designed our own buffer overflow attack detection mechanism, called STRIDE, which is described in Section 4.

Buffer overflow detection can be relatively costly, so it could be applied after anomaly detection. However, it has extreme locality, since each network stream can be processed individually. So if the cost can be distributed to a number of processors, it is reasonable to place buffer overflow detection before worm substring identification.

3.2 ASCII Nul Filter

Typical buffer overflow attacks must not contain the ASCII nul character because it would prevent the overflow by terminating the copying into the vulnerable buffer.

Checking for the ASCII nul character is a very cheap filter, and can be applied as a filter within the processing loop of the main module. A counter will have to be maintained along with the incremental computation of Rabin fingerprints. The counter is increased on every processed byte, except on ASCII nul, in which case

it is reset to zero. The computed fingerprints are only considered further if the counter has a value greater than the assumed minimum vulnerable buffer size.

3.3 Scanning Detection

Worms to day have used variations of random scanning as a method of finding targets to infect. Detecting nodes that frequently probe other nodes, and then processing only their traffic, could be applied as a filter before worm substring identification. However, hitlist worms, worms that use a precompiled list of targets, can minimize failed connections and possibly evade scanning detection. On the other hand, it should be possible to detect scanning worms faster, if the scanning is exploited by the detection mechanisms. Therefore, we plan to allow using an optional scanning detection preprocessing filter, and recommend running two instances of the system, one calibrated to high sensitivity to detect scanning worms and one less sensitive but not blind to worms that evade scan detection.

3.4 Flow Limit

Arguments similar to the ones used for scanning detection hold for the flow limit heuristic, which relies on the observation that all worm attacks to day were contained in the initial portion of the stream that carried them. It can be evaded, but it can give a boost to detection of worms which are exposed.

3.5 Grouping per Destination Port

Worms to day have targetted specific ports. This can be exploited by the system by not relating strings that are sent to different ports. However, a worm that spreads using multiple ports, for example a topological worm that spreads from one peer-to-peer node to its neighbours by targeting the peer-to-peer service itself, would have its strings appears as belonging to several less aggressive worms (one for each port) and perhaps evade detection.

3.6 White-listing

Known content strings that cause false positives can be white-listed.

Additionally, in a LAN environment certain hosts can cause false positives. All traffic from these hosts hosts could be filtered out.

3.7 Flow Sampling

The system should monitor as much traffic as possible in order to detect an outbreak early and reliably. As the amount of monitored traffic increases, temporary and

local popularity bursts are flattened out and a worm outbreak can be detected easier.

It is usefull, however, to monitor more traffic that the amont require for timely detection because of the concern of resisting against faked worm outbreaks. With a low threshold, a few connections are enough for triggering a worm alert, but this problem can be handled by sampling as much flows as are required for reliable detection from a much larger pool of flows. Only the sampling device has to operate at the speed of the high bandwidth line. Without knowing which flows are sampled, the attacker would have to create enough worm-like flows against the entire traffic.

Chapter 4

Sled Detection

In this chapter we describe STRIDE, our new sled detection mechanism which, compared to previous approaches, is able to detect more types of sleds with less false positives.

4.1 Classification of Sleds

The sled is a sequence of instructions responsible for directing the flow of control towards the core code of a buffer overflow attack. Although execution of the sled can start at any position, it always ends up "sliding" inside the core code of the attack. There are many different ways for a sled to achieve its functionality. In this section, we present several types of sleds in order of increasing (perceived) difficulty to detect.

4.1.1 Simple NOP Sled

The simplest sled consists of a series of NOP (no-operation) instructions. A NOP instruction has no effect on program behavior: it simply advances the program counter. Execution of the sled may start at any position, and the NOPs are used to transfer control, step by step, to the shellcode right after the sled. This simple sled has been demonstrated in the buffer overflow examples of [4] and has been used in many other attacks.

4.1.2 One-byte NOP-equivalents Sled

A NOP sled can be easily obfuscated by replacing literal NOP instructions with one-byte instructions which have no significant effect, and, for the purposes of the attacker, are practically equivalent to NOPs. For example, instructions that increase or decrease a register which is not used by the attacker, instructions that set or clear a flag, and instructions that push or pop a register, can all be used in a sled instead of NOPs.

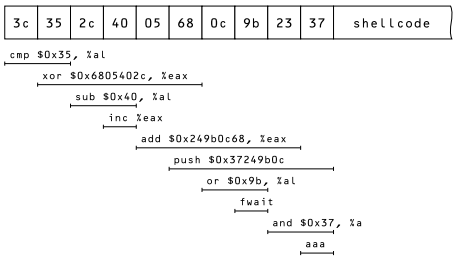

Figure 4.1: An example of a small sled, executable at every byte offset, which is constructed by interleaving one-byte and multi-byte NOP-equivalent instructions.

Current polymorphic buffer overflow attack generators use such sleds to avoid detection. The ADMmutate [6] engine uses this technique with a list of of 55 one-byte NOP-equivalent instructions. The Metasploit framework [3] extends the ADMmutate engine with 3 additional single-byte NOP replacements. We have enumerated 66 such instructions in the Intel IA-32 architecture [1]. Although not yet seen in the wild, obfuscated sleds are already available to attackers.

4.1.3 Multi-byte NOP-equivalents Sled

A straightforward extension to one-byte NOP-equivalent sleds is to use multi-byte NOP-equivalent instructions, which, like their one-byte counterparts, simply just advance the program counter in order to reach the core of the exploit. However, it is not possible to use *any* multi-byte NOP equivalent instruction available in the instruction set, because a sled must be executable at *every* offset. Therefore, a straightforward way to generate multi-byte NOP-equivalents sleds is to restrict the operands of multi-byte instructions to correspond only to the opcodes of one-byte NOP-equivalent instructions, or to the opcodes of multi-byte NOP-equivalents. Consider for example the multi-byte NOP-equivalents sled shown in Figure 4.1. If control is transferred to the leftmost byte, it will execute instructions `cmp $0x35, %al`, `sub $0x40, %al`, `add $0x249b0c68, %eax`, etc. Note that the first argument of the first instruction `cmp $0x35, %al`, is `0x35`, which corresponds to the opcode of instruction `xor`. Therefore, if control is transferred to the penultimate byte from the left, it will execute instructions `xor`, `or`, `and`, etc. leading to the end of the sled. This is true for all instructions in this type of sleds: their arguments are such that if control is transferred to any byte inside the sled, the execution will eventually lead to the end of the sled.

4.1.4 Four-byte Aligned Sled

Although traditional NOP sleds had to be executable at each and every byte, stack alignment can relax this restriction by constraining the possible placements of the vulnerable buffer. The default behavior of modern compilers is to align the stack at

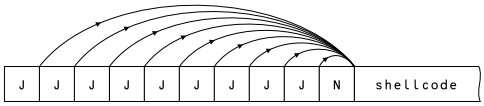

Figure 4.2: The ideal trampoline-sled: flow of control is directed to the shellcode in a single step from any position in the sled.

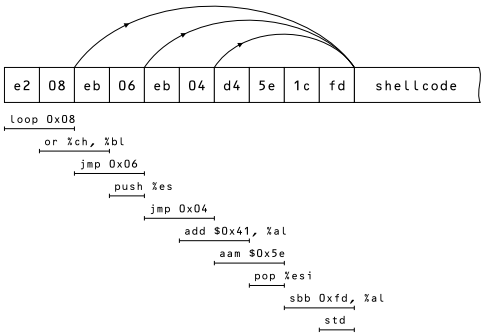

Figure 4.3: An example of a small trampoline-sled that is executable at every byte offset. Control transfer instructions are placed at every second byte and their relative address operand is chosen so that it is a valid NOP-equivalent opcode.

word (4-byte) boundaries [5]. Reference [7] discusses the possibility of exploiting stack alignment to construct sleds that have to be executable every 4 bytes. Pairs of non-destructive 2-byte instructions can be used as NOP-equivalents, but it is also possible to use longer instructions with techniques similar to the multi-byte instruction sled discussed earlier. Code sequences starting at non-word-aligned offsets may contain any kind of instruction, including instructions with destructive side-effects or even illegal ones, which can hinder detection.

4.1.5 Trampoline-sled

Although typical sleds transfer control to the shellcode by sliding it along their body —hence the name sled—, the same functionality can be achieved by jumping directly to the shellcode, as illustrated in Figure 4.2. The body of such a sled consists of control transfer instructions with relative addresses, all pointing directly to the shellcode. As a result, the flow of control will reach the shellcode in a *single* step from any point it may have entered the sled.

Trampoline-sleds can be directly implemented, relying on four-byte alignment, by cramming a jump instruction together with its operands into every four-byte-long slot of the sled. Given that trampoline-sleds have to be executable at every offset, they must carefully chose the operands of the jump instructions to be valid NOP-equivalent opcodes, as explained in Section 4.1.3 An example of a small trampoline-sled that is executable at every byte offset is illustrated in Figure 4.3.

The shortest control transfer instructions available are two bytes long. For ex-

ample, instructions such as `jmp` and `loop` take a one-byte operand that specifies the relative address of the jump target. The use of two-byte control transfer instructions places an additional restriction on the maximum jump displacement that can be used for sleds executable at each byte. Generally, the operand of these instructions is encoded as a signed 8-bit immediate value, which allows for a maximum forward relative offset of 127 bytes. Additionally, since the operand must at the same time act as a one-byte NOP-equivalent instruction, the maximum jump displacement is further reduced to the NOP-equivalent opcode with the greater signed integer value that is less than 128. The two NOP replacements with the largest such opcodes that we have come across are `push imm8` and `push imm32`, which result to an offset of 106 and 104 bytes, respectively. Trampoline sleds are still feasible, though, by solely using jumps with relatively large positive displacements, which result to forward execution "bounces". Thus, the flow of control "jumps" and "strides" towards the shellcode.

4.1.6 Obfuscated Trampoline-sled

Since the number of control transfer instructions that can be used for the construction of trampoline-sleds is limited, one could argue that such sleds can be detected by searching for the specific opcodes of these instructions, much in the same way that Fnord does for NOP-equivalents (cf. Section 4.2.2).

The entropy of the basic trampoline-sled can be increased in order to evade detection, by interleaving NOP-equivalent instructions along with the jump instructions. In this way, the shellcode is not reached in a single step, but in a number of steps which can be tuned by the attacker. This will result to a sparse distribution of the control transfer instructions, which renders simple detection methods ineffective.

4.1.7 Static Analysis Resistant Sleds

Sleds of this type attempt to evade detection by making it difficult for detection heuristics to statically infer the outcome of the execution of the sled. When the sled is actually executed, its behavior is that intended by the attacker, correctly leading to the shellcode. This can be achieved by either using branches whose target cannot be determined statically or by using self-modifying code.

Static analysis cannot follow branches that cannot be determined statically, such as register or memory indirect jumps, because the contents of the registers or memory are not known during the analysis. Therefore, it cannot continue with the inspection of the corresponding code paths and cannot determine their outcome. Such jumps, however, must specify the target as an absolute address.

Also, a sled could modify itself so that invalid instructions, appearing under static analysis to terminate a code path, are overwritten during execution by previous instructions and are actually executed normally. However, the sled must rely on stack alignment to avoid the execution of illegal instructions before they

Sled Type	Scheme			
	Snort	Fnord	APE	STRIDE
1. NOP instructions	Yes	Yes	Yes	Yes
2. One-byte NOP-equivalents	No	Yes	Yes	Yes
3. Multi-byte NOP-equivalents	No	No	Yes	Yes
4. Four-byte Aligned	No	No	Yes	Yes
5. Trampoline-sled	No	No	No	Yes
6. Obfuscated Trampoline-sled	No	No	No	Yes
7. Static Analysis Resistant	No	No	No	After extension

Table 4.1: Comparative effectiveness of various sled detection schemes.

are fixed-up. Again, like indirect branches, write operations require an absolute address.

To overcome the absolute address problem, present in both indirect branches and self-modifying instructions, the `esp` register, which holds the stack frame's absolute address, can be used to find the buffer and sled addresses. However, the use of the `esp` register could hint for static analysis resistant sleds, but, in fact, the absolute address of the sled can be found even without using this register: knowing the injected return address and maintaining a counter while sliding through the sled provides knowledge of the absolute address of the current sled position. This seems to be relatively hard to implement, especially considering the need for 4-byte alignment.

4.2 Existing Approaches

In this section we briefly present three techniques which have been proposed for sled detection: NIDS signatures, the Fnord mutated sled detection plugin, and APE. Table 4.1 summarizes the effectiveness of each technique, along with our proposed detection mechanism, for each sled type.

4.2.1 NIDS Signatures

Detecting simple NOP sleds such as those described in section 4.1.1 is relatively straightforward. On the Intel IA-32 architecture, `nop` is a single-byte instruction with opcode 0x90. Thus, to detect a simple sled consisting only of `nop` instructions, a pattern matching rule searching for a sufficiently long sequence of bytes with value 0x90 is enough. Indeed, such rules exist for popular NIDS, such as Snort [11].

4.2.2 Fnord

The Fnord [12] mutated sled detection plugin for Snort detects sleds by searching network traffic for long series of one-byte NOP-equivalent instructions. It is, therefore, capable to detect type-2 sleds, such as those described in Section 4.1.2. It may be the case that its list of NOP-equivalents could be extended with the opcodes of multi-byte NOP-equivalents, making it capable to detect type-3 sleds such as those described in Section 4.1.3, but we use the standard version here. However, Fnord definitely fails to detect type-4 sleds and above, that exploit the alignment of stack variables.

There also exist various other tools that offer similar sled detection capabilities with Fnord [2, 8]. Since these tools, along with Fnord, all rely on the NOP-equivalents list contained in ADMmutate in order to detect mutated sleds, it is sufficient to consider just one of them.

4.2.3 Abstract Payload Execution - APE

APE [15] is a detection mechanism that enables the detection of sleds by looking for sufficiently long series of valid instructions: instructions which decode correctly and whose memory operands are within the address space of the process being protected against attacks. To reduce its runtime execution overhead, APE uses sampling to pick a small number of positions in the data from which it will start abstract execution. The number of successfully executed instructions from each position is called the Maximum Executable Length (MEL). When APE encounters a conditional branch, it follows both branches and considers the longest one as the MEL. If the destination of the branch can not be determined statically, APE terminates execution and uses the MEL value computed so far. A sled is detected if a sequence has a MEL value greater than 35. Although APE can be used to detect sleds of type-1 through type-4, it fails, however, to detect sleds of type-5 (trampoline sleds), type-6 (obfuscated trampoline), and type-7 (static-analysis-resistant sleds).

Indeed, although the purpose of type-5, and type-6 sleds is to transfer program control to the shellcode in as few steps as possible using jump instructions, the mechanism that is used by the APE scheme is based on the detection of a sufficiently long execution sequence of instructions, and thus, trampoline-sleds evade detection by having a short sequence of executed instructions. Static analysis resistant sleds also confuse APE, because it errs on the unsafe side when it cannot decide about a code sequence.

4.3 STRIDE

STRIDE is given some input data, such as an alert generated by the main module, and searches each and every position of the data to find a sled. If a sled is found, the input data are considered part of an attack.

To detect a sled spanning over at least n bytes and starting at position i of the input data, STRIDE searches for all sequences of instructions of length $n - j$ bytes starting at offset $i + j$ of the input data, for all $j \in \{0 \ldots n - 1\}$. If STRIDE finds all n sequences of instructions to be `valid sequences`, it then concludes that a sled of length n starts at position i.

We call a code sequence, starting at a certain point i in the input data, a "valid sequence of instructions of length n at position i," (1) if it either decodes correctly for n bytes without encountering privileged instructions, or (2) if a jump instruction is encountered along the way.

Informally, a valid sequence of instructions is a sequence of instructions which can be used to construct a sled. Such a sequence may only contain valid instructions, and may not contain privileged instructions, i.e. instructions which can be invoked only by the operating system kernel.

Figure 4.4 gives the pseudo-code for STRIDE. The main routine, `stride`, consists of a loop which tries to find a sled of length `sled_length` at each and every position of input data `input`. Routine `find_sled (data,len)` finds a sled by attempting to valid all valid sequences of length `len-i` which start at position `data+i`, for al values of `i`. Aligned sleds are accounted-for by checking for valid sequences at every four bytes instead of at every byte but the check is applied for all four possible displacements.

4.4 Complementary Techniques

STRIDE can only be applied to buffer-overflow-based attacks which use sleds. If an attack does not make use of a sled, then it can not be detected by STRIDE. In this section we discuss some complementary buffer overflow detection techniques, for detecting buffer overflow attacks without a sled.

The so-called jmp esp technique can be used to avoid using a sled at all; it works as follows. At the time of the attack the esp register points to the current stack frame — right after the overwritten return address. By making the return address point to a jmp esp or call esp instruction somewhere in memory, control can be directed to the shellcode without the need for a sled, because when the function returns, the jmp esp instruction is executed and control is transfered exactly to the bytes after the overwritten return adress, where the shellcode can be placed. The technique may make the attack depend on the OS version or the version of the vulnerable application, depending on where the instruction used resides. The Metasploit [3] project's web site even includes a tool to search for the addresses of appropriate instructions that are available across different OS versions. However, such a list of addresses can also be searched-for in network traffic, as in the Buttercup system [9], and used to detect attempts to use this technique.

Another way to avoid using a sled it to brute-force the return address: to repeatedly try the attack with a different return address each time. Yet another related possibility of confusing detection methods is that of partial sleds, sleds that are not

```
stride(input, input_size, sled_length)
{
  for (i=0 ; i < input_size-sled_length; i++) {
      if (find_sled(input+i,sled_length))
           return TRUE ;
  }
  return FALSE ;
}

find_sled(data, len)
{
  for (j = 0 ; j < 4 ; j++) {
    for (i = j ; i < len ; i+=4)  {
      if (!valid_sequence(data+i, len-i) )
        return FALSE ;
    }
  }
return TRUE;
}

is_valid_sequence(data,len)
{
   // decode "len" instructions in buffer "data"
   res= decode(data, len)
   if (res == VALID_DECODE) return TRUE;
   if (res == ENDS_IN_JMP) return TRUE;
   return FALSE;
}
```

Figure 4.4: Pseudo-code for STRIDE algorithm

executable at every offset; the attacker may choose to sacrifice some efficiency in order to confuse detection schemes. Both techniques may result in reduced efficiency of the attack and crashed connections — the latter could be detected at the network level.

Yet another way to avoid using a sled is the so-called *jump-to-libc* technique: instead of executing a shellcode, the attack transfers control to an application or system function, with appropriate parameters placed on the stack. For example, an attacker could invoke the system() function and execute an arbitrary command. A sled is not needed because the absolute address of the invoked function is known beforehand. However, the function does not expect its parameters to be encrypted so machine-language polymorphism is not applicable to such attacks.

Chapter 5

Preliminary Measurements

In this chapter we evaluate the effectiveness and efficiency of our approach and investigate its parameter space using real network traffic traces.

5.1 Network Traffic Traces

Real network traffic traces, gathered from FORTH's Local Area Network in early 2004, have been used for our evaluation. The monitored networks contain about 100 hosts. The trace is about 5 Gbytes large and contains 11.7 million network packets that represent traffic from web clients, peer-to-peer programs, SMB shares, IMAP, printers, etc. The traffic characteristics of the traces are shown in table 5.1.

5.2 Experiments

In this section we explore the parameter space of the worm detector that we have described. The parameters of the experiments are:

- substring length
- sampling mask
- distinct destinations threshold
- queue size

The measured quantities are the following:

Trace	TCP Packets	TCP (Payload) Bytes	Duration	Attacks
TRACE-I	11,746,790	5,108,857,877	2h	102

Table 5.1: Characteristics of the trace used in the experiments.

- **false positives**, which are defined to be the number of flows that were flagged but did not correspond to any real worm,
- **detection delay**, which is defined as the elapsed worm attacks up to the detection of the worm.

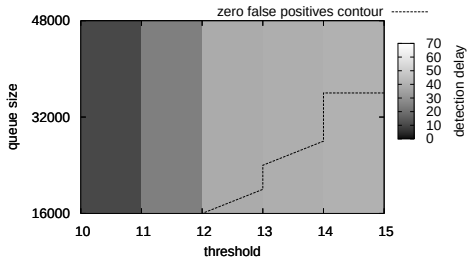

Figure 5.1: Detection delay and zero false positives contour line with a string length of 200.

5.2.1 Worm Detection Effectiveness

Figure 5.1 explores detection delay and false positives as a function of threshold and queue size, for TRACE-I and TRACE-II, and a substring length of 200 bytes. The zero false positives contour is projected onto the parameters space, so that the area below it corresponds to threshold and queue size combinations with zero false positives.

We observe that decreasing the threshold decreases the detection delay. This is expected, since more worm attacks are required to trigger detection. However, decreasing the threshold may also result in false positives. This is expected too, since there exist legitimate strings that are sent to more than one destinations.

We also notice the effect of queue size on the false positives. Even for higher thresholds, an increased queue size may result in false positives. This is because a larger queue size will hold strings with a lower rate of new destinations.

The bottom-right part of the graph, where the two areas of detection and zero false positives overlap, represents a combination of parameters that result in worm detection without false-positives.

Next we investigate the CPU time spent for the unoptimized version of the worm detection system to process a trace of network packets 5 Gbytes large. We see that the un-optimized version takes close to 16 minutes which corresponds to a processing rate of 38.5 Mbps. We expect, however, a ten-fold increase in performance through the use of fingerprint value sampling.

Chapter 6

Non-Detection-Related Functionality

6.1 Configuration

The system will use a configuration file.

The parameters that will be configurable include:

- reporting method and its parameters (e.g. logfile location)
- enabled and disabled modules
- module parameters such as the various thresholds

Chapter 7

Future Optimizations

In this chapter, we describe future optimizations that should be considered, but not necessarily for a first implementation. The first such consideration is provisions for operating the system behind a load balancer in order to handle high network speeds. The second future enhancement is support for repeating suspicious traffic to a honeypot system, for verification.

7.1 Load Balancing

Traffic monitoring, and especially the detection of repetitive substrings, is a resource demanding task. At high network speeds, the available computational resources are limited. The current design relies on sampling of streams for reducing the load to a manageable level, but there is also the option of distributing the computation to different nodes of a computer farm using a load balancer.

The load balancer would have to assign entire streams to individual nodes. This can be achieved without stream reassembly or session tracking, by using the fields of packet headers that identify a session to dispatch packets to the appropriate nodes. Packets with the same source and destination addresses and port numbers would be directed to the same node. Indeed, stream reassembly can be completely offloaded to the nodes.

The distribution raises the issue of communication among the nodes. Maintaining a common queue across the nodes would lead to excessive communication. Instead, each node should have its own queue, and communicate only the fingerprints that reach a local threashold. For n nodes, the local threshold can have a value of $threshold/n$, because if the streams are randomly assigned to nodes, we expect the instances of a fingerprint to be distributed equally among the nodes.

Note that as the volume of the monitored traffic increases, the threshold used to trigger detection can also be increased. So the $threshold/n$ ratio will not necessarilly diminish with increasing n.

7.2 Replay Module

Any system based on anomaly detection is bound to have false positives. The goal of this module is to provide support for redirecting suspicious traffic to a honeypot system, for further verification. It should not impose any modification on the system, rather it would receive the output from the detection system and verify it, before forwarding it to an enforcement system.

Chapter 8

Schedule

In this chapter, we provide the anticipated schedule for the development of the system.

- Prototype versions for the main filter and for the sled detection have been developed already.
- The unifying framework for the various detection heuristics will be developed during January and February 2005.
- Reporting functionality will be developed during March 2005.
- A first version of the complete system will be ready on April 2005.
- The final version of the complete system will be ready on May 2005.

Chapter 9

Conformance to the Requirements

In this chapter, we discuss how the design conforms to the requirements laid out in the Specifications document. We list each requirement and discuss the relevant provisions in the design.

9.1 Functional Requirements

9.1.1 Detected Attacks

F 1.1 *The system should detect attacks that use unfragmented packets, as well as fragmented packets to spread their payload over multiple packets in order to obfuscate their signatures.*

By employing flow reassembly mechanisms, the design ensures that detection cannot be evaded by breaking signatures across packet boundaries.

F 1.2 *The system should detect worms with an attack that contains at least 300 consecutive bytes.*

The relevant parameter of the system is substring size, and the preliminary evaluation shows that a substring size of 300 bytes is sufficient to detect worms.

F 1.3 *The system must detect worms that spread over the TCP protocol. However, it is highly desirable, but not initially required, to include support for UDP worms as well.*

There is support for the TCP protocol, including session tracking and stream reassembly. We are considering UDP support as a future extension.

F 1.4 *The system is not expected to detect stealth worms.*

The system is based on repetitive content. Stelth worms, however, do not generate adequate amounts of traffic per time unit.

F 1.5 *Strings of small length are often popular without belonging to a worm. For example, many unrelated HTTP requests contain the string* GET /*, or* HTTP/1.1. *Therefore, a minimum detectable string length must be established.*

The system has a substring length threshold that controls the minimum detectable signature length.

9.1.2 Detection Delay

F 2.1 *The detection delay of the system, measured in elapsed attacks before an alert has been triggered, should be evaluated theoretically and experimentally for worms with different levels of aggressiveness.*

We have performed preliminary evaluation of the detection delay with a real worm, but the complete evaluation will be provided in latter phases using the complete system.

9.1.3 False Positives

F 3.1 *Timely detection is worthless in the presence of false positives, therefore the system is required to have a zero false positives rate.*

Again, preliminary evaluation has shown that it is possible to detect worms without false positives. In addition, we have considered a number of mechanisms to further reduce false positives, including buffer overflow detection.

F 3.2 *As a last means of preventing false positives, the system should support a white-list that allows handling persistent false positives. Strings listed in the white-list should not be considered as worm signatures.*

The system will compute rabin fingerprints of the white-listed strings and place appropriate flags in the corresponding hash table entries.

9.1.4 Configuration - Customization

F 4.1 *It should be possible to adjust the sensitivity of the system using a threshold. The system could be adjusted for faster detection with the cost of the false positives rate going up.*

This tradeoff has been demonstrated in the preliminary measurements, and the requirement is supported by allowing the tuning of parameters.

F 4.2 *It must be possible to adjust the amount of information that will be recorded, so as to cater for those that worry about their privacy being revealed.*

The minimal amount of recorded information consists of the offending signature. We are currently not considering of providing more information.

F 4.3 *It should be possible to configure whether log-files, result packets, or both will be used to report alerts.*

The configuration file allows tuning of this behaviour.

F 4.4 *It should be possible to configure the log-file location and the destination to which result packets are sent.*

The configuration file allows tuning of this behaviour.

9.1.5 Security Constrains

F 5.1 *The network where the system is hosted must not be exposed to vulnerabilities because of the presence of the early warning system.*

F 5.2 *The system must also be resilient to malicious traffic targeted to attack the system itself. Finally it must be carefully engineered so that it will be immune to any kind of attack. This is particular important as a poorly configured/engineered system can pose a huge security risk for the whole network. Thus is essential that the system is programmed with security practices in mind.*

The chief security design concern is the abuse of the system by causing false positives that match legitimate traffic. Acting on those false positives would effectively mount a denial-of-service attack on the legitimate traffic. The main mechanism of defence against such an attack is sampling, which forces the attacker to generate considerable traffic. Additional filters such as buffer overflow detection reduce significantly the possibility of matching legitimate traffic.

9.1.6 Privacy

F 6.1 *The data gathered should be strictly used for analyzing, identifying a possible attack, implementing ways to protect and for absolutely no other reason. The EAR framework will operate in a way that the data do not have to be shared among various individuals and/or companies thus minimizing the risk of revealing important information about the end-user or corporate data.*

F 6.2 *The data analysis must take place within the organization data center which is considered to be a trusted body.*

The system has been designed as a centralized facility and the only information that might be publicized are the detected signatures.

F 6.3 *The issued alerts and reports must not include any information that will reveal the identity of the user (eg. the IP addresses belonging to the infected hosts). Furthermore, only content that belongs to traffic that has been identified as traffic initiated by a worm is going to be included by the system in alerts.*

F 6.4 *The system should be able to integrate with an external application that will take over the task of transmitting the alerts to the administrators.*

F 6.5 *The necessary information should be recorded in log files. The format of log files should be specified in detail when the system has been developed completely, it shall contain enough information to create a filtering signature. The task of the application will vary according to the kind of output required and the actions that need to be taken. The general o*

9.2 Performance Constrains

9.2.1 Monitoring Capacity

P 1.1 *The system must be capable of operating at 100 Mbits/sec. However, it is highly desirable to achieve operation speeds up to 1 Gbit/sec.*

The preliminary mesurements show that 200 Mbits/sec can be processed by the detection mechanisms. During the implementation further optimizations will take place.

The rest of the requirements are not relevant to the design of the system.

References

[1] IA-32 Intel Architecture Software Developer's Manual vol. 1-3. http://developer.intel.com/design/pentium4/manuals/index_new.htm.

[2] Prelude IDS. http://www.prelude-ids.org/.

[3] Metasploit project, 2004. http://www.metasploit.com/.

[4] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996. http://www.phrack.org/phrack/49/P49-14.

[5] Kang Su Gatlin. Windows data alignment on IPF, x86, and x86-64, February 2003. MSDN Library, http://msdn.microsoft.com/.

[6] K2. ADMmutate. http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz.

[7] Oleg Kolesnikov, Dick Dagon, and Wenke Lee. Advanced polymorphic worms: Evading IDS by blending in with normal traffic, 2004. http://www.cc.gatech.edu/~ok/w/ok_pw.pdf.

[8] NGSEC. NIDSFindShellcode. http://www.ngsec.com/downloads/misc/NIDSfindshellcode.tgz.

[9] A. Pasupulati, J. Coit, K. Levitt, S. F. Wu, S. H. Li, R. C. Kuo, and K. P. Fan. Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In *Proceedings of The IEEE/IFIP Network Operations and Management Symposium 2004 - NOMS 2004*, pages 235–248, April 2004.

[10] M.O. Rabin. Fingerptinting by random polynomials. Technical Report 15-81, Center for Research in Computing Technology - Harvard University, 1981.

[11] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of USENIX LISA '99*, November 1999. (software available from http://www.snort.org/).

[12] Dragos Ruiu. Fnord: Multi-architecture mutated NOP sled detector, February 2002. http://www.cansecwest.com/spp_fnord.c.

[13] C. Shannon and D. Moore. The spread of the witty worm, 2004. http://www.caida.org/analysis/security/witty/.

[14] Neil T. Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 87–95. ACM Press, 2000.

[15] Thomas Toth and Christopher Krügel. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID)*, October 2002.