STRIDE: POLYMORPHIC SLED DETECTION THROUGH INSTRUCTION SEQUENCE ANALYSIS

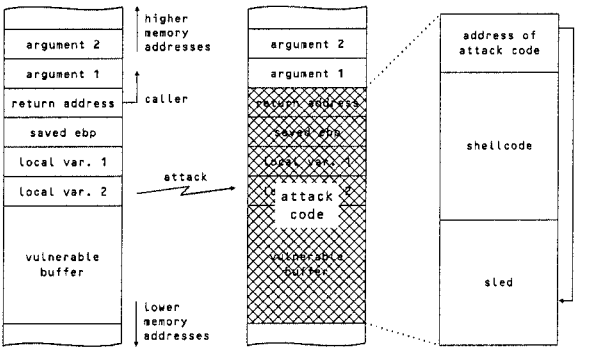
P. Akritidis[1], E. P. Markatos[1], M. Polychronakis[1], and K. Anagnostakis[2]
[1]*Institute of Computer Science Foundation for Research and Technology Hellas, P.O. Box 1385 Heraklio, GR-711-10 Greece, {akritid, markatos, mikepo}@ics.forth.gr*
[2]*Distributed Systems Laboratory, CIS Department, Univ. of Pennsylvania, 200 S. 33rd Street, Phila, PA 19104, anagnost@dsl.cis.upenn.edu*

Abstract: Despite considerable effort, buffer overflow attacks remain a major security threat today, especially when coupled with self-propagation mechanisms as in worms and viruses. This paper considers the problem of designing network-level mechanisms for detecting polymorphic instances of such attacks. The starting point for our work is the observation that many buffer overflow attacks require a "sled" component to transfer control of the system to the exploit code. While previous work has shown that it is possible to detect certain types of sleds, including obfuscated instances, this paper demonstrates that the proposed detection heuristics can be thwarted by more elaborate sled obfuscation techniques. To address this problem, we have designed a new sled detection heuristic, called STRIDE, that offers three main improvements over previous work: it detects several types of sleds that other techniques are blind to, has a lower rate of false positives, and is significantly more computationally efficient, and hence more suitable for use at the network-level.

Keywords: security; intrusion detection; buffer overflow detection.

1. INTRODUCTION

Buffer overflow attacks, popularized in 1996 by Aleph One[1], have been a major security concern ever since, because exploiting a buffer overflow vulnerability allows an attacker located anywhere on the Internet to execute arbitrary code on the compromised system. The highly interconnected



environment of the Internet currently creates tremendous exploitation

Figure 1. Anatomy of a stack-based buffer overflow attack. By overflowing the buffer, the return address can be overwritten with a value pointing somewhere within the sled. The flow of control will be transferred to the start of the shellcode from any location in the sled.

opportunities. In fact, such vulnerabilities in networked services are currently the main means used for propagation of Internet worms.

A tutorial on buffer overflow attacks was provided by Aleph One in 1996[1]. A buffer overflow attack takes advantage of insufficient bounds checking on a buffer located on the stack to overflow the buffer and overwrite the return address of the currently executing function. Figure 1 shows the typical layout of the stack both *before* (left) as well as *after* the buffer overflow attack (middle and right). The attack involves injecting data into the buffer which resides in the lower addresses of the stack. The amount of data injected is larger than the buffer size and the resulting overflow overwrites at least the `local` variables, the `saved ebp`, and the `return address`, resulting in the stack shown in Figure 1 (middle). The return address is hijacked to point to malicious code that is injected by the attacker, usually within the same overflowed buffer, illustrated as the shaded area in Figure 1 (middle). Besides *stack-based* buffer overflow attacks, it is also possible (although, perhaps more difficult) to engineer similar attacks on heap-based and statically-allocated buffers.

Despite considerable efforts in end-system preventive measures[2-8], adoption of these techniques is proceeding at an alarmingly slow pace. Dealing with this threat therefore requires additional perimeter defense mechanisms, as provided by firewalls and network intrusion detection systems. However, obfuscation techniques make it hard to apply simple rule-based detection techniques as currently used in network intrusion detection.

Buffer overflows often have features that can be seen as a weakness that can be exploited by detection heuristics. A key observation is that, although the location of the injected code relative to the start of the buffer is known to

the attacker, the *absolute address*, which represents the start of the injected code, is only *approximately* known because the location of the start of the buffer relative to the start of the stack varies between systems, even for the same executable program.

To overcome the lack of exact knowledge on where to divert control, the attacker needs to append the malicious code fragment to a sequence of NOP instructions, typically around a few hundred bytes long. The overwritten return address always transfers control somewhere inside this sequence, and thus, after sliding through the NOP instructions, control will eventually reach the worm code. Due to the sliding metaphor, this sequence is usually called a *sled*. The *exact* location within the sled where execution will start does not matter: as long as the return address causes the system to jump anywhere within the sled, it will always reach the core of the exploit.

To detect buffer overflows, Network Intrusion Detection Systems (NIDSes) rely on signatures, characteristic strings, and regular expressions, such as code sequences included in an attack's shellcode that can identify the attack. However, obfuscation similar to polymorphism[9,10], used in viruses since the early 90s, renders signature-based techniques for zero-day worm detection obsolete[11,12]. Polymorphism usually encrypts the shellcode with a different random key each time and prepends it with a decryption routine. When the malicious program starts executing, the decryption routine will execute first, which in turn will decrypt the shellcode, which will then start executing. Since the decryption routine itself cannot be encrypted, some systems base their zero-day worm detection on detecting the decryption routine itself. Unfortunately, decryption routines are usually obfuscated using *metamorphism*. Metamorphism substitutes (sequences of) instructions with equivalent (sequences of) instructions, making the decryption routine difficult to fingerprint. Metamorphism is also used to obfuscate sleds, by, for example, substituting NOP instructions, with other equivalent instruction sequences.

Many existing detection mechanisms have also focused on detecting the sled component in order to detect buffer overflow attacks. For example, signatures to match simple sleds have been included in the shellcode rule set of the Snort NIDS[13]. In addition, Snort has been extended with the Fnord plugin[14] that searches for obfuscated sleds. Finally, Toth and Kruegel proposed the Abstract Payload Execution (APE) method[15] which further improves the sensitivity of obfuscated sled detection.

The rest of the paper is organized as follows. First, in Section 2 we present a classification of obfuscated sleds, and in Section 3 we discuss some existing detection techniques. Then, in Section 4 we propose STRIDE, a novel detection mechanism. In Section 5 we evaluate the detection mechanisms using generated attacks and real network traces, and in Section

6 we discuss limitations of sled detection in general and of STRIDE in particular. Finally, we conclude in Section 7.

2. 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.

2.1 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[1] and has been used in many other attacks.

2.2 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. Current polymorphic buffer overflow attack generators use such sleds to avoid detection. The ADMmutate[16] engine uses this technique with a list of of 55 one-byte NOP-equivalent instructions. The Metasploit Framework[17] extends the ADMmutate engine with 3 additional single-byte NOP replacements. We have enumerated 66 such instructions in the Intel IA-32 architecture[18]. Although not yet seen in the wild, obfuscated sleds are readily available to attackers.

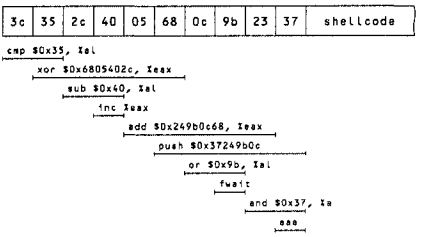


Figure 2. An example of a small sled, executable at every byte offset, which is constructed by interleaving one-byte and multi-byte NOP-equivalent instructions.

2.3 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 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 2. 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.

2.4 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 word (4-byte) boundaries[19]. Reference[11] 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

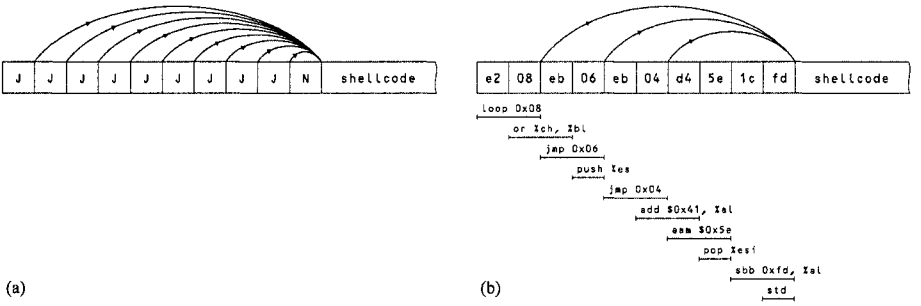


Figure 3. (a) The ideal trampoline-sled: flow of control is directed to the shellcode in a single step from any position in the sled. (b) An example of a small trampoline-sled that is executable at every byte offset. Control transfer instructions are placed at every second byte and their relative address operand is chosen so that it is a valid NOP-equivalent opcode.

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.

2.5 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 3(a). The body of such a sled consists of control transfer instructions with relative addresses, all pointing directly to the shellcode. Thus, 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. Even if a trampoline-sled has to be executable at every offset, an attacker can carefully choose the operands of the jump instructions to be valid NOP-equivalent opcodes, as explained in Section 2.3. An example of a small trampoline-sled that is executable at every byte offset is illustrated in Figure 3(b).

The shortest control transfer instructions available are two bytes long. For example, 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.

2.6 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 3.2).

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.

2.7 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 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.

Table 1. Comparative effectiveness of various sled detection schemes.

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*

3. SLED DETECTION MECHANISMS

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 1 summarizes the effectiveness of each technique, along with our proposed detection mechanism, for each sled type.

3.1 NIDS Signatures

Detecting simple NOP sleds such as those described in section 2.1 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

* STRIDE can detect sleds that use indirect jumps, and we discuss how it may be extended to detect self modifying sleds in Section 6.

long sequence of bytes with value 0x90 is enough. Indeed, such rules exist for popular NIDSes, such as Snort[13].

3.2 Fnord

The Fnord[14] 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 2.2. 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 2.3, 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[20, 21]. 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.

3.3 Abstract Payload Execution

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

```
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. Pseudo-code for STRIDE algorithm

instructions. Static analysis resistant sleds also confuse APE, because it errs on the unsafe side when it cannot decide about a code sequence.

4. THE STRIDE DETECTION ALGORITHM

In this section 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.

STRIDE is given some input data, such as a URL, 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," if it either (1) decodes correctly for n bytes without encountering privileged instructions, or if (2) 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 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.

STRIDE is related to previous approaches in several ways. Like Snort[13] and Fnord[14], STRIDE is able to find long sequences of NOP(-equivalent) instructions. Like APE[15], STRIDE is able to decode the input data and identify sequences of instructions that may be part of a sled. However, STRIDE has two major differences from APE:

Table 2. Detection rate of the various detection schemes for traces containing 10,000 different generated sleds of a single type.

Sled Type in Trace	Scheme			
	Snort	Fnord	APE	STRIDE
NOP instructions	100%	100%	100%	100%
One-byte NOP-equivalents	0%	55.4%	100%	100%
Multi-byte NOP-equivalents	0%	0%	100%	100%
Four-byte Aligned	0%	0%	100%	100%
Trampoline-sled	0%	0%	0%	100%
Obfuscated Trampoline-sled	0%	0%	Fig. 5	100%

- APE detects a sled when it finds a sufficiently long execution sequence of instructions. Therefore, although it is able to detect NOP-based sleds, APE can not detect trampoline sleds, because they jump directly to their destination code and, therefore, do not exhibit long execution sequences. In contrast, STRIDE may consider even short execution sequences (such as a single jump instruction) to be part of a valid sled.
- STRIDE verifies that each and every byte of a sled (apart from the cases of word-aligned sleds) is the start of a valid sequence of instructions. On the contrary, for APE it is enough to find only *one* sufficiently long execution sequence to consider it a valid sled.

5. EXPERIMENTAL EVALUATION

We evaluate the accuracy of the detection rate of our proposed algorithm STRIDE, Snort's shellcode signatures[13], Fnord mutated sled detection plugin for Snort[14], and APE, by generating 10,000 different sleds of each type using the Metasploit Framework v2.2[17], modified to generate sleds ranging from type-1 (simple NOP sleds) up to type-6 (obfuscated trampoline).

We also evaluate the false positives rate of the four methods as in[15], by applying them to HTTP URIs. The URIs were captured from our institution's LAN, which contains about 150 hosts. Sled detection methods which are based on instruction decoding, employ the decoder used in[22].

5.1 Detection Rate

The results of applying all four detection methods on the generated sleds are shown in Table 2. We observe that Snort's shellcode signatures detect simple NOP sleds with 100% success, but fail to detect more elaborate sleds. Fnord is able to detect simple NOP sleds with 100% success too, and in addition is able to detect sleds with one-byte NOP-equivalent instructions with a 55.4% rate. Although it could have achieved a 100% rate for one-byte NOP-equivalent sleds, it achieves a lower-rate due to an incomplete NOP-equivalent instruction list. Fnord also fails to detect sleds with multi-byte NOP-equivalent instructions, but it should be possible to update its list of NOP-equivalents to include them as well. However, this is as far as Fnord can get. Indeed, Table 2 shows that Fnord fails to detect sophisticated sleds, such as 4-byte aligned and trampoline sleds.

Table 2 suggests that the APE method is able to detect simple NOP sleds, sleds with one-byte and multi-byte NOP-equivalent instructions, as well as four-byte aligned sleds with a 100% success rate. However, APE cannot detect trampoline sleds. This was expected, because trampoline sleds reach the core attack code by executing only a small number of jump instructions, while APE bases its detection method on the sequential execution of a long sequence of instructions.

It is interesting, however, to point out that although APE can not detect trampoline sleds, it is able to detect some of the more difficult obfuscated trampoline sleds. Indeed, as Figure 5 shows, APE is able to detect as many as 6% of the obfuscated trampoline sleds for small MEL. This is because the NOP-equivalent instructions that are used for the obfuscation cause an increase of the overall execution steps of the sled, which can now reach a

Finally, STRIDE is able to detect simple NOP sleds, sleds with one-byte or multi-byte NOP-equivalent instructions, as well as four-byte aligned sleds and plain or obfuscated trampoline sleds with 100% success, as expected.

low MEL threshold. Nevertheless, the detection rate of APE is still very low, at 6%, even for the minimum suggested MEL value.

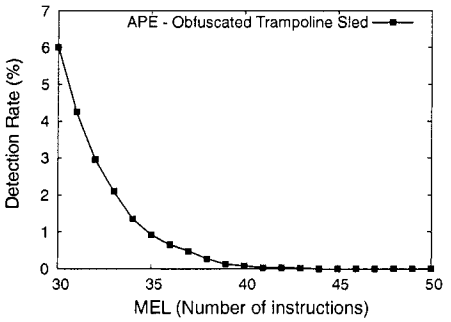


Figure 5. Detection rate for APE when applied to obfuscated sleds as a function of MEL.

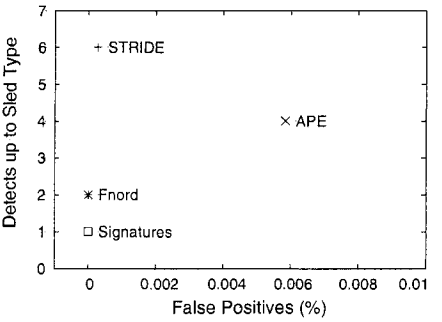


Figure 6. Comparative effectiveness of the various detection schemes. The results for APE are for a MEL value of 35 with 100 samples per kilobyte and for STRIDE for sled length 130 bytes.

5.1 False Positives

The results of the false positives rate evaluation for the four methods with real traces are shown in Figure 6. In this experiment STRIDE has a sled length parameter of 130 bytes and APE has a MEL value of 35 instructions with 100 samples per kilobyte. With these parameters APE is sensitive, like STRIDE, to sled lengths of about 130 bytes and above.

The Snort shellcode signatures have zero false positives, because there was no sufficiently long NOP-sequence in our traces. Fnord also has almost 0% false positives, because there were very few sequences of bytes in the traces which corresponded to sequences of NOP-equivalent instructions. Although both Snort and Fnord have an attractive practically 0% of false positives rate, they are severely limited in their ability to detect elaborate sleds, such as trampoline sleds. Figure 6 shows that APE has a false positive rate of 0.006%. Finally, STRIDE has a false positive rate of 0.00027%, close to an order of magnitude smaller than APE. Overall, we see that STRIDE seems to strike a good balance between true positives and false positives. That is, it is able to find more true positives than any other method, while keeping the false positives as low as those of Fnord and Snort.

The interested reader should notice that the exact value of false positives for APE and STRIDE depends heavily on their parameters. To explore the influence of the parameters to the false positive rate of APE and STRIDE,

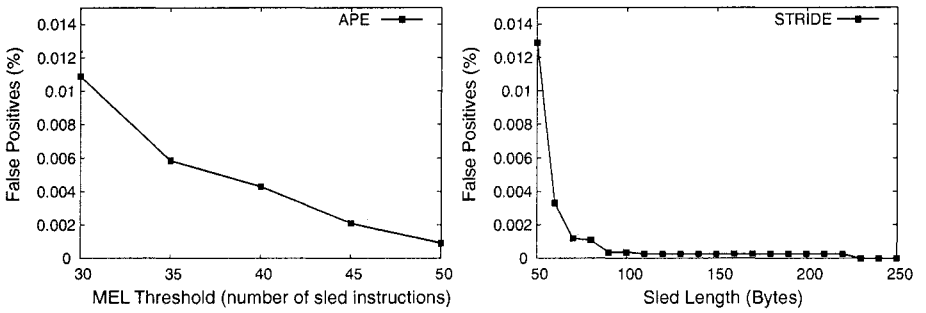


Figure 7. False positives rate for APE and STRIDE with varying parameters.

we investigate the false positives rate for both methods as a function of MEL and sled length, and display the results in Figure 7. We see that as the size of MEL increases, the percentage of false positives for APE decreases. However, we should point out that larger MEL values also decrease the number of detected true positives, as can be seen in Figure 5. Figure 7 also shows that the percentage of false positives for STRIDE decreases with the sled length, and reaches zero for sled length larger than 230 bytes. This is an encouraging result, since typical sleds are usually longer than 250 bytes. Overall, our results suggest that STRIDE is able to have a true positive rate of 100% (Table 2), while having a false positive rate of (close to) 0%.

5.2 PERFORMANCE

Besides being accurate, a worm detection method should also be fast, so as to be able to detect worms in real-time. To evaluate the speed of STRIDE we measured the CPU time consumed by STRIDE with a sled length value of 200 bytes, and compared it to the execution time of APE with a MEL count of 35 on a Pentium 4 machine (2.6GHz clock speed, 512KB cache size) for a trace with 1,093,249 requests. The CPU time for processing this trace was 25 sec for APE (22.9 usec per request) and 4.85 sec for STRIDE (4.4 usec per request). We see that STRIDE outperforms APE by a factor of 5. This is mostly due to the different handling of branch instructions by the two algorithms. Indeed, when APE encounters a branch instruction, whose target can be determined statically it follows both branches, a decision, which may potentially lead to the exploration of an exponential number of execution paths. Unlike APE, when STRIDE encounters a branch instruction, it assumes that it found a valid sequence, without making any

attempt to follow the branch. By being conservative, STRIDE avoids the exponential explosion and significantly reduces the associated run-time cost.

6. DISCUSSION

Although our evaluation has shown that STRIDE has some benefits, we should mention that it still has limitations. For example, 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 addition, STRIDE still cannot detect self-modifying sleds. It is, however, possible, to extend it with a decoder that is capable of identifying memory write operations and handle the sequences that contain them the way it currently handles jump instructions: consider them valid, because they cannot be proved invalid with static analysis. Finally, a worm writer could blind STRIDE by adding invalid instruction sequences at suitable locations in the sled. Note that this would most likely lead to a fraction of the infection attempts crashing the remote process and would most certainly slow down the spread of the worm, while also exposing it to other detection components that look for anomalous behavior at the process-level.

7. SUMMARY AND CONCLUDING REMARKS

We have presented STRIDE, a new approach for network-level detection of buffer overflow attacks, which relies on the identification of the sled component that is usually part of such attacks. Because it operates at the network-level, STRIDE can be used for detecting worms that replicate through buffer overflow exploits, even if they involve elaborate obfuscation. Our analysis allows us to make three main observations:

- *STRIDE can detect several classes of sleds that cannot be identified by previous proposals.* As presented in Section 2.5, trampoline sleds can be used by attackers in order to evade current sled-based detection mechanisms. STRIDE detects such sleds, even in their obfuscated variations.
- *STRIDE achieves high detection rates while maintaining low false positive rates.* Snort and Fnord have few false positives but can only detect basic sled types. APE detects more complex sleds, but has an order of magnitude more false positives compared to STRIDE, while also missing two classes of sleds. Our approach can detect all types of sleds presented in this paper, except for Static Analysis Resistant sleds, with a detection rate of 100%, and a false positive rate that reaches 0% for

reasonable algorithm parameters. As suggested in Section 6, it may also be possible to detect Static Analysis Resistant sleds. This question, however, requires additional analysis and is outside the scope of this paper.

- *STRIDE is more efficient in terms of processing cost.* As shown in Table 3, STRIDE has relatively low computational cost, outperforming APE by a factor of 5. This suggests that STRIDE can operate on high-speed links and remain effective even under heavy loads, at a reasonable cost.

The high accuracy, low false positive rate, and low processing cost achieved by STRIDE suggest that it is likely to be highly useful as part of an automated network-level defense mechanism against both targeted attacks and large-scale zero-day worm outbreaks, especially as worms become more aggressive and more sophisticated.

ACKNOWLEDGEMENTS

This work was supported in part by the IST project NoAH (011923) funded by the European Union and the GSRT project EAR (USA-022) funded by the Greek Secretariat for Research and Technology. The work of K. Anagnostakis is also supported by OSD/ONR CIP/SW URI through ONR Grant N00014-04-1-0725. P. Akritidis, E. P. Markatos, and M. Polychronakis are also with the University of Crete. The work of K. Anagnostakis was done while at ICS-FORTH.

REFERENCES

1. Aleph One. Smashing the stack for fun and profit. Phrack, 7(49), Nov. 1996. http://www.phrack.org/phrack/49/P49-14.
2. A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In Proceedings of USENIX Annual Technical Conference, June 2000.
3. C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In Proceedings of the 10th USENIX Security Symposium, August 2001.
4. J. J. C. Cowan, S. Beattie and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In Proceedings of the 12th USENIX Security Symposium, August 2003.
5. C. Cowan, C. Pu, D. Maier, M. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In Proc. of the 7th USENIX Security Symposium, January 1998.
6. M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In Proceedings of the 10th USENIX Security Symposium, August 2001.

7. I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In Proc. of the 5th USENIX Security Symposium, 1996.
8. V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program sheperding. In Proceedings of the 11th USENIX Security Symposium, 2002.
9. T. Detristan, T. Ulenspiegel, Y. Małcom, and M. Underduk. Polymorphic shellcode engine using spectrum analysis. Phrack, 11(61), Aug. 2003. http://www.phrack.org/phrack/61/p61-0x09_Polymorphic_Shellcode_Engine.txt.
10. C. Nachenberg. Understanding and managing polymorphic viruses. White Paper, July 1996. http://www.symantec.com/avcenter/reference/striker.pdf.
11. O. Kolesnikov, D. Dagon, and W. Lee. Advanced polymorphic worms: Evading IDS by blending in with normal traffic, 2004. http://www.cc.gatech.edu/ɑok/w/ok_pw.pdf.
12. P. Szor and P. Ferrie. Hunting for metamorphic. White Paper, Sept. 2001. http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf.
13. M. Roesch. Snort: Lightweight intrusion detection for networks. In Proceedings of USENIX LISA 99, Nov. 1999. (software available from http://www.snort.org/).
14. D. Ruiu. Fnord: Multi-architecture mutated NOP sled detector, Feb. 2002. http://www.cansecwest.com/spp_fnord.c.
15. T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID), Oct. 2002.
16. K2. ADMmutate. http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz.
17. Metasploit project, 2004. http://www.metasploit.com/.
18. IA-32 Intel Architecture Software Developer's Manual vol. 1-3. http://developer.intel.com/design/ pentium4/manuals/index_new.htm.
19. K. S. Gatlin. Windows data alignment on IPF, x86, and x86-64, Feb. 2003. MSDN Library, http://msdn.microsoft.com/.
20. Prelude IDS. http://www.prelude-ids.org/.
21. NIDSFindShellcode. http://www.ngsec.com/downloads/misc/NIDSfindshellcode.tgz.
22. T. Toth. Apache buffer overflow detector module, Mar. 2002. http://www.infosys.tuwien.ac.at/Staff/tt/abstract_execution/.