NOZZLE: A Defense Against Heap-spraying Code Injection Attacks

Paruj Ratanaworabhan
Cornell University
paruj@csl.cornell.edu

Benjamin Livshits
Microsoft Research
livshits@microsoft.com

Benjamin Zorn
Microsoft Research
zorn@microsoft.com

Abstract

Heap spraying is a security attack that increases the exploitability of memory corruption errors in type-unsafe applications. In a heap-spraying attack, an attacker coerces an application to allocate many objects containing malicious code in the heap, increasing the success rate of an exploit that jumps to a location within the heap. Because heap layout randomization necessitates new forms of attack, spraying has been used in many recent security exploits. Spraying is especially effective in web browsers, where the attacker can easily allocate the malicious objects using JavaScript embedded in a web page. In this paper, we describe NOZZLE, a runtime heap-spraying detector. NOZZLE examines individual objects in the heap, interpreting them as code and performing a static analysis on that code to detect malicious intent. To reduce false positives, we aggregate measurements across all heap objects and define a global heap health metric.

We measure the effectiveness of NOZZLE by demonstrating that it successfully detects 12 published and 2,000 synthetically generated heap-spraying exploits. We also show that even with a detection threshold set six times lower than is required to detect published malicious attacks, NOZZLE reports no false positives when run over 150 popular Internet sites. Using sampling and concurrent scanning to reduce overhead, we show that the performance overhead of NOZZLE is less than 7% on average. While NOZZLE currently targets heap-based spraying attacks, its techniques can be applied to any attack that attempts to fill the address space with malicious code objects (e.g., stack spraying [42]).

1 Introduction

In recent years, security improvements have made it increasingly difficult for attackers to compromise systems. Successful prevention measures in runtime environments and operating systems include stack protection [10], improved heap allocation layouts [7, 20], address space layout randomization [8, 36], and data execution prevention [21]. As a result, attacks that focus on exploiting memory corruptions in the heap are now popular [28].

Heap spraying, first described in 2004 by SkyLined [38], is an attack that allocates many objects containing the attacker's exploit code in an application's heap. Heap spraying is a vehicle for many high profile attacks, including a much publicized exploit in Internet Explorer in December 2008 [23] and a 2009 exploit of Adobe Reader using JavaScript embedded in malicious PDF documents [26].

Heap spraying requires that an attacker use another security exploit to trigger an attack, but the act of spraying greatly simplifies the attack and increases its likelihood of success because the exact addresses of objects in the heap do not need to be known. To perform heap spraying, attackers have to be able to allocate objects whose contents they control in an application's heap. The most common method used by attackers to achieve this goal is to target an application, such as a web browser, which executes an interpreter as part of its operation. By providing a web page with embedded JavaScript, an attacker can induce the interpreter to allocate their objects, allowing the spraying to occur. While this form of spraying attack is the most common, and the one we specifically consider in this paper, the techniques we describe apply to all forms of heap spraying. A number of variants of spraying attacks have recently been proposed including sprays involving compiled bytecode, ANI cursors [22], and thread stacks [42].

In this paper, we describe NOZZLE, a detector of heap spraying attacks that monitors heap activity and reports spraying attempts as they occur. To detect heap spraying attacks, NOZZLE has two complementary components. First, NOZZLE scans individual objects looking for signs of malicious intent. Malicious code commonly includes a landing pad of instructions (a so-called NOP sled) whose execution will lead to dangerous shellcode. NOZZLE focuses on detecting a sled through an analysis of its control flow. We show that prior work on sled detection [4, 16, 31, 43] has a high false positive rate when applied to objects in heap-spraying attacks (partly due to

the opcode density of the x86 instruction set). NOZZLE interprets individual objects as code and performs a static analysis, going beyond prior sled detection work by reasoning about code reachability. We define an attack surface metric that approximately answers the question: "If I were to jump randomly into this object (or heap), what is the likelihood that I would end up executing shellcode?"

In addition to local object detection, NOZZLE aggregates information about malicious objects across the entire heap, taking advantage of the fact that heap spraying requires large-scale changes to the contents of the heap. We develop a general notion of global "heap health" based on the measured attack surface of the applicatoin heap contents, and use this metric to reduce NOZZLE's false positive rates.

Because NOZZLE only examines object contents and requires no changes to the object or heap structure, it can easily be integrated into both native and garbage-collected heaps. In this paper, we implement NOZZLE by intercepting calls to the memory manager in the Mozilla Firefox browser (version 2.0.0.16). Because browsers are the most popular target of heap spray attacks, it is crucial for a successful spray detector to both provide high successful detection rates and low false positive rates. While the focus of this paper is on low-overhead online detection of heap spraying, NOZZLE can be easily used for offline scanning to find malicious sites in the wild [45]. For offline scanning, we can combine our spraying detector with other checkers such as those that match signatures against the exploit code, etc.

1.1 Contributions

This paper makes the following contributions:

- We propose the first effective technique for detecting heap-spraying attacks through runtime interpretation and static analysis. We introduce the concept of attack surface area for both individual objects and the entire heap. Because directing program control to shellcode is a fundamental property of NOP sleds, the attacker cannot hide that intent from our analysis.
- We show that existing published sled detection techniques [4, 16, 31, 43] have high false positive rates when applied to heap objects. We describe new techniques that dramatically lower the false positive rate in this context.
- We measure Firefox interacting with popular web sites and published heap-spraying attacks, we show that NOZZLE successfully detects 100% of 12 published and 2,000 synthetically generated heap-spraying exploits. We also show that even with a detection threshold set six times lower than is required to detect known malicious attacks, NOZZLE reports no false positives when tested on 150 popular Alexa.com sites.
- We measure the overhead of NOZZLE, showing that without sampling, examining every heap object slows execution 2–14 times. Using sampling and concurrent scanning, we show that the performance overhead of NOZZLE is less than 7% on average.
- We provide the results of applying NOZZLE to Adobe Reader to prevent a recent heap spraying exploit embedded in PDF documents. NOZZLE succeeds at stopping this attack without any modifications, with a runtime overhead of 8%.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 provides background on heap spraying attacks. Section 3 provides an overview of NOZZLE and Section 4 goes into the technical details of our implementation. Section 5 summarizes our experimental results. While NOZZLE is the first published heap spraying detection technique, our approach has several limitations, which we describe fully in Section 6. Finally, Section 7 describes related work and Section 8 concludes.

2 Background

Heap spraying has much in common with existing stack and heap-based code injection attacks. In particular, the attacker attempts to inject code somewhere in the address space of the target program, and through a memory corruption exploit, coerce the program to jump to that code. Because the success of stack-based exploits has been reduced by the introduction of numerous security measures, heap-based attacks are now common. Injecting and exploiting code in the heap is more difficult for an attacker than placing code on the stack because the addresses of heap objects are less predictable than those of stack objects. Techniques such as address space layout randomization [8, 36] further reduce the predictability of objects on the heap. Attackers have adopted several strategies for overcoming this uncertainty [41], with heap spraying the most successful approach.

Figure 1 illustrates a common method of implementing a heap-spraying attack. Heap spraying requires a memory corruption exploit, as in our example, where an attacker has corrupted a vtable method pointer to point to an incorrect address of their choosing. At the same time, we assume that the attacker has been able, through entirely legal methods, to allocate objects with contents of their choosing on the heap. Heap spraying relies on populating the heap with a large number of objects containing the attacker's code, assigning the vtable exploit to jump to an

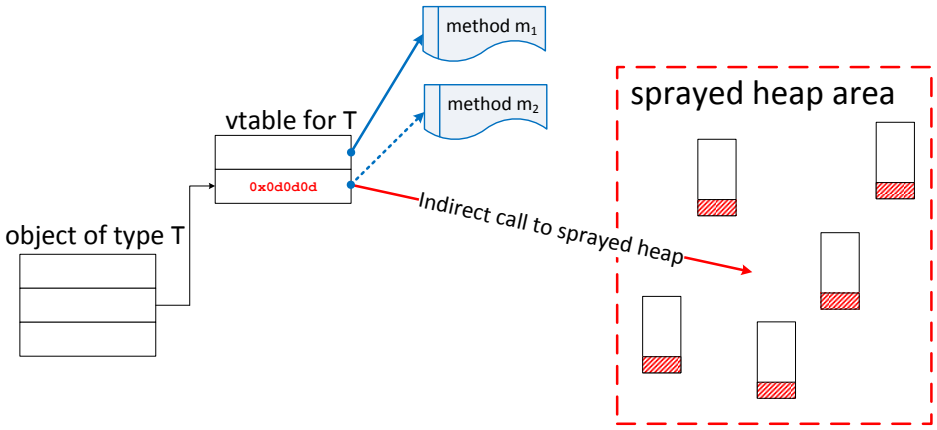

Figure 1: Schematic of a heap spraying attack.

```
 1. <SCRIPT language="text/javascript">
 2.   shellcode = unescape("%u4343%u4343%...");
 3.   oneblock = unescape("%u0D0D%u0D0D");
 4.
 5.   var fullblock = oneblock;
 6.   while (fullblock.length<0x40000) {
 7.       fullblock += fullblock;
 8.   }
 9.
10. sprayContainer = new Array();
11. for (i=0; i<1000; i++) {
12.     sprayContainer[i] = fullblock + shellcode;
13. }
14. </SCRIPT>
```

Figure 2: A typical JavaScript heap spray.

arbitrary address in the heap, and relying on luck that the jump will land inside one of their objects. To increase the likelihood that the attack will succeed, attackers usually structure their objects to contain an initial NOP sled (indicated in white) followed by the code that implements the exploit (commonly referred to as shellcode, indicated with shading). Any jump that lands in the NOP sled will eventually transfer control to the shellcode. Increasing the size of the NOP sled and the number of sprayed objects increases the probability that the attack will be successful.

Heap spraying requires that the attacker control the contents of the heap in the process they are attacking. There are numerous ways to accomplish this goal, including providing data (such as a document or image) that when read into memory creates objects with the desired properties. An easier approach is to take advantage of scripting languages to allocate these objects directly. Browsers are particularly vulnerable to heap spraying because JavaScript embedded in a web page authored by the attacker greatly simplifies such attacks.

The example shown in Figure 2 is modelled after a previously published heap-spraying exploit [44]. While we are only showing the JavaScript portion of the page, this payload would be typically embedded within an HTML page on the web. Once a victim visits the page, the JavaScript payload is automatically executed. Lines 2 allocates the shellcode into a string, while lines 3–8 of the JavaScript code are responsible for setting up the spraying NOP sled. Lines 10–13 create JavaScript objects each of which is the result of combining the sled with the shellcode. It is quite typical for published exploits to contain a long sled (256 KB in this case). Similarly, to increase the effectiveness of the attack, a large number of JavaScript objects are allocated on the heap, 1,000 in this case. Figure 10 in Section 5 provides more information on previously published exploits.

3 Overview

While type-safe languages such as Java, C#, and JavaScript reduce the opportunity for malicious attacks, heap-spraying attacks demonstrate that even a type-safe program can be manipulated to an attacker's advantage. Unfortunately, traditional signature-based pattern matching approaches used in the intrusion detection literature are not very effective when applied to detecting heap-spraying attacks. This is because in a language as flexible as JavaScript it is easy to hide the attack code by either using encodings or making it polymorphic; in fact, most JavaScript worms observed in the wild use some form of encoding to disguise themselves [19, 34]. As a result, effective detection techniques typically are not syntactic. They are performed at runtime and employ some level of semantic analysis or runtime interpretation. Hardware support has even been provided to address this problem, with widely used architectures supporting a "no-execute bit", which prevents a process from executing code on specific pages in its address space [21]. We dis-

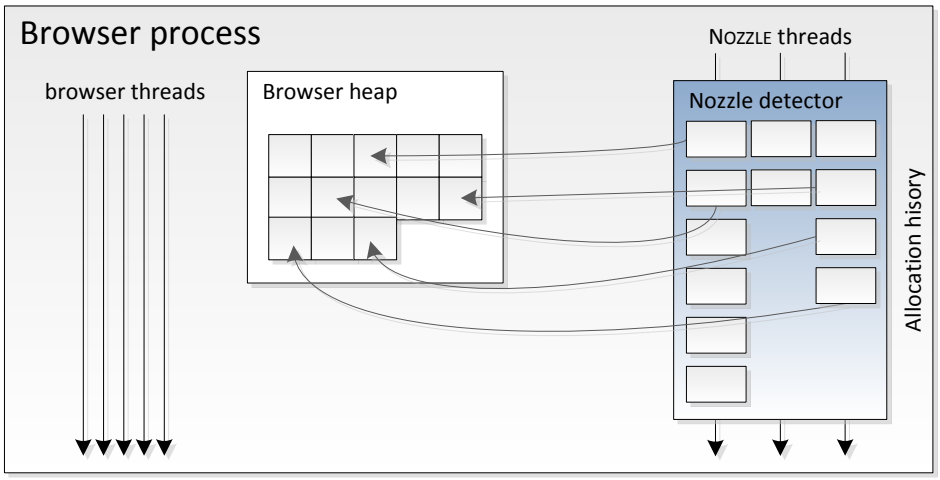

Figure 3: NOZZLE system architecture.

cuss how NOZZLE complements existing hardware solutions in Section 7. In this paper, we consider systems that use the x86 instruction set architecture (ISA) running the Windows operating system, a ubiquitous platform that is a popular target for attackers.

3.1 Lightweight Interpretation

Unlike previous security attacks, a successful heap-spraying attack has the property that the attack influences the contents of a large fraction of the heap. We propose a two-level approach to detecting such attacks: scanning objects locally while at the same time maintaining heap health metrics globally.

At the individual object level, NOZZLE performs lightweight interpretation of heap-allocated objects, treating them as though they were code. This allows us to recognize potentially unsafe code by interpreting it within a safe environment, looking for malicious intent.

The NOZZLE lightweight emulator scans heap objects to identify valid x86 code sequences, disassembling the code and building a control flow graph [35]. Our analysis focuses on detecting the NOP sled, which is somewhat of a misnomer. The sled can be composed of arbitrary instructions (not just NOPs) as long as the effect they have on registers, memory, and the rest of the machine state do not terminate execution or interfere with the actions of the shellcode. Because the code in the sled is intended to be the target of a misdirected jump, and thus has to be executable, the attacker cannot hide the sled with encryption or any means that would prevent the code from executing. In our analysis, we exploit the fundamental nature of the sled, which is to direct control flow specifically to the shellcode, and use this property as a means of detecting it. Furthermore, our method does not require detecting or assume there exists a definite partition between the shellcode and the NOP sled.

Because the attack jump target cannot be precisely controlled, the emulator follows control flow to identify basic blocks that are likely to be reached through jumps from multiple offsets into the object. Our local detection process has elements in common with published methods for sled detection in network packet processing [4, 16, 31, 43]. Unfortunately, the density of the x86 instruction set makes the contents of many objects look like executable code, and as a result, published methods lead to high false positive rates, as demonstrated in Section 5.1.

We have developed a novel approach to mitigate this problem using global heap health metrics, which effectively distinguishes benign allocation behavior from malicious attacks. Fortunately, an inherent property of heap-spraying attacks is that such attacks affect the heap globally. Consequently, NOZZLE exploits this property to drastically reduce the false positive rate.

3.2 Threat Model

We assume that the attacker has access to memory vulnerabilities for commonly used browsers and also can lure users to a web site whose content they control. This provides a delivery mechanism for heap spraying exploits. We assume that the attacker does not have further access to the victim's machine and the machine is otherwise uncompromised. However, the attacker does *not* control the precise location of any heap object.

We also assume that the attacker knows about the NOZZLE techniques and will try to avoid detection. They may have access to the browser code and possess detailed knowledge of system-specific memory layout properties

such as object alignment. There are specific potential weaknesses that NOZZLE has due to the nature of its runtime, statistical approach. These include time-of-check to time-of-use vulnerabilities, the ability of the attacker to target their attack under NOZZLE's thresholds, and the approach of inserting junk bytes at the start of objects to avoid detection. We consider these vulnerabilities carefully in Section 6, after we have presented our solution in detail.

4 Design and Implementation

In this section, we formalize the problem of heap spray detection, provide improved algorithms for detecting suspicious heap objects, and describe the implementation of NOZZLE.

4.1 Formalization

This section formalizes our detection scheme informally described in Section 3.1, culminating in the notion of a *normalized attack surface*, a heap-global metric that reflects the overall heap exploitability and is used by NOZZLE to flag potential attacks.

Definition 1. *A sequence of bytes is* legitimate*, if it can be decoded as a sequence of valid x86 instructions. In a variable length ISA this implies that the processor must be able to decode every instruction of the sequence. Specifically, for each instruction, the byte sequence consists of a valid opcode and the correct number of arguments for that instruction.*

Unfortunately, the x86 instruction set is quite dense, and as a result, much of the heap data can be interpreted as legitimate x86 instructions. In our experiments, about 80% of objects allocated by Mozilla Firefox contain byte sequences that can be interpreted as x86 instructions.

Definition 2. *A* valid instruction sequence *is a legitimate instruction sequence that does* not *include instructions in the following categories:*

- *I/O or system calls (*`in`*,* `outs`*, etc)*
- *interrupts (*`int`*)*
- *privileged instructions (*`hlt`*,* `ltr`*)*
- *jumps outside of the current object address range.*

These instructions either divert control flow out of the object's implied control flow graph or generate exceptions and terminate (privileged instructions). If they appear in a path of the NOP sled, they prevent control flow from reaching the shellcode via that path. When these instructions appear in the shellcode, they do not hamper the control flow in the NOP sled leading to that shellcode in any way.

Semi-lattice	L	bitvectors of length N
Top	$\top$	$\bar{1}$
Initial value	$init(B_i)$	$\bar{0}$
Transfer function	$TF(B_i)$	$0 \ldots 010 \ldots 0$ (ith bit set)
Meet operator	$\wedge(x, y)$	$x \vee y$ (bitwise or)
Direction		*forward*

Figure 4: Dataflow problem parametrization for computing the surface area (see Aho et al.).

Previous work on NOP sled detection focuses on examining possible attacks for properties like valid instruction sequences [4, 43]. We use this definition as a basic object filter, with results presented in Section 5.1. Using this approach as the sole technique for detecting attacks leads to an unacceptable number of false positives, and more selective techniques are necessary.

To improve our selectivity, NOZZLE attempts to discover objects in which control flow through the object (the NOP sled) frequently reaches the same basic block(s) (the shellcode, indicated in Figure 1), the assumption being that an attacker wants to arrange it so that a random jump into the object will reach the shellcode with the greatest probability.

Our algorithm constructs a control flow graph (CFG) by interpreting the data in an object at offset Δ as an instruction stream. For now, we consider this offset to be zero and discuss the implications of malicious code injected at a different starting offset in Section 6. As part of the construction process, we mark the basic blocks in the CFG as valid and invalid instruction sequences, and we modify the definition of a basic block so that it terminates when an invalid instruction is encountered. A block so terminated is considered an invalid instruction sequence. For every basic block within the CFG we compute the *surface area*, a proxy for the likelihood of control flow passing through the basic block, should the attacker jump to a random memory address within the object.

Algorithm 1. Surface area computation.

Inputs: *Control flow graph C consisting of*

- *Basic blocks $B_1, \ldots, B_N$*
- *Basic block weights, $\bar{W}$, a single-column vector of size N where element W_i indicates the size of block B_i in bytes*
- *A validity bitvector $\bar{V}$, a single-row bitvector whose ith element is set to one only when block B_i contains a valid instruction sequence and set to zero otherwise.*
- *$MASK_1, \ldots, MASK_N$, where $MASK_i$ is a single-row bitvector of size N where all the bits are one except at the i^{th} position where the bit is zero.*

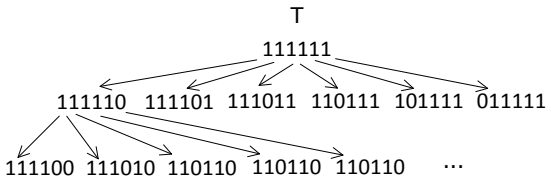

Figure 5: Semi-lattice used in Example 1.

Outputs: *Surface area for each basic block* $SA(B_i), B_i \in C$.

Solution: We define a parameterized dataflow problem using the terminology in Aho et al. [2], as shown in Figure 4. We also relax the definition of a conventional basic block; whenever an invalid instruction is encountered, the block prematurely terminates. The goal of the dataflow analysis is to compute the reachability between basic blocks in the control graph inferred from the contents of the object. Specifically, we want to determine whether control flow could possibly pass through a given basic block if control starts at each of the other $N-1$ blocks. Intuitively, if control reaches a basic block from many of the other blocks in the object (demonstrating a "funnel" effect), then that object exhibits behavior consistent with having a NOP sled and is suspicious.

Dataflow analysis details: The dataflow solution computes $out(B_i)$ for every basic block $B_i \in C$. $out(B_i)$ is a bitvector of length N, with one bit for each basic block in the control flow graph. The meaning of the bits in $out(B_i)$ are as follows: the bit at position j, where $j \neq i$ indicates whether a possible control path exists starting at block j and ending at block i. The bit at position i in B_i is always one. For example, in Figure 6, a path exists between block 1 and 2 (a fallthrough), and so the first bit of $out(B_2)$ is set to 1. Likewise, there is no path from block 6 to block 1, so the sixth bit of $out(B_1)$ is zero.

The dataflow algorithm computes $out(B_i)$ for each B_i by initializing them, computing the contribution that each basic block makes to $out(B_i)$, and propagating intermediate results from each basic block to its successors (because this is a forward dataflow computation). When results from two predecessors need to be combined at a join point, the meet operator is used (in this case a simple bitwise or). The dataflow algorithm iterates the forward propagation until the results computed for each B_i do not change further. When no further changes occur, the final values of $out(B_i)$ have been computed. The iterative algorithm for this forward dataflow problem is guaranteed to terminate in no more than the number of steps equal to the product of the semi-lattice height and the number of basic blocks in the control flow graph [2].

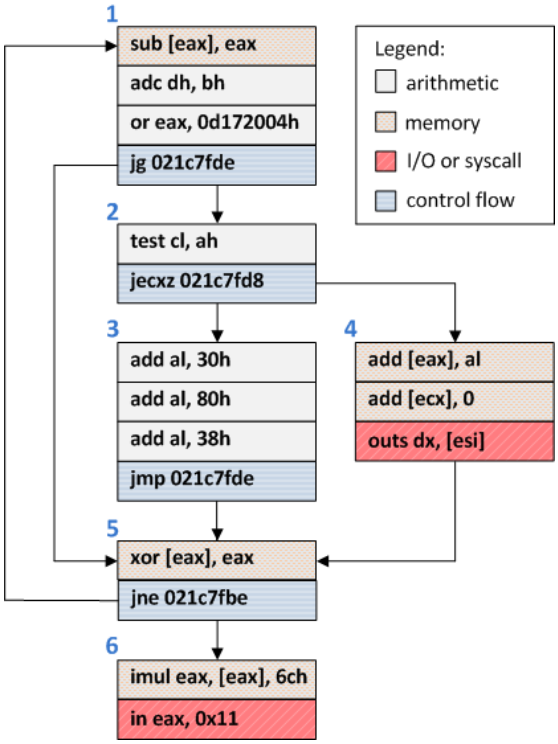

Figure 6: The control flow graph for Example 1.

Having calculated $out(B_i)$, we are now ready to compute the surface area of the basic block B_i. The surface area of a given block is a metric that indicates how likely the block will be reached given a random control flow landing on this object. The surface area of basic block B_i, $SA(B_i)$, is computed as follows:

$$SA(B_i) = (out(B_i) \wedge \bar{V} \wedge MASK_i) \cdot \bar{W}$$

where $out(B_i)$ is represented by a bitvector whose values are computed using the iterative dataflow algorithm above. $\bar{V}$, $\bar{W}$, and $MASK_i$ are the algorithm's inputs. $\bar{V}$ is determined using the validity criteria mentioned above, while $\bar{W}$ is the size of each basic block in bytes. $MASK_i$ is used to mask out the contribution of B_i's weight to its own surface area. The intuition is that we discard the contribution from the block itself as well as other basic blocks that are not valid instruction sequences by logically bitwise ANDing $out(B_i)$, $\bar{V}$, and $MASK_i$. Because the shellcode block does not contribute to actual attack surface (since a jump inside the shellcode is not likely to result in a successful exploit), we do not include the weight of B_i as part of the attack surface. Finally, we perform vector multiplication to account for the weight each basic block contributes—or does not—to the surface area of B_i.

In summary, the surface area computation based on the dataflow framework we described accounts for the contribution each basic block, through its weight and validity,

has on every other blocks reachable by it. Our computation method can handle code with complex control flow involving arbitrary nested loops. It also allows for the discovery of malicious objects even if the object has no clear partition between the NOP sled and the shellcode itself.

Complexity analysis. The standard iterative algorithm for solving dataflow problems computes $out(B_i)$ values with an average complexity bound of $O(N)$. The only complication is that doing the lattice meet operation on bitvectors of length N is generally an $O(N)$ and *not a constant time* operation. Luckily, for the majority of CFGs that arise in practice — 99.08% in the case of Mozilla Firefox opened and interacted on `www.google.com` — the number of basic blocks is fewer than 64, which allows us to represent dataflow values as long integers on 64-bit hardware. For those rare CFGs that contain over 64 basic blocks, a generic bitvector implementation is needed.

Example 1 Consider the CFG in Figure 6. The semilattice for this CFG of size 6 is partially shown in Figure 5. Instructions in the CFG are color-coded by instruction type. In particular, system calls and I/O instructions interrupt the normal control flow. For simplicity, we show $\bar{W}_i$ as the number of instructions in each block, instead of the number of bytes. The values used and produced by the algorithm are summarized in Figure 7. The $out'(B_i)$ column shows the intermediate results for dataflow calculation after the first pass. The final solution is shown in the $out(B_i)$ column. □

Given the surface area of individual blocks, we compute the *attack surface area* of object `o` as:

$$SA(o) = max(SA(B_i), B_i \in C)$$

For the entire heap, we accumulate the attack surface of the individual objects.

Definition 3. *The* attack surface area *of heap H, $SA(H)$, containing objects $o_1, \ldots, o_n$ is defined as follows:*

$$\sum_{i=1,\ldots,n} SA(o_i)$$

Definition 4. *The* normalized attack surface area *of heap H, denoted as $NSA(H)$, is defined as: $SA(H)/|H|$.*

The normalized attack surface area metric reflects the overall heap "health" and also allows us to adjust the frequency with which NOZZLE runs, thereby reducing the runtime overhead, as explained below.

4.2 Nozzle Implementation

NOZZLE needs to periodically scan heap object content in a way that is analogous to a garbage collector mark phase. By instrumenting allocation and deallocation routines, we maintain a table of live objects that are later scanned asynchronously, on a different NOZZLE thread.

We adopt garbage collection terminology in our description because the techniques are similar. For example, we refer to the threads allocating and freeing objects as the mutator threads, while we call the NOZZLE threads scanning threads. While there are similarities, there are also key differences. For example, NOZZLE works on an unmanaged, type-unsafe heap. If we had garbage collector write barriers, it would improve our ability to address the TOCTTOU (time-of-check to time-of-use) issue discussed in Section 6.

4.2.1 Detouring Memory Management Routines

We use a binary rewriting infrastructure called Detours [14] to intercept functions calls that allocate and free memory. Within Mozilla Firefox these routines are `malloc`, `calloc`, `realloc`, and `free`, defined in `MOZCRT19.dll`. To compute the surface area, we maintain information about the heap including the total size of allocated objects.

NOZZLE maintains a hash table that maps the addresses of currently allocated objects to information including size, which is used to track the current size and contents of the heap. When objects are freed, we remove them from the hash table and update the size of the heap accordingly. Note that if NOZZLE were more closely integrated into the heap allocator itself, this hash table would be unnecessary.

NOZZLE maintains an ordered work queue that serves two purposes. First, it is used by the scanning thread as a source of objects that need to be scanned. Second, NOZZLE waits for objects to mature before they are scanned, and this queue serves that purpose. Nozzle only considers objects of size greater than 32 bytes to be put in the work queue as the size of any harmful shellcode is usually larger than this

To reduce the runtime overhead of NOZZLE, we randomly sample a subset of heap objects, with the goal of covering a fixed fraction of the total heap. Our current sampling technique is based on sampling by object, but as our results show, an improved technique would base sampling frequency on bytes allocated, as some of the published attacks allocate a relatively small number of large objects.

4.2.2 Concurrent Object Scanning

We can reduce the performance impact of object scanning, especially on multicore hardware, with the help of multiple scanning threads. As part of program detouring, we rewrite the `main` function to allocate a pool of N scanning threads to be used by NOZZLE, as shown in Figure 2.

B_i	$TF(B_i)$	$\bar{V}_i$	$\bar{W}_i$	$out'(B_i)$	$out(B_i)$	$out(B_i) \wedge \bar{V} \wedge MASK_i$	$SA(B_i)$
1	100000	1	4	100000	111110	011010	8
2	010000	1	2	110000	111110	101010	10
3	001000	1	4	111000	111110	110010	8
4	000100	0	3	110100	111110	111010	12
5	000010	1	2	111110	111110	111000	10
6	000001	0	2	111111	111111	111010	12

Figure 7: Dataflow values for Example 1.

This way, a mutator only blocks long enough when allocating and freeing objects to add or remove objects from a per-thread work queue.

The task of object scanning is subdivided among the scanning threads the following way: for an object at address a, thread number

$$(a >> p) \ \% \ N$$

is responsible for both maintaining information about that object and scanning it, where p is the number of bits required to encode the operating system page size (typically 12 on Windows). In other words, to preserve the spatial locality of heap access, we are distributing the task of scanning *individual pages* among the N threads. Instead of maintaining a global hash table, each thread maintains a local table keeping track of the sizes for the objects it handles.

Object scanning can be triggered by a variety of events. Our current implementation scans objects once, after a fixed delay of one object allocation (i.e., we scan the previously allocated object when we see the next object allocated). This choice works well for JavaScript, where string objects are immutable, and hence initialized immediately after they are allocated. Alternately, if there are extra cores available, scanning threads could pro-actively rescan objects without impacting browser performance and reducing TOCTTOU vulnerabilities (see Section 6).

4.3 Detection and Reporting

NOZZLE maintains the values $NSA(H)$ and $SA(H)$ for the currently allocated heap H. The criteria we use to conclude that there is an attack in progress combines an absolute and a relative threshold:

$$(NSA(H) > th_{norm}) \wedge (SA(H) > th_{abs})$$

When this condition is satisfied, we warn the user about a potential security attack in progress and allow them to kill the browser process. An alternative would be to take advantage of the error reporting infrastructure built into modern browsers to notify the browser vendor.

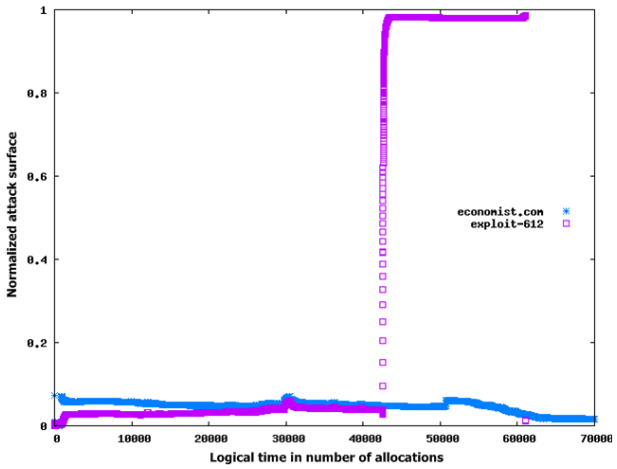

Figure 8: Global normalized attack surface for `economist.com` versus a published exploit (612).

These thresholds are defined based on a comparison of benign and malicious web pages (Section 5.1). The guiding principle behind the threshold determination is that for the attacker to succeed, the exploit needs to be effective with reasonable probability. For the absolute threshold, we choose five megabytes, which is roughly the size of the Firefox heap when opening a blank page. A real attack would need to fill the heap with at least as many malicious objects, assuming the attacker wanted the ratio of malicious to non-malicious objects to be greater than 50%.

5 Evaluation

The bulk of our evaluation focuses on applying NOZZLE to the Firefox web browser. Section 5.5 talks about using NOZZLE to protect Adobe Acrobat Reader.

We begin our evaluation by showing what a heap-spraying attack looks like as measured using our normalized attack surface metric. Figure 8 shows the attack surface area of the heap for two web sites: a benign site (`economist.com`), and a site with a published heap-spraying attack, similar to the one presented in Figure 2. Figure 8 illustrates how distinctive a heap-spraying attack

is when viewed through the normalized attack surface filter. The success of NOZZLE depends on its ability to distinguish between these two kinds of behavior. After seeing Figure 8, one might conclude that we can detect heap spraying activity based on how rapidly the heap grows. Unfortunately, benign web sites as `economist.com` can possess as high a heap growth rate as a rogue page performing heap spraying. Moreover, unhurried attackers may avoid such detection by moderating the heap growth rate of their spray. In this section, we present the false positive and false negative rate of NOZZLE, as well as its performance overhead, demonstrating that it can effectively distinguish benign from malicious sites.

For our evaluations, we collected 10 heavily-used benign web sites with a variety of content and levels of scripting, which we summarize in Figure 9. We use these 10 sites to measure the false positive rate and also the impact of NOZZLE on browser performance, discussed in Section 5.3. In our measurements, when visiting these sites, we interacted with the site as a normal user would, finding a location on a map, requesting driving directions, etc. Because such interaction is hard to script and reproduce, we also studied the false positive rate of NOZZLE using a total of 150 benign web sites, chosen from the most visited sites as ranked by Alexa [5][1]. For these sites, we simply loaded the first page of the site and measured the heap activity caused by that page alone.

To evaluates NOZZLE's ability to detect malicious attacks, we gathered 12 published heap-spraying exploits, summarized in Figure 10. We also created 2,000 synthetically generated exploits using the Metasploit framework [12]. Metasploit allows us to create many malicious code sequences with a wide variety of NOP sled and shellcode contents, so that we can evaluate the ability of our algorithms to detect such attacks. Metasploit is parameterizable, and as a result, we can create attacks that contain NOP sleds alone, or NOP sleds plus shellcode. In creating our Metasploit exploits, we set the ratio of NOP sled to shellcode at 9:1, which is quite a low ratio for a real attack but nevertheless presents no problems for NOZZLE detection.

5.1 False Positives

To evaluate the false positive rate, we first consider using NOZZLE as a global detector determining whether a heap is under attack, and then consider the false-positive rate of NOZZLE as a local detector that is attempting to detect individual malicious objects. In our evaluation, we compare NOZZLE and STRIDE [4], a recently published local detector.

[1]Our tech report lists the full set of sites used [32].

Site URL	Download (kilobytes)	JavaScript (kilobytes)	Load time (seconds)
`economist.com`	613	112	12.6
`cnn.com`	885	299	22.6
`yahoo.com`	268	145	6.6
`google.com`	25	0	0.9
`amazon.com`	500	22	14.8
`ebay.com`	362	52	5.5
`facebook.com`	77	22	4.9
`youtube.com`	820	160	16.5
`maps.google.com`	285	0	14.2
`maps.live.com`	3000	2000	13.6

Figure 9: Summary of 10 benign web sites we used as NOZZLE benchmarks.

Date	Browser	Description	milw0rm
11/2004	IE	IFRAME Tag BO	612
04/2005	IE	DHTML Objects Corruption	930
01/2005	IE	.ANI Remote Stack BO	753
07/2005	IE	`javaprxy.dll` COM Object	1079
03/2006	IE	`createTextRang` RE	1606
09/2006	IE	VML Remote BO	2408
03/2007	IE	ADODB Double Free	3577
09/2006	IE	WebViewFolderIcon `setSlice`	2448
09/2005	FF	0xAD Remote Heap BO	1224
12/2005	FF	`compareTo()` RE	1369
07/2006	FF	`Navigator` Object RE	2082
07/2008	Safari	Quicktime Content-Type BO	6013

Figure 10: Summary of information about 12 published heap-spraying exploits. BO stands for "buffer overruns" and RE stands for "remote execution."

5.1.1 Global False Positive Rate

Figure 11 shows the maximum normalized attack surface measured by NOZZLE for our 10 benchmark sites (top) as well as the top 150 sites reported by Alexa (bottom). From the figure, we see that the maximum normalized attack surface remains around 6% for most of the sites, with a single outlier from the 150 sites at 12%. In practice, the median attack surface is typically much lower than this, with the maximum often occurring early in the rendering of the page when the heap is relatively small. The `economist.com` line in Figure 8 illustrates this effect. By setting the spray detection threshold at 15% or above, we would observe no false positives in any of the sites measured.

5.1.2 Local False Positive Rate

In addition to being used as a heap-spray detector, NOZZLE can also be used locally as a malicious object detector. In this use, as with existing NOP and shellcode detectors such as STRIDE [4], a tool would report an object as potentially malicious if it contained data that could be interpreted as code, and had other suspicious properties. Previous work in this area focused on detection of malware in network packets and URIs, whose content is very different than heap objects. We evaluated NOZZLE

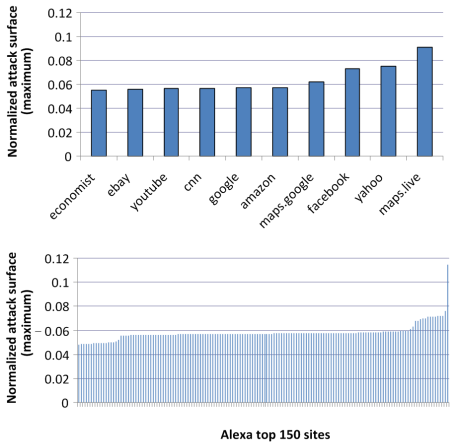

Figure 11: Global normalized attack surface for 10 benign benchmark web sites and 150 additional top Alexa sites, sorted by increasing surface. Each element of the X-axis represents a different web site.

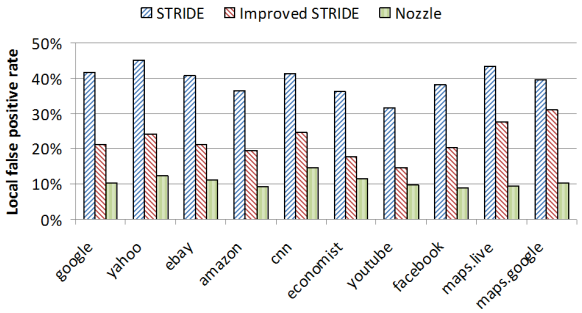

Figure 12: Local false positive rate for 10 benchmark web sites using NOZZLE and STRIDE. Improved STRIDE is a version of STRIDE that uses additional instruction-level filters, also used in NOZZLE, to reduce the false positive rate.

and STRIDE algorithm, to see how effective they are at classifying benign heap objects.

Figure 12 indicates the false positive rate of two variants of STRIDE and a simplified variant of NOZZLE. This simplified version of NOZZLE only scans a given heap object and attempts to disassemble and build a control flow graph from its contents. If it succeeds in doing this, it considers the object suspect. This version does not include any attack surface computation. The figure shows that, unlike previously reported work where the false positive rate for URIs was extremely low, the false positive rate for heap objects is quite high, sometimes above 40%. An improved variant of STRIDE that uses more information about the x86 instruction set (also used in NOZZLE) reduces this rate, but not below 10% in any case. We conclude that, unlike URIs or the content of network packets, heap objects often have contents that can be entirely interpreted as code on the x86 architecture. As a result, existing methods of sled detection do not directly apply to heap objects. We also show that even NOZZLE, without incorporating our surface area computation, would have an unacceptably high false positive rate.

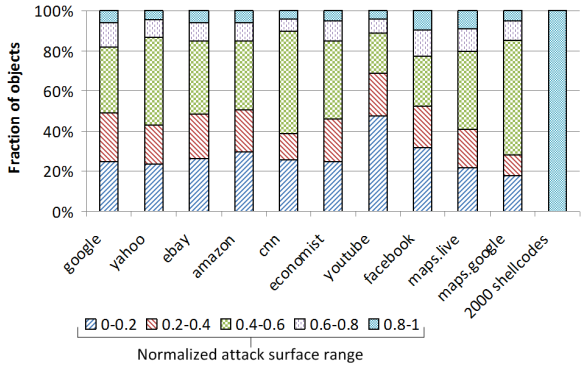

Figure 13: Distribution of filtered object surface area for each of 10 benchmark web sites (benign) plus 2,000 synthetic exploits (see Section 5.2). Objects measured are only those that were considered valid instruction sequences by NOZZLE (indicated as false positives in Figure 12.

To increase the precision of a local detector based on NOZZLE, we incorporate the surface area calculation described in Section 4. Figure 13 indicates the distribution of measured surface areas for the roughly 10% of objects in Figure 12 that our simplified version of NOZZLE was not able to filter. We see from the figure that many of those objects have a relatively small surface area, with less that 10% having surface areas from 80-100% of the size of the object (the top part of each bar). Thus, roughly 1% of objects allocated by our benchmark web sites qualify as suspicious by a local NOZZLE detector, compared to roughly 20% using methods reported in prior work. Even at 1%, the false positive rate of a local NOZZLE detector is too high to raise an alarm whenever a single instance of a suspicious object is observed, which motivated the development of our global heap health metric.

5.2 False Negatives

As with the false positive evaluation, we can consider NOZZLE both as a local detector (evaluating if NOZZLE is capable of classifying a known malicious object correctly), and as a global detector, evaluating whether it correctly detects web pages that attempt to spray many copies of malicious objects in the heap.

Figure 14 evaluates how effective NOZZLE is at avoid-

Configuration	min	mean	std
Local, NOP w/o shellcode	0.994	0.997	0.002
Local, NOP with shellcode	0.902	0.949	0.027

Figure 14: Local attack surface metrics for 2,000 generated samples from Metasploit with and without shellcode.

Configuration	min	mean	std
Global, published exploits	0.892	0.966	0.028
Global, Metasploit exploits	0.729	0.760	0.016

Figure 15: Global attack surface metrics for 12 published attacks and 2,000 Metasploit attacks integrated into web pages as heap sprays.

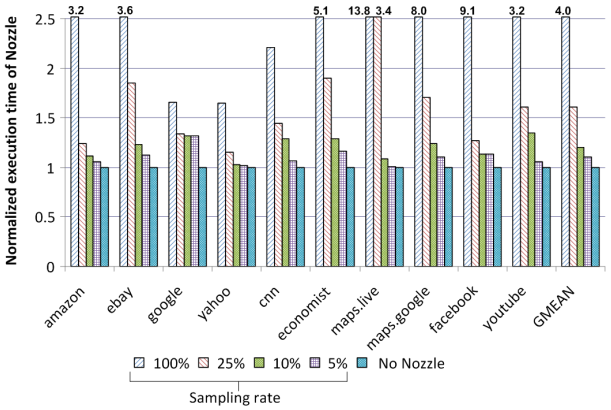

Figure 16: Relative execution overhead of using NOZZLE in rendering a typical page of 10 benchmark web sites as a function of sampling frequency.

ing local false negatives using our Metasploit exploits. The figure indicates the mean and standard deviation of the object surface area over the collection of 2,000 exploits, both when shellcode is included with the NOP sled and when the NOP sled is measured alone. The figure shows that NOZZLE computes a very high attack surface in both cases, effectively detecting all the Metasploit exploits both with and without shellcode.

Figure 15 shows the attack surface statistics when using NOZZLE as a global detector when the real and synthetic exploits are embedded into a web page as a heap-spraying attack. For the Metasploit exploits which were not specifically generated to be heap-spraying attacks, we wrote our own JavaScript code to spray the objects in the heap. The figure shows that the published exploits are more aggressive than our synthetic exploits, resulting in a mean global attack surface of 97%. For our synthetic use of spraying, which was more conservative, we still measured a mean global attack surface of 76%. Note that if we set the NOP sled to shellcode at a ratio lower than 9:1, we will observe a correspondingly smaller value for the mean global attack surface. All attacks would be detected by NOZZLE with a relatively modest threshold setting of 50%. We note that these exploits have global attack surface metrics 6–8 times larger than the maximum measured attack surface of a benign web site.

5.3 Performance

To measure the performance overhead of NOZZLE, we cached a typical page for each of our 10 benchmark sites. We then instrument the start and the end of the page with the JavaScript `newDate().getTime()` routine and compute the delta between the two. This value gives us how long it takes to load a page in milliseconds. We collect our measurements running Firefox version 2.0.0.16 on a 2.4 GHz Intel Core2 E6600 CPU running Windows XP Service Pack 3 with 2 gigabytes of main memory. To minimize the effect of timing due to cold start disk I/O, we first load a page and discard the timing measurement. After this first trial, we take three measurements and present the median of the three values. The experiments were performed on an otherwise quiescent machine and the variance between runs was not significant.

In the first measurement, we measured the overhead of NOZZLE without leveraging an additional core, i.e., running NOZZLE as a single thread and, hence, blocking Firefox every time a memory allocation occurs. The resulting overhead is shown in Figure 16, both with and without sampling. The overhead is prohibitively large when no sampling is applied. On average, the no sampling approach incurs about 4X slowdown with as much as 13X slowdown for `maps.live.com`. To reduce this overhead, we consider the sampling approach. For these results, we sample based on object counts; for example, sampling at 5% indicates that one in twenty objects is sampled. Because a heap-spraying attack has global impact on the heap, sampling is unlikely to significantly reduce our false positive and false negative rates, as we show in the next section. As we reduce the sampling frequency, the overhead becomes more manageable. We see an average slowdown of about 60%, 20% and 10% for sampling frequency of 25%, 10% and 5%, respectively, for the 10 selected sites.

For the second measurement, taking advantage of the second core of our dual core machine, we configured NOZZLE to use one additional thread for scanning, hence, unblocking Firefox when it performs memory allocation. Figure 17 shows the performance overhead of NOZZLE with parallel scanning. From the Figure, we see that with no sampling, the overhead of using NOZZLE ranges from 30% to almost a factor of six, with a geometric mean of

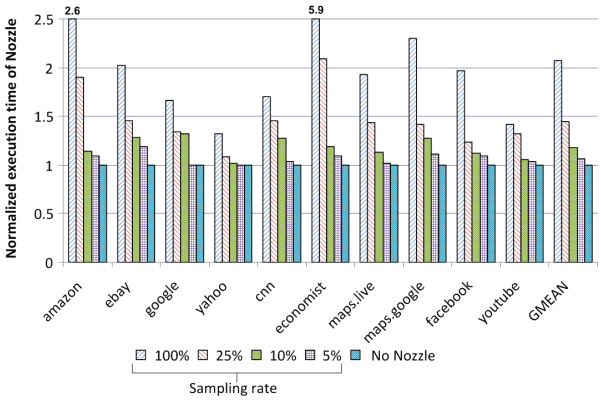

Figure 17: Overhead of using NOZZLE on a dual-core machine.

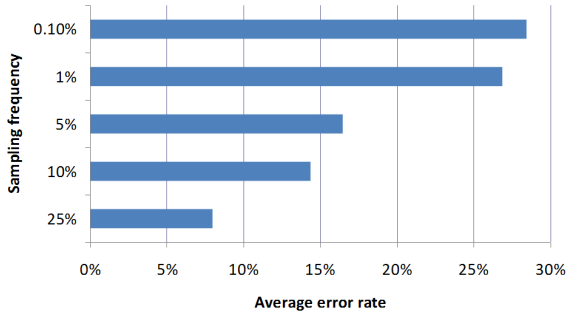

Figure 18: Average error rate due to sampling of the computed average surface area for 10 benign benchmark web sites.

two times slowdown. This is a significant improvement over the serial version. When we further reduce the sampling rate, we see further performance improvement as with the first measurement. Reducing the sampling rate to 25%, the mean overhead drops to 45%, while with a sampling rate of 5%, the performance overhead is only 6.4%.

5.4 Impact of Sampling on Detection

In this section, we show the impact of sampling on the amount of error in the computation of the attack surface metric for both benign and malicious inputs.

Figure 18 shows the error rate caused by different levels of sampling averaged across the 10 benign web sites. We compute the error rate $E = |Sampled - Unsampled|/Unsampled$. The figure shows that for sample rates of 0.1% or above the error rate is less than 30%. To make this concrete, for a benign website, instead of calculating the normalized attack surface correctly as 5%, with a 0.1% sampling rate, we would instead calculate the normalized attack surface as 6.5%, still far below any threshold we might use for signaling an attack. Noting that the malicious pages have attack surfaces that are 6–8 times larger than benign web pages, we conclude that sampling even at 5% is unlikely to result in significant numbers of false positives.

	Sampling Rate				
	100%	25%	10%	5%	1%
12 Published	0	0	0	0	50%
2,000 Metasploit	0	0	0	0	0

Figure 19: False negative rate for 12 real and 2,000 Metasploit attacks given different object sampling rates.

In Figure 19, we show the impact of sampling on the number of false negatives for our published and synthetic exploits. Because existing exploits involve generating the heap spray in a loop, the only way sampling will miss such an attack is to sample at such a low rate that the objects allocated in the loop escape notice. The figure illustrates that for published attacks sampling even at 5% results in no false negatives. At 1%, because several of the published exploits only create on the order of tens of copies of very large spray objects, sampling based on object count can miss these objects, and we observe a 50% (6/12) false negative rate. Sampling based on bytes allocated instead of objects allocated would reduce this false negative rate to zero.

5.5 Case Study: Adobe Reader Exploit

In February 2009, a remote code execution vulnerability was discovered in Adobe Acrobat and Adobe Reader [26]. The attack, which is still active on the Internet as of the time of this writing, exploited an integer overflow error and was facilitated by a JavaScript heap spray. Without making any modifications to NOZZLE, we used Detours to instrument the commercially-distributed binary of Adobe Reader 9.1.0 (`acrord32.exe`) with NOZZLE. The instrumentation allowed us to monitor the memory allocations being performed by the embedded JavaScript engine and detect possible spraying attacks. To test whether NOZZLE would detect this new attack, we embedded the heap spraying part of the published attack [6], disabling the JavaScript that caused the integer overflow exploit.

NOZZLE correctly detected this heap spraying attack, determining that the attack surface of the heap was greater than 94% by the time the heap spray was finished. No modifications were made either to the NOZZLE implementation or the surface area calculation to enable NOZZLE to detect this attack, which gives us confidence that NOZZLE is capable of protecting a wide range of software, going well beyond just web browsers.

To facilitate overhead measurements, we created a large document by concatenating six copies of the ECMA 262 standard — a 188-page PDF document [11] — with itself. The resulting document was 1,128 pages long and took 4,207 kilobytes of disk space. We added scripting code to the document to force Adobe Reader to "scroll" through this large document, rendering every page sequentially. We believe this workload to be representative of typical Adobe Reader usage, where the user pages through the document, one page at a time.

We measured the overhead of NOZZLE running in Adobe Reader on an Intel Core 2 2.4 GHz computer with 4 GB of memory running Windows Vista SP1. We measured elapsed time for Adobe Reader with and without NOZZLE on a lightly loaded computer and averaged five measurements with little observed variation. Without NOZZLE, Adobe Reader took an average of 18.7 seconds to render all the pages, and had a private working set of 18,772 kilobytes as measured with the Windows Task Manager. With a sampling rate set to 10% and multiprocessor scanning disabled, Adobe Reader with NOZZLE took an average of 20.2 seconds to render the pages, an average CPU overhead of 8%. The working set of Adobe Reader with NOZZLE average 31,849 kilobytes, an average memory overhead of 69%. While the memory overhead is high, as mentioned, we anticipate that this overhead could easily be reduced by integrating NOZZLE more closely with the underlying memory manager.

6 Limitations of NOZZLE

This section discusses assumptions and limitations of the current version of NOZZLE. In summary, assuming that the attackers are fully aware of the NOZZLE internals, there are a number of ways to evade its detection.

- As NOZZLE scans objects at specific times, an attacker could determine that an object has been scanned and arrange to put malicious content into the object only *after* it has been scanned (a TOCTTOU vulnerability).
- As NOZZLE currently starts scanning each object at offset zero, attackers can avoid detection by writing the first few bytes of the malicious object with a series of uninterpretable opcodes.
- Since NOZZLE relies on the use of a threshold for detection, attackers can populate the heap with fewer malicious objects to stay just under the detection threshold.
- Attackers can find ways to inject the heap with sprays that do not require large NOP sleds. For example, sprays with jump targets that are at fixed offsets in every sprayed page of memory are possible [39].
- Attackers can confuse NOZZLE's surface area measurement by designing attacks that embed multiple shellcodes within the same object or contain cross-object jumps.

Below we discuss these issues, their implications, and possible ways to address them.

6.1 Time-of-check to Time-of-use

Because NOZZLE examines object contents only at specific times, this leads to a potential time-of-check to time-of-use (TOCTTOU) vulnerability. An attacker aware that NOZZLE was being used could allocate a benign object, wait until NOZZLE scans it, and then rapidly change the object into a malicious one before executing the attack.

With JavaScript-based attacks, since JavaScript `String`s are immutable, this is generally only possible using JavaScript `Array`s. Further, because NOZZLE may not know when objects are completely initialized, it may scan them prematurely, creating another TOCTTOU window. To address this issue, NOZZLE scans objects once they mature on the assumption that most objects are written once when initialized, soon after they are allocated. In the future, we intend to investigate other ways to reduce this vulnerability, including periodically rescanning objects. Rescans could be triggered when NOZZLE observes a significant number of heap stores, perhaps by reading hardware performance counters.

Moreover, in the case of a garbage-collected language such as JavaScript or Java, NOZZLE can be integrated directly with the garbage collector. In this case, changes observed via GC *write barriers* [29] may be used to trigger NOZZLE scanning.

6.2 Interpretation Start Offset

In our discussion so far, we have interpreted the contents of objects as instructions starting at offset zero in the object, which allows NOZZLE to detect the current generation of heap-spraying exploits. However, if attackers are aware that NOZZLE is being used, they could arrange to fool NOZZLE by inserting junk bytes at the start of objects. There are several reasons that such techniques will not be as successful as one might think. To counter the most simplistic such attack, if there are invalid or illegal instructions at the beginning of the object, NOZZLE skips bytes until it finds the first valid instruction.

Note that while it may seem that the attacker has much flexibility to engineer the offset of the start of the malicious code, the attacker is constrained in several important ways. First, we know that it is likely that the attacker is trying to maximize the probability that the attack will succeed. Second, recall that the attacker has to corrupt a control transfer but does not know the specific address in an

object where the jump will land. If the jump lands on an invalid or illegal instruction, then the attack will fail. As a result, the attacker may seek to make a control transfer to every address in the malicious object result in an exploit. If this is the case, then NOZZLE will correctly detect the malicious code. Finally, if the attacker knows that NOZZLE will start interpreting the data as instructions starting at a particular offset, then the attacker might intentionally construct the sled in such a way that the induced instructions starting at one offset look benign, while the induced instructions starting at a different offset are malicious. For example, the simplist form of this kind of attack would have uniform 4-byte benign instructions starting at byte offset 0 and uniform malicious 4-byte instructions starting at byte offset 2. Note also that these overlapped sequences cannot share any instruction boundaries because if they did, then processing instructions starting at the benign offset would eventually discover malicious instructions at the point where the two sequences merged.

While the current version of NOZZLE does not address this specific simple case, NOZZLE could be modified to handle it by generating multiple control flow graphs at multiple starting offsets. Furthermore, because x86 instructions are typically short, most induced instruction sequences starting at different offsets do not have many possible interpretations before they share a common instruction boundary and merge. While it is theoretically possible for a determined attacker to create a non-regular, non-overlapping sequence of benign and malicious instructions, it is not obvious that the malicious sequence could not be discovered by performing object scans at a small number of offsets into the object. We leave an analysis of such techniques for future work.

6.3 Threshold Setting

The success of heap spraying is directly proportional to the density of dangerous objects in the program heap, which is approximated by NOZZLE's normalized attack surface metric. Increasing the number of sprayed malicious objects increases the attacker's likelihood of success, but at the same time, more sprayed objects will increase the likelihood that NOZZLE will detect the attack. What is worse for the attacker, failing attacks often result in program crashes. In the browser context, these are recorded on the user's machine and sent to browser vendors using a crash agent such as Mozilla Crash reporting [24] for per-site bucketing and analysis.

What is interesting about attacks against browsers is that from a purely financial standpoint, the attacker has a strong incentive to produce exploits with a high likelihood of success. Indeed, assuming the attacker is the one discovering the vulnerability such as a dangling pointer or buffer overrun enabling the heap-spraying attack, they can sell their finding directly to the browser maker. For instance, the Mozilla Foundation, the makers of Firefox, offers a cash reward of $500 for every exploitable vulnerability [25]. This represents a conservative estimate of the financial value of such an exploit, given that Mozilla is a non-profit and commercial browser makes are likely to pay more [15]. A key realization is that in many cases heap spraying is used for direct financial gain. A typical way to monetize a heap-spraying attack is to take over a number of unsuspecting users' computers and have them join a botnet. A large-scale botnet can be sold on the black market to be used for spamming or DDOS attacks.

According to some reports, to cost of a large-scale botnet is about $.10 per machine [40, 18]. So, to break even, the attacker has to take over at least 5,000 computers. For a success rate α, in addition to 5,000 successfully compromised machines, there are $5,000 \times (1-\alpha)/\alpha$ failed attacks. Even a success rate as high as 90%, the attack campaign will produce failures for 555 users. Assuming these result in crashes reported by the crash agent, this many reports from a single web site may attract attention of the browser maker. For a success rate of 50%, the browser make is likely to receive 5,000 crash reports, which should lead to rapid detection of the exploit!

As discussed in Section 5, in practice we use the relative threshold of 50% with Nozzle. We do not believe that a much lower success rate is financially viable from the standpoint of the attacker.

6.4 Targeted Jumps into Pages

One approach to circumventing NOZZLE detection is for the attacker to eliminate the large NOP sled that heap sprays typically use. This may be accomplished by allocating page-size chunks of memory (or multiples thereof) and placing the shellcode at fixed offsets on every page [39]. While our spraying detection technique currently will not discover such attacks, it is possible that the presence of possible shellcode at fixed offsets on a large number of user-allocated pages can be detected by extending NOZZLE, which we will consider in future work.

6.5 Confusing Control Flow Patterns

NOZZLE attempts to find basic blocks that act as sinks for random jumps into objects. One approach that will confuse NOZZLE is to include a large number of copies of shellcode in an object such that no one of them has a high surface area. Such an approach would still require that a high percentage of random jumps into objects result in non-terminating control flow, which might also be used as a trigger for our detector.

Even more problematic is an attack where the attacker includes inter-object jumps, under the assumption that,

probabilistically, there will be a high density of malicious objects and hence jumps outside of the current object will still land in another malicious object. NOZZLE currently assumes that jumps outside of the current object will result in termination. We anticipate that our control flow analysis can be augmented to detect groupings of objects with possible inter-object control flow, but we leave this problem for future work.

6.6 Summary

In summary, there are a number of ways that clever attackers can defeat NOZZLE's current analysis techniques. Nevertheless, we consider NOZZLE an important first step to detecting heap spraying attacks and we believe that improvements to our techniques are possible and will be implemented, just as attackers will implement some of the possible exploits described above.

The argument for using NOZZLE, despite the fact that hackers will find ways to confound it, is the same reason that virus scanners are installed almost ubiquitously on computer systems today: it will detect and prevent many known attacks, and as new forms of attacks develop, there are ways to improve its defenses as well. Ultimately, NOZZLE, just like existing virus detectors, is just one layer of a defense in depth.

7 Related Work

This section discusses exploit detection and memory attack prevention.

7.1 Exploit Detection

As discussed, a code injection exploit consists of at least two parts: the NOP sled and shellcode. Detection techniques target either or both of these parts. Signature-based techniques, such as Snort [33], use pattern matching, including possibly regular expressions, to identify attacks that match known attacks in their database. A disadvantage of this approach is that it will fail to detect attacks that are not already in the database. Furthermore, polymorphic malware potentially vary its shellcode on every infection attempt, reducing the effectiveness of pattern-based techniques. Statistical techniques, such as Polygraph [27], address this problem by using improbable properties of the shellcode to identify an attack.

The work most closely related to NOZZLE is Abstract Payload Execution (APE), by Toth and Kruegel [43], and STRIDE, by Akritidis et al. [4, 30], both of which present methods for NOP sled detection in network traffic. APE examines sequences of bytes using a technique they call *abstract execution* where the bytes are considered valid and correct if they represent processor instructions with legal memory operands. APE identifies sleds by computing the execution length of valid and correct sequences, distinguishing attacks by identifying sufficiently long sequences.

The authors of STRIDE observe that by employing jumps, NOP sleds can be effective even with relatively short valid and correct sequences. To address this problem, they consider all possible subsequences of length n, and detect an attack only when all such subsequences are considered valid. They demonstrate that STRIDE handles attacks that APE does not, showing also that tested over a large corpus or URIs, their method has an extremely low false positive rate. One weakness of this approach, acknowledged by the authors is that "...a worm writer could blind STRIDE by adding invalid instruction sequences at suitable locations..." ([30], p. 105).

NOZZLE also identifies NOP sleds, but it does so in ways that go beyond previous work. First, prior work has not considered the specific problem of sled detection in heap objects or the general problem of heap spraying, which we do. Our results show that applying previous techniques to heap object results in high false positive rates. Second, because we target heap spraying specifically, our technique leverages properties of the entire heap and not just individual objects. Finally, we introduce the concept of surface area as a method for prioritizing the potential threat of an object, thus addressing the STRIDE weakness mentioned above.

7.2 Memory Attack Prevention

While NOZZLE focuses on detecting heap spraying based on object and heap properties, other techniques take different approaches. Recall that heap spraying requires an additional memory corruption exploit, and one method of preventing a heap-spraying attack is to ignore the spray altogether and prevent or detect the initial corruption error. Techniques such as control flow integrity [1], write integrity testing [3], data flow integrity [9], and program shepherding [17] take this approach. Detecting all such possible exploits is difficult and, while these techniques are promising, their overhead has currently prevented their widespread use.

Some existing operating systems also support mechanisms, such as Data Execution Prevention (DEP) [21], which prevent a process from executing code on specific pages in its address space. Implemented in either software or hardware (via the no-execute or "NX" bit), execution protection can be applied to vulnerable parts of an address space, including the stack and heap. With DEP turned on, code injections in the heap cannot execute.

While DEP will prevent many attacks, we believe that NOZZLE is complementary to DEP for the following rea-

sons. First, security benefits from defense-in-depth. For example, attacks that first turn off DEP have been published, thereby circumventing its protection [37]. Second, compatibility issues can prevent DEP from being used. Despite the presence of NX hardware and DEP in modern operating systems, existing commercial browsers, such as Internet Explorer 7, still ship with DEP disabled by default [13]. Third, runtime systems that perform just-in-time (JIT) compilation may allocate JITed code in the heap, requiring the heap to be executable. Finally, code injection spraying attacks have recently been reported in areas other than the heap where DEP cannot be used. Sotirov and Dowd describe spraying attacks that introduce exploit code into a process address space via embedded .NET User Controls [42]. The attack, which is disguised as one or more .NET managed code fragments, is loaded in the process text segment, preventing the use of DEP. In future work, we intend to show that NOZZLE can be effective in detecting such attacks as well.

8 Conclusions

We have presented NOZZLE, a runtime system for detecting and preventing heap-spraying security attacks. Heap spraying has the property that the actions taken by the attacker in the spraying part of the attack are legal and type safe, allowing such attacks to be initiated in JavaScript, Java, or C#. Attacks using heap spraying are on the rise because security mitigations have reduced the effectiveness of previous stack and heap-based approaches.

NOZZLE is the first system specifically targeted at detecting and preventing heap-spraying attacks. NOZZLE uses lightweight runtime interpretation to identify specific suspicious objects in the heap and maintains a global heap health metric to achieve low false positive and false negative rates, as measured using 12 published heap spraying attacks, 2,000 synthetic malicious exploits, and 150 highly-visited benign web sites. We show that with sampling, the performance overhead of NOZZLE can be reduced to 7%, while maintaining low false positive and false negative rates. Similar overheads are observed when NOZZLE is applied to Adobe Acrobat Reader, a recent target of heap spraying attacks. The fact that NOZZLE was able to thwart a real published exploit when applied to the Adobe Reader binary, without requiring any modifications to our instrumentation techniques, demonstrates the generality of our approach.

While we have focused our experimental evaluation on heap-spraying attacks exclusively, we believe that the our techniques are more general. In particular, in future work, we intend to investigate using our approach to detect a variety of exploits that use code masquarading as data, such as images, compiled bytecode, etc. [42].

In the future, we intend to further improve the selectivity of the NOZZLE local detector, demonstrate NOZZLE's effectiveness for attacks beyond heap spraying, and further tune NOZZLE's performance. Because heap-spraying attacks can be initiated in type-safe languages, we would like to evaluate the cost and effectiveness of incorporating NOZZLE in a garbage-collected runtime. We are also interested in extending NOZZLE from detecting heap-spraying attacks to tolerating them as well.

Acknowledgements

We thank Emery Berger, Martin Burtscher, Silviu Calinoiu, Trishul Chilimbi, Tal Garfinkel, Ted Hart, Karthik Pattabiraman, Patrick Stratton, and Berend-Jan "SkyLined" Wever for their valuable feedback during the development of this work. We also thank our anonymous reviewers for their comments.

References

[1] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the Conference on Computer and Communications Security*, pages 340–353, 2005.

[2] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 263–277, 2008.

[4] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. G. Anagnostakis. STRIDE: Polymorphic sled detection through instruction sequence analysis. In R. Sasaki, S. Qing, E. Okamoto, and H. Yoshiura, editors, *Proceedings of Security and Privacy in the Age of Ubiquitous Computing*, pages 375–392. Springer, 2005.

[5] Alexa Inc. Global top sites. `http://www.alexa.com/site/ds/top_sites`, 2008.

[6] Arr1val. Exploit made by Arr1val proved in Adobe 9.1 and 8.1.4 on Linux. `http://downloads.securityfocus.com/vulnerabilities/exploits/34736.txt`, Feb. 2009.

[7] E. D. Berger and B. G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 158–168, 2006.

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

[9] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 147–160, 2006.

[10] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. *Foundations of Intrusion Tolerant Systems*, pages 227–237, 2003.

[11] ECMA. ECMAScript language specification. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, 1999.

[12] J. C. Foster. *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research*. Syngress Publishing, 2007.

[13] M. Howard. Update on Internet Explorer 7, DEP, and Adobe software. `http://blogs.msdn.com/michael_howard/archive/2006/12/12/update-on-internet-explorer-7-dep-and-adobe-software.aspx`, 2006.

[14] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *In Proceedings of the USENIX Windows NT Symposium*, pages 135–143, 1999.

[15] iDefense Labs. Annual vulnerability challenge. `http://labs.idefense.com/vcp/challenge.php`, 2007.

[16] I.-K. Kim, K. Kang, Y. Choi, D. Kim, J. Oh, and K. Han. A practical approach for detecting executable codes in network traffic. In S. Ata and C. S. Hong, editors, *Proceedings of Managing Next Generation Networks and Services*, volume 4773 of *Lecture Notes in Computer Science*, pages 354–363. Springer, 2007.

[17] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the USENIX Security Symposium*, pages 191–206, 2002.

[18] J. Leyden. Phatbot arrest throws open trade in zombie PCs. `http://www.theregister.co.uk/2004/05/12/phatbot_zombie_trade`, May 2004.

[19] B. Livshits and W. Cui. Spectator: Detection and containment of JavaScript worms. In *Proceedings of the USENIX Annual Technical Conference*, pages 335–348, June 2008.

[20] A. Marinescu. Windows Vista heap management enhancements. In *BlackHat US*, 2006.

[21] Microsoft Corporation. Data execution prevention. `http://technet.microsoft.com/en-us/library/cc738483.aspx`, 2003.

[22] Microsoft Corporation. Microsoft Security Bulletin MS07-017. `http://www.microsoft.com/technet/security/Bulletin/MS07-017.mspx`, Apr. 2007.

[23] Microsoft Corporation. Microsoft Security Advisory (961051). `http://www.microsoft.com/technet/security/advisory/961051.mspx`, Dec. 2008.

[24] Mozilla Developer Center. Crash reporting page. `https://developer.mozilla.org/En/Crash_reporting`, 2008.

[25] Mozilla Security Group. Mozilla security bug bounty program. `http://www.mozilla.org/security/bug-bounty.html`, 2004.

[26] Multi-State Information Sharing and Analysis Center. Vulnerability in Adobe Reader and Adobe Acrobat could allow remote code execution. `http://www.msisac.org/advisories/2009/2009-008.cfm`, Feb. 2009.

[27] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 226–241, 2005.

[28] J. D. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004.

[29] P. P. Pirinen. Barrier techniques for incremental tracing. In *Proceedings of the International Symposium on Memory Management*, pages 20–25, 1998.

[30] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *Proceedings of Symposium on Recent Advances in Intrusion Detection*, pages 87–106, 2007.

[31] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level polymorphic shellcode detection using emulation. *Journal in Computer Virology*, 2(4):257–274, 2007.

[32] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. Technical Report MSR-TR-2008-176, Microsoft Research, Nov. 2008.

[33] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the USENIX conference on System administration*, pages 229–238, 1999.

[34] Samy. The Samy worm. `http://namb.la/popular/`, Oct. 2005.

[35] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 45–54, 2002.

[36] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the Conference on Computer and Communications Security*, pages 298–307, 2004.

[37] Skape and Skywing. Bypassing windows hardware-enforced DEP. *Uninformed Journal*, 2(4), Sept. 2005.

[38] SkyLined. Internet Explorer IFRAME src&name parameter BoF remote compromise. `http://skypher.com/wiki/index.php?title=Www.edup.tudelft.nl/~bjwever/advisory/_iframe.html.php`, 2004.

[39] SkyLined. Personal communication, 2009.

[40] Sophos Inc. Stopping zombies, botnets and other email- and web-borne threats. `http://blogs.piercelaw.edu/tradesecretsblog/SophosZombies072507.pdf`, 12 2006.

[41] A. Sotirov. Heap feng shui in JavaScript. In *Proceedings of Blackhat Europe*, 2007.

[42] A. Sotirov and M. Dowd. Bypassing browser memory pro-

tections. In *Proceedings of BlackHat*, 2008.

[43] T. Toth and C. Krügel. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of Symposium on Recent Advances in Intrusion Detection*, pages 274–291, 2002.

[44] R. van den Heetkamp. Heap spraying. `http://www.0x000000.com/index.php?i=412&bin=110011100`, Aug. 2007.

[45] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with Strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS 2006)*, Feb. 2006.