Nozzle: A Defense Against Heap-spraying Code Injection Attacks

Paruj Ratanaworabhan
Cornell University
Ithaca, NY

Benjamin Livshits and Benjamin Zorn
Microsoft Research
Redmond, WA

November 19, 2008

Microsoft Research Technical Report MSR-TR-2008-176

Abstract

Heap spraying is a new security attack that significantly increases the exploitability of existing memory corruption errors in type-unsafe applications. With heap spraying, attackers leverage their ability to allocate arbitrary objects in the heap of a type-safe language, such as JavaScript, literally filling the heap with objects that contain dangerous exploit code. In recent years, spraying has been used in many real security exploits, especially in web browsers. In this paper, we describe Nozzle, a runtime monitoring infrastructure that detects attempts by attackers to spray the heap. Nozzle uses lightweight emulation techniques to detect the presence of objects that contain executable code. To reduce false positives, we developed a notion of global "heap health".

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 a more general class of attacks in which an attacker attempts to fill the address space with dangerous code objects.

1 Introduction

In recent years, security improvements in systems, including more secure programming practices, stack protection [9], improved heap allocation layouts [6, 19], address space layout randomization [7, 31], and data execution prevention [20], have made it increasingly difficult for attackers to compromise systems. New attacks, primarily focused on exploiting memory corruptions in the heap, are now popular [25].

Heap spraying, originally proposed by SkyLined [33], is a security attack using a strategy of allocating many objects containing the attacker's exploit code in an application's heap. 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.

Heap spraying is an unusual security exploit in that the actions taken by the attacker in the spraying part of the attack are legal and type safe. Thus code executing in a type-safe language such as JavaScript, Java, or C# can be used to perform the spray. Since its introduction in 2004, heap spraying has been used widely to simplify exploits of security vulnerabilities in web browsers. Recently, variants of spraying attacks have been proposed, where the attack is set up so that data such as compiled bytecode, ANI cursors [21], and thread stacks are interpreted as code [35].

In this paper, we describe NOZZLE, a runtime infrastructure that detects heap spraying attacks by exploiting the fact that spraying places many copies of objects with specific characteristics into the heap. NOZZLE uses a combination of methods including statistics, object examination, and lightweight emulation to estimate whether a given object is part of a spraying attack. Because heap spraying involves large-scale changes to the heap contents, we exploit this characteristic to reduce our false positive and false negative detection rates. We develop a general notion of global "heap health" based on the measured attack surface of the heap contents.

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 very high successful detection rates and very low false positive rates.

1.1 Contributions

This paper makes the following contributions:

- We propose the first effective technique for detecting heap-spraying attacks through lightweight runtime interpretation and introduce the concept of attack surface area.
- We show that existing published NOP sled detection techniques have high false positive rates when applied to heap objects and we describe effective techniques that dramatically lower the false positive rate in this context.
- Measuring 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 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.

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. Section 6 considers broader implications of memory spraying attacks. Section 7 describes related work and Section 8 concludes. Appendix 8 provides some additional results.

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

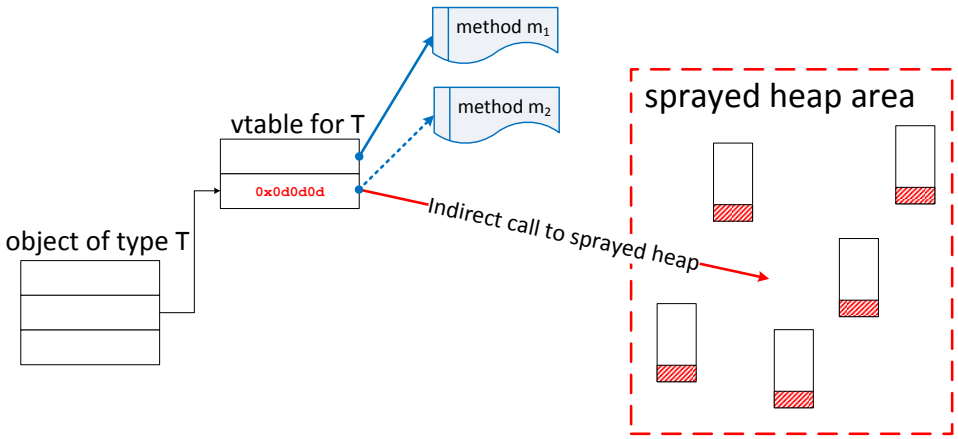

Figure 1: Schematic of a Heap Spraying Attack.

layout randomization [7, 31] further reduce the predictability of objects on the heap. Attackers have adopted several strategies for overcoming this uncertaintly [34], 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 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

```
 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.

directly. Browsers are particularly vulnerable to heap spraying because JavaScript embedded in a web page authored by the attacker greatly simplifies such attacks.

Example 1. The example shown in Figure 2 is modeled after a previously published heap-spraying exploit [37]. 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 13 in Section 5 provides more information on previously published exploits.

3 Overview

Despite the popularity of type-safe languages such as Java, C#, and JavaScript, the recent upsurge in heap-spraying attacks demonstrates that lan-

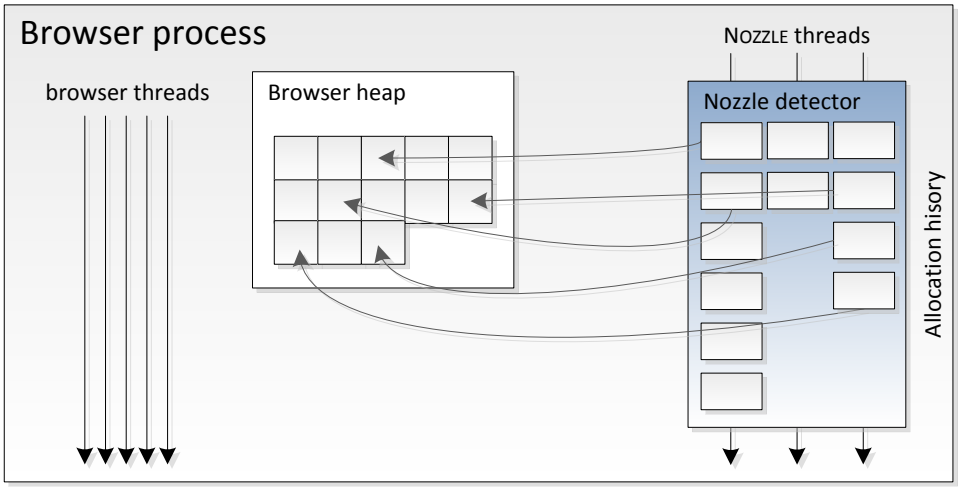

Figure 3: NOZZLE system architecture.

guage type safety is not a panacea. 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 [18, 29]. 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 architecures supporting a "no-execute bit", which prevents a process from executing code on specific pages in its address space [20]. We discuss why such support is not always effective in Section 7. In this paper, we consider systems that use the x86 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 [30]. 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, 15, 27, 36]. 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 the fact 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 also may have access to the browser code and possess detailed knowledge of system-specific memory layout properties such as object alignment. We discuss these implications in detail in Section 6.1.

4 Nozzle 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 process address space.*

Previous work on NOP sled detection focuses on examining possible attacks for properties like valid instruction sequences [4, 36]. 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 valid instruction sequence (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.

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.).

Our algorithm constructs a control flow graph (CFG) by interpreting the data in an object at offset Δ as an instruction stream. For the rest of this paper, 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 to terminate a basic block after an invalid instruction is encountered. For every basic block within the CFG we compute the *surface area*, a proxy for the likelihood of control flow ending at 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, $\bar{W}$, and a basic block validity bitvector $\bar{V}$, with its i^{th} component representing whether B_i is a valid instruction sequence.*

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

Solution: We define a parametrized dataflow problem using the terminology in Aho et. al. [2], as shown in Figure 4. This produces $out(B_i)$ for every basic block $B_i \in C$.

Next, the surface area of basic block B_i, $SA(B_i)$, is computed as follows:

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

where $out(B_i)$ is represented by a bitvector whose values are computed using the iterative dataflow algorithm above and $\bar{V}$ and $\bar{W}$ are algorithm inputs. $\bar{V}$ is determined using the validity criteria mentioned above, while $\bar{W}$ is the size of each basic block in bytes. The intuition is that we discard basic blocks that are not valid instruction sequences by logically bitwise ANDing $out(B_i)$ and $\bar{V}$. Finally, we use vector multiplication to account for the weight each basic block contributes—or does not—to the surface area of B_i.

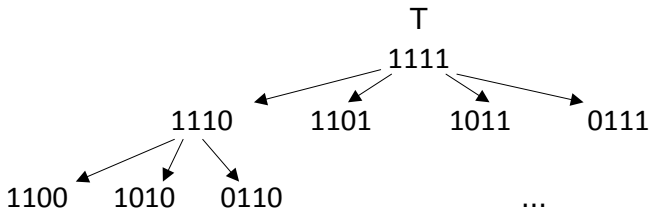

Figure 5: Semi-lattice used in Example 1..

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 semi-lattice for this CFG of size 4 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. In this example, we illustrate that we have broken block 1 at the invalid `in` instruction, with the remainder of the block contributing to the control flow graph. 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.

For simplicity of exposition, we include the weight of block B_i itself in the computation presented here. In practice, 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.

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

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

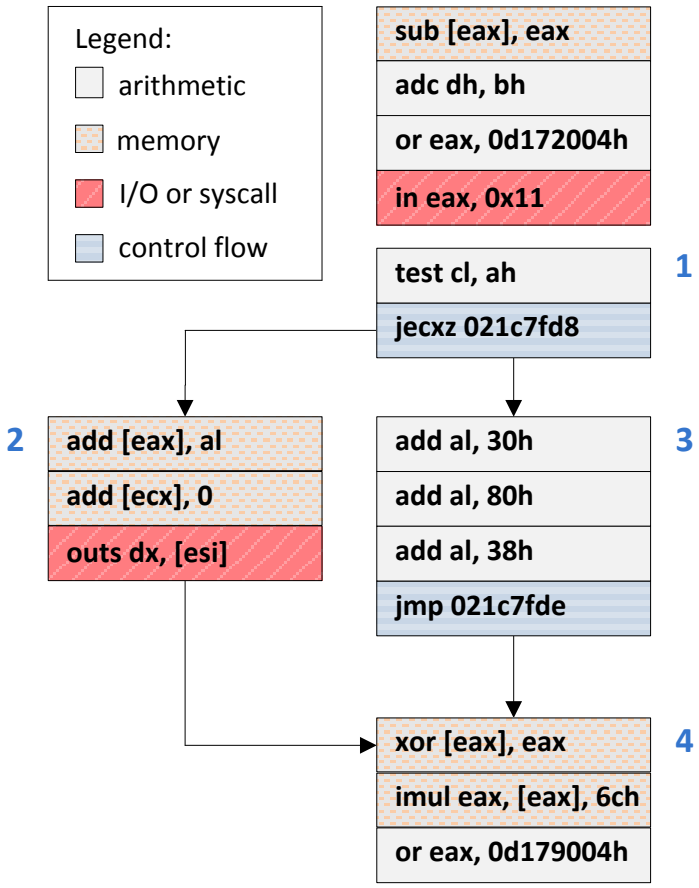

Figure 6: The control flow graph for Example 1..

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:*

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

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"

B_i	$TF(B_i)$	$\bar{V}_i$	$\bar{W}_i$	$out(B_i)$	$out(B_i) \wedge \bar{V}$	$SA(B_i)$
1	1000	1	2	1000	1000	2
2	0100	0	3	1100	1000	2
3	0010	1	4	1010	1010	6
4	0001	1	3	1111	1011	9

Figure 7: Values for Example 1..

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 TOCTOU (time of check to time of use) issue discussed in Section 6.1.

4.2.1 Detouring Memory Management Routines

We use a binary rewriting infrastructure called Detours [12] 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. 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

$$(a>>p) \texttt{ \% } N$$

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 proactively rescan objects without impacting browser performance and reducing TOCTOU vulnerabilities (see Section 6.1).

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 of the issue. Either of these approaches is superior to silently killing the process, the way DEP/NX protection in Windows responds to memory execution protection violations.

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, observing that the size of the Firefox heap when opening to a blank page is approximately six megabytes. We consider the economic basis for a viable attack in Section 6.2.

5 Evaluation

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 filter of our attack surface metric. The success of NOZZLE depends on its ability to distinguish between these two kinds of behavior. After seeing Figure 8, one might be inclined to think that we can easily 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.

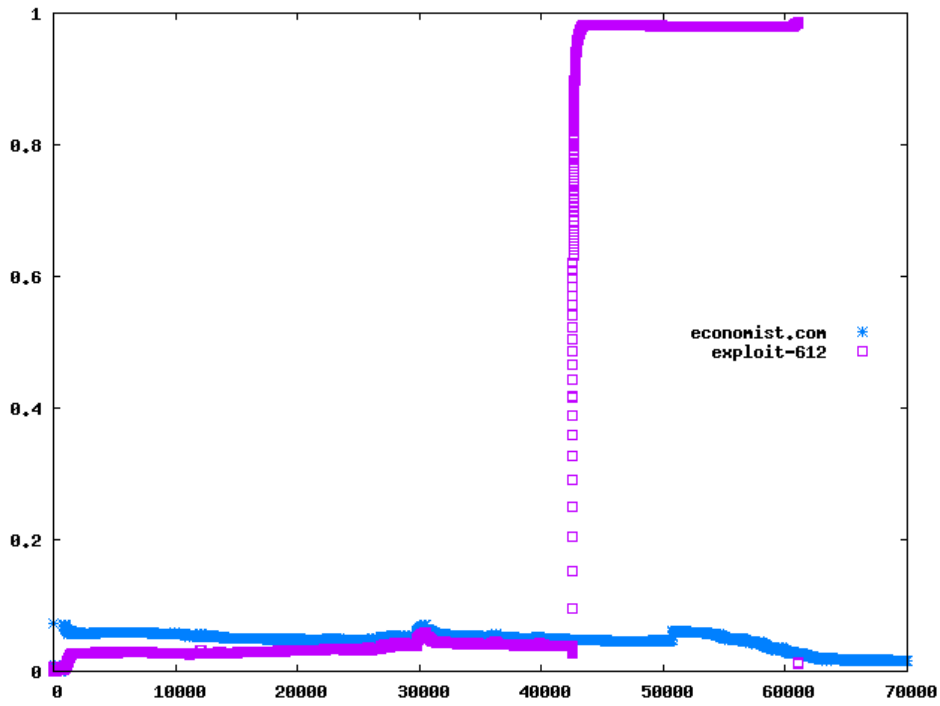

Figure 8: Global normalized attack surface for `economist.com` versus a published exploit (612). The Y-axis indicates the computed normalized attack surface. The X-axis indicates logical time as measured in object allocations.

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.

5.1 False Positives

Because web sites are so diverse, a heap-spray detector detection technique for the broswer must have a very low false positive rate. To measure the false positive rate of NOZZLE, 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 web sites, chosen from the most visited sites as ranked by Alexa [5]. For these sites, we simply loaded the first page of the site and measured the heap activity caused by that page alone.

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.

5.1.1 Global False Positive Rate

Figure 10 shows the maximum normalized attack surface measured by NOZZLE for our 10 benchmark sites (top) as well as the top 150 sites reported

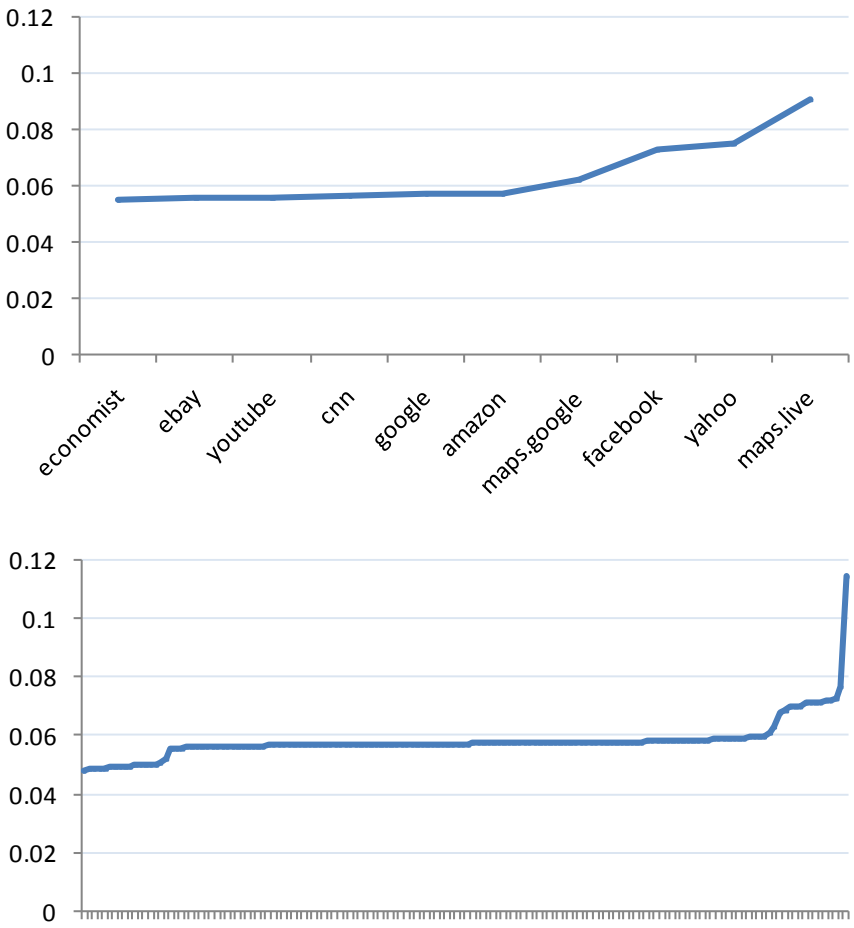

Figure 10: Global normalized attack surface for 10 benign benchmark web sites and 150 additional top Alexa sites, sorted by increasing surface. The Y-axis indicates the maximum normalized attack surface measured during the visit to the site, where each element of the X-axis represents a different web site.

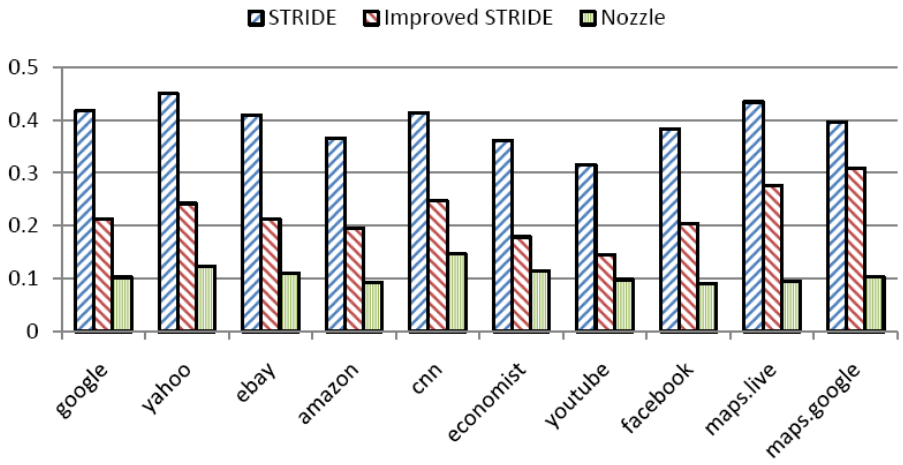

Figure 11: Local false positive rate for 10 benchmark web sites using Nozzle and STRIDE. Improved STRIDE is a version of STRIDE that uses additional instruction-level filters, also used in Nozzle, to reduce the false positive rate.

by Alexa (bottom). From the figure, we see that the maximum normalized attack surface remains around 5% for most of the sites, with a single outlier from the 150 sites at around 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 20% 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 and STRIDE, a recently published NOP sled detection algorithm, to see how effective they are classifying benign heap objects.

Figure 11 indicates the false positive rate of two variants of STRIDE and a simplified variant of Nozzle, which does not include any surface area computation. The figure shows that, unlike previously reported work where the false positive rates for URIs was extremely low, the false positive rate for heap objects is quite high, sometimes about 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 from this 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 malicious code 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.

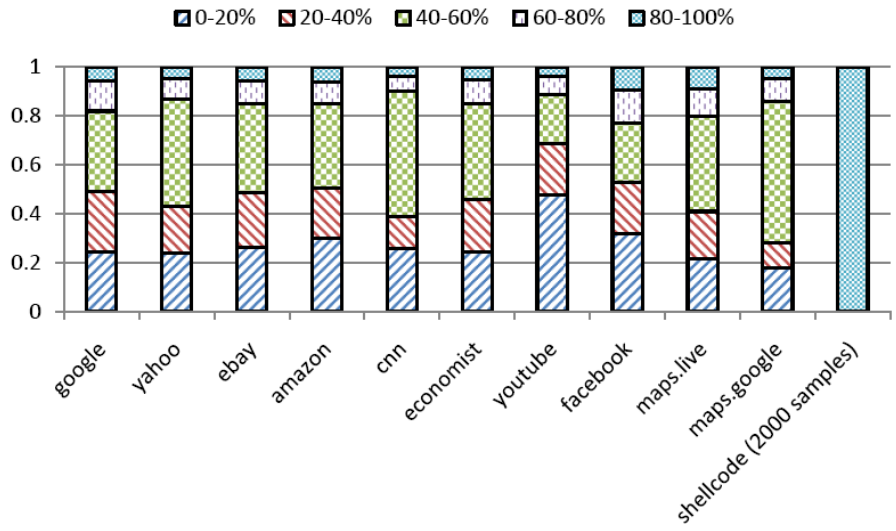

Figure 12: Distribution of filtered object surface area for each of 10 benchmark web sites (benign) plus 2,000 synthetic exploits (see Section 5.2). Objects measured are only those that were considered valid instruction sequences by Nozzle.

To increase the precision of a local detector based on Nozzle, we incorporate the surface area calculation described in Section 4. Figure 12 indicates the distribution of measured surface areas for the roughly 10% of objects in Figure 11 that our simplified version of Nozzle was not able to filter. We see from the figure that many of those objects have a relatively

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

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

To evaluate the false negative rate of Nozzle, we gathered 12 published heap-spraying exploits, summarized in Figure 13. We also created 2,000 synthetically generated exploits using the Metasploit framework [10]. 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.

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

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.

objects in the heap.

Figure 14 evaluates how effective NOZZLE is at avoiding 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 global attack surface of 98%. For our synthetic use of spraying, which was more conservative, we still measured a mean global attack surface of 76%. 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 Overhead

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. In these measurements, because we had a dual core machine, we configured Nozzle to use one additional thread for scanning. 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.

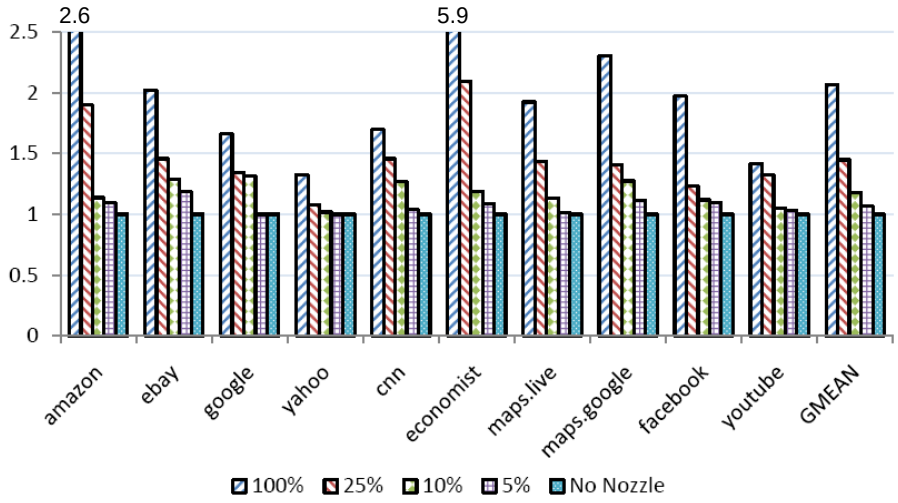

Figure 16: Relative execution overhead of using Nozzle in rendering a typical page of 10 benchmark web sites as a function of sampling frequency.

Figure 16 shows the performance overhead of Nozzle, both with and without sampling. 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 two times slowdown. To reduce this overhead, we consider the impact of sampling on the overhead. 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. Reducing the sampling rate to 25%, the mean overhead drops to 45%, while with a

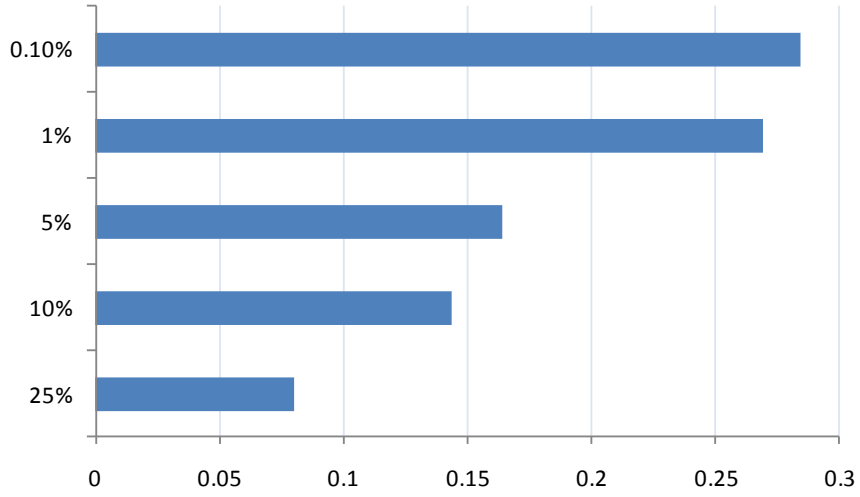

Figure 17: Average error rate due to sampling of the computed average surface area for 10 benign benchmark web sites.

sampling rate of 5%, the performance overhead is only 6.4%.

5.4 Impact of Sampling on Detection

Previously, we showed that sampling signifantly improves the CPU performance of using Nozzle. Here, 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 17 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%. 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.

In Figure 18, 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 Rate				
	100%	25%	10%	5%	1%
12 Published	0	0	0	0	50%
2,000 Metasploit	0	0	0	0	0

Figure 18: False negative rate for 12 real and 2,000 Metasploit attacks given different object sampling rates in Nozzle.

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. As mentioned, sampling based on bytes allocated instead of objects allocated would reduce this false negative rate to zero.

6 Discussion

In this section, we consider additional implications of using our approach.

6.1 Assumptions and Limitations

This section lists assumptions that Nozzle makes and discusses their implications.

TOCTOU issues. Because Nozzle examines object contents only at specific times, this leads to a potential time-of-check-to-time-of-use (TOCTOU) vulnerability. An attacker aware that Nozzle was being used could be 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 string is an immutable type, 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 TOCTOU 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* may be used as a trigger for NOZZLE to rescan objects.

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 some number of junk bytes at the start of objects. In order for NOZZLE to be comprehensive when inspecting an object, it must probe into multiple offsets other than zero. We consider resisting this attack here.

Each initial instruction offset where a malicious attack might start defines an equivalence class of instruction layouts. For example, the induced layout starting at offset one might be different than the induced layout starting at offset zero, except in the case where the first instruction at offset zero is a single-byte instruction, in which case the two induced layouts would be equivalent. Note that a layout starting at offset zero is likely to have many other layouts in its equivalence class (specifically, any layout that has an initial instruction starting at an instruction boundary present in the offset zero layout).

Site URL	**Probe frac. (%)**
`economist.com`	17.9
`cnn.com`	8.4
`yahoo.com`	8.2
`google.com`	10.8
`amazon.com`	32.4
`ebay.com`	11.1
`facebook.com`	10.4
`youtube.com`	12.1
`maps.google.com`	24.5
`maps.live.com`	8.7
Metasploit sleds	28.3

Figure 19: Average fraction of an object that needs to be probed for complete coverage in 10 benchmark web sites and 2,000 NOP sleds.

To understand how many such equivalence classes exist, we studied the 10 benchmark web sites and 2,000 NOP sleds generated from Metasploit to see how many offsets need to be probed to cover all the equivalence classes of instruction layouts for a given object (see Figure 19). We see a range of values from the minimum of 8% (yahoo) to the maximum of 32% (amazon). These results suggest that exhaustively checking every offset equivalence class would be prohibitive for a browser heap (typically tens of megabytes in size). Therefore, we believe a practical solution to address this problem would be to do sampling within an object by randomly probing offsets at the start of different equivalent classes. We plan to incorporate multiple offset probes into future implementations

of NOZZLE.

6.2 Threshold Setting

The success of heap spraying is directly proportional to the density of the dangerous objects in the program heap, as measured by the normalized attack surface employed by Nozzle. The more sprayed objects there are, the more likely an attack is to succeed in a probabilistic sense. At the same time, a densely-sprayed heap will have a high normalized surface area and will be immediately detected by Nozzle. If the attacker tries to disguise her actions by having a sparsely populated heap so as to fall under the Nozzle radar, she is not going to take over many of machines. As a result, the attacker is between a rock and a hard place; to make matters worse for the attacker, failing attacks most often result in program crashes. In the browser context, these are recorded on the user's machine and sent over to browser vendors using a crash agent such as Mozilla Crash reporting [22] 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 [23]. 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 [13]. 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 [17, 14]. 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, we believe that this many reports from a single web site should 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.

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 [28], 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 [24], 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 [36], and STRIDE, by Akritidis et al. [4, 26], 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..." ([26], 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 [8], and program shepherding [16] 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) [20], 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.

Even with DEP, however, there are reasons to use NOZZLE. First, attacks that first turn off DEP have been published, thereby circumventing its protection [32]. Second, because DEP is an all-or-nothing solution, 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 [11]. Third, a run-time system that performs just-in-time (JIT) compilation such as the Sun JVM allocates objects with read-write-execute permission, therefore, making DEP irrelevant. Thus, it can be used as a vehicle to perform heap spraying in a system with DEP. For example, instead of using JavaScript to do spraying, we may elect to use Java applets to do this, instead. Finally, code injection spraying attacks have recently been reported in areas other than the heap where DEP cannot be used. Sotirov describes spraying attacks that introduce exploit code into a process address space via embedded .NET User Controls [35]. 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.

While we have focused our experimental evaluation on heap-spraying attacks exclusively, the techniques this paper proposes are considerably more general. In particular, we can detect a variety of exploits that use code masquarading as data, such as images, compiled bytecode, etc. [35].

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 Martin Burtscher, Silviu Calinoiu, Trishul Chilimbi, and Ted Hart for their valuable feedback during the development of this work.

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, *International Conference on Information Security (SEC 2005)*, pages 375–392. Springer, 2005.

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

[6] 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.

[7] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the USENIX Security Symposium*, pages 8–8, 2003.

[8] 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.

[9] 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.

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

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

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

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

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

[15] 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, *APNOMS*, volume 4773 of *Lecture Notes in Computer Science*, pages 354–363. Springer, 2007.

[16] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, 2002.

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

[18] B. Livshits and W. Cui. Spectator: Detection and containment of JavaScript worms. In *Proceedings of the Usenix Annual Technical Conference*, July 2008.

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

[20] Microsoft Corporation. Data execution prevention. `technet.microsoft.com/en-us/library/`

cc738483.aspx, 2003.

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

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

[23] Mozilla Security Group. Mozilla security bug bounty program. www.mozilla.org/security/bug-bounty.html, 2004.

[24] 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.

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

[26] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In C. Krgel, R. Lippmann, and A. Clark, editors, *RAID*, pages 87–106, 2007.

[27] 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.

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

[29] Samy. The Samy worm. namb.la/popular/, Oct. 2005.

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

[31] 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.

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

[33] SkyLined. Internet explorer iframe src&name parameter BoF remote compromise. skypher.com/wiki/index.php?title =Www.edup.tudelft.nl/~bjwever/advisory_iframe.html.php, 2004.

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

[35] A. Sotirov and M. Dowd. Bypassing browser memory protections. In *Proceedings of BlackHat*, 2008.

[36] T. Toth and C. Krügel. Accurate buffer overflow detection via abstract payload execution. In *RAID*, pages 274–291, 2002.

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

Appendix

A: Alexa Sites

Figure 20 shows the list of 150 sites we used for our false positive evaluation. Note that we had to filter our some sites from the original Alexa.com site listing to avoid "business-inappropriate" sites.

B: Normalized Attack Surface

Figures 21–23 show the normalized attack surface metric over time (measured in the number of allocations) for a variety of commonly used sites.

C: How STRIDE Detection Generates False Alarms

Consider the following byte sequence (in hex) of size 32 taken from a heap object on visiting `google.com`:

```
00251998 dc e3 12 60 00 00 00 00
002519a0 1c 00 c0 40 10 00 00 00
002519a8 00 00 00 00 00 00 00 00
002519b0 00 00 00 00 e0 e0 e0 e0
```

STRIDE inspects a suspicious object starting at offset 0, 1, 2, and 3. At each offset, it decodes sequences of bytes by probing into different sub-offsets in the increment of four until the last sequence at that offset has size less than or equal to four. If all sequences are valid, STRIDE flags the object as malicious. A valid sequence cannot contain privileged instructions, and is decodable from start to finish or contains a branch along the way.

Given the above object, STRIDE effectively generates four different equivalence classes of instruction layout starting at offset 0, 1, 4, and 11. All the four layouts induced are as follows:

starting offset = 0

```
00251998 dce3            fsubr   st(3),st
0025199a 126000          adc     ah,byte ptr [eax]
0025199d 0000            add     byte ptr [eax],al
0025199f 001c00          add     byte ptr [eax+eax],bl
002519a2 c0401000        rol     byte ptr [eax+10h],0
002519a6 0000            add     byte ptr [eax],al
002519a8 0000            add     byte ptr [eax],al
002519aa 0000            add     byte ptr [eax],al
```

1 http://www.yahoo.com
2 http://www.google.com
3 http://www.youtube.com
4 http://www.live.com
5 http://www.facebook.com
6 http://www.msn.com
7 http://www.myspace.com
8 http://www.wikipedia.org
9 http://www.blogger.com
10 http://www.yahoo.co.jp
11 http://www.baidu.com
12 http://www.rapidshare.com
13 http://www.microsoft.com
14 http://www.google.co.in
15 http://www.google.de
16 http://www.hi5.com
17 http://www.qq.com
18 http://www.ebay.com
19 http://www.google.fr
20 http://www.sina.com.cn
21 http://www.google.co.uk
22 http://www.mail.ru
23 http://www.fc2.com
24 http://www.aol.com
25 http://www.vkontakte.ru
26 http://www.google.com.br
27 http://www.wordpress.com
28 http://www.orkut.com.br
29 http://www.google.it
30 http://www.flickr.com
31 http://www.yandex.ru
32 http://www.google.cn
33 http://www.photobucket.com
34 http://www.google.es
35 http://www.google.co.jp
36 http://www.amazon.com
37 http://www.naver.com
38 http://www.go.com
39 http://www.craigslist.org
40 http://www.friendster.com
41 http://www.odnoklassniki.ru
42 http://www.google.com.mx
43 http://www.taobao.com
44 http://www.imdb.com
45 http://www.skyrock.com
46 http://www.cnn.com
47 http://www.bbc.co.uk
48 http://www.orkut.co.in
49 http://www.googlesyndication.com
50 http://www.163.com
51 http://www.youku.com
52 http://www.ask.com
53 http://www.imageshack.us
54 http://www.adobe.com
55 http://www.google.ca
56 http://www.uol.com.br
57 http://www.rakuten.co.jp
58 http://www.espn.go.com
59 http://www.sohu.com
60 http://www.ebay.de
61 http://www.dailymotion.com
62 http://www.netlog.com
63 http://www.mixi.jp
64 http://www.metroflog.com
65 http://www.daum.net
66 http://www.rambler.ru
67 http://www.vmn.net
68 http://www.apple.com
69 http://www.yahoo.com.cn
70 http://www.rediff.com
71 http://www.livedoor.com
72 http://www.orkut.com
73 http://www.google.com.tr
74 http://www.megavideo.com
75 http://www.fastclick.com
76 http://www.fotolog.net
77 http://www.livejournal.com
78 http://www.about.com
79 http://www.globo.com
80 http://www.soso.com
81 http://www.mininova.org
82 http://www.nytimes.com
83 http://www.nicovideo.jp
84 http://www.wretch.cc
85 http://www.ameblo.jp
86 http://www.google.com.au
87 http://www.nasza-klasa.pl
88 http://www.bebo.com
89 http://www.goo.ne.jp
90 http://www.google.pl
91 http://www.google.co.id
92 http://www.google.com.sa
93 http://www.yourfilehost.com
94 http://www.mediafire.com
95 http://www.imagevenue.com
96 http://www.comcast.net
97 http://www.ku6.com
98 http://www.google.ru
99 http://www.ebay.co.uk
100 http://www.free.fr
101 http://www.veoh.com
102 http://www.4shared.com
103 http://www.xunlei.com
104 http://www.clicksor.com
105 http://www.terra.com.br
106 http://www.megaupload.com
107 http://www.google.nl
108 http://www.perfspot.com
109 http://www.google.com.ar
110 http://www.google.co.th
111 http://www.doubleclick.com
112 http://www.deviantart.com
113 http://www.metacafe.com
114 http://www.sogou.com
115 http://www.thepiratebay.org
116 http://www.mop.com
117 http://www.zshare.net
118 http://www.geocities.com
119 http://www.amazon.co.jp
120 http://www.download.com
121 http://www.orange.fr
122 http://www.2ch.net
123 http://www.tagged.com
124 http://www.tudou.com
125 http://www.tribalfusion.com
126 http://www.gmx.net
127 http://www.pconline.com.cn
128 http://www.clicksor.net
129 http://www.homeway.com.cn
130 http://www.amazon.de
131 http://www.weather.com
132 http://www.biglobe.ne.jp
133 http://www.conduit.com
134 http://www.cyworld.com
135 http://www.google.co.za
136 http://www.geocities.jp
137 http://www.aim.com
138 http://www.studiverzeichnis.com
139 http://www.maktoob.com
140 http://www.infoseek.co.jp
141 http://www.kaixin001.com
142 http://www.alibaba.com
143 http://www.sourceforge.net
144 http://www.dell.com
145 http://www.google.com.eg
146 http://www.onet.pl
147 http://www.tinypic.com
148 http://www.gamespot.com
149 http://www.ig.com.br
150 http://www.zol.com.cn

Figure 20: List of the top 150 websites from Alexa used in the NOZZLE false positive evaluation.

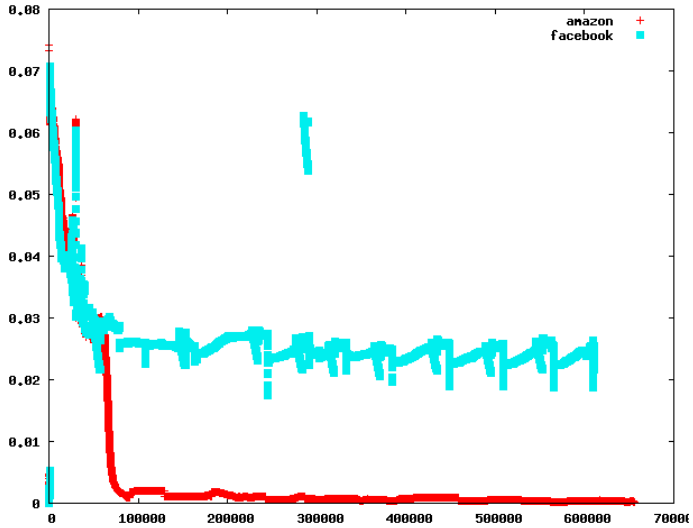

Figure 21: Normalized surface of attack over time: Amazon, Facebook, CNN, Economist.

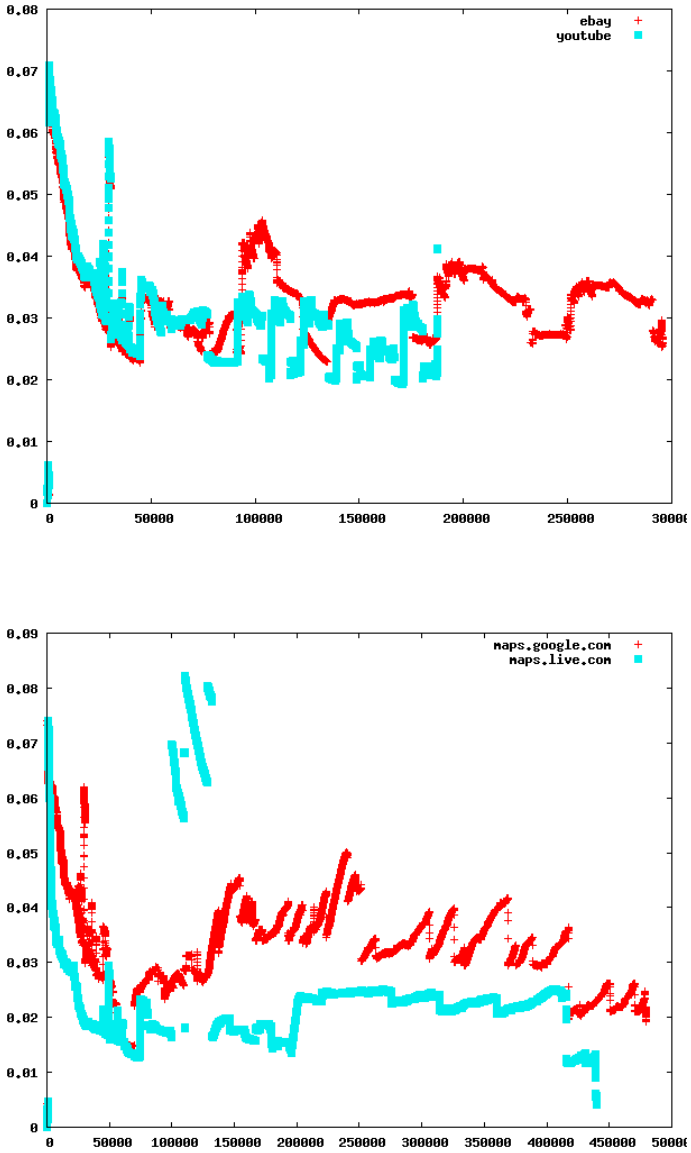

Figure 22: Normalized surface of attack over time: ebay.com, youtube.com, Google maps, and Live Maps.

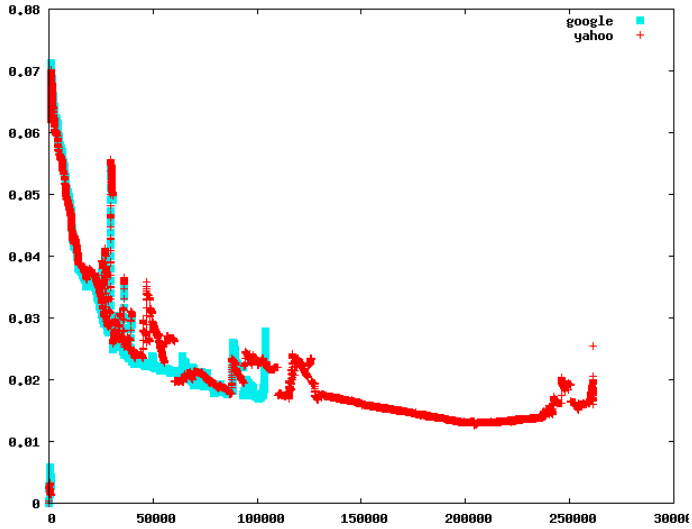

Figure 23: Normalized surface of attack over time: Google.com and Yahoo.com.

```
002519ac 0000            add     byte ptr [eax],al
002519ae 0000            add     byte ptr [eax],al
002519b0 0000            add     byte ptr [eax],al
002519b2 0000            add     byte ptr [eax],al
002519b4 e0e0            loopne  00251996
002519b6 e0e0            loopne  00251998
```

starting offset = 1

```
00251999 e312            jecxz   002519ad
0025199b 60              pushad
0025199c 0000            add     byte ptr [eax],al
0025199e 0000            add     byte ptr [eax],al
002519a0 1c00            sbb     al,0
002519a2 c0401000        rol     byte ptr [eax+10h],0
002519a6 0000            add     byte ptr [eax],al
002519a8 0000            add     byte ptr [eax],al
002519aa 0000            add     byte ptr [eax],al
002519ac 0000            add     byte ptr [eax],al
002519ae 0000            add     byte ptr [eax],al
```

```
002519b0 0000            add     byte ptr [eax],al
002519b2 0000            add     byte ptr [eax],al
002519b4 e0e0            loopne  00251996
002519b6 e0e0            loopne  00251998
```

starting offset = 4

```
0025199c 0000            add     byte ptr [eax],al
0025199e 0000            add     byte ptr [eax],al
002519a0 1c00            sbb     al,0
002519a2 c0401000        rol     byte ptr [eax+10h],0
002519a6 0000            add     byte ptr [eax],al
002519a8 0000            add     byte ptr [eax],al
002519aa 0000            add     byte ptr [eax],al
002519ac 0000            add     byte ptr [eax],al
002519ae 0000            add     byte ptr [eax],al
002519b0 0000            add     byte ptr [eax],al
002519b2 0000            add     byte ptr [eax],al
002519b4 e0e0            loopne  00251996
002519b6 e0e0            loopne  00251998
```

starting offset = 11

```
002519a3 40              inc     eax
002519a4 1000            adc     byte ptr [eax],al
002519a6 0000            add     byte ptr [eax],al
002519a8 0000            add     byte ptr [eax],al
002519aa 0000            add     byte ptr [eax],al
002519ac 0000            add     byte ptr [eax],al
002519ae 0000            add     byte ptr [eax],al
002519b0 0000            add     byte ptr [eax],al
002519b2 0000            add     byte ptr [eax],al
002519b4 e0e0            loopne  00251996
002519b6 e0e0            loopne  00251998
```

All of these instruction layouts are considered valid for STRIDE, and, hence, it will flag this object as malicious. NOZZLE, on the other hand, realizes that this cannot be so because the two conditional branches `loopne` have targets beyond the range of this object.