The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

DICE: A NONDETERMINISTIC MEMORY ALIGNMENT DEFENSE AGAINST HEAP TAICHI

A Thesis in

Computer Science and Engineering

by

Wei Zhong

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2011

The thesis of Wei Zhong was reviewed and approved* by the following:

Sencun Zhu
Associate Professor of Computer Science and Engineering
Thesis Advisor

Daniel Kifer
Assistant Professor of Computer Science and Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

ABSTRACT

Heap spraying is a security attack that mostly accounts for the popularity of exploits targeting web browsers and Adobe family products over the last few years. Such an attack, when combined with memory corruption techniques, can lead to arbitrary code execution on victim's host. A typical heap spraying attack populates application's heap by allocating large number of heap objects, each composed of NOP sled and malicious code, and then relies on a corrupted pointer to transfer control flow to a location within NOP sled, eventually causing the malicious code to be executed. A recent enhanced heap spraying attack that exploits the allocation granularity of system has a more precise control of the location of malicious code, thus effectively removes the requirement of using NOP sled and renders existing defenses powerless.

In this paper, we present Dice, a runtime guard that prevents enhanced heap spraying attack and detects any attempt that targets bypassing our countermeasure. Dice works by randomizing the location of heap object to be allocated and examines the content of heap object for a particular long NOP sled. Our prototype is implemented in SpiderMonkey and integrated into Firefox. We measure the effectiveness of Dice by performing a wide variety of benchmarks and compare the results with genuine Firefox. With sampling employed, the performance slowdown is less than 5% on average, and the memory overhead in the worst case is roughly 2%.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to take this opportunity to thank all the people who made this thesis possible and who helped me survive graduate life during past years.

First of all, I would like to express my deepest appreciation to my parents, Ruimin Zhong and Lingjun Qi. Although we were physically apart, they have been there all the time providing unconditional support and encouragement, and helping me go through the most stressful moments.

I would also like to acknowledge my thesis advisor, Dr. Sencun Zhu, for his courtesy in pointing out valuable papers for me to read and for his great patience in guiding me to conduct research.

I am also thankful to my academic advisor, Dr. Daniel Kifer, who has played an important role in supervising my academic progress and providing countless suggestions on job hunting, writing research articles, and so on.

Next, I would like to thank all the members I worked with in Network Operations unit of Graduate School during the past two years. Particularly I thank Chris Brown, Dmitry Podkuiko, Judy Evock, Ryan Y. Coleman and Heather Duty for being extremely enthusiastic in assisting me working on campus-wide web development projects.

Last but not least, I want to express my gratitude to all of my friends who helped me settle down when I first came here and carried me through the hardest time during past years.

Chapter 1

Introduction

1.1 Introduction

Nowadays heap spraying attacks have increasingly gained its popularity among attackers and helped attackers to execute arbitrary code on victim's host. It contributed tremendously to the popularity of exploits targeting web browsers and Adobe family products over the last few years. Heap spraying was first introduced in 2004 by Skylined [1], and it was written to exploit the vulnerability in Microsoft Internet Explorer *MS04-040* [2] and *MS05-020* [3]. Later in 2007, Sotirov [4] took Internet Explorer heap internals into consideration and made remarkable improvements on heap spraying attacks with respect to the reliability of exploits. In contrast to traditional code-injection buffer overflow attacks, heap spraying attacks are more generic and easier to manipulate for three reasons: (1) A number of countermeasures against buffer overflow attacks have been proposed and partially adopted by Operating System as default. Protection techniques falling under this category include address space layout randomization (ASLR) [9], Data Execution Prevention (DEP) [10], compiler and linker level protections such as StackShield [11], StackGuard [12], Stack Smashing Protector [13] and Libsafe [14]. (2) Unlike buffer overflow attacks targeting routinely-maintained servers, heap spraying attacks targeting normal users that are less sophisticated and whose hosts are less secure are much more likely to succeed. (3) Heap spraying attacks normally require victims to open a file (Webpage, office document, pdf, etc.) where malicious code is embedded. The

language supported for writing malicious code is capable of identifying the version and brand of products (e.g., Javascript can tell whether IE or Firefox is being used), thus making it easier for exploiting corresponding vulnerability across a variety of products.

The threat landscape of heap spraying attacks is heterogeneous in that any language that supports dynamic allocation (including but not limited to Javascript, VBscript and ActionScript) can be potentially used as a vehicle for the purpose of spraying heap. The success of many notable attacks being seen in the wild recently such as *Operation Aurora* [5], where 34 organizations were targeted and a zero-day IE vulnerability [6] was exploited in early 2010, are attributed to this technique. Other famous products supporting aforementioned languages such as Adobe Acrobat/Reader and Adobe Flash Player are suffering from heap spraying attacks in recent years [7, 8]. Additionally, "with ActionScript in flash, attackers have all the tools in their hands to develop workable exploits for Microsoft Office vulnerabilities." [15]

Heap spraying is a memory manipulation technique. For instance, through precisely manipulating a scripting language supported by products such as web browser, attackers could allocate tens or hundreds of megabytes of a consecutive sequence of memory blocks in victim's process heap. The contents of those memory blocks are under full control of attackers and each block is in the form of "NOP sled + shellcode". Allocation of objects on process heap is of legal use and could not be modeled as malicious behavior. Such technique also has the property of allocating memory blocks in the roughly same location for each instance of heap spraying, which directly makes this type of exploitation more reliable. Meanwhile, each heap spraying attack has to be coupled with another memory corruption technique (Stack Overflow, Format String Overflow, Integer Overflow, etc.) with the assistance of which control flow is likely to be transferred to anywhere within NOP sled, which eventually leads to the execution of

shellcode. The act of spraying heap with large amount of NOP sleds greatly increases the likelihood of landing within NOP sled for the corrupted instruction pointer.

We break down existing heap spraying defenses into two classes: (1) shellcode-based detection and (2) NOP sled-based detection. Shellcode-based detection detects heap spraying attacks by identifying the existence of shellcode within allocated memory blocks. One example of this is Drive-by [16]. Egele et al. utilized a ready-made x86 instruction emulation library to statically disassemble and identify JavaScript string buffers that contain shellcode. However, such detection approach is not resilient to self-modifying or polymorphic shellcode [17], and one of its major assumptions does not hold at present. NOP sled-based detection relies on the assumption that attackers have no knowledge of the location of crafted shellcode even though they are capable enough to redirect instruction pointer to a desired location. Accordingly, attackers have to populate heap with large amount of NOP sleds appended with shellcode in order to increase the likelihood of landing within NOP sled. Nozzle [18] is a run-time detector belonging to this category. To be specific, Nozzle used lightweight runtime interpretation to decode byte stream from start for each heap object, it then constructed a control flow graph (CFG) using the decoded instruction sequence and made use of an existing dataflow analysis algorithm to derive the size of landing pads (NOP sleds) for each heap object. Nozzle will report an attack if the accumulative size of landing pads reaches a threshold.

However, the assumption that locations of allocated memory blocks are undeterminable does not hold any more. This insight was first offered by Sotirov et al. [19] and later elaborated in Heap Taichi [20]. In a nutshell, modern operating systems like Windows align “each region of reserved process address space to begin on an integral boundary defined by the value of the system allocation granularity” [21]. Our experiment further ascertains the fact that in Windows, any allocation of memory block larger than

512 KB will result in an alignment to 64K-byte boundary. Consequently, the locations of memory blocks to be allocated are more predicable than expected. Heap Taichi took this observation one step further and proposed a new type of heap spraying attack that, without losing the reliability of exploitation, requires few NOP sleds which is far below the detection threshold set forth in Nozzle.

Our focus/design goal in this paper is to present an approach that prevents the heap spraying attack described in Heap Taichi and detects any attempt that tries to bypass prevention-based solution. By introducing non-determinism into allocation of memory blocks, the locations of allocated memory blocks are no longer predictable for attackers. We also assume attackers are aware of our prevention-based countermeasure. We provide an analytical study of potential actions attackers would possibly take to bypass our prevention-based countermeasure and demonstrate such actions will nevertheless be detected by our detector due to the use of a mandatory length of NOP sled of our discretion. We implement Dice in Mozilla Firefox browser and adopt sampling to reduce performance overhead.

1.2 Contribution

This paper makes the following contributions:

- We present a prevention-based defense that introduces non-determinism into heap allocation, which effectively thwarts enhanced heap spraying attack that exploits allocation granularity.
- We also present an anomaly-based detection defense which would detect a hypothetical attack that seeks to bypass prevention-based defense. In principle, for a successful attack, the attacker has to use a NOP sled, the length of which is at our disposal ahead of time.

- Our NOP sled-based detection also makes it resilient to enhanced spraying attack that makes use of polymorphic shellcode, a scenario where shellcode-based detection is usually rendered useless.
- We measure the instrumented Firefox interacting with heavily-used web sites and show that the detection threshold used in existing shellcode-based detection literature [16] is too low to have negligible false positive rate.
- We evaluate the performance overhead of our prototype by performing macrobenchmarks as well as microbenchmarks. With sampling enabled, the average performance slowdown is less than 5%. An additional analytic study of memory overhead shows the extra memory consumption in the worst case is 2%.

1.3 Paper Organization

The rest of thesis is organized as follow. Chapter 2 provides background about related basics and an enhanced heap spraying attack. Chapter 3 describes an overview of Dice and its implementation. Chapter 4 evaluates our implementation in terms of memory usage and performance overhead. Chapter 5 shortly discusses the contributions and limitations in our work. Chapter 6 elaborates related work. We conclude our thesis in Chapter 7.

Chapter 2

Background

In section 2.1, we first review memory basics and the notion of allocation granularity. We then recapitulate a traditional heap spraying attack and formalize the problem of enhanced heap spraying attack in section 2.2. Finally, we shortly explain why existing defense like ASLR does not help much in section 2.3.

2.1 Memory Basics and Allocation Granularity

All addresses in a process virtual address space must stay under one of three statuses during process lifetime: free, reserved and committed [21, 36]. Reserving addresses, as its name suggests, reserves a range of addresses for future use and protects them from being allocated by other memory requests. Committing addresses commits physical memory to reserved addresses to satisfy memory allocation requests. On windows platforms, applications can reserve and commit addresses through a family of VirtualAlloc APIs (e.g., VirtualAlloc, VirtualAllocEx, VirtualAllocExNuma, etc.). VirtualAlloc APIs are page granularity functions used for allocating large memory blocks. A heap manager is provided in Windows to address the need of managing allocations inside large memory blocks reserved using page granularity functions. HeapAlloc APIs are the most frequently used functions in this area. Take web browser as an example,

heap objects such as string[1] are allocated implicitly through HeapAlloc and VirtualAlloc will be automatically called if only reserved addresses run out.

On all Windows platforms up to Windows 7, when a region of virtual address space is reserved (commonly through VirtualAlloc), the reserved memory block must be aligned to an integral boundary defined by the value of the system *allocation granularity* [21]. This value on Windows platforms is typically 64[2] KB, and it is chosen for providing better performance and support for future processors with larger page size. Such memory alignment behavior has serious implication for memory layout management: large memory blocks allocated through VirtualAlloc are guaranteed to be aligned to 64 KB boundaries.

Here we borrow the notions of *heap block* and *heap object* defined in Heap Taichi [20]. *Heap block* refers to the memory block allocated through VirtualAlloc family APIs. *Heap object* refers to object allocated by HeapAlloc family APIs. They observe that a request of heap object larger than 512 KB will be serviced by allocating a separate heap block for this object. Namely, a heap object larger than 512 KB will be aligned to 64 KB boundaries. We define the term *atomic attack size* as the minimum size of heap object that results in being allocated on a separate heap block. For the determination of atomic attack size, we leverage HeapLib [4], a JavaScript library that provides alloc and free functions which map directly to calls to the system allocator, and attach WinDbg [35] to IE to keep track of the starting address of newly allocated heap objects. As illustrated in Figure 1, heap objects at the size of 512 KB are aligned to 64 KB boundaries[3] while heap

[1] Only if string is instantiated as a result of string manipulation operations such as concatenation will it be allocated on heap.

[2] Windows kernel-mode code is not subject to this restriction and can reserve memory at the granularity of a single page.

[3] The first 0x24 bytes are reserved for allocation header and metadata.

objects at the size of 500 KB do not. Our experimental results show atomic attack size is in a range between 500 KB and 512 KB, so for simplicity, we consider 500 KB as the value of atomic attack size.

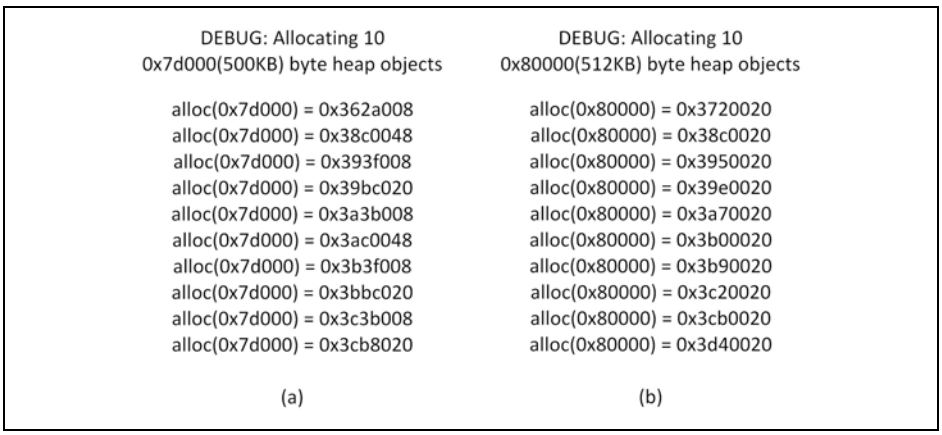


Figure 1. Determiniation of atomic attack size.

2.2 Traditional and Enhanced Heap Spraying Attack

As introduced above, heap spraying attacks are commonly found in the context of Web browsers, Adobe family products and so on. A successful heap spraying attack typically grants attacker the privilege of executing arbitrary code on victim's host. A traditional heap spraying exploit must incorporate two steps. The first step involves a memory manipulation technique that sprays the victim's process heap with large number of "NOP sled + shellcode" blocks. This is usually done through scripting language supported in products. As shown on the right-hand side of Figure 2, the region of white area represents NOP sled while the region of gray area stands for shellcode. The second step refers to a memory corruption technique that exploits a type of vulnerability in victim's process, forcing the control flow to transfer to somewhere within NOP sled. Prevalent vulnerabilities being seen in the wild include but not limited to buffer overflow

vulnerability [37, 38], use-after-free vulnerability [38, 39], and format string overflow vulnerability [8]. As shown on the left-hand side of Figure 2, buffer overflow vulnerability is exploited and the return address on the stack is corrupted, which eventually makes control flow transferred to a position (0x0c0c0c0c) within sprayed heap region. From attacker's standpoint, not knowing the specific locations of shellcode necessitates use of long NOP sled since landing in the middle of shellcode or landing outside sprayed heap region will incur program crashing.

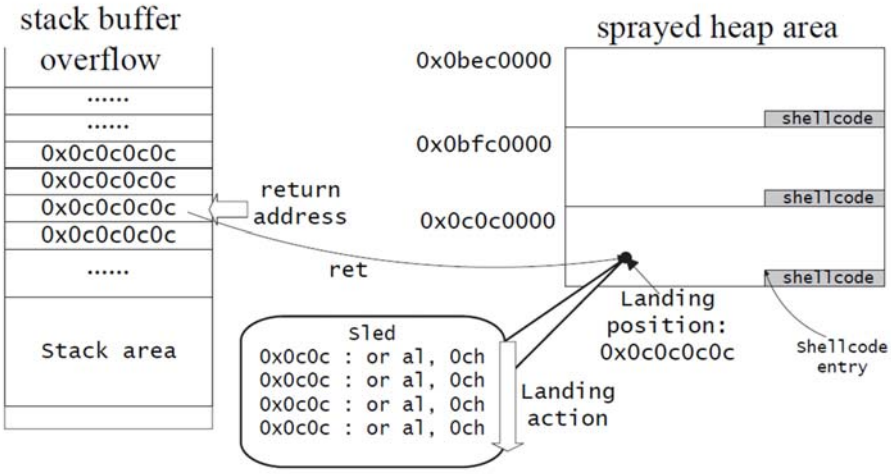


Figure 2. Traditional heap spraying exploit. Source: Heap Taichi.

Nevertheless, by leveraging allocation granularity, enhanced heap spraying attacks do not require use of long NOP sled at all. Suppose that a 512 KB long heap object is constructed as shown in Figure 3 and process heap is populated with large number of such objects. For each and every shellcode within such heap object, the value associated with it indicates the offset from the starting address of that heap object. Apparently, shellcode appears at a fixed relative position (0x24 byte) every 64 KB. Recall that the value of atomic attack size is 500 KB, this heap object will certainly be

aligned to a 64 KB boundary (0x00150000 in this case) since its size is above threshold. The precise control of shellcode at predicable locations drastically eliminates the need of using long NOP sleds. In our example, overwriting return address with 0xXXXX0024 is sufficient enough to ensure control flow will be transferred to shellcode, which causes the victim's host to be compromised.

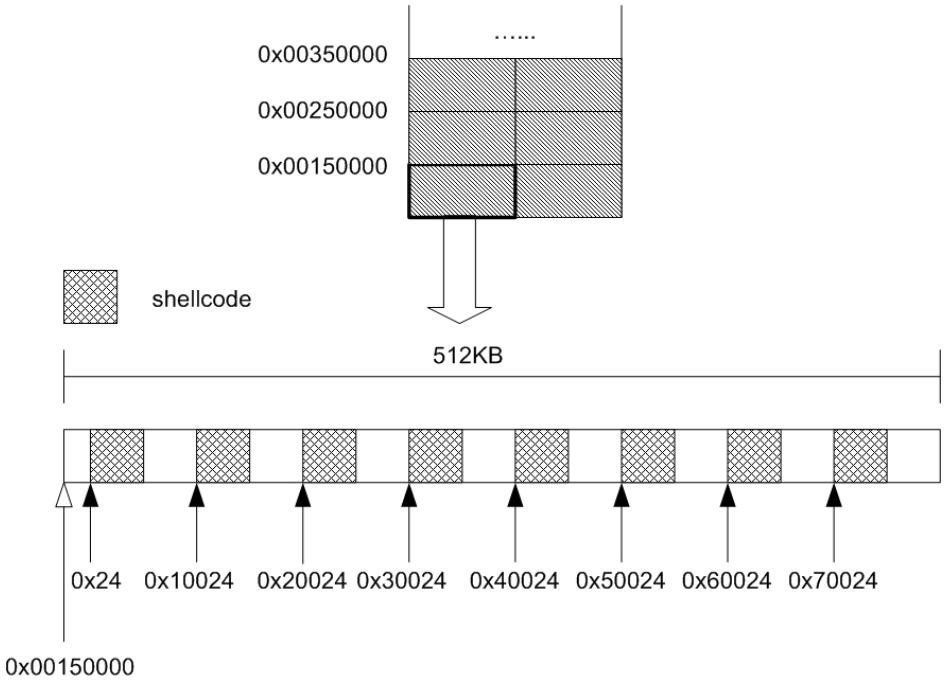


Figure 3. Enhanced heap spraying exploit.

Figure 4 shows a JavaScript code snippet that sprays heap objects as described above. Line 2 embeds the shellcode into a string object. Line 3-5 calculates the size of interval between two shellcodes and stuffs the interval with byte sequence which can not be interpreted as NOP sled. Line 6-8 constructs a 512 KB heap object composed of 8 chunks, each is made up of a piece of shellcode followed by a non-interpretable byte sequence. Line 9-12 repeatedly allocates crafted heap object until 128 MB heap memory has been taken up. The JavaScript's special representation of string in Unicode requires

discreet handle of size-related computation. In line 4, the length of shellcode is evaluated in Unicode (2 bytes) and, in order to capture the real memory consumption in byte, it has to be multiplied by 2. Conversely, in line 11, since the size of heap object to be constructed is represented in Unicode and hence, we need to divide the memory consumption of crafted *heap_chunk* by two after cutting the memory overhead (0x24) and Null terminator (0x2) from it.

```
1    function heapSpray() {
2        var shellcode = unescape("%u56e8…");
3        var alloc_granularity = 0x10000;
4        var interval = alloc_granularity - shellcode.length * 2;
5        var white_region = createBlockWithSize(interval);
6        var heap_chunk = shellcode + white_region;
7        for(i = 0; i < 3; i++)
8            heap_chunk += heap_chunk;
9        var memory = new Array();
10       for(j = 0; j < 0x100; j++)
11           memory[i] = heap_chunk.substring(0,
                              8 * alloc_granularity/2 - 0x24/2 - 0x2/2);
12   }
```

Figure 4. JavaScript code spraying heap with enhanced exploit.

2.3 ASLR Weakness

Previous works [19, 20] show ASLR does not affect enhanced heap spraying attacks. Essentially, ASLR introduces unpredictability by shifting the beginning of process heap by a parameter which is randomly chosen on each instance of a program. This parameter has to be in the range between 0 and 2MB and in the meantime, it must be a multiple of 64 KB. Since our heap object is already aligned to 64 KB boundary, the relative position of shellcode does not change at all. Spraying heap with abundant illustrated heap objects will largely decrease the effectiveness of ASLR-based defense.

Chapter 3

Design and Implementation

This Chapter briefly discusses the threat model of this thesis. A formalization of Dice including an analytical study of hypothetical attack is presented. We also describe the implementation of Dice at the end of this Chapter. The design goal of this chapter is to prevent enhanced heap spraying attack that exploits allocation granularity and detect attacks that intends to bypass Dice.

3.1 Threat Model

Simply speaking, we make three assumptions about our hypothetical attacker's abilities. Firstly, the attacker is fully acquainted with the latest workable memory corruption vulnerabilities, possibly even zero-day vulnerability, and is able to control the contents of exploits (e.g., JavaScript code snippet in web browser or ActionScript code snippet in Adobe Flash) that will be running on victim's host. Secondly, the attacker has no physical access to victim's host and delivers heap spraying attacks remotely. The channels used for delivering heap spraying exploits are diverse. For example, web exploit kit [41] is one of the most favorite tools the attack uses to lure normal users into visiting malicious web pages. Another detailed introduction of distribution channels for PDF malware can be found in [8]. Thirdly, the attacker is aware of our prevention approach and will manage to bypass it.

3.2 Dice

This section formally defines the terminology commonly used in this thesis and formalizes a two-layer defensive scheme for defending against enhanced heap spraying attack. The criteria of detecting and reporting an ongoing enhanced heap spraying attack is formulated in the end.

3.2.1 Terminology

First of all, we make an assumption that the attacker targets IA-32 [48] hosts due to its universal popularity in personal computers. We refer to the definition of *valid instruction sequence* in [18] and slightly adjust it to fit our purpose.

Definition 1. *A sequence of bytes is called NOP sled, if each and every byte position is the start of a valid x86 instruction. A valid x86 instruction must contain valid opcode and the correct number of arguments for that instruction. In addition, a valid x86 instruction does not include instructions in the following categories:*

- *I/O or system calls (*`in, insv, outsb`*, etc)*
- *interrupts (*`int, int3`*, etc)*
- *privileged instructions (*`lar, lsl, clts`*, etc)*

These instructions are excluded from valid x86 instructions for the reason that execution of such instructions would immediately incur diversion of control flow out of current heap object, thus directly preventing control flow from reaching shellcode. In reality, NOP sled is somehow misleading and it can be more versatile than a sequence of NOP (x90) bytes. A thorough study of constituents and polymorphic form of NOP sled is discussed in [42, 43].

3.2.2 Formalization of Defensive Scheme

As seen in section 2.2, the core issue of enhanced heap spraying attacks is that heap objects larger than atomic attack size will be aligned to 64 KB boundary, which effectively makes the locations of shellcode predicable. Therefore, the key idea underlying our prevention-based defense is to randomize the locations of shellcode to some extent, by prefixing a random sized block to suspicious heap objects. Figure 5 illustrates this idea. The length of random sized block is expressed as a parameter *r* which

```
r = rand() % RANDOM_OFFSET * 2
```

is determined by randomly selecting a value between 0 and RANDOM_OFFSET, and multiplies it with 2. The value of RANDOM_OFFSET is chosen to be 10K for the reason well documented in section 4.1. Under such circumstance, the attacker has one out of 10K chance of correctly guessing the offset position of shellcode, which thwarts enhanced heap spraying attack to a large degree. Dice is named after such a circumstance. To put it another way, the attacker has to guess the correct number before a 10K-sided die is thrown out. Correspondingly, the random sized block is referred to as *dice block*.

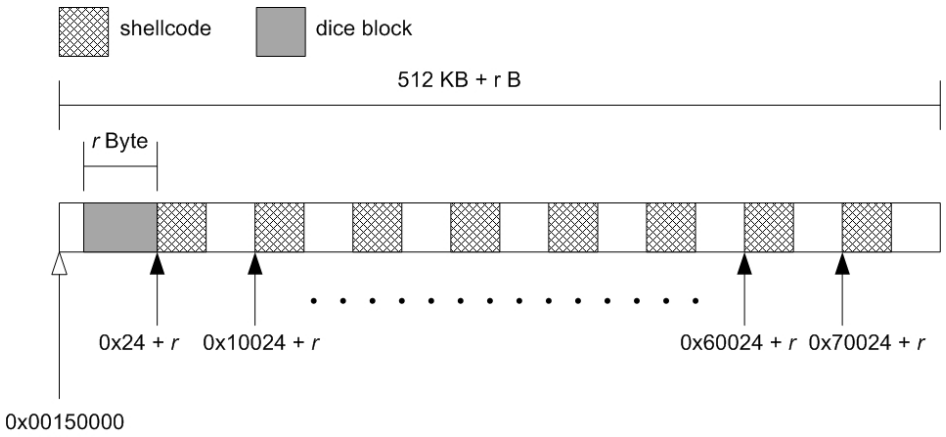


Figure 5. An example of prefixing a heap object with dice block.

As mentioned in section 3.1, we assume an informative attacker knows our prevention-based defense and would make an utmost effort to bypass it. Similarly as use of NOP sled in stack-based buffer overflow attacks [37, 38], an attacker may prefix a particularly long NOP sled to each and every shellcode in heap objects as an attempt to assisting a wrong guess to stealthily slide into shellcode, as shown in Figure 6. Such an attack employs limited number of NOP sleds and hence, makes it fall out of scope of traditional heap spraying attack. Fortunately, we find that, for a successful exploit leveraging NOP sled in this manner, the specific length of NOP sled to be inserted is dominated by parameter r_{max}. That is to say, the length of NOP sled must be as long as r_{max} bytes and the corrupted return address should be shifted right by r_{max} bytes. Figure 6 depicts two extreme cases where the control flow is transferred to the start and the end of NOP sled in the presence of dice block. Either case achieves the goal of having control flow reach the shellcode. For this kind of hypothetical attack, we incorporate an anomaly-based detection solution into our defense. The key thought is to measure the length of NOP sled for suspicious heap objects in a few benign web pages and then determine a threshold that normal length of NOP sled found in benign web pages usually does not reach. As discussed above, extending the maximum length of *dice block,* r_{max}, to the value of threshold obliges the attacker to insert corresponding length of NOP sled in front of each and every shellcode in order to make a successful exploit. Such an excessive length of NOP sled is abnormal and conspicuous that once found, our detector would fire an alarm. In other words, as long as maximum of r is made equal to threshold, our anomaly-based detection approach can be converted to identifying the use of NOP sled with length equal to/larger than r_{max} within suspicious heap objects.

On the whole, our two-layer defensive scheme works as follows: (1) In browser, when an allocation request for heap object comes in and the size of heap object, n, meets

the atomic attack size threshold, it will be serviced by allocating memory of size $n + r$. A random sized dice block of size r is prefixed to the real memory content of heap object. When read, heap object is accessed through a pointer and its value would not be affected by prefixed byte sequence. (2) Our anomaly-based detector then sequentially scans and decodes the byte sequence of heap object. An alarm would be fired only if a NOP sled of size r_{max} is found. Section 4.1 elaborates the process of tuning r_{max}.

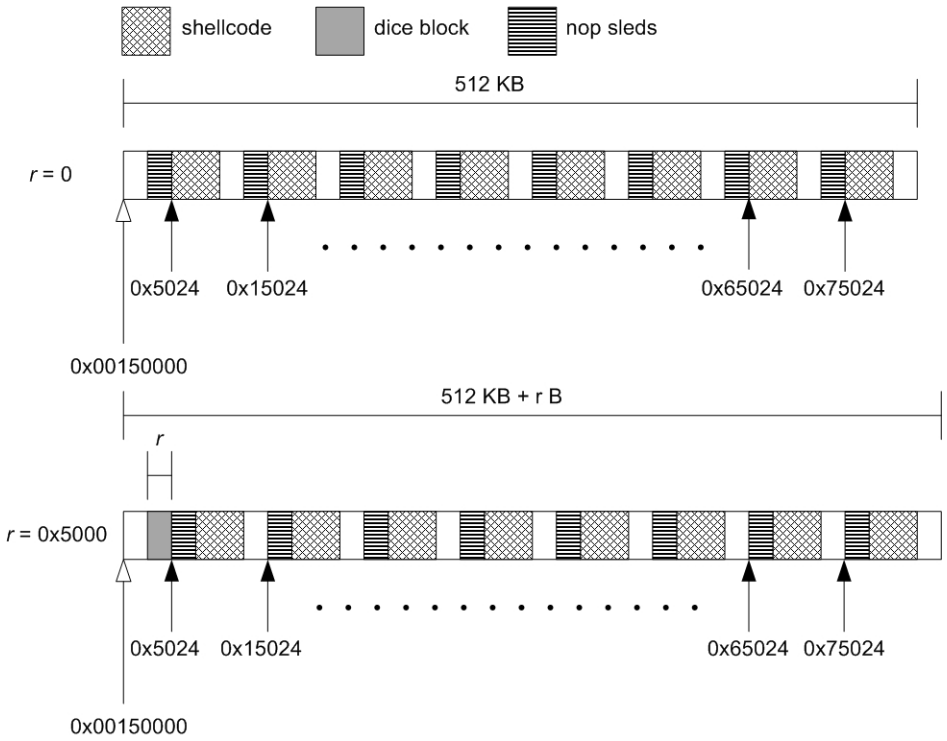


Figure 6. An example of bypassing aforementioned prevention approach.

3.2.3 Detection and Reporting

For currently allocated heap object H, Dice keeps track of the size of H and, if necessary, the length of the longest NOP sled within H. Two conditions have to be satisfied simultaneously for concluding that an enhanced heap spraying attack is underway:

$$(\text{size}(H) > \text{ATOMIC\_ATTACK\_SIZE} \wedge \text{nop\_sleds}(H) >= r_{max})$$

Calculating the length of the longest NOP sled within *H* involves a sequential scan. During scan, a disassemble function is invoked at each and every byte position in the attempt to translate byte sequence into interpreted x86 instruction. This procedure is CPU-intensive and should be called as few as possible to avoid incurring performance overhead. As shown above, this procedure is being called only when the size of *H* is above atomic attack size. Actually, among top 100 benign websites ranked by Alexa [44] we tested in section 4.1, none of them contain a string object with length above this threshold.

Atomic attack size is chosen to be 500 KB as explained in section 2.1. The threshold determination of r_{max} is based on the experimental results of visiting a few benign websites. Recall that attacker has to insert equivalently long NOP sled as r_{max} in front of each shellcode to bypass our prevention-based countermeasure. The guiding principle of determining r_{max} is that it should be made large enough to have negligible false positive ratio while maintaining low memory overhead.

3.3 Implementation

Previous section illustrates the exercise of JavaScript code to spray browser's heap using strings that are filled with x86 instructions. In this thesis, we merely consider strings as heap objects that attacker would make use of to spray heap for two reasons: (1) String is the most common object supported by a wide range of products including various browsers and PDF readers, and hence is the most frequently used heap object for spraying purpose in traditional heap spraying attacks. (2) String is selected for proving

the feasibility and effectiveness of our solution. The two-layer defensive scheme is conceptually generic and can be applied to other heap objects.

The prototype implementation of our defensive scheme has been implemented in JavaScript engine SpiderMonkey (Ver. 1.8.2) [46] and was integrated into Mozilla Firefox (Ver. 3.6.19pre) [45], a noted open source web browser. Our prototype implementation targets JavaScript String object as primary threat and tries to prevent and detect any attempt that performs heap spraying using this object.

3.3.1 JavaScript String Basics

```
Struct JSString{
     size_t length;
     jschar *char;
};
```

SpiderMonkey implements JavaScript String object using a structured type. This structure is solely made up of two members: the *length* variable indicates the length of string, and *char* variable is a special pointer that points to the beginning of the string. The representation of string in JavaScript engine differs from usual C-style string in two aspects. Firstly, the length of string is stored as an independent variable and is irrelevant to how string is represented in memory. Secondly, the string is composed of an array of Unicode characters rather than ASCII characters. When a string is created, SpiderMonkey will allocate sufficient memory space to accommodate the requested sequence of Unicode characters and then set the *char* pointer to the beginning of this sequence.

3.3.2 Prevention-based Defense

Such a special inner representation of string largely reduces the amount of work to be done. To deploy our prevention-based defense, a heap allocation request can be serviced by allocating much more memory than required, as illustrated in Figure 7. The size of extra leading space (*dice block*) is randomly determined at each run and is unpredictable ahead of time. Filling the memory right after *dice block* with original memory content and adjusting string pointer to an appropriate position makes such change entirely transparent to users.

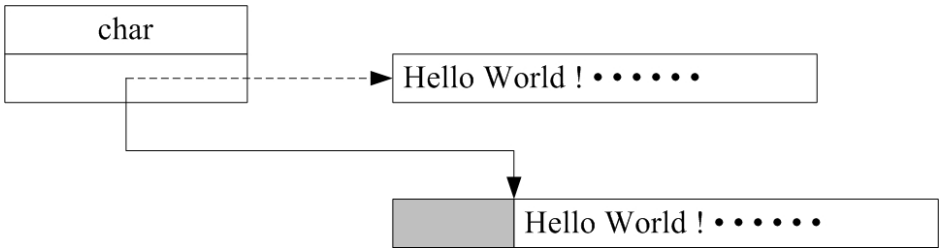


Figure 7. Deploy prevention-based defense in SpiderMonkey.

3.3.3 Anomaly-based Detection

Only when the size of heap allocation request exceeds atomic attack size is our detector triggered. The detector performs a serial scan on the real byte sequence of heap object and searches for an extraordinarily large NOP sled. libdasm [47] was leveraged to take charge of instruction decoding, which is crucial for the identification of valid instruction sequence. Such object scanning is performed only once immediately after the deployment of prevention-based defense. The fact being exploited is that string is implemented as immutable. Namely, under no circumstances should the memory content of initialized string be modified. Any change of an initialized string would result in an allocation of new string with modified value. As a result, our prototype monitors all

string manipulation functions in case a clear heap object generates a new one filled with NOP sleds. Table 1 lists all functions our detector attaches to in instrumented SpiderMonkey and their associated functionality.

Table 1. List of functions to be monitored in SpiderMonkey.

Function Name	**Functionality**
`js_ConcatStrings`	Concatenates two JS strings and returns the result
`js_UndependString`	Creates a new JS string for given dependent JS string
`js_str_escape`	Encodes a string and makes it portable
`str_unescape`	Decodes an encoded string
`str_toSource`	Retrieves the source code of an object
`js_toLowerCase`	Converts a JS string to lowercase letters
`js_toUpperCase`	Converts a JS string to uppercase letters
`str_fromCharCode`	Converts Unicode values to characters
`str_replace`	Replaces the matched substring with a new substring
`js_NewStringCopyN`	Copies n characters from a character array into new string
`js_NewStringCopyZ`	Copies a null-terminated character array into new string

The fundamental rule for ruling out other functions and choosing aforementioned functions as our subjects is that these are the only functions that involve memory management routines including malloc and realloc, which are the only ways to allocate string object on heap.

Algorithm 1 check whether the given string contains NOP sleds at the size of detection threshold r_{max}

```
Input: string to be tested, length of string n, detection threshold r_max
    offset ← 0; cul_length ← 0
    if string is empty
        return false
    while offset < n and cul_length < r_max do
        inst ← decode instruction at offset position of string
        if inst is not empty and
            inst is not a privileged instruction and
            inst is not a interrupt instruction and
            inst is not I/O or system call instruction then
            cul_length ← cul_length + 1
        else
            cul_lenth ← 0
        offset ← offset +1
    if cul_length < r_max then
        return false
    else
        return true
```

Algorithm 1 shows the algorithm for our detector to check if the length of NOP sled within a given string exceeds a threshold. This algorithm includes a search over string and would not be called off unless it reaches the end of string or the cumulative length of NOP sled reaches predefined threshold. During each run of while loop, the algorithm checks if a byte sequence at a specific position can be interpreted into a valid instruction. If not, the cumulative length of NOP sled is reset and the counting process starts over again. Otherwise, the instruction is further examined to ensure it does not fall under instruction groups defined in section 3.2.1. If all of these conditions are satisfied, the cumulative length of NOP sled is increased by 1.

3.3.4 Performance Optimization

As we mentioned before, decoding byte sequence at each and every byte position is a time-consuming operation. For applications such as web browser that requires quick

content rendering, response time is one of the most important indicator for judging the effectiveness of our prototype. To reduce the performance overhead, we adopt sampling technique in our prototype.

Sampling technique can be generally broken down into two classes: (1) Sampling by object; (2) Sampling by allocation granularity. The first class randomly selects heap object and checks the existence of long NOP sled. The downside of this approach is it may miss the real malicious heap object if the attacker populates the memory with a good mixture of numerous but small benign heap objects and few but extremely long malicious heap objects. Consequently, it is inevitable to check large heap object above certain threshold one by one and hand over the rest to sampling. The second class employs sampling at a finer granularity level. As shown in Figure 6, NOP sled has to be inserted every 64 KB to increase the possibility that an enhanced heap spraying attack exploiting allocation granularity will succeed. Therefore, an alternative is to randomly select 64 KB block within heap object and check the existence of long NOP sled. The resulting speedup is proportional to the ratio of heap object size to allocation granularity, which is expected to be at least 8X. Combining both techniques helps minimize the performance overhead to a great extent. Nonetheless, we simply apply the second type to our prototype.

Chapter 4

Experiments

This chapter is organized as follows: In section 4.1, top 100 sites ranked by Alexa are visited and further analyzed to determine the detection threshold of our detector. Section 4.2 evaluates the performance degradation incurred by Dice. Section 4.3 presents an analytical study of memory overhead of Dice.

4.1 Parameter Tuning

The detector in our prototype distinguishes benign heap objects from malicious ones based on the maximum length of NOP sled contained in the object. In past literatures, threshold tuning process involves testing the prototype implementation against benign subjects as well as malicious subjects. The threshold value should be set precisely and appropriately so as to minimize false positive rate and false negative rate. Our prototype is distinctive from the past in the sense that the length of NOP sled to be inserted for constructing malicious heap object is subject to our control. From an attacker's point of view, raising the maximum size of *dice block*, indicated by threshold value r_{max}, requires the insertion of longer NOP sled in front of each shellcode. The threshold value in our prototype is determined by measuring the maximum length of NOP sled of string objects in top 100 web sites reported by Alexa and setting it to a value beyond the limit of observed results. In our experiment, we simply visited the first page of each site and, for every string object larger than 1 KB, noted down the maximum

length of NOP sled concealed under that object. For each individual site, the maximum length of NOP sled is our only concern and is kept for analysis purpose. Figure 8 depicts the distribution of the maximum length of NOP sleds among top 100 Alexa sites. We see that about 90% web sites contains NOP sled no larger than 1 KB while 99% web sites contains NOP sled below 10 KB. Figure 9 shows the recorded maximum length of NOP sled for each individual site. The single outlier from this collection reaches 19 KB. Therefore, by setting the threshold value to 20 KB, our detector avoids flagging benign web site as malicious and is capable of detecting any attempt that targets bypassing Dice.

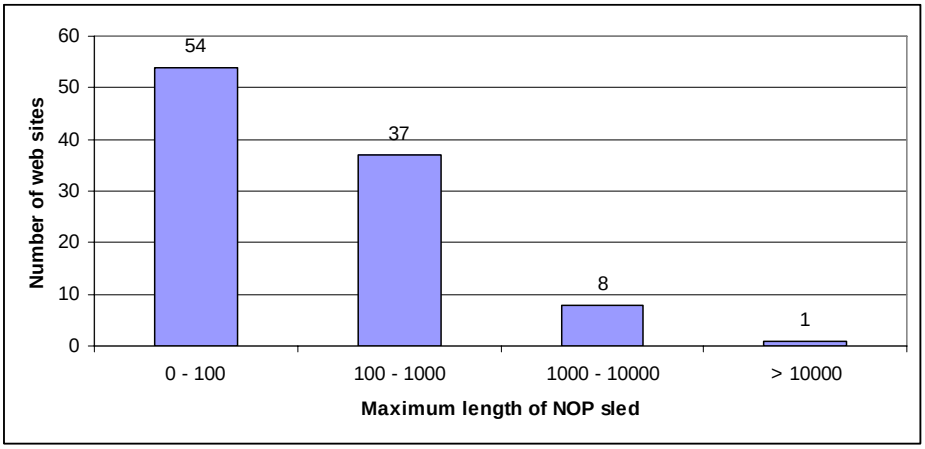


Figure 8. Distribution of the maximum length of NOP sled among top 100 Alexa sites. The X-axis classifies the maximum length of NOP sled into four categories.

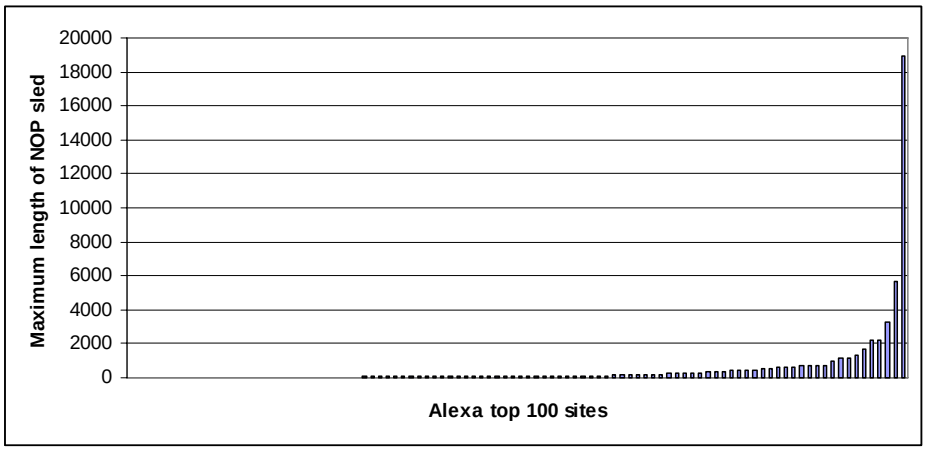


Figure 9. Maximum length of NOP sled for top 100 sites recorded by Alexa. The X-axis represents each individual website and Y-axis represents the length of NOP sled in byte.

4.2 Performance Benchmarks

In addition to the correctness of our prototype, another essential factor needs to be tested is the effectiveness of the instrumented browser. In this section, both macrobenchmarks and microbenchmarks are performed for evaluating the performance overhead of Dice. All benchmarks were performed in Ubuntu 10.04 equipped with Intel Pentium Dual 1.87Ghz CPU and 512MB RAM.

4.2.1 Macrobenchmarks

To measure the impact of Dice in the real world from browser performance point of view, we run *Namoroka*, project name for genuine Firefox 3.6, as well as Dice against 10 benchmark sites chosen from either Nozzle or Alexa site. The metric for performance measurement is the load time of the index page of each site. In our experiment, each index page is downloaded and instrumented by inserting JavaScript `new Date()` function at the beginning of HTML head section and the end of HTML body section.

Load time is evaluated as the difference of two `Date` values. For each individual page, our recorded load time is computed by visiting each page 50 times and taking the average of results. In view of reflecting the real load time of each page, we shift refresh the browser to avoid any influence by caching.

As shown in Table 2, the number added or subtracted from the average load time delimits the upper limit and lower limit of a 95% confidence interval for a given benchmark site. In other words, we are 95% confident that the average load time of a larger sample set (≥50) would lie within this range. Figure 10 illustrates the confidence interval of the load time of economist.com with 95% confidence level for both browsers.

The average performance slowdown of Dice is 0.2%. A closer study of 10 benchmark sites reveals that none of them contains string object at the size greater than atomic attack size. Thus, the extra workload of our instrumented browser only comes from size check of string objects. The occasional occurrence of lower performance overhead of Dice can be primarily ascribed to the bare use of string objects in that site, which effectively reduces the workload of our prototype.

Table 2. Performance overhead of Dice compared to Namoroka.

Benchmark Sites	Load Time (ms) in Namoroka	Load Time (ms) in Dice	Performance Slowdown
cnn.com	2767.26 ± 92.94	2668.64 ± 128.97	-3.56%
ebay.com	837.36 ± 61.23	817.22 ± 44.93	-2.41%
economist.com	5383.69 ± 147.66	5561.74 ± 187.43	+3.31%
facebook.com	289.22 ± 5.29	295.96 ± 5.68	+2.33%
google.com	369.96 ± 19.36	373.56 ± 11.39	+0.97%
maps.google.com	1165.10 ± 20.88	1162.12 ± 27.86	-0.26%
maps.live.com	1193.28 ± 18.85	1223.04 ± 23.17	+2.49%
microsoft.com	1064.12 ± 19.07	1075.20 ± 36.26	+1.04%
yahoo.com	1775.74 ± 38.24	1759.88 ± 37.57	-0.89%
youtube.com	1813.74 ± 31.17	1795.46 ± 29.76	-1.01%
Average			+0.20%

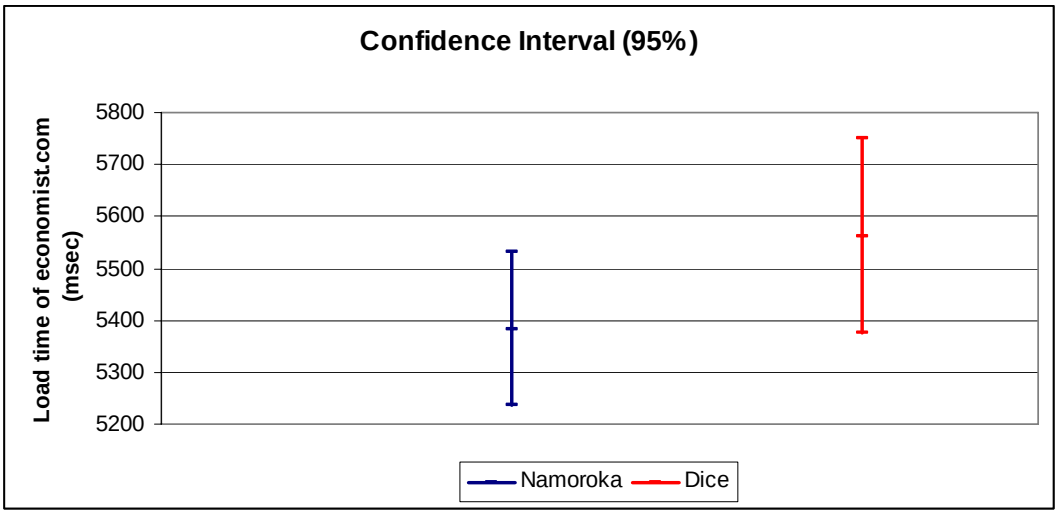


Figure 10. Confidence interval of the load time of economist.com with 95% confidence level for Namoroka and Dice.

4.2.2 Microbenchmarks

We resort to microbenchmarks to better understand the scope and degree of performance penalty brought about by Dice. Three JavaScript performance benchmarks released by different major browser vendors were performed: SunSpider [49] released by WebKit, V8 [50] released by Google and Dromaeo [51] released by Mozilla.

SunSpider is a JavaScript benchmark designed to compare different versions of the same browser and different browsers to each other. The tests are classified into a wide variety of domains (e.g., crypto, math, string, etc.) and each domain, which is composed of several subtests, is running 5 times each with the results measured in milliseconds. Similar to the benchmark performed in BuBBle, Table 3 shows the aggregate runtime overhead for domains with insignificant performance increase and detailed runtime overhead for subtests that have been heavily affected. The 10.7% slowdown of regexp domain is mainly caused by the additional check and scan implemented in string replacement function. The higher overhead of string subsets is foreseeable since the

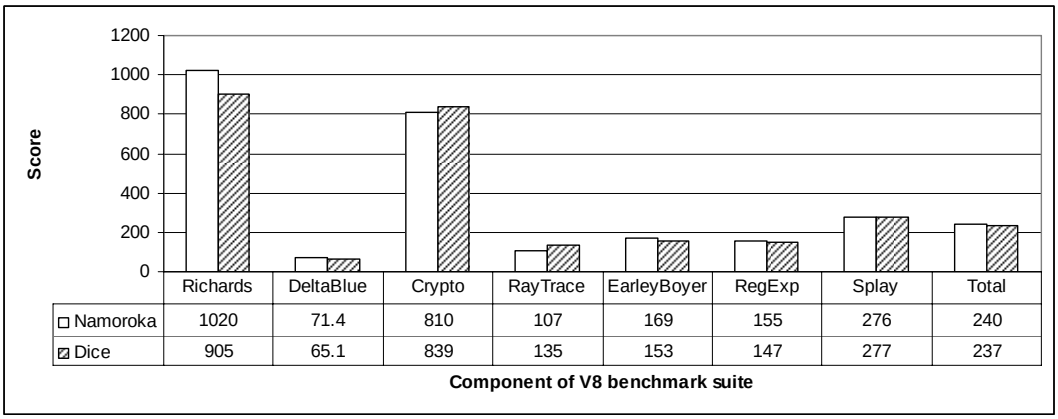
detector we attached to aforementioned string manipulation functions would perform extra instruction decoding as needed. However, Table 3 shows the overall slowdown of performance is 2.03% and hence, is ignorable.

Table 3. SunSpider 0.9.1 JavaScript benchmark results shown by means and 95% confidence interval.

Test	Runtime (ms) in Namoroka	Runtime (ms) in Dice	Performance Slowdown
3d	273.6 ± 2.5%	280.9 ± 11.9%	+2.67%
access	296.6 ± 12.0%	298.1 ± 11.0%	+0.51%
bitops	65.5 ± 5.5%	65.4 ± 5.1%	-0.15%
controlflow	59.6 ± 2.6%	59.5 ± 1.5%	-0.17%
crypto	100.5 ± 2.9%	100.9 ± 6.3%	+0.40%
date	301.5 ± 1.5%	308.8 ± 10.2%	+2.42%
math	79.4 ± 3.2%	78.2 ± 3.1%	-1.51%
regexp	99.9 ± 34.4%	110.6 ± 20.4%	+10.7%
string	634.8 ± 6.7%	656.4 ± 7.3%	+3.4%
base64	22.2 ± 2.5%	22.0 ± 8.0%	-0.90%
fasta	141.1 ± 7.1%	153.7 ± 16.8%	+8.93%
tagcloud	200.2 ± 9.1%	203.6 ± 3.1%	+1.70%
unpackcode	151.0 ± 1.5%	166.5 ± 19.9%	+10.26%
validate-input	120.3 ± 17.2%	110.6 ± 8.1%	-8.06%
Total	1911.4 ± 4.1	1967.9 ± 1.8%	+2.03%

The V8 Benchmark Suite is designed to tune the JavaScript engine of Google Chrome. The experimental result is a list of scores based on a reference system (score 100), showing the performance for each component of benchmark suite. Higher score indicates better performance. Rather than running for a predetermined number of iterations, each component is run continuously until a second of time has passed. As shown in Figure 10, for each individual component, no one was running noticeably faster than the other. This is because the fast-running tests chosen in V8 hardly contain string manipulation benchmark. Even through the crypto benchmark applies encryption and decryption over strings, the majority of them may not reach the size above threshold.

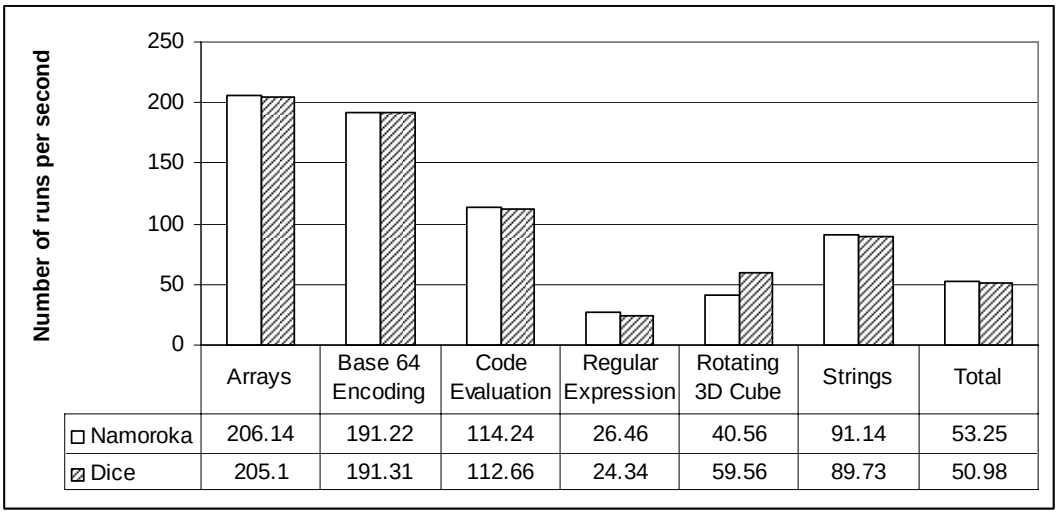
Accordingly, Dice imposes negligible performance penalty on JavaScript engine in most cases.



	Richards	DeltaBlue	Crypto	RayTrace	EarleyBoyer	RegExp	Splay	Total
Namoroka	1020	71.4	810	107	169	155	276	240
Dice	905	65.1	839	135	153	147	277	237

Figure 11. V8 Benchmark Suite test results.

Dromaeo runs a range of tests, each running at least 5 times. The results were measured in the number of runs per second. Similar to the V8 benchmark suite test results, Dice exhibited a comparable performance compared to the *Namoroka*. The only difference is that the involvement of string intensive operations including base64 encoding/decoding and routine string manipulation did not cause apparent performance degradation for Dice. This may be attributed to the limited length of strings involved, which is far below the threshold that triggers the detector.



	Arrays	Base 64 Encoding	Code Evaluation	Regular Expression	Rotating 3D Cube	Strings	Total
Namoroka	206.14	191.22	114.24	26.46	40.56	91.14	53.25
Dice	205.1	191.31	112.66	24.34	59.56	89.73	50.98

Figure 12. Dromaeo benchmark test results.

4.3 Memory Overhead

We culminate this chapter in an analytical study of the memory overhead of our prototype. As described in chapter 3, *dice block* is a random sized contiguous block of memory prefixed to the real content of heap object like string. This kind of block is wasted and would be used under no circumstance during the lifespan of web page. Since the size of *dice block* ranges from 0 to 20 KB, the average wasted space would be 10 KB. In the worst case that the size of requested heap object exactly matches atomic attack size, the ratio of wasted memory to practically utilized memory is 2% (10/500), a tolerable trade-off for raising security. Any use of heap objects above or below atomic attack size would cut down that ratio to a lower value.

Chapter 5

Discussion

This section highlights two contributions we made in this thesis and shortly discusses two limitations our prototype faces and what can be done in the future.

5.1 Discretionary Control of NOP Sled

Our experimental results conflicts with the findings described in Drive-by in terms of the maximum length of NOP sled contained in benign pages. Drive-by declares that, due to the use of 16-bit Unicode characters to store text in JavaScript, a Unicode character sequence representing the same text as usual ASCII character sequence is less likely to contain interpretable byte sequence (= NOP sled) at a size greater than 32 bytes and this is mainly because every other byte has to contain the value of 0x00. Thus, they monitored all string allocations in JavaScript engine and used 32 bytes as detection threshold for recognizing shellcode. However, this is not true as 0x00 can be interpreted as a valid opcode of ADD instruction. Meanwhile, we observed that 0x00 is still a valid operand for the family of jump instructions. The fact that the existence of 0x00 does not hamper instruction decoding at all is affirmed by our experimental result that nearly half of the benign sites we visited contained NOP sled, length of which is greater than 100 bytes.

One of the major contributions we made in this thesis is the discretional control of the length of NOP sled inserted by attacker, as explained in section 3.2.2. In short, the

maximum length of *dice block* directly forces the attacker to put equivalently long NOP sled in front of each shellcode. Unlike traditional heuristic-based study, the detection threshold is set at our choice and false positive rate is largely reduced.

5.2 Resilience to Self-modification

Self-modification is an obfuscation technique that prevents shellcode from being detected by static analysis. The shellcode is generally encrypted by a key and is decrypted and executed on the fly. Our prototype is resilient to self-modification since we flag an attack by identifying NOP sled instead of shellcode alone. More specifically, an attacker relies on landing within NOP sled to reach and execute shellcode. For this reason, the byte sequence at each and every position within NOP sled must be interpretable, and can not be obfuscated through self-modification.

5.3 Limitation

One of the limitations in our prototype is it fails to detect a special kind of heap spraying attack which populates heap with objects at a size below atomic attack size. However, such an attack does not exploit allocation granularity and, as claimed by our design goal of this thesis, is beyond our concern. Moreover, this kind of attack requires use of huge amount of NOP sleds and hence, would be caught by Nozzle.

The other limitation is a motivated attacker might try to interpolate invalid instruction at few selective positions of NOP sled and bridge the gap through jump instruction. Consequently, the length of interpretable byte sequence is reduced to a value below the detection threshold, which immediately evades being detected as malicious while still maintaining high NOP sled surface area. The fact that a successful attack

depends on the large surface area of NOP sled does not change. We believe by calculating the NOP sled surface area in the same way the attack surface area is computed in Nozzle, such an attack would be detected anyway.

Chapter 6

Related Work

This Chapter discusses generic memory protection mechanisms that may potentially be useful for defending against heap spraying attacks, existing detection and prevention approaches specifically designed against heap spraying attacks and other defenses against memory corruption techniques.

6.1 Generic Memory Protection

As discussed, a successful heap spraying attack entails the execution of shellcode from within a process heap. Existing memory protection mechanisms such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) implemented in modern operating systems serves as good candidates. DEP is a security feature included in modern operating systems with the intention of preventing an application or service from executing code from a non-executable memory region (e.g., thread stack, process heap, etc.). With DEP enabled, memory pages that are marked as non-executable could no longer be used to execute injected code.

Nevertheless, techniques that can bypass DEP are being seen in the wild: (1) Skape et al. [22] pointed out that, for the purpose of making existing third-party applications avoid running into compatibility issues when DEP is introduced, Microsoft provides a couple of windows APIs that can enable or disable DEP on a per-process basis. Whether or not DEP is enabled for a process is determined at execution time. By

launching a return-into-libc derived attack, attackers can call those APIs with proper arguments, eventually fulfilling the task of disabling NX support at runtime from within a process. (2) Sotirov et al. [19] introduced several exploits that can bypass DEP on Windows Vista. For example, shellcode can be put in the text section of a .NET binary and get loaded at an executable page in the browser process. (3) Buchanan et al. [23] described return-oriented programming (ROP), a generalization of return-into-libc that allows an attacker to execute code in the presence of DEP. This is mainly because code snippets to be executed are dispersed in executable memory pages such as code segment and are later chained together by attackers. A library that automates ROP exploits was developed by Sole [24]. (4) Moreover, Just-In-Time (JIT) compiler may write interpreted code in the heap, which requires those pages to be executable. One technique exploiting this fact was coined JIT spraying, a process of coercing the JIT engine to write many executable pages with embedded shellcode, by Blazakis [25]. (5) Recently, an exploitation technique that can bypass both ASLR and DEP in IE 8 on Windows 7 was proposed in PWN2OWN 2010 [40]. Its key idea is to use existing MS code to call VirtualProtect() and change memory protect settings for shellcode to attacker's advantage.

ASLR enhances security by moving executable images as well as DLLs into random locations when a system boots. By default, Windows Vista and later versions will randomize system DLLs and EXEs [26]. This technique makes attacks that rely on transferring control flow to a target address such as the entry of shared library in return-to-libc attack hard to execute because target address is less predictable. A wrong guess of target address may lead to program crashing, which significantly reduces the reliability of foregoing exploits. But nonetheless, works [19, 20, 27] have been showed that ASLR is not as robust as expected. Whitehouse [27] concluded that the frequency distribution of values for heap allocations and PEB addresses are not uniformly distributed. Other works

[19, 20] showed that attackers with full understanding of ASLR supported for heap allocation may still launch heap spraying attacks without being detected by Nozzle. Our focus of this paper is to present an approach that can catch these sophisticated attacks.

6.2 Heap Spraying Detection and Prevention

Existing countermeasures against heap spraying attacks generally fall into two groups: (1) NOP sled-based detection and (2) shellcode-based detection. Nozzle [18] is an example of first type. In particular, Nozzle is the first run-time system specifically designed to detect and prevent heap spraying attacks. The idea behind this system is, given a heap object, Nozzle uses lightweight runtime interpreter to decode byte stream of this object from start. The decoded instruction sequences are treated as program based on which a CFG is constructed. Afterwards, a dataflow analysis algorithm is further applied to derived CFG with the goal of calculating the size of landing pad for each basic block. The landing pad is used as a metric that indicates how likely a basic block will be reached given a random control flow landing on this object. Based on the assumption that the basic block which contains shellcode should have the largest landing pad, the maximum calculated size is considered as surface area of NOP sled in this object. From a global point of view, Nozzle develops the notion of *heap health*, a metric indicates the scale of the surface area of NOP sled throughout the whole heap, and it will report an attack if *heap health* reaches a threshold. Despite informative limitations described in Nozzle, the most critical limitation it did not consider is Nozzle fails to detect heap spraying attacks which use few amount of NOP sleds that is far below tunable threshold. Such attack is elaborated in background section of this paper and is the very attack we intend to defend against.

An exemplary work of second type is Drive-by [16]. Drive-by exploits the fact that in order to populate memory with large amount of "NOP sled + shellcode", shellcode must be put in the form of string in the first place. Hence, they monitor all strings that are allocated by the JavaScript interpreter and check the presence of shellcode for each string. Because mainstream JavaScript engines implement strings as immutable (i.e., whenever a string is changed or replaced, an additional string variable is instantiated with the modified content), the possibility of a string manifesting itself as benign before the check and transformed into shellcode afterwards is eliminated. For this reason, each string is examined once. Furthermore, Drive-by set the threshold for recognizing shellcode within a string buffer as 32 bytes. This may not seem straightforward at first sight considering x86 instruction set is so dense that any string should have been interpreted as legitimate x86 instructions with length above this value. As a matter of fact, JavaScript uses 16-bit Unicode characters to represent string. Even if a given sequence of ASCII characters results in valid x86 instructions most of the time, the JavaScript representation of the same characters most likely does not, since every other byte would contain the value 0x00. Such conclusion is supported by their experimental results showing that benign pages typically do not contain strings that map to valid x86 instruction sequences, and we use this as one of the most important assumptions in our work. One major challenge their work confronts at present is prior assumption that it is impossible for attackers to evade detection by distributing shellcode fragments, each with length less than 32 bytes, over multiple strings does not hold anymore. This is because attackers leveraging allocation granularity is able to control the positions of shellcode fragments and chain those together using jumps without the help of NOP sled. By introducing non-determinism into allocation of memory blocks, we have confidence that our prototype would prevent such attacks.

Compared to Nozzle and Drive-by, BuBBle [34] is a countermeasure that takes an entirely different approach. Their underlying idea is identical to that of DEP but is implemented in a different way: execution should stop whenever control flow is transferred to a non-executable memory region. Their goal is achieved by substituting characters in strings at random positions by special interrupting values (0xCC). These special interrupting values will raise an exception when it is executed as an instruction. The modified string is stored in memory. A custom data structure that keeps track of the original values and where in the string they are stored is used for restoring original string when string is read by application. Also, any string manipulation operation that results in an instantiation of a new string must apply this countermeasure to the new string before it is stored back in memory. As a result, scripts that require intensive string manipulations will incur a performance penalty.

Heap Taichi [20] improves heap spraying attacks by significantly reducing the entropy of the positions of memory blocks. They leverage the implication of modern operating systems' memory allocation granularity and introduce a new heap spraying technique that exploits the weakness of memory alignment. To be more specific, if a heap object to be allocated is bigger than a certain threshold (512 KB in their experiment), Windows always allocates a separate heap block for this object, which is aligned to 64 KB boundaries. Accordingly, the starting address of this heap object is more or less predictable rather than true randomness. Such attacks subvert assumptions made in prior works including Nozzle and Drive-by, which makes this type of attack effectively evade existing detection tools. They propose to use finer memory allocation granularity at memory managers of all levels for prevention purpose. Our work tries to defend against such attacks from another perspective.

6.3 Memory Corruption Detection and Prevention

Recall that heap spraying alone is a memory manipulation technique that structures memory layout in a way to attacker's advantage. Such behavior itself is of no danger and would not compromise victim's host. For the sake of achieving a successful heap spraying attack, it must be coupled with another memory corruption technique to cause program's control flow to be redirected to a target address. From another point of view, heap spraying attacks can be prevented if any use of memory corruption techniques can be prevented or detected. Techniques such as control flow integrity [28], write integrity testing [29], data flow integrity [30] and program shepherding [31] take precaution against memory corruptions by restricting control-flow transfer to predetermined safe destination only. Although other defense against memory corruption techniques such as Integer Overflow and Format String Overflow are respectively proposed in IntScope [32] and FormatShield [33], it is still hard to find a panacea that can detect all possible exploits with cost-effective overhead, making these solutions far from being widely deployed.

Chapter 7

Conclusion

To the best of our knowledge, Dice is the first system specifically designed to prevent and detect Heap Taichi, an enhance heap spraying attack that exploits allocation granularity of system. At runtime, Dice monitors all string allocations in JavaScript engine and checks if the size of requested allocation memory exceeds atomic attack size. If so, it allocates more memory than as required and prefixes a random sized memory block to the real content of string for the sake of randomizing the start position of string and making it less predicable to the attacker. Our detector is further triggered to examine whether the attacker uses a predetermined length of NOP sled to bypass the above defense and would fire an alarm once the detection threshold is reached. We employ sampling technique to reduce the performance penalty incurred by Dice. A variety range of benchmarks are performed and we show that the performance slowdown is less than 5% on average and the memory overhead in the worst case is roughly 2%.

Although we consider string as the only heap object in this thesis, the two-layer defensive scheme is conceptually generic and can be applied to other heap objects with ease. Another generic idea worthy to be taken away is that if a low detection threshold value is unable to distinguish from benign data from malicious one, as discussed in section 5.1, we may raise the security bar to force the attacker to add additional noticeable elements for detection purpose.

References

[1] Microsoft Internet Explorer buffer overflow exploit. http://www.exploit-db.com/exploits/612, 2004.

[2] Microsoft security bulletin MS04-040. http://www.microsoft.com/technet/security/bulletin/ms04-040.mspx, 2004.

[3] Microsoft security bulletin MS05-020. http://www.microsoft.com/technet/security/bulletin/ms05-020.mspx, 2005.

[4] A. Sotirov. Heap feng shui in JavaScript. In *Blackhat*, *USA*, 2007.

[5] G. Kurtz. Operation aurora hit Google, Others. http://siblog.mcafee.com/cto/operation-%E2%80%9Caurora%E2%80%9D-hit-google-others/, 2010.

[6] Microsoft security advisory (979352). http://microsoft.com/technet/security/advisory/979352.mspx, 2010.

[7] H. Li. Smashing Adobe's heap memory management systems for profit. http://www.fortiguard.com/analysis/pdfanalysis.html, 2009.

[8] K. Selvaraj and N. Gutierrez. The rise of PDF malware. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the_rise_of_pdf_malware.pdf, 2010.

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

[10] Microsoft Corporation. Data execution prevention. http://technet.microsoft.com/enus/library/cc738483.aspx, 2003.

[11] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 56, 2000.

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

[13] Wikipedia. Buffer overflow protection. http://en.wikipedia.org/wiki/Buffer_overflow_protection#GCC_Stack-Smashing_Protector_.28ProPolice.29.

[14] SecurityFocus. Libsafe. http://www.securityfocus.com/archive/1/395999, 2005.

[15] B. Liu. Flash Mob: Spraying the heap. http://blog.fortinet.com/flash-mob-spraying-the-heap/, 2009.

[16] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of DIMVA*, pages 88-106, 2009.

[17] M. Polychronakis, K.G. Anagnostakis, and E.P. Markatos. Network-level polymorphic shellcode detection using emulation. In *Proceedings of DIMVA*, pages 54-73, 2006.

[18] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle. A defense against heap-spraying code injection attacks. In *Proceedings of the Usenix Security Symposium*, USENIX, 2009.

[19] A. Sotirov and M.Dowd. Bypassing browser memory protections. In *BlackHat*, USA, 2008.

[20] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou. Heap Taichi: Exploiting memory allocation granularity in heap-spraying attacks. In *Proceedings of ACSAC*, pages 327-336, 2010.

[21] M. E. Russinovich and D. A. Solomon. Windows internals: Covering Windows Server 2008 and Windows Vista. Fifth Edition. Microsoft Press, 2009.

[22] Skape and Skywing. Bypassing Windows hardware-enforced data execution prevention. *Uninformed Journal*, 2(4), Sept. 2005.

[23] E. Buchanan, R. Roemer, S. Savage, and H. Shacham. Return-oriented programming: exploitation without code injection. In *BlackHat*, USA, 2008.

[24] P. Sole. Defeating DEP, the immunity debugger way. http://www.immunityinc.com/downloads/DEPLIB.pdf, 2008.

[25] D. Blazakis. Interpreter exploitation: Pointer inference and jit spraying. In *Blackhat*, USA, 2010.

[26] M. Howard, M. Miller, J. Lambert, and M. Thomlinson. Windows ISV software security defenses. http://msdn.microsoft.com/en-us/library/bb430720.aspx, 2010.

[27] Whitehouse. An analysis of address space layout randomization on Windows Vista. In *Symantec Advanced Threat Research*, 2007.

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

[29] 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.

[30] 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.

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

[32] T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, 2009.

[33] P. Kohli and B. Bruhadeshwar. FormatShield: A binary rewriting defense against format string attacks. In *Proceedings of ACISP*, pages 376-390, 2008.

[34] F. Gadaleta, Y. Younan, and W. Joosen. BuBBle: A JavaScript engine level countermeasure against heap-spraying attacks. In *Proceedings of ESSoS*, pages 1-17, 2010.

[35] Download and install debugging tools for Windows. http://msdn.microsoft.com/en-us/windows/hardware/gg463009.

[36] R. Kath. Managing virtual memory. http://msdn.microsoft.com/en-us/library/ms810627.aspx, 1993.

[37] A. One. Smashing the stack for fun and profit. *Phrack Magazine vol.7*, issue 49, file 14 of 16, 1996.

[38] A. Cugliari and M. Graziano. Smashing the stack in 2010. http://mariano-graziano.llab.it/docs/stsi2010.pdf, 2010.

[39] Lupin. Heap spray exploit tutorial: Internet Explorer use after free aurora vulnerability. http://grey-corner.blogspot.com/2010/01/heap-spray-exploit-tutorial-internet.html, 2010.

[40] P. Vreugdenhil. Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit. http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf, *Pwn2Own 2010*, 2010.

[41] M86 Security. Web exploits: There's an App for That. http://www.m86security.com/documents/pdfs/security_labs/m86_web_exploits_report.pdf, 2010.

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

[43] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. G. Anagnostakis. STRIDE: Polymorphic sled detection through instruction sequence analysis. In *Proceedings of Security and Privacy in the Age of Ubiquitous Computing*, pages 375-392, Springer, 2005.

[44] Alexa Inc. Global top sites. http://www.alexa.com/ top_sites, 2011.

[45] Mozilla Foundation. Getting Mozilla source code using mercurial. https:// developer.mozilla.org/En/Developer_Guide/Source_Code/Mercurial.

[46] Mozilla Foundation. SpiderMonkey. https://developer.mozilla.org/en/ SpiderMonkey.

[47] Libdasm. http://code.google.com/p/libdasm/, 2010.

[48] Intel IA-32 architectures software developer's manual volum1: basic architecture.

[49] Webkit. SunSpider JavaScript benchmark – version 0.9.1. http://www.webkit. org/perf/sunspider/sunspider.html.

[50] Google. V8 benchmark suite – version 6. http://v8.googlecode.com/svn/data/ benchmarks/v6/run.html.

[51] Mozilla Foundation. Dromaeo JavaScript performance testing. http://dromaeo. com/?dromaeo.