Heap Taichi: Exploiting Memory Allocation Granularity in Heap-Spraying Attacks

Yu Ding[1,2], Tao Wei[1,2]*, TieLei Wang[1,2], Zhenkai Liang[3], Wei Zou[1,2],

[1]Institute of Computer Science and Technology, Peking University
[2]Key Laboratory of Network and Software Security Assurance(Peking University), Ministry of Education, Beijing 100871, China
[3]Department of Computer Science, School of Computing, National University of Singapore

ABSTRACT

Heap spraying is an attack technique commonly used in hijacking browsers to download and execute malicious code. In this attack, attackers first fill a large portion of the victim process's heap with malicious code. Then they exploit a vulnerability to redirect the victim process's control to attackers' code on the heap. Because the location of the injected code is not exactly predictable, traditional heap-spraying attacks need to inject a huge amount of executable code to increase the chance of success. Injected executable code usually includes lots of NOP-like instructions leading to attackers' shellcode. Targeting this attack characteristic, previous solutions detect heap-spraying attacks by searching for the existence of such large amount of NOP sled and other shellcode.

In this paper, we analyze the implication of modern operating systems' memory allocation granularity and present Heap Taichi, a new heap spraying technique exploiting the weakness in memory alignment. We describe four new heap object structures that can evade existing detection tools, as well as proof-of-concept heap-spraying code implementing our technique. Our research reveals that a large amount of NOP sleds is not necessary for a reliable heap-spraying attack. In our experiments, we showed that our heap-spraying attacks are a realistic threat by evading existing detection mechanisms. To detect and prevent the new heap-spraying attacks, we propose enhancement to existing approaches and propose to use finer memory allocation granularity at memory managers of all levels. We also studied the impact of our solution on system performance.

1. INTRODUCTION

Heap spraying is a new attack technique commonly used in recent attacks to web browsers [4–8, 36]. In a heap-spraying attack, attackers allocate objects containing their malicious code in the victim process's heap, and then trigger a vulnerability to force the victim process to execute code from the heap region. Compared to traditional buffer overflow attacks, heap spraying is simpler, as there is no need to know the detailed layout of memory regions surrounding the buffer vulnerable to overflow, but the heap object locations are not predictable. In order to increase the chance of success, existing heap-spraying techniques allocate lots of heap blocks filled with a large amount NOP-like instructions (e.g. `0x90`, `0x0c 0x0a`), called *NOP sled*, and followed by the malicious shellcode. The NOP sled serves as the *landing area* of the shellcode, which leads the execution to the shellcode if the victim process jumps to anywhere in the NOP sled.

Since a typical heap object used in a heap-spraying attack is in the form of "NOP sled + shellcode," the *large amount of NOP sled* and *existence of shellcode* are the main characters used by heap-spraying attack detectors. Accordingly, existing approaches to detect heap-spraying attacks mainly fall into two types: *sled-oriented* and *shellcode-oriented*. Shellcode-oriented methods detect heap-spraying attacks by detecting the existence of shellcode. For example, Egele et al. [21] detect heap-spraying attacks by inspecting the JavaScript string objects to identify shellcode using lightweight emulation [9]. However, this type of approach have difficulty in dealing with shellcode obfuscation techniques, such as, shellcode encoding [28, 34], encryption [46], polymorphism [20, 24], and other obfuscation schemes [17, 27, 31, 37].

A more successful type of techniques to detect heap-spraying attacks are sled-oriented [11, 29, 32, 42]. Such techniques focus on identifying large chunks of NOP sled. In particular, NOZZLE [32] uses static analysis to build the control-flow graph (CFG) of heap memory blocks and measures the size of NOP sled, called *surface area*, across a process's entire heap region. If the percentage of surface area is above a certain threshold, NOZZLE reports an attack. NOZZLE assumes that heap-spraying attacks must inject a large number of executable codes (especially NOP sled) because attackers cannot predict the location of their malicious code. Next, we will show that this assumption is not always valid.

We observe that modern operating system memory allocation behavior is more predictable than we usually believe, even in the presence of address space layout randomization (ASLR). For instance, the Windows-family systems (from Windows XP to Windows 7) enforce a memory allocation granularity of 64K bytes [22, 33], which makes all memory blocks *directly* allocated by Windows (using API `VirtualAlloc`) aligned to a 64K-byte boundary. As a result, addresses of such heap blocks are less random. For example, a particular address in a 1MB block only has 16 possible locations, much less than the one million possible locations if the heap block can be allocated at random addresses. We discuss this in detail in Section 3.

*Corresponding author. Email: *weitao@icst.pku.edu.cn*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA
Copyright 2010 ACM 978-1-4503-0133-6/10/12 ...$10.00.

A new attack. Based on the above analysis, we present a new heap-spraying technique, called *Heap Taichi*, which can evade existing detection mechanisms. By precisely manipulating the heap layout, Heap Taichi only needs to put executable code at a small number of offsets in a heap block, and thus makes the "large of NOP sled" feature in traditional heap-spraying attacks unnecessary.

To demonstrate the feasibility of Heap Taichi, we made proof-of-concept heap-spraying attacks using Heap Taichi. Our experiments showed that the surface area of a Heap Taichi attack is significantly less than the acceptable threshold used in existing solutions. We also studied the impact of different memory-allocation granularity on heap-spraying attacks and system performance, and found that larger memory allocation granularity gives attackers more flexibility without significant gain in performance.

To address this problem, we proposed methods to enhance existing heap-spraying attack detection techniques by considering memory allocation granularity, and experimented with new ways of memory allocation.

Contributions:

- We analyze the implication of modern operating systems' memory allocation granularity on heap-spraying attacks, and present a new heap-spraying technique utilizing the weakness of memory alignments, which can effectively evade existing detection tools.

- We present four heap object structures that do not require a large amount of NOP sled. We provide insight into the relationship between memory alignment size and heap-spraying attack surface areas.

- We implement proof-of-concept Heap Taichi, and measure the attack surface areas of these attacks. Experiments showed that our heap-spraying attacks are a realistic threat, which can evade existing detection tools.

2. HEAP SPRAYING AND DEFENSE

In this section, we describe a typical heap-spraying attack, and discuss existing defense mechanisms.

2.1 Heap-spraying attacks

Throughout the paper, we use the term *heap region* to refer to all the memory areas of a process's heap. We use the term *heap block* to refer to the memory block allocated for heap, e.g., the blocks allocated by Windows's memory management through the `VirtualAlloc` family APIs. We call individual objects allocated on the heap *heap objects*, e.g., objects allocated by the API `HeapAlloc`. Therefore, a heap region consists of several heap blocks, and a heap block contains one or more heap objects.

Figure 1 illustrates a typical heap-spraying attack found by our web crawler. The attack is launched by malicious JavaScript in a web page, targeting a vulnerability in the Internet Explorer version 6 or version 7 [18]. In the first step of this attack, attackers create a large amount of heap objects. Each heap object is filled with a large number of NOP-like instructions (`0x0c0c`, the instruction `or al, 0ch`) followed by a block of malicious shellcode. Illustrated in the right-hand side of Figure 1, the large white areas are the NOP-like instructions, while the grey areas are the shellcode. If attackers can hijack the process's execution to any byte in the range of NOP-like instructions, the malicious shellcode will be executed. Although attackers cannot know the exact address of the injected code, when the browser process's heap region is very large, certain range of address, such as `0x0c0c0c0c`, will be in the region of allocated heap objects, as is illustrated in Figure 1.

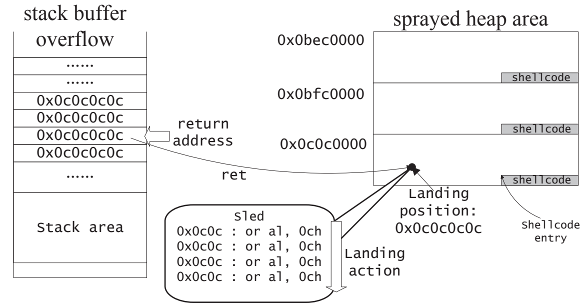

Figure 1: The traditional heap spraying with stack buffer overflow exploit.

After the heap is prepared with malicious shellcode, the second step of this attack is to exploit a vulnerability in the victim process, forcing the victim process to transfer control to the sprayed heap region. Any vulnerability that can be exploited to affect the control flow can be used in this step. Here we show an example using a stack-based buffer overflow vulnerability, illustrated in the left-hand side of Figure 1. By exploiting the buffer overflow vulnerability, attackers inject lots of `0x0c` characters onto the stack, overwriting the return address. When the program returns using the corrupted return address, its execution is redirected to the address `0x0c0c0c0c`, which is in the NOP sled of a sprayed heap object. The victim program will continue through the NOP sled and reach attackers' shellcode.

Thanks to the defense mechanisms against buffer overflow attacks, it is very hard for attackers to know the exact address of their shellcode. Therefore, they cannot use the stack overflow to execute their shellcode directly. In contrast, heap-spraying attacks do not require attackers to know the detailed layout of the data structures of the targeted memory region. But the object addresses on the heap are harder to predict, even with the deployment of ASLR [22, 33, 40, 44]. To increase the chance of success in the second step of the attack, attackers usually put a long NOP sled before the shellcode in their heap objects, and they have to inject a large amount of heap objects containing shellcode, so that the jump target of the attack will be covered by injected code with a high probability. Otherwise, if the victim process jumps into the middle of shellcode, or even jumps out of the heap region sprayed by the attacker, the victim process often crashes because of invalid memory access or invalid instructions.

In the rest of the paper, we use the following terms to describe the behavior of a heap-spraying attack. 1) We call the execution after the exploit and before running the shellcode a *landing action*. In traditional heap-spraying attacks, the landing action usually runs on the huge sled area, byte by byte. The traditional sled is filled with NOP like bytes, such as `0x90` (NOP), `0x0c0c` (or al, 0ch) and these bytes lead to smooth landing actions. On the contrary, landing actions executing some jump instructions, such as `jmp`, are called bumpy landing. 2) The place where the landing action starts is called *landing position*, or *landing point*. 3) The notion of *surface area* is defined in the NOZZLE paper [32] as the number of available landing positions in one heap object. 4) The *normalized attack surface area (NSA)* is a heap object's surface area divided by the heap object's size. The normalized attack surface area represents the percentage of sled in a memory block. It also represents the possibility of successfully executing the shellcode when execu-

tion randomly falls into a heap object. 5) The *shellcode entry* is the starting point of the shellcode.

2.2 Existing defense mechanisms

Existing defense mechanisms against heap-spraying attacks can be classified into two main types based on the analysis they perform on heap objects. Approaches of the first type detect shellcode by searching for common patterns of shellcode. Approaches of the second type analyze the control flow structure of heap objects to identify common structures used in heap-spraying attacks.

Egele et al. [21] is an example of the first type. It monitors all strings objects allocated in a browser's JavaScript engine, and reports an attack when there is shellcode detected in string objects created by the script. To detect shellcode, it uses the libemu library to identify suspicious and valid instruction sequences longer than 32 bytes. As is discussed in the paper [21], attackers can evade detection by breaking shellcode into multiple fragments smaller than 32 bytes and linking them with indirect jump/call instructions. However, unless attackers can precisely control the landing position, they still need a large portion of NOP sled to make a reliable attack, which is well over the 32 byte threshold.

NOZZLE [32] is an example of the second type. Given a heap object, it disassembles possible x86 instructions in the object and build a control flow graph (CFG). As we have described earlier in this section, the heap block used in a typical heap-spraying attack contains a block of shellcode, and the rest of the heap block contains instructions leading to the shellcode. NOZZLE searches for this property in the CFG by identifying the location S that can be reaching from most of other locations in the heap object. The total number of locations that lead to S is the *surface area* of the heap object. In other words, NOZZLE draws the CFG of the heap block. For each basic block in the CFG, it counts the number of instructions that connect to the basic block. NOZZLE then calculates the surface area of the entire heap. When the surface-area-to-heap-size ratio is greater than a threshold, NOZZLE reports a heap-spraying attack. This approach is more accurate than the first type approaches, because it looks for more intrinsic properties of heap-spraying attacks: when the location of shellcode is not predictable, it is necessary to include large surface areas to increase the chance for success.

Both types of existing solutions assume *attackers have little information about the address of their shellcode*. With this assumption, attackers cannot break sled and shellcode into small pieces to evade the approach of Egele et al.; they also cannot evade NOZZLE by only including very little NOP sled instructions. This assumption is valid if heap objects are allocated randomly without restriction. However, the randomness of heap object allocation is limited by memory alignment enforced in operating systems. Next, we discuss its impact on heap memory allocation and describe an attack that can evade both types of defense mechanisms.

3. HEAP SPRAYING WITH LITTLE SURFACE AREAS

Memory alignment is commonly adopted in modern operating systems for better memory performance. With memory alignment, a memory block allocated for a process cannot start from arbitrary addresses. Instead, the addresses must be multiples of the alignment size defined by the system.

In this section, we analyze the memory allocation behavior of the Windows platform and its implication on heap-spraying attacks. Then we describe a new attack technique that can evade existing heap-spraying detection mechanisms. Note that other operating systems such as Linux have a similar memory allocation behavior to Windows, which differs mainly in the default memory alignment size.

3.1 Windows memory allocation granularity

Windows memory alignment is controlled by the *allocation granularity*. On all existing Windows platforms, the value of allocation granularity[1] is always 64K [33]. This size 64K was chosen in part for supporting future processors with large page sizes [22], as well as solving relocation problems on existing processors [3]. The memory allocation granularity only affects user-mode code; kernel-mode code can allocate memory at the granularity of a single page [22]. As a result of the Windows memory allocation granularity, almost all of the base addresses of non-free regions are aligned with 64K boundaries. In a process's memory space, only few regions (allocated by kernel-mode code [33]) are not aligned. Even with ASLR enabled [40], the alignment of memory region addresses is not affected. On Linux systems, the memory allocation granularity is 4K bytes.

Therefore, taking Windows as an example, all heap blocks allocated by user-mode code start from 64K boundaries. Note that heap objects allocated by `HeapAlloc` can still start at random addresses in a heap block, but we have an interesting observation: when a heap object is bigger than a certain threshold, 512K in our experiment, Windows always allocates a separate heap block for this object. That is, the addresses of large heap objects are also aligned according to the allocation granularity, thus more predictable.

What is the implication of such a memory allocation behavior on heap-spraying attacks? Recall that in the second step of a heap-spraying attack, after the attacker triggers a control-hijacking exploit successfully, the victim process's EIP register is loaded with a value assigned by the attacker. If the starting addresses of heap objects are fully random, the EIP can fall anywhere in a heap object. For example, when the heap object's size is 512K bytes, the hijacked EIP can point to any byte of the 512K bytes. This is the main reason for requiring a large amount of NOP-like instructions in heap-spraying attacks. However, the Windows memory allocation granularity makes large heap objects' addresses much more predictable. If an EIP assigned by an attacker have few possible locations in a large heap object, the attacker only need to put jump-equivalent instructions at those locations to guide the victim process to execute malicious shell code, which breaks the assumptions, relied on by previous defense mechanisms. As a result, the large block of NOP sled is no longer necessary for a heap-spraying attack with high chances to succeed.

In fact, an EIP assigned by an attacker can only point to *EIGHT* possible locations in a 512K-byte heap object, which is explained in Figure 2 using the address `0x0c0c0c0c`. Due to the 64K (0x10000) Windows memory allocation granularity, a 512K-byte heap object covering the address `0x0c0c0c0c` can only start from `0x0c050000`, `0x0c060000`, ..., `0x0c0c0000`. Therefore, the offset of the address `0x0c0c0c0c0` inside the object have eight possible values: `0x70c0c`, `0x60c0c`, ..., `0x00c0c`. On each of these offsets, if the attacker puts a few bytes, say 20 bytes (the unconditional jump instruction takes five bytes on 32-bit x86), of jumping instructions, the resulting surface area is very little: 160 bytes out of a 512K-byte object.

[1] It can be retrieved by the `GetSystemInfo` API (the `dwAllocationGranularity` member of the returned `SYSTEM_INFO` structure).

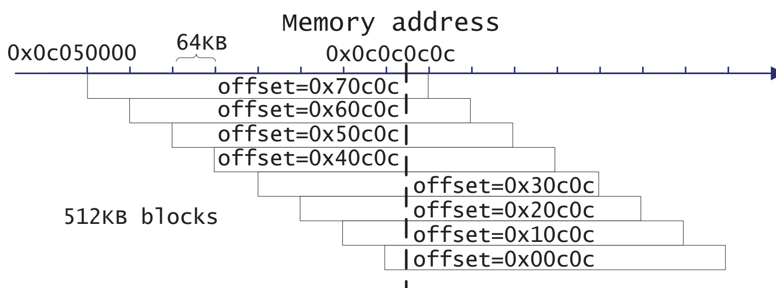

Figure 2: Possible offset of `0x0c0c0c0c` in a 512KB heap blocks.

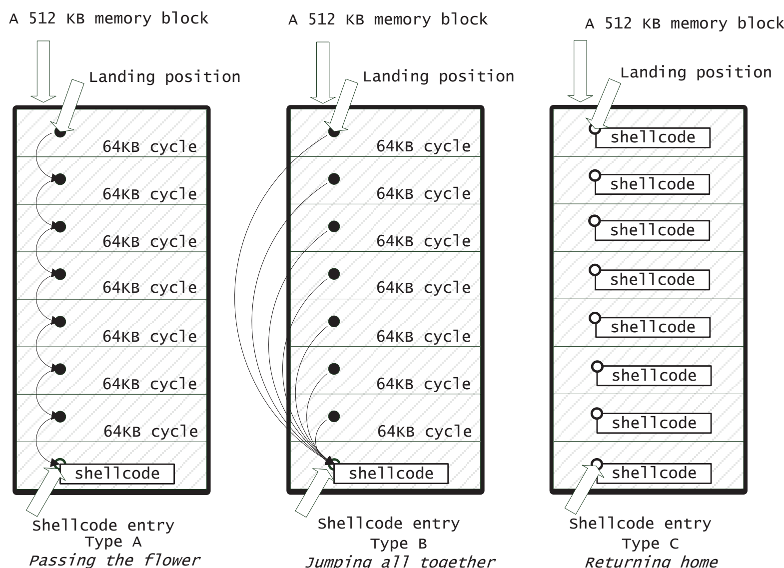

Figure 3: Structures of new heap-spraying memory blocks.

3.2 Structure of malicious heap objects with little surface area

As is discussed in the previous section, given a specific address $addr$ in the heap region, the landing action can only start at few offsets in a large heap object. Executable code at other offsets will never be the direct jump target when the process transfers control to the address $addr$. With this new insight, we describe a few new structures of malicious heap objects that result in very little surface area.

The general idea is to put jump-equivalent instructions at possible landing positions to guide execution into attackers' shellcode. The shellcode is a small piece of code connected by jump-family instructions, which can evade the approaches that detect valid instruction sequences. Figure 3 shows three types of the new heap block structures that have little surface area. In this figure, each rectangle with bold boundary stands for a heap object. The shadow areas are bytes with random values. The possible landing positions are represented as solid dots. Shellcode is represented as white rectangles, with a circle indicating its entry point.

- In the Type A structure, attackers first copy the block of malicious shellcode into the heap object. The landing positions are chained together to reach the shellcode entry. That is, each landing position is a set of jump-equivalent instructions that point to the next landing position. The instructions at the last landing position lead to the shellcode entry.
- In the Type B structure, attackers put jump-equivalent instructions at the possible landing positions. Each group of jump instruction will jump to the shellcode entry.
- In the Type C structure, the malicious shellcode is directly put at each landing position. By using this kind of memory blocks, the landing action is eliminated and the shellcode is executed immediately after the exploit is triggered.

In the Type C structure, although there are several copies of shellcode, the surface area is as small as one copy because the copies of shellcode are not connected. The Type C structure requires the shellcode size to be smaller than the alignment granularity. To launch such an attack on an operating system using a small alignment granularity, say 32 bytes, we introduce the Type D heap object structure, which is an improved Type C structure.

Shown in Figure 4, the main idea of this structure is that we can split the shellcode into pieces and link these pieces with jump instructions. We place jump instructions at each landing point to jump to the shellcode. Similar to the Type C structure, although there are lots of shellcode copies in the heap block, the measurable surface area is small. We illustrate this type of structure by an example. Assuming the memory allocation granularity is 32 bytes, we construct a 512K-byte heap block using the Type D structure, which includes 1024 copies of shellcode. In the heap block, we need to create $512K/32 = 16384$ landing points. Each landing point connects to one of the shellcode copies sequentially or arbi-

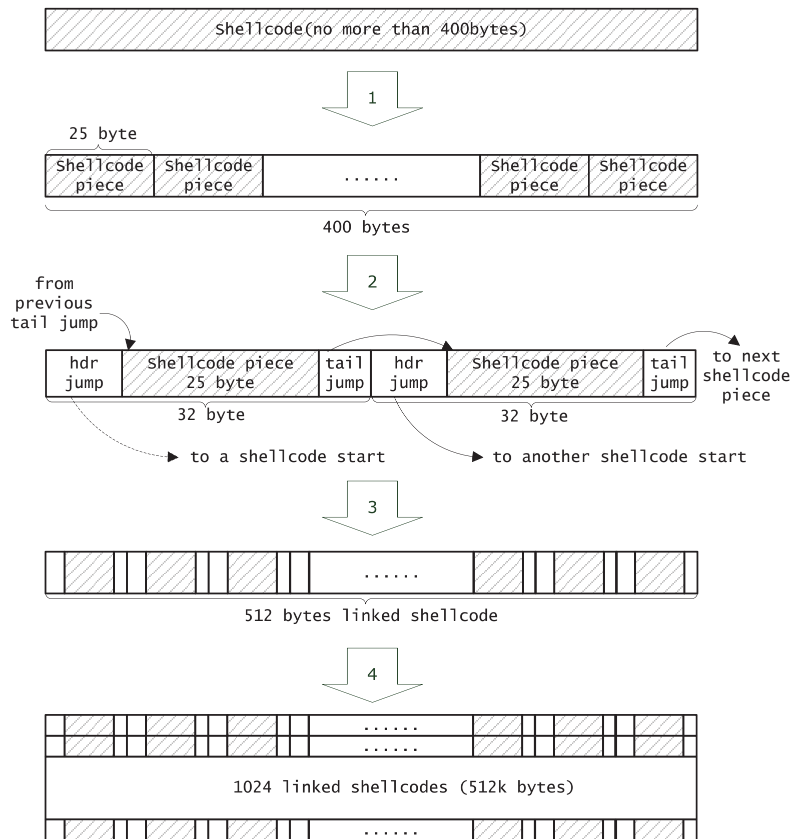

Figure 4: Type D layout 'Dropping around'

trarily. This transformation is still a "sled construction" technique, which plants landing points inside the shellcode. The shellcode features are not changed after these landing points inserted.

Type D structures can be created using the following technique. Given a piece of shellcode[2], we first split it into pieces, where each piece is less than or equal to 25 bytes. If a piece is less than 25 bytes, we append a few NOP-like instructions to it to make the size of all pieces 25 bytes. To connect the shellcode pieces, we enclose each shellcode piece between a prologue and an epilogue, shown in Step 2 of the figure. The prologue is called "*header (hdr) jump*" and it's a jump instruction (5 bytes, jump near, relative, displacement relative to next instruction) pointing to the shellcode's starting position. We need to distribute the header jumps to the start of 1024 copies of shellcode evenly. In the attack, the prologues are put at landing points. The epilogue is called "*tail jump*" and it's a jump instruction (2 bytes, jump short, relative, displacement relative to next instruction). In the attack, the epilogues connect the shellcode pieces. The tail jump only jumps $2 + 5 = 7$ bytes forward. So with the prologue and epilogue, each shellcode piece is extended to $25 + 5 + 2 = 32$ bytes. In the third step, we combine 16 such 32-byte pieces to form shellcode of $16 \times 32 = 512$ bytes. We call it a *512-byte linked shellcode*. To fit the selected original shellcode into such a block, the shellcode size should be less than $25 \times 16 = 400$ bytes. Finally, we merge 1024 linked shellcode pieces into one heap memory block. There are $1024 \times 16 = 16384$ header jumps inside the heap memory block and they are the landing positions.

The final heap memory block will be used in our new heap-spraying attack. The possible landing positions are at each 32 byte boundary. So we could exploit to address such as `0x0c0c0c20`, `0x0c0c0c40`, `0x0c0c0c60`, and etc. When the execution starts from any one of the landing positions, it will reach the shellcode.

[2]The size of shellcode ranges from dozens to hundreds [12].

Alignment size	Type A	Type B	Type C	Type D
64 kbytes	√	√	√	√
32 bytes	√	√	×	√
8 bytes	√	√	×	×
4 bytes	√	×	×	×

Table 1: Relationship between layout types and alignment size.

We summarize the relationship between four heap object structures and the memory alignment boundaries in table 1. When the alignment size is 64K bytes, all four heap object structures can be used. More generally, all four heap object structures can be used as long as the alignment size is larger than the size of the shellcode. When the alignment size is smaller than the size of the shellcode, the Type C layout does not work anymore, but the Type D is still effective.

In the new attack discussed in this paper, the sprayed heap objects are mostly filled with bytes that cannot be treated as NOP sled or bytes that cannot be interpreted as legal x86 instructions. NOZZLE can only find memory blocks that have a normalized surface area much lower than its threshold.

3.3 Surface area calculation

Our calculation involves the following variables: heap memory block size $size_{block}$, alignment size $size_{alignment}$, shellcode size $size_{sc}$, and normalized attack Surface Area NSA. We use NSA_{typeX} to represent the normalized attack surface area of Type X.

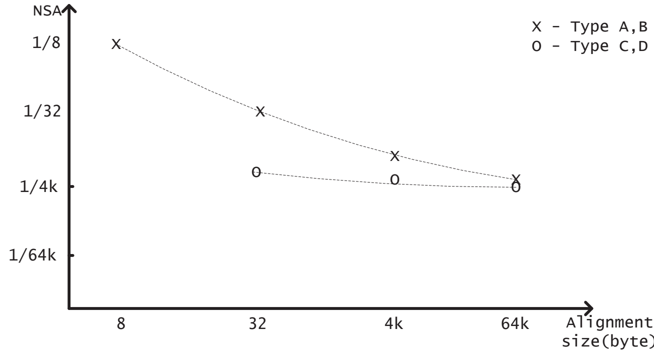

Figure 5: Normalized attack surface.

$$
\begin{aligned}
NSA_{typeA} &\approx NSA_{typeB} \\
&\approx \frac{\frac{size_{block}}{size_{alignment}} + size_{sc}}{size_{block}} \\
&= \frac{1}{size_{alignment}} + \frac{size_{sc}}{size_{block}} \\
NSA_{typeC} &\approx \frac{size_{sc}}{size_{alignment}} \\
NSA_{typeD} &\approx \frac{\frac{size_{sc}}{size_{alignment}} + size_{sc}}{size_{block}} \\
&= \frac{size_{sc}}{size_{block}} \times (1 + \frac{1}{size_{alignment}}) \\
&\approx \frac{size_{sc}}{size_{block}}
\end{aligned}
$$

From the formulas we can see that

- The NSA of Type A and B consists of two parts. The first term of the NSA is only relevant to alignment size, and the second term is relevant to both shellcode size and block size. The NSA of Type A and Type B increases when the alignment size or block size decrease, or when the shellcode size increases.

- The NSA of Type C is only relevant to the size of shellcode and memory block size. The NSA is proportional to the size of shellcode, and is inversely proportional to the size of the memory block size.

- The NSA of Type D is more complex, but it is clear that the NSA is inversely proportional to the size of the memory block. We also found that the NSA of Type D is much smaller than that of Type A and B.

There are three independent variables in these formulas and the function graph is hard to plot. To draw the graph, we must fix two of them. We assume that the heap memory block size is 1M bytes and the shellcode size is 256 bytes. Figure 5 shows the function graphs. The X-axis indicates the alignment size in bytes and the Y-axis indicates the normalized attack surface area (NSA). To simplify the calculation, we assume all the size of all instructions is one. Therefore, the surface area of practical samples may be two or three times of the theoretical value. As is showed in Figure 5, the normalized attack surface of all new heap objects is lower than the threshold of NOZZLE (50%).

3.4 Detecting Heap Taichi attacks

Enhanced NOZZLE *detection.*

From the above discussion, we can see that the assumptions made by NOZZLE are not necessary for a reliable heap-spraying attack.

NOZZLE can be enhanced to detect some of the new attacks by considering the effect of memory allocation granularity. The key is that all the landing positions should not be treated as the same. Instead, an enhanced NOZZLE algorithm should count the numbers of landing positions on each offset inside an "alignment-size segment" and record these numbers into an array. For example, on a 64K-byte aligned system, in a 1M-byte heap memory block, the three landing positions at `0x00c0c`, `0x10c0c`, `0x20c0c` number the count at `0x0c0c` as three. In the example of case study, the array at offset `0x0c0c` is counted as 8. Then we calculate the success rate on each offset. In the example of case study, the success rate on `0x0c0c` is $8 \div 8 = 100\%$ and on other positions the success rates are $0 \div 8 = 0\%$. Any success rate over 50% means a potential threat that may trigger a shellcode with a high success rate. The improved NOZZLE report a potential heap-spraying attack when it finds an offset with success rate over 50%.

However, the enhanced algorithm cannot deal with the Type C and D attack, where there are many copies of shellcode in one heap memory block. The landing positions are different from each other when analyzed statically because that they connect to different shellcode copies and these shellcode copies are not connected in the CFG. So, in Type C and D attack, the enhanced NOZZLE calculates the success rate at offset `0x0c0c` as $1 \div 8 = 12.5\%$. We report our evaluation results of this enhanced algorithm in Section 4.2.

Heap memory allocation in finer granularity.

The main problem behind this new type of attack is the predictability of heap addresses resulted from the coarse granularity of memory allocation. So a natural solution to prevent Heap Taichi attacks and similar attacks is to aligning memory allocation at a smaller-sized boundary. But we found it not easy to achieve in our experiments, because several application-level libraries align allocated memory objects by themselves.

There are many heap managers on different levels of a program, each of which has its own heap management strategy. For example, at the kernel level, there are "heap manager" in Windows, "SLUB allocator" in Linux, and Address Space Layout Permutation (ASLP) [26] in Linux. At the library level, there are libraries like jemalloc [23] and tcmalloc [35]. At the program level, we found that Firefox implemented a memory allocator based on object lifetimes named "JSArena" [25]. The heap manager on each level always manages its own "chunks" and also tries to get the chunks aligned on its own boundaries. Therefore, the granularity enforced by lower levels may be ignored in higher levels. For instance, jemalloc wraps `VirtualAlloc` and keeps its chunks aligned at 2M-byte boundaries. If the `VirtualAlloc` returns a memory block not aligned at the 2M-byte boundary, jemalloc frees the chunk and repeats the allocation until the returned memory block is aligned at the 2M-byte boundary.

To our understanding, the main reason for user-level alignment is performance. However, our performance evaluation (discussed in Section 4.3) showed that the gain in performance by the user-level alignment is not significant (less than 5% in our experiment). Therefore, memory managers at all levels should use finer memory allocation granularity for better security, a trade-off by sacrificing a limited amount of performance.

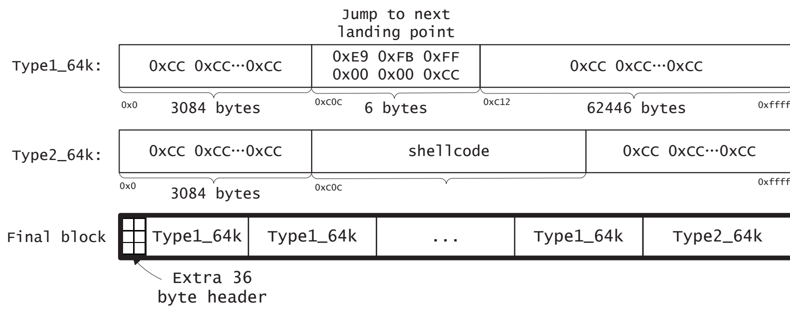

Figure 6: A sample structure of memory blocks with little surface area.

```
1   function heapspray(){
2    var scstring = unescape("%u9090...");
3    var alignment_size = 0x10000;
4    var pre_len = 0x00000c0c;
5    var post_len = alignment_size - 0x00000c0c - 0x6;
6    var head_offset = 0x24;
7    var jmp_str = unescape("%uFBE9%u00FF%uCC00");
8    var type1_str = CreateCCstringwithsize(pre_len) + jmp_str + CreateCCstringwithsize(post_len);
9    var type2_str = CreateCCstringwithsize(pre_len) + scstring +
        CreateCCstringwithsize(post_len + 0x6 - scstring.length * 2);
10   var type1_total_str = DuplicateStr(type1_str, 15);
11   type1_total_str = type1_total_str.substr(head_offset / 2,
                       type1_total_str.length - head_offset / 2);
                       // cut off the header bytes
12   var m = new Array();
13   for(i = 0; i < 200; i++)
14      m[i] = type1_total_str + type2_str;
15  }
```

Figure 7: Sample JavaScript spraying heap with Type A blocks.

4. EXPERIMENT AND EVALUATION

In this section, we describe our experiments of Heap Taichi, which generated heap objects that can bypass existing detection mechanisms. We also measure their normalized attack surface with different alignment sizes in the experiments.

4.1 Case study: A sample JavaScript code creating Type A heap objects

In this section, we give a JavaScript example of spraying a browser's heap with our Type A heap objects. This attack can also be done in other languages, including VBScript and ActionScript.

Figure 6 illustrates the structure used in this example. The malicious heap object's size is 1M bytes, consisting of two types of 64K-byte memory blocks. The first type only contains jump instructions at the landing positions, pointing to the landing position in the next block. The second type of block contains the shellcode at the landing position. We use the address `0x0c0c0c0c` as the jump target in step two of the attack. According to our analysis in Section 3, the landing position is at the offset `0x0c0c` of each 64K-byte block. We construct the final block by concatenating 15 type-1 blocks and one type-2 block, forming a heap object of 1M bytes. Note that each heap object allocated by Javascript has a 36-byte header (a Windows heap management header and a Javascript heap management header); we need to remove 36 bytes at the beginning of the final block, so that the offsets of landing positions will not be shifted by the header.

Figure 7 shows a piece of JavaScript code that implements a Heap Taichi attack, performing the heap object construction and heap spraying. The function `CreateCCstringwithsize` is used to create a string filled with value `0xCC` and the function `DuplicateStr` is used to create a long string. We fill the blocks with `0xCC`, because it is the opcode of x86's `INT 3` instruction, regarded as a terminator of a sequence of shellcode by existing approaches. We can fill these blocks with random bytes, because they are not used anyway. Because JavaScript strings use unicode encoding where each character takes 16 bits, we need to divide the length measured in bytes by two to get the correct length of unicode strings. Line 7 constructs the `type1_64k` block, and line 8 constructs the `type2_64k` block. Then line 9 and line 10 prepare the first half of the final block. Thirty six bytes are cut from the first half to accommodate the heap header. Finally, the heap is sprayed in line 12 to line 14 by an array of 200 strings containing the final block, taking up 200M bytes of the browser's heap region.

The `scstring` is filled with shellcode that libemu [9] cannot detect, which is captured by our drive-by download monitoring system [47]. The main reason that libemu cannot detect such shellcode is that libemu just emulates shellcode and once the shellcode includes instructions like `xor eax, [edi]` where register `eax` can only be determined at run-time, libemu cannot work well. For more evasion techniques, we refer readers to [29]. We extracted 44 shellcode pieces from those cached web pages, and 12 of them can't be detected by libemu. We choose a 236-byte shellcode to fill the `scstring`. Thus this script can bypass defending techniques based on libemu shellcode detection. We have also scanned this shellcode using 12 anti-virus products, and none of them could recognize it as a malicious code.

Sample ID	Heap Object Type	Alignment size
A64k	Type A	64k bytes
B64k	Type B	64k bytes
C64k	Type C	64k bytes
A4k	Type A	4k bytes
B4k	Type B	4k bytes
C4k	Type C	4k bytes
A32	Type A	32 bytes
B32	Type B	32 bytes
D32	Type D	32 bytes
A8	Type A	8 bytes
B8	Type B	8 bytes

Table 2: Samples used in surface area measurement

Block type	Alignment size			
	8 bytes	32 bytes	4K bytes	64K bytes
Type A	14%	3.6%	0.030%	0.0068%
Type B	25%	3.6%	0.030%	0.0068%
Type C			0.0055%	0.0054%
Type D		0.015%		

Table 3: Normalized attack surface area in our experiments

In our experiment, we modified a cached drive-by download web page by replacing its heap-spraying script with the one shown in Figure 7. Then we browsed the page using IE6 on Windows XP. The script reliably executed the shellcode, which downloaded and installed a bot on the victim machine.

4.2 Surface area measurement experiments

We build several example heap blocks of all the four heap structures and various alignment sizes, shown as Table 2. For example, A64k is the one given in the last subsection. B64k uses type B structure, a modified version of A64k with different jump instructions. C64k uses Type C structure, which can be achieved by replacing all `type1_str` with `type2_str` in our example JavaScript. The shellcode used in our experiment has 236 bytes, including 101 instructions. Its maximum attack surface area is 56.

To measure the normalized attack surface area (NSA), we implemented NOZZLE's surface area measurement algorithm. Table 3 summarizes the measured NSAs. We also plotted the results in Figure 8, where the Y-axis indicates the NSA of the attack vectors and the X-axis indicates the test cases. In Figure 8, we also marked several thresholds used in NOZZLE. When alignment size is 32 or higher, the normalized attack surface areas of the samples are far below the 50% threshold in NOZZLE.

When alignment size is 8, the Type C and Type D heap objects cannot be created. B8 exceeds the 20% "no false positive threshold" of NOZZLE, and A8 is on the border. The enhanced NOZZLE detection should cooperate with 8-byte or 4-byte heap allocation granularity. In the Type A and Type B objects, there are many landing positions connects to one copy of shellcode. The enhanced NOZZLE detects all of them and reports a heap spraying attack.

Also, we can see the difference in ratio between these results and the theoretical calculation in Section 3 is less than 3.0, which is close to the average instruction length. Therefore, the experiments confirm our theoretical analysis.

4.3 Performance of fine-grained memory allocation granularity

To evaluate the performance of 8-byte alignments, we built the Firefox 3.6.3 with jemalloc enabled and also modified jemalloc and SpiderMonkey with 8-byte randomization. Then we measured the modified Firefox's Javascript performance with Sunspider Javascript Benchmark and V8 Benchmark. The result showed that the performance overhead is less than 5%. Researchers [26] also reported acceptable performance overhead of an 8-byte aligned randomization using other benchmarks.

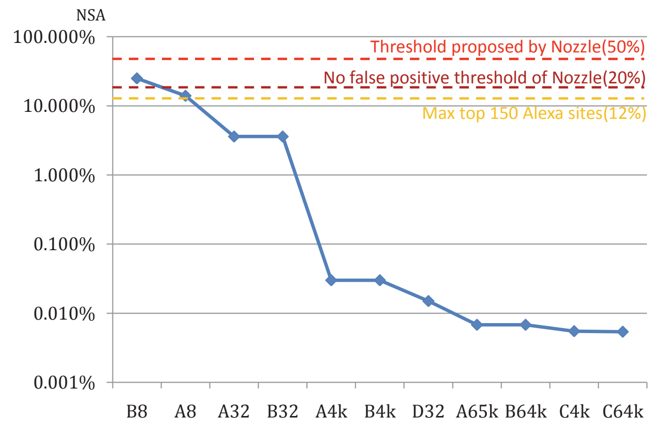

Figure 8: Sorted normalized attack surface area

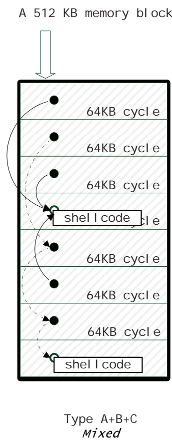

Figure 9: A sample mixed layout

5. DISCUSSION

5.1 Variations of Heap Taichi

Section 3 describes four basic memory layouts of Heap Taichi. Attackers may create new attacks by extending Heap Taichi.

At instruction level, attackers can replace those jump instruction with different instruction sequences, and fill arbitrary instructions between landing points and shellcode. At layout level, attackers can use the basic layouts to compose new layouts. For example, Figure 9 shows a "mixed" layout by combining Type A, B and C. Type C includes multiple copies of shellcode but keeps a small surface area; Type A and B layouts introduce more surface area but fewer shellcode copies. Attackers could use all these types in one heap block to balance these characteristics to evade detections.

5.2 Difficulty of detecting Heap Taichi attack

Under the 64K-byte alignment size, there are only 16 landing points in one 1M heap block as analyzed in Section 3. Attackers

could use mixed layouts similar to the example in Figure 9: place 3 to 5 shellcode pieces in one 1M heap block, and let landing points lead to any one of these shellcode pieces. In average, four landing points flow into one shellcode copy. Under this situation, there are no obvious anomalies in statistics compared to benign heap blocks, and it is very hard for methods like NOZZLE to detect this kind of attack without a high false positive rate.

As a consequence, to detect Heap Taichi under 64k memory alignment is as hard as to detect four shellcode copies in a 1M-byte heap object in real time. This could be a real challenge, and there is no practical solution which achieve both low false negative and low false positive so far.

6. RELATED WORK

6.1 Heap spraying with ASLR and DEP

The Address Space Layout Randomization(ASLR) technique [22, 33, 40, 44] is widely used in recent Windows versions such as Windows Vista and Windows 7. Analyses [40, 44] show that the randomization of heap area is quite weak. For each heap memory block, the system creates a five bit random number (between 0 and 31) and multiplies it with 64K, and then adds the product to the initial allocation base. This technique affects heap-spraying attacks, because it creates unpredictable gaps between the memory blocks. But attackers can deal with it by allocating a huge memory block and structure it carefully, so that the risk of landing in the gaps would be significantly reduced.

The ASLR-based defense is not effective on the new attack discussed in this paper. Because of the Windows memory allocation granularity, heap blocks are still aligned to 64K boundaries even after randomization. That means, the relative landing positions in each heap object is unchanged. As long as attackers can spray enough memory area using the heap region, the attack can still have a high success rate.

Data Execution Prevention (DEP) [1] is complementary to ASLR. It is an effective scheme to prevent an application or service from executing code from a non-executable memory region. Since shellcode is injected into non-executable memory region, most code injection attacks cannot work anymore when both DEP and ASLR are turned on. However, the attack techniques that can bypass DEP and ASLR are continually proposed. For example, Nenad Stojanovski et al. [41] showed that initial implementation of the software for DEP in Windows XP is actually not at all secure, and many attacks (such as return-to-libc like attack) can bypass DEP protection. Furthermore, Alexander Sotirov and Mark Dowd [39] implemented several exploitation techniques to bypass the protections and reliably execute remote code on Windows Vista. Dion Blazakis [15] illustrated two novel techniques (i.e., pointer inference and JIT spraying) to Windows Vista with IE8. Recently, during the PWN2OWN hacking contest 2010 [10], both IE 8 and Firefox 3 web browsers running on the Windows 7 system (both DEP and ASLR enabled) were successfully compromised. We believe that the attacks against DEP and ASLR cannot be completely avoided due to the vulnerabilities in operating systems or security-critical applications.

6.2 Heap-spraying attack and detection

Our approach is closely related to existing work on heap behavior manipulation, heap-spraying detection, as well as x86 executable code detection.

Heap behavior manipulation. A successful heap-spraying attack requires attackers to be able to predict the heap organization and, more importantly, locations of allocated heap objects. Sotirov [38] introduced a technique to use JavaScript to manipulate browser heap's layout, and implemented this technique into a JavaScript library for setting up heap state before triggering a vulnerability. Daniel et al. [19] developed a technique to reliably position a function pointer after a heap buffer that is vulnerable to buffer overflow. In this paper, we leverage a weakness on Windows heap allocation due to the large memory allocation granularity enforced on Windows systems, which makes heap allocation more predictable for attackers.

Executable code detection. Recent researches such as [28, 37] have proved that detecting arbitrary shellcode by static code features is difficult and even infeasible. In the context of network packets, several solutions [11, 30, 42] can detect executable code in the payload, but they cause high false positives in the context of heap objects [32], which makes them unsuitable for heap-spraying detection. In section 2.2, we have discussed several detection methods in detail.

6.3 Memory exploit detection and prevention

Note that heap spraying itself cannot directly cause the malicious payload to be executed. A successful attack needs another vulnerability to trigger the change of control flow to the sprayed heap. Detecting and preventing such vulnerabilities can stop heap spraying.

Buffer overflow is the common vulnerability exploited to redirect victim process's control flow. Traditional buffer overflow attacks target the pointer variables on stack or heap. A large number of solutions [45] have been proposed to address this problem. Among these efforts, address space layout randomization (ASLR) [2, 13, 14] provides general protection against memory exploits by randomizing the location of memory objects. It is now widely adopted in major operating systems. Note that address space layout randomization makes the location of memory objects, including heap objects, unpredictable, thus forcing heap-spraying attacks to inject a huge amount of heap objects containing code to increase the chance of success. This forms the basis for existing heap-spraying detection solutions.

Another common vulnerability exploited in browsers is integer overflow. Many integer overflow vulnerabilities are disclosed in recent years, and some integer overflow detection and prevention methods are proposed [16, 43]. Integer overflow leads to heap overflow in many cases, and heap spraying could construct step stones when exploiting these vulnerabilities.

In practice, it is very hard to eliminate all such vulnerabilities. Also, the runtime overhead prevents many of these approaches from being deployed widely. Therefore, the solution from this paper complements the approaches in memory exploit prevention.

7. CONCLUSION

Heap-spraying code injection attacks are commonly used in websites with exploits and drive-by downloads. This technique provides the attacker an easy-to-use code injection method which can be implemented in many type-safe languages. Since traditional heap spraying attacks require large number of NOP sled to increase the possibility of successful attacks, existing detection solutions mainly check for large amount of executable instructions on the heap.

By analyzing the operating systems' memory allocation mechanism, we found that the large amount of NOP sled is not necessary for heap spraying attacks if the memory alignment size is large enough. We introduced a new technique to launch heap-spraying

attack, which only injects a little amount of executable instruction, making it undetectable by existing approaches. We discussed the four basic types of attack modes and provide insight into the relationship between memory alignment size and heap spraying attack surface areas. We verified the technique by a proof-of-concept implementation. Even when the alignment size is 32 bytes, our attack can evade existing detection techniques. As a solution, we propose to enforce finer memory allocation granularity at memory managers of all levels, trading a limited amount performance for better security.

Acknowledgments The authors would like to thank the anonymous reviewers for their valuable comments. This work was supported in part by National Natural Science Foundation of China under the grant No. 61003216, National Development and Reform Commission under the project "A monitoring platform for web safe browsing", and Singapore Ministry of Education under the NUS grant R-252-000-367-133.

8. REFERENCES

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

[2] The PaX team. http://pax.grsecurity.net.

[3] Why is address space allocation granularity 64k? http://blogs.msdn.com/oldnewthing/archive/2003/10/08/55239.aspx.

[4] Microsoft Internet Explorer .ANI file "anjh" header BoF exploit, 2004. http://skypher.com/wiki/index.php?title=www.edup.tudelft.nl/~bjwever/details_msie_ani.html.php.

[5] Microsoft Internet Explorer DHTML object handling valuerabilities (MS05-20), 2004. http://skypher.com/wiki/index.php?title=www.edup.tudelft.nl/~bjwever/advisory_msie_R6025.html.php.

[6] Microsoft Internet Explorer IFRAME src&name parameter BoF remote compromise, 2004. http://skypher.com/wiki/index.php?title=www.edup.tudelft.nl/~bjwever/advisory_iframe.html.php.

[7] Microsoft Internet Explorer javaprxy.dll COM object vulnerability, 2005. http://www.frsirt.com/english/advisories/2005/0935.

[8] Microsoft Internet Explorer "msdds.dll" remote code execution, 2005. http://www.frsirt.com/english/advisories/2005/1450.

[9] libemu - shellcode detection, 2007. http://libemu.carnivore.it.

[10] Pwn2own 2010, 2010. http://dvlabs.tippingpoint.com/blog/2010/02/15/pwn2own-2010.

[11] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis. STRIDE: Polymorphic sled detection through instruction sequence analysis. In *Security and Privacy in the Age of Ubiquitous Computing*, 2005.

[12] C. Anley, J. Heasman, F. Lindner, and G. Richarte. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley, 2004.

[13] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceeding of 12th USENIX Security Symposium*, 2003.

[14] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of 14th USENIX Security Symposium*, 2005.

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

[16] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song. RICH: Automatically protecting against integer-based vulnerabilities. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS)*, 2007.

[17] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1998.

[18] CVE, 2007. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0038.

[19] M. Daniel, J. Honoroff, and C. Miller. Engineering heap overflow exploits with JavaScript. In *Proceedings of the 2nd USENIX Workshop on Offensive Technologies*, 2008.

[20] T. Detristan, T. Ulenspiegel, and Yann_malcom. Polymorphic shellcode engine using spectrum analysis. *Phrack 11,57-15 (2001).*

[21] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browser against drive-by downloads: Mitigating heap-srpaying code injection attacks. In *Proceedings of the 6th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2009.

[22] M. E.Russinovich and D. A.solomon. *Microsoft Wndows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows Xp, and Windows 2000*. Microsoft Press, 2008.

[23] J. Evans. A scalable concurrent malloc(3) implementation for freebsd. In *BSDCan conference*, 2006.

[24] P. Fogla and W. Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, 2006.

[25] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Softw. Pract. Exper.*, 20(1):5–12, 1990.

[26] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *ACSAC'06: Proceedings of the 22th Annual Computer Security Applications Conference*, 2006.

[27] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, 2003.

[28] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[29] M. Polychronakis, K. Anagnostakis, and E. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[30] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the 3rd Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2006.

[31] I. V. Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, Berkeley, CA, USA, 2007.

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

[33] J. Richter and C. Nasarre. *Windows via C/C++ 5th edition*. Microsoft Press, 2008.

[34] RIX. Writing ia32 alphanumeric shellcodes. *Phrack 11,57-15 (2001).*

[35] P. M. Sanjay Ghemawat, 2005. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[36] SecurityFocus. Mozilla Firefox 3.5 'TraceMonkey' component remote code execution vulnerability, 2009. http://www.securityfocus.com/bid/35660.

[37] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo. On the infeasibility of modeling polymorphic shellcode. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007.

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

[39] A. Sotirov. Bypassing browser memory protections in windows vista. In *Blackhat, USA*, 2008.

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

[41] N. Stojanovski, M. Gusev, D. Gligoroski, and Svein.J.Knapskog. Bypassing data execution prevention on microsoftwindows xp sp2. In *The Second International Conference on Availability, Reliability and Security (ARES)*, 2007.

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

[43] 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 (NDSS)*, 2009.

[44] O. Whitehouse. An analysis of address space layout randomization on windows vista™. In *Symantec Advanced Threat Research*, 2007.

[45] Y. Younan, W. Joosen, and F. Piessens. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical Report CW386, Department of Computer Science, Katholieke Universiteit Leuven, 2004.

[46] A. Young and M. Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 129, Washington, DC, USA, 1996. IEEE Computer Society.

[47] J. Zhuge, T. Holz, C. Song, J. Guo, X. Han, and W. Zou. Studying malicious websites and the underground economy on the chinese web. In *Proceedings of the 7th Workshop on the Economics of Information Security (WEIS'08)*, 2008.