# BuBBle: A Javascript Engine Level Countermeasure against Heap-Spraying Attacks

Francesco Gadaleta, Yves Younan, and Wouter Joosen

IBBT-Distrinet, Katholieke Universiteit Leuven, 3001, Leuven Belgium*

**Abstract.** Web browsers that support a safe language such as Javascript are becoming a platform of great interest for security attacks. One such attack is a heap-spraying attack: a new kind of attack that combines the notoriously hard to reliably exploit heap-based buffer overflow with the use of an in-browser scripting language for improved reliability. A typical heap-spraying attack allocates a high number of objects containing the attacker's code on the heap, dramatically increasing the probability that the contents of one of these objects is executed. In this paper we present a lightweight approach that makes heap-spraying attacks in Javascript significantly harder. Our prototype, which is implemented in Firefox, has a negligible performance and memory overhead while effectively protecting against heap-spraying attacks.

**Keywords:** heap-spraying, buffer overflow, memory corruption attacks, browser security.

## 1 Introduction

Web browsing has become an very important part of today's computer use. Companies like Google™and Yahoo are evidence of this trend since they offer full-fledged software inside the browser. This has resulted in a very rich environment within the browser that can be used by web programmers. However, this rich environment has also lead to numerous security problems such as cross site scripting and cross site request forgeries (CSRF). The browser is often written in C or C++, which exposes it to various vulnerabilities that can occur in programs written in these languages, such as buffer overflows, dangling pointer references, format string vulnerabilities, etc. The most often exploited type of C vulnerability is the stack-based buffer overflow. In this attack, an attacker exploits a buffer overflow in an array, writing past the bounds of the memory allocated for the array, overwriting subsequent memory locations. If the attackers are able to overwrite the return address or a different type of code pointer (such

* This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science policy, the Research Fund K.U.Leuven, the Interdisciplinary Institute for Broadband Technology and the European Science Foundation (MINEMA network).

F. Massacci, D. Wallach, and N. Zannone (Eds.): ESSoS 2010, LNCS 5965, pp. 1–17, 2010.
© Springer-Verlag Berlin Heidelberg 2010

as a function pointer), they can gain control over the program's execution flow, possibly redirecting it to their injected code. While stack-based buffer overflows are still an important vulnerability in C programs, they have become harder to exploit due to the application of many different countermeasures such as StackGuard [30], ProPolice [14], ASLR [5], etc. The goal of these countermeasures is to protect areas of potential interest for attackers from being modified or to prevent attackers from guessing where their injected code is located, thus preventing them from directing control flow to that location after they have overwritten such an area of potential interest. Attackers have subsequently focussed on different types of vulnerabilities. One important type of vulnerability is the heap-based buffer overflow. However, due to the changing nature of the heap, these vulnerabilities are notoriously hard to exploit. Especially in browsers, where the heap can look completely different depending on which and how many sites the user has visited, it is hard for an attacker to figure out where in the heap space his overflow has occurred. This makes it hard for attackers to figure out where their injected code is located. The application of countermeasures like ASLR (Address Space Layout Randomization) on the heap has made it even harder to reliably exploit these vulnerabilities. ASLR is a technique by which positions of key data areas in a process's address space, such as the heap, the stack or libraries, are arranged at random positions in memory. All attacks based on the knowledge of target addresses (e.g. *return-to-libc* attacks in the case of randomized libraries or attacks that execute injected *shellcode* in the case of a randomized heap/stack) may fail if the attacker cannot guess the exact target address. Recently a new attack emerged that combines the rich environment found in the browser to facilitate exploits of C vulnerabilities, sometimes resulting in the successful bypass of countermeasures like ASLR that are supposed to protect against these type of vulnerabilities. Heap-spraying attacks use the Javascript engine in the browser to replicate the code they want executed a large amount of times inside the heap memory, dramatically increasing the possibility that a particular memory location in the heap will contain their code. Several examples of heap-spraying attacks have already affected widely used web browsers like Safari, Internet Explorer and Mozilla Firefox.

This paper presents an approach that protects against heap-spraying attacks based on the observation that the attack relies on the fact that the heap will contain homogenous data inserted by the attacker. By introducing diversity in the heap at random locations and by modifying the way that Javascript stores data on the heap at these locations, we can build an effective protection against these exploits at low cost. We implemented this countermeasure in the Javascript engine of Firefox, Tracemonkey[1] The overhead of our approach is very low, measuring the overhead of the countermeasure on a number of popular websites which use a significant amount of Javascript, showed that our approach has an average overhead of 5%. The rest of the paper is organized as follows: Section 2 discusses the problem of heap-based buffer overflows and heap-spraying in more detail, while Section 3.1 discusses our approach and Section 3.2 our prototype implementation. Section 4 evaluates our prototype implementation, while Section 5 compares our approach to related work. Section 6 concludes.

[1] In the remainder of the paper we will refer to Spidermonkey, since Tracemonkey is based on the former engine to which it adds native-code compilation, resulting in a massive speed increase in loops and repeated code.

## 2 Problem Description

### 2.1 Heap-Based Buffer Overflows

The main goal of a heap-spraying attack is to inject malicious code somewhere in memory and jump to that code to trigger the attack. Because a memory corruption is required, heap-spraying attacks are considered a special case of heap-based attacks. Exploitable vulnerabilities for such attacks normally deal with dynamically allocated memory. A general way of exploiting a heap-based buffer overflow is to overwrite management information the memory allocator stores with the data. Memory allocators allocate memory in chunks. These chunks are located in a doubly linked list and contain memory management information (chunkinfo) and real data (chunkdata). Many different allocators can be attacked by overwriting the chunkinfo.

Since the heap memory area is less predictable than the stack it would be difficult to predict the memory address to jump to execute the injected code.Some countermeasures have contributed to making these vulnerabilities even harder to exploit [35,13].

### 2.2 Heap-Spraying Attacks

An effective countermeasure against attacks on heap-based buffer overflow is Address Space Layout Randomization (ASLR) [5]. ASLR is a technique which randomly arranges the positions of key areas in a process's address space. This would prevent the attacker from easily predicting target addresses. However, attackers have developed more effective strategies that can bypass these countermeasures. Heap spraying [27] is a technique that will increase the probability to land on the desired memory address even if the target application is protected by ASLR. Heap spraying is performed by populating the heap with a large number of objects containing the attacker's injected code. The act of spraying simplifies the attack and increases its likelihood of success. This strategy has been widely used by attackers to compromise security of web browsers, making attacks to the heap more reliable than in the past [4,8,22,25,32] while opening the door to bypassing countermeasures like ASLR.

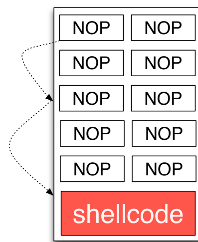


**Fig. 1.** NOP sled and shellcode appended to the sequence

A heap-spraying attack attempts to increase the probability to jump to the injected code (`shellcode`). To achieve this, a basic block of NOP[2] instructions is created. The

[2] Short for No Operation Performed, is an assembly language instruction that effectively does nothing at all [17].

size of this block is increased by appending the block's contents to itself, building the so called *NOP sled*. Finally `shellcode` is appended to it. A jump to any location within the *NOP sled* will transfer control to the `shellcode` appended at the end. The bigger the sled the higher the probability to land in it and the attack to succeed. A schema of a NOP sled with the shellcode appended to it is provided in Fig.1. The second phase of the attack consists of populating the heap of the browser with a high number of these objects, by using the legal constructs provided by the scripting language supported by the web browser. Figure 2 shows the schema of a heap-spraying attack during heap population. Although in this paper we will refer to heap-spraying the memory space of a web browser, this exploit can be used to spray the heap of any process that allows the user to allocate objects in memory. For instance Adobe Reader has been affected by a heap-spraying vulnerability, by which malicious PDF files can be used to execute arbitrary code [26]. Moreover, heap-spraying is considered an unusual security exploit since the action of spraying the heap is considered legal and permitted by the application. In our specific scenario of a heap-spraying attack in a web browser, memory allocation may be the normal behavior of benign web pages. A web site using AJAX (Asynchronous Javascript And XML) technology, such as facebook.com or plain Javascript such as economist.com, ebay.com, yahoo.com and many others, would seem to spray the heap with a large number of objects during their regular operation. Since a countermeasure should not prevent an application from allocating memory, heap-spraying detection is a hard problem to solve.

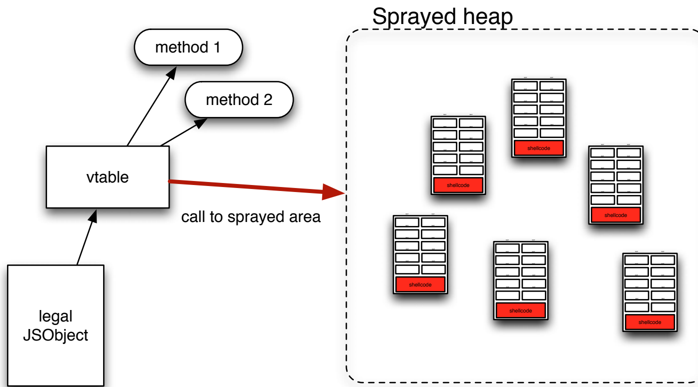


**Fig. 2.** A heap-spraying attack: heap is populated of a large number of $NOP-shellcode$ objects. The attack may be triggered by a memory corruption. This could potentially allow the attacker to jump to an arbitrary address in memory. The attack relies on the chance that the jump will land inside one of the malicious objects.

Because the structure of the heap depends on how often the application has allocated and freed memory before the spraying attack, it would be difficult to trigger it without knowing how contents have been arranged. This would reduce the attack to a guess of the correct address the malicious object has been injected to. But by using a client-side scripting language, such as Javascript, it is also possible to create the ideal

circumstances for such an attack and arrange the heap to get the structure desired by the attacker, as described in [28,11].

Fig. 3 shows an example of a typical heap-spraying attack in Javascript.

```
1. var sled;
2. var spraycnt = new Array();
3. sled = <NOP_instruction>;
4. while(sled.length < _size_)
5.     {
6.       sled+=sled;
7.     }
8. for(i=0; i< _large_; i++)
9.     {
10.      spraycnt[i] = sled+shellcode;
11.    }
```

**Fig. 3.** A Javascript code snippet to perform a basic heap-spraying attack usually embedded in a HTML web page

# 3 BuBBle: Protection against Heap-Spraying

In this section we describe our approach to prevent the execution of shellcode appended to a NOP sled, when a heap-spraying attack and a memory corruption have occured. Our general approach is described in Section 3.1 and our implementation is discussed in Section 3.2.

## 3.1 Approach

An important property of a heap-spraying attack is that it relies on homogeneity of memory. This means that it expects large parts of memory to contain the same information (i.e., it's nop-shellcode). It also relies on the fact that landing anywhere in the nopsled will cause the shellcode to be executed. Our countermeasure breaks that assumption by introducing diversity on the heap, which makes it much harder to perform a heap-spraying attack. The assumption is broken by inserting special interrupting values in strings at random positions when the string is stored in memory and removing them when the string is used by the application. These special interrupting values will cause the program to generate an exception when it is executed as an instruction. Because these special values interrupt the strings inside the memory of the application, the attacker can no longer depend on the nopsled or even the shellcode being intact. If these values were placed at fixed locations, the attacker could attempt to bypass the code by inserting jumps over specific possible locations within the code. Such an attack, however is unlikely, because the attacker does not know exactly where inside the shellcode control has been transferred. However, to make the attack even harder, the special interrupting values are placed at random locations inside the string. Since an attacker does not know at which locations in the string the special interrupting values are stored,

he can not jump over them in his nop-shellcode. This lightweight approach thus makes heap-spraying attacks significantly harder at very low cost.

We have implemented this concept in the Javascript engine of Firefox, an opensource web browser. The internal representation of Javascript strings was changed in order to add the interrupting values to the contents when in memory and remove them properly whenever the string variable is used or when its value is read. Interruption is regulated by a parameter which can be chosen at browser build time. We have chosen this to be $25$ bytes, which is the smallest useful shellcode we found in the wild [33]. This parameter will set the interval at which to insert the special interrupting values. Given the length $n$ of the string to transform $i$ ($i = \lceil \frac{n}{25} \rceil$) intervals are generated (with $n > 25$). A random value is selected for each interval. These numbers will represent the positions within the string to modify. The parameter sets the size of each interval, thus the number of positions that will be modified per string. By choosing a lower value for the parameter the amount of special interrupting values that are inserted will be increased. Setting the size of each interval to the length of the smallest shellcode does not guarantee that the positions will be at distance of $25$ bytes. It may occur that a position $p$ is randomly selected from the beginning of its interval $i_p$ and the next position $q$ from the end of its interval $i_q$. In this case $(q - p)$ could be greater than $25$, allowing the smallest shellcode to be stored in between. However heap-spraying attacks are based on large amounts of homogeneous data, not simply on inserting shellcode. Thus being able to insert this shellcode will not simply allow an attacker to bypass this approach. When the characters at random positions are changed, a support data structure is filled with *metadata* to keep track of original values and where in the string they are stored. The modified string is then stored in memory. Whenever the string variable is used, the engine will perform an inverse function, to restore the string to its original value to the caller. This task is achieved by reading the metadata from the data structure bound to the current Javascript string and replacing the special interrupting values with their original values on a copy of the contents of the string. With this approach different strings can be randomized differently, giving the attacker even less chances to figure out the locations of the special values in the string. Because each string variable stays modified as long as it is stored in memory and a copy of this string variable is only restored to its original value when the application requests access to that a string. When the function processing the string stores the result back to memory, the new string is again processed by our countermeasure. If the function discards the string, it will simply be freed. Moreover the Javascript engine considered here implements strings as immutable type. This means that string operations do not modify the original value. Instead, a new string with the requested modification is returned.

### 3.2 Implementation

In this section we discuss the implementation details of our countermeasure. It has been implemented on Mozilla Firefox (Ver. 3.7 Beta 3) [15], a widely used web browser and its ECMA-262-3-compliant engine, Tracemonkey (Ver. 1.8.2)[18].

An attacker performing a heap-spraying attack attempts to arrange a contiguous block of values of his choice in memory. This is required to build a sled that would not be interrupted by other data. To achieve this, a monolithical data structure is required.

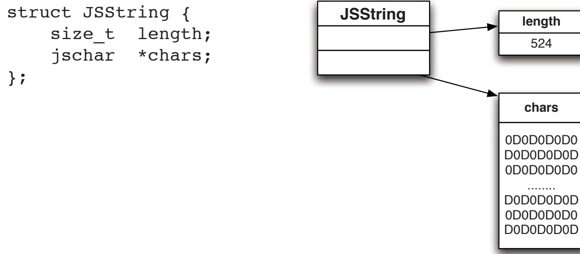


**Fig. 4.** Spidermonkey's JSString type is considered a threat for a heap-spraying attack since member *chars* is a pointer to a vector of size $(length + 1)$* *sizeof(jschar)*

Javascript offers several possibilities to allocate blocks in memory. The types supported by Spidermonkey are numbers, objects and strings. An overview about how the Javascript engine represents Javascript objects in memory is given in [19].

The string type represents a threat and can be used to perform a potentially dangerous heap-spraying. Figure 4 depicts what a `JSString`, looks like. It is a data structure composed of two members: the `length` member, an integer representing the length of the string and the `chars` member which points to a vector having byte size `(length + 1) * sizeof(jschar)`. When a string is created, `chars` will be filled with the real sequence of characters, representing that contiguous block of memory that the attacker can use as a sled. We have instrumented the `JSString` data structure with the fields needed for BuBBle to store the metadata: a flag $transformed$ will be set to 1 if the character sequence has been transformed and an array $rndpos$ is used to store the random positions of the characters that have been modified within the sequence.

Our countermeasure will save the original value of the modified character to $rndpos$, change its value (at this point the string can be stored in memory) and will restore the original value back from $rndpos$ whenever the string is read.

This task is performed respectively by two functions: `js_Transform(JSString*)` and `js_Restore(JSString*)`. The value of the character to modify is changed to the 1-byte value `0xCC`. This is the assembly language instruction for x86 processors to

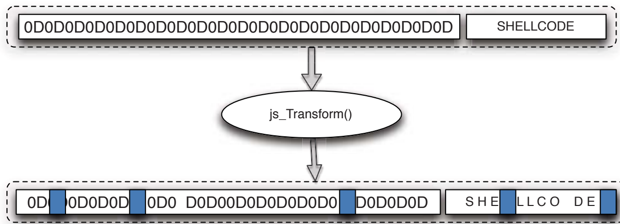


**Fig. 5.** Representation of the transformed string in memory: characters at random positions are changed to special interrupting values. The potential execution of the object's contents on the heap would be interrupted by the special value.

| | 0 | 1 | 2 | 3 | 4 | 5 | | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|
| rndpos = | 4 | "0" | 28 | "D" | 52 | "0" | ... | 507 | "E" |

**Fig. 6.** How metadata is stored to array $rndpos$: index $i$ within the array contains the value of the position in the string; index $(i + 1)$ contains its original value

generating a software breakpoint. If a heap-spraying attack was successfully triggered and the byte `0xCC` at a random position was executed, an interrupt handler is called to detect and report the attack. The web browser could expose an alert popup to encourage the user close the application and notify the browser vendor of the detected issue. The number of characters to randomize depends on the $degree$ parameter. This parameter was chosen based on the length of the smallest shellcode found (to date, 25 bytes long[3]), but can be tuned to select the level of security and the overhead that will be introduced by the countermeasure. If $size$ is the length of the string to transform, the number of intervals is given by $\lceil \frac{size}{24} \rceil$. A random value for each interval will be the position of the character that will be changed. For performance reasons we generate $50$ random values in a range between $(0, 24)$ at program startup and use these values as offsets to add to the first index of each interval to compute the random position within that interval. The random values are regenerated whenever the Garbage Collector reclaims memory. This prevents the attacker from learning the values over time as they may already have changed. The value of the $i^{th}$ random position is stored at $rndpos[2i]$, while the original value of the $i^{th}$ character is stored at $rndpos[2i+1]$ (Fig. 6). Function `js_Transform(str)` will use the values stored in the str$\rightarrow rndpos[]$ array to restore the string to its original value.

# 4 Evaluation

In Section 4.1 we discuss our performance overhead, while in Section 4.2 we report an analytical study of the memory overhead in the worst case[4].

All benchmarks were performed on an Intel Core 2 Duo 2Ghz, 4GB RAM, running Debian Gnu/Linux.

## 4.1 Performance Benchmarks

To measure the performance overhead of BuBBle we performed two types of benchmarks. We collected results of macrobenchmarks on 8 popular websites and accurate timings of microbenchmarks running SunSpider, Peacekeeper Javascript Benchmark, and V8 Benchmark Suite.

**Macrobenchmarks:** To collect timings of BuBBle's overhead in a real life scenario, we run a performance test similar to the one used to measure the overhead of Nozzle [23]. We downloaded and instrumented the HTML pages of eight popular web sites

[3] The smallest setuid and execve shellcode for GNU/Linux Intel x86 to date can be found at `http://www.shell-storm.org/shellcode/files/shellcode-43.php`

[4] With *worst case* we mean the case where it is guaranteed that the smallest shellcode cannot be stored on the heap without being interrupted by random bytes.

by adding the Javascript `newDate()` routine at the beginning and the end of the page, and computed the delta between the two values. This represents the time it takes to load the page and execute the Javascript. Since the browser caches the contents of the web page, that value will be close to how long it takes to execute the Javascript. We then ran the benchmark 20 times for each site, 10 times with BuBBle disabled and 10 times with BuBBle enabled. Table 1 shows that the average performance overhead over these websites is 4.8%.

**Table 1.** Performance overhead of BuBBle in action on 8 popular web sites

| Site URL | Load (ms) | Load(ms) BuBBle | Perf. overh. |
|---|---|---|---|
| economist.com | 17304 | 18273 | +5.6% |
| amazon.com | 11866 | 12423 | +4.7% |
| ebay.com | 7295 | 7601 | +4.2% |
| facebook.com | 8739 | 9167 | +4.9% |
| maps.google.com | 15098 | 15581 | +3.2% |
| docs.google.com | 426 | 453 | +6.3% |
| cnn.com | 12873 | 13490 | +4.8% |
| youtube.com | 12953 | 13585 | +4.9% |
| **Average** | | | **+4.82** |

**Microbenchmarks:** Microbenchmarks, which allow us to better assess the overheads introduced by BuBBle in different situations were also performed. These microbenchmarks were performed by running three different benchmarks: the SunSpider Javascript Benchmarks [31], the Peacekeeper benchmarks [9] and the V8 benchmarking suite [16].

*SunSpider:* SunSpider is used by Mozilla Firefox to benchmark the core Javascript language only, without the DOM or other browser dependent APIs. The tests are divided into multiple domains: testing things such as 3D calculations, math, string operations, etc.. Table 2 contains the runtime in milliseconds of running the various benchmarks that are part of SunSpider. The results for each domain are achieved by performing a number of subtests. However for most domains the overhead of the subsets is close to 0%. Thus, to save space in Table 2, we have removed the results of the subtests and simply included the results of the domain (which is sum of all the subtests). However, because we modify the way strings are represented in memory and do a number of transformations on strings, we have included the subtests which test the performance of string operations.

The results in Table 2 show that the overhead for BuBBle in areas other than string manipulation are negligible. The overheads for string operations on the other hand vary from 3% to 27%. This higher overhead of 27% for base64 is due to the way the base64 test is written: the program encodes a string to base64 and stores the result. When the program starts, it generates a character by adding getting a random number, multiplying the number by 25 and adding 97. This character is converted to a string and added to an existing string. This is done until a string of 8192 characters is created. Then to do the encoding, it will loop over every 3rd character in a string and then perform the

**Table 2.** Microbenchmarks performed by SunSpider Javascript Benchmark Suite

| Test | | Runtime(ms) | BuBBle Runtime (ms) | Perf. overh. |
|---|---|---|---|---|
| 3d | | 568.6ms +/- 1.4% | 569.0ms +/- 1.2% | **+0.17%** |
| bitops | | 66.4ms +/- 1.8% | 67ms +/- 1.8% | **+0.89%** |
| controlflow | | 13.8ms +/- 1.9% | 14.0ms +/- 1.6% | **+1.44%** |
| math | | 63.2ms +/- 1.0% | 63.6ms +/- 1.7% | **+0.62%** |
| regexp | | 84.2ms +/- 2.0% | 84.4ms +/- 2.9% | **+0.23%** |
| string | | | | |
| | base64 | 74.8ms +/- 2.9% | 102.2ms +/- 1.9% | +27.3% |
| | fasta | 280.0ms +/- 1.5% | 283.4ms +/- 0.7% | +1.24% |
| | tagcloud | 293.2ms +/- 2.6% | 299.6ms +/- 0.8% | +2.20% |
| | unpack-code | 352.0ms +/- 0.8% | 363.8ms +/- 3.1% | +3.24% |
| | validate-input | 119.8ms +/- 2.4% | 132.2ms +/- 1.0% | +9.30% |
| | | 1119.8ms +/- 0.9% | 1181.2ms +/- 1.0% | **+5.19%** |

encoding of those three characters to 4 base64 encoded characters. In every iteration of the loop, it will do 7 accesses to a specific character in the original string, 4 access to a string which contains the valid base64 accesses and finally it will do 4 += operations on the result string. Given that our countermeasure will need to transform and restore the string multiple times, this causes a noticeable slowdown in this application.

**Table 3.** Peacekeeper Javascript Benchmarks results (the higher the better)

| Benchmark | Score | BuBBle Score | Perf. overh. |
|---|---|---|---|
| Rendering | 1929 | 1919 | +0.5% |
| Social Networking | 1843 | 1834 | +0.5% |
| Complex graphics | 4320 | 4228 | +2.2% |
| Data | 2047 | 1760 | +14.0% |
| DOM operations | 1429 | 1426 | +0.2% |
| Text parsing | 1321 | 1298 | +2.0% |
| **Total score** | 1682 | 1635 | **+2.8** |

*Peacekeeper:* Peacekeeper is currently used to tune Mozilla Firefox. It will assign a score based on the number of operations performed per second. The results of the Peacekeeper benchmark are located in Table 3: for most tests in this benchmark, the overhead is negligible, except for the Data test which has an overhead of 14%. The Data test is a test which will do all kinds of operations on an array containing numbers and one test which performs operations on an array containing strings of all possible countries in the world. The operations on the strings containing all possible countries are what contribute to the slow down in this benchmark: whenever a country is used, the string is restored, whenever one is modified the resulting new string is transformed.

*V8:* The V8 Benchmark Suite is used to tune V8, the Javascript engine of Google Chrome. The scores are relative to a reference system (100) and as with Peacekeeper,

**Table 4.** V8 Benchmark Suite results (the higher the better)

| Benchmark | Score | BuBBle Score | Perf. overh. |
|---|---|---|---|
| Richards | 151 | 143 | +5.6% |
| DeltaBlue | 173 | 167 | +3.6% |
| Crypto | 110 | 99.6 | +10.4% |
| Ray Trace | 196 | 193 | +1.5% |
| EarlyBoyer | 251 | 242 | +3.7% |
| RegExp | 174 | 173 | +0.6% |
| Splay | 510 | 501 | +1.8% |
| **Total score** | 198 | 193 | **+2.6** |

the higher the score, the better. Again, most overheads are negligible except for Crypto, which has an overhead of 10.4%. Crypto is a test encrypts a string with RSA. To encrypt this string the application does a significant number of string operations, resulting in transformation and restoration occurring quite often.

These benchmarks show that for string intensive javascript applications that do little else besides run string operations, the overhead can be significant, but not a show stopper. In all other cases the overhead was negligible.

### 4.2 Memory Overhead

This section discusses the memory overhead of our countermeasure. This is done by providing both an analytical description of our worst case scenario and providing a measurement of the memory overheads that the benchmarks incur.

**Theoretical memory overhead:** An analytical study of memory usage has been conducted in the case of our highest level of security. This is achieved when we want to prevent the execution of the smallest shellcode by changing at least one character every 24 bytes. If $s$ is the lenght of the smallest shellcode, the `js_Transform()` function will change the value of a random character every $(s-k)$ bytes, $k = 1...(s-1)$. In a real life scenario $k = 1$ is sufficient to guarantee a lack of space for the smallest shellcode. If the length of the original string is $n$ bytes, the number of positions to transform will be $i = \lceil \frac{n}{s} \rceil$. The array used to store the position and the original value of the transform character will be $2i$ bytes long.

*Memory usage: a numerical example* Given the following data:

```
-----------------------------------------------------
original string length:  n = 1 MB = 1.048.576 bytes
smallest shellcode length: s = 25 bytes
injected sequence length:  r = 1  byte
-----------------------------------------------------
```

The number of positions will be $i = \lceil \frac{1MB}{s} \rceil = 43691$ and the memory overhead for the worst case will be $8.3\%$.

**Table 5.** Memory overhead of BuBBle in action on three Javascript benchmarks suites

| Benchmark | Mem. usage (MB) | BuBBle mem. usage (MB) | Mem. overh. |
|---|---|---|---|
| Sunspider | 88 | 93 | +5.6% |
| V8 | 219 | 229 | +4.2% |
| Peacekeeper | 148 | 157 | +6.5% |
| **Average** | | | **+5.3%** |

**Memory overhead for the benchmarks:** Table 5 contains measurements of the maximum memory in megabyte that the benchmarks used during their runtime. These values were measured by starting up the browser, running the benchmarks to completion and then examining the *VmHWM* entry in $/proc/ < pid > /status$. This entry contains the peak resident set size which is the maximum amount of RAM the program has used during its lifetime. Our tests were run with swap turned off, so this is equal to the actual maximum memory usage of the browser. These measurements show that the overhead is significantly less than the theoretical maximum overhead.

### 4.3 Security Evaluation

In this section we give a security evaluation of BuBBle. When running the Javascript snippet of Fig.3 we are able to spray the heap in all cases: spraying means allocating memory and this is not considered an action to be detected. However, when attempting to execute the contents of sprayed objects, by a memory corruption, the attack will fail. The instruction pointer landed within a sled will execute the byte instruction `0xCC` at a random position. This 1-byte instruction will call the interrupt procedure and execution will be halted. The execution of the `0xCC` sequence is sufficient to consider the system under attack and to detect an unexpected execution. In fact, a legal access would purge the string of the `0xCC` sequence. A drawback of our countermeasure is that a heap-spraying attack can still be performed by using a language other than Javascript such as Java or C#. However, the design in itself gives a reasonably strong security guarantee against heap-spraying attacks to be implemented for other browser supported languages. Another way to store malicious objects to the heap of the browser would be by loading images or media directly from the Internet. But this would generate a considerable amount of traffic and loading time, making the attack clearly observable.[5]

## 5 Related Work

Several countermeasures have been designed and implemented to specifically protect against heap-based attacks. Others have been designed to prevent memory corruption in general. We provide an overview of some countermeasures against heap overflow attacks in Section 5.2. A description of some countermeasures specifically designed to protect against heap-spraying attacks in web browsers is provided in Section 5.1.

[5] Heap spraying by content download might generate a traffic of hundreds of MBs. We are confident that also a broadband internet access would make the attack observable.

### 5.1 Heap-Spraying Defences

**Nozzle:** Nozzle is the first countermeasure specifically designed against heap-spraying attacks to web browsers [23]. It uses emulation techniques to detect the presence of malicious objects. This is achieved by the analysis of the contents of any object allocated by the Web browser. The countermeasure is in fact implemented at memory allocator level. This has the benefit of protecting against a heap-spraying attack by any scripting language supported by the browser. Each block on the heap is disassembled and a control flow graph of the decoded instructions is built. A `NOP-shellcode` object may be easily detected by this approach because one basic block in the control flow graph will be reachable by several directions (other basic blocks). For each object on the heap a measure of the likelihood of landing within the same object is computed. This measure is called `attack surface area`. The surface area for the entire heap is given by the accumulation of the surface area of individual blocks. This metric reflects the overall heap *health*. This countermeasure is more compatible than DEP and would help to detect and report heap-spraying attacks by handling exceptions, without just crashing. This approach has although some limitations. Because Nozzle examines objects only at specific times, this may lead to the so called TOCTOU-vulnerability (Time-Of-Check-Time-Of-Use). This means that an attacker can allocate a benign object, wait for Nozzle to examine it, then change it to contain malicious content and trigger the attack. Moreover Nozzle examines only a subset of the heap, for performance reasons. But this approach will lead to a lower level of security. The performance overhead of Nozzle examining the whole heap is unacceptable. Another limitation of Nozzle is the assumption that a heap-spraying attack allocates a relatively small number of large objects. A design based on this assumption would not protect against a heap-spraying which allocates a large number of small objects which will have the same probability to succeed.

**Shellcode detection:** Another countermeasure specifically designed against heap-spraying attacks to web browsers is proposed by [12]. This countermeasure is based on the same assumptions that (1) a heap-spraying attack may be conducted by a special crafted HTML page instrumented with Javascript and (2) Javascript strings are the only way to allocate contiguous data on the heap. Thus all strings allocated by the Javascript interpreter are monitored and checked for the presence of shellcode. All checks have to be performed before a vulnerability can be abused to change the execution control flow of the application. If the system detects the presence of shellcode, the execution of the script is stopped. Shellcode detection is performed by *libemu*, a small library written in C that offers basic x86 emulation. Since *libemu* uses a number of heuristics to discriminate random instructions from actual shellcode, false positives may still occur. Moreover an optimized version of the countermeasure that achieves accurate detection with no false positives is affected by a significant performance penalty of 170%.

### 5.2 Alternative Countermeasures

**Probabilistic countermeasures:** Many countermeasures make use of randomness when protecting against attacks. Canary-based countermeasures [21,24] use a secret

random number that is stored before an important memory location: if the random number has changed after some operations have been performed, then an attack has been detected. Memory-obfuscation countermeasures [6,10] encrypt (usually with XOR) important memory locations or other information using random numbers. Memory layout randomizers [5,7,34] randomize the layout of memory: by loading the stack and heap at random addresses and by placing random gaps between objects. Instruction set randomizers [3] encrypt the instructions while in memory and will decrypt them before execution. While these approaches are often efficient, they rely on keeping memory locations secret. However, programs that contain buffer overflows could also contain "buffer overreads" (e.g. a string which is copied via *strncpy* but not explicitly null-terminated could leak information) or other vulnerabilities like format string vulnerabilities, which allow attackers to print out memory locations. Such memory leaking vulnerabilities could allow attackers to bypass this type of countermeasure. Another drawback of these countermeasures is that, while they can be effective against remote attackers, they are easier to bypass locally, because attackers could attempt brute force attacks on the secrets.

**DEP:** Data Execution Prevention [29] is a countermeasure to prevent the execution of code in memory pages. It is implemented either in software or hardware, via the NX bit. With DEP enabled, pages will be marked non-executable and this will prevent the attacker from executing shellcode injected on the stack or the heap of the application. If an application attempts to execute code from a page marked by DEP, an access violation exception will be raised. This will lead to a crash, if not properly handled. Unfortunately several applications attempt to execute code from memory pages. The deployment of DEP is less straightforward due to compatibility issues raised by several programs [2].

**Separation and replication of information:** Countermeasures that rely on separation or replication of information will try to replicate valuable control-flow information [36,37] or will separate this information from regular data. This makes it harder for an attacker to overwrite this information using an overflow. Some countermeasures will simply copy the return address from the stack to a separate stack and will compare it to or replace the return addresses on the regular stack before returning from a function. These countermeasures are easily bypassed using indirect pointer overwriting where an attacker overwrites a different memory location instead of the return address by using a pointer on the stack. More advanced techniques try to separate all control-flow data (like return addresses and pointers) from regular data, making it harder for an attacker to use an overflow to overwrite this type of data. While these techniques can efficiently protect against buffer overflows that try to overwrite control-flow information, they do not protect against attacks where an attacker controls an integer that is used as an offset from a pointer, nor do they protect against non-control-data attacks.

**Execution monitors:** In this section we describe two countermeasures that monitor the execution of a program and prevent transferring control-flow which could be unsafe. Program shepherding [20] is a technique that monitors the execution of a program and will disallow control-flow transfers[6] that are not considered safe. An example of

[6] Such a control flow transfer occurs when e.g., a *call* or *ret* instruction is executed.

a use for shepherding is to enforce return instructions to only return to the instruction after the call site. The proposed implementation of this countermeasure is done using a runtime binary interpreter. As a result, the performance impact of this countermeasure is significant for some programs, but acceptable for others. Control-flow integrity [1] determines a program's control flow graph beforehand and ensures that the program adheres to it. It does this by assigning a unique ID to each possible control flow destination of a control flow transfer. Before transferring control flow to such a destination, the ID of the destination is compared to the expected ID, and if they are equal, the program proceeds as normal. This approach, while strong and in the same efficiency range as our approach, does not protect against non-control data attacks.

## 6 Conclusion

Heap-spraying attacks expect to have large parts of the heap which are homogenous. By introducing heterogeneity where attackers expect this homogeneity, we can make heap-based buffer overflows a lot harder. By modifying the way the strings are stored in memory in the Javascript engine, we can achieve an effective countermeasure that introduces this heterogeneity. This is done by inserting special values at random locations in the string, which will cause a breakpoint exception to occur if they are executed. If an attacker tries to perform a heap-spraying attack, his injected code will now have been interrupted at a random location with such a breakpoint exception, allowing the browser to detect and report that an attack has occurred. Benchmarks show that this countermeasure has a negligible overhead both in terms of performance and memory overhead.

## References

1. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security, Alexandria, Virginia, U.S.A., November 2005, pp. 340–353. ACM, New York (2005)
2. Anisimov, A.: Defeating microsoft windows xp sp2 heap protection and dep bypass, `http://www.ptsecurity.com`
3. Barrantes, E.G., Ackley, D.H., Forrest, S., Palmer, T.S., Stefanović, D., Zovi, D.D.: Randomized instruction set emulation to disrupt binary code injection attacks. In: Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003), Washington, D.C., U.S.A., October 2003, pp. 281–289. ACM, New York (2003)
4. Berry-Bryne, S.: Firefox 3.5 heap spray exploit (2009), `http://www.milw0rm.com/exploits/9181`
5. Bhatkar, S., Duvarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: Proceedings of the 12th USENIX Security Symposium, Washington, D.C., U.S.A., August 2003, pp. 105–120. USENIX Association (2003)
6. Bhatkar, S., Sekar, R.: Data space randomization. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 1–22. Springer, Heidelberg (2008)
7. Bhatkar, S., Sekar, R., DuVarney, D.C.: Efficient techniques for comprehensive protection from memory error exploits. In: 14th USENIX Security Symposium, Baltimore, MD, August 2005, USENIX Association (2005)

8. Blog, M.A.L.: New backdoor attacks using pdf documents (2009), http://www.avertlabs.com/research/blog/index.php/2009/02/19/new-backdoor-attacks-using-pdf-documents/
9. Futuremark Corporation. Peacekeeper The Browser Benchmark, http://service.futuremark.com/peacekeeper/
10. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: PointGuard: protecting pointers from buffer overflow vulnerabilities. In: Proceedings of the 12th USENIX Security Symposium, Washington, D.C., U.S.A., August 2003, pp. 91–104. USENIX Association (2003)
11. Daniel, M., Honoroff, J., Miller, C.: Engineering heap overflow exploits with javascript. In: WOOT 2008: Proceedings of the 2nd conference on USENIX Workshop on offensive technologies, Berkeley, CA, USA, pp. 1–6. USENIX Association (2008)
12. Egele, M., Wurzinger, P., Kruegel, C., Kirda, E.: Defending browsers against drive-by downloads: mitigating heap-spraying code injection attacks. In: Flegel, U., Bruschi, D. (eds.) DIMVA 2009. LNCS, vol. 5587, pp. 88–106. Springer, Heidelberg (2009)
13. Erlingsson, Ú.: Low-level software security: Attacks and defenses. Technical Report MSR-TR-2007-153, Microsoft Research (November 2007)
14. Etoh, H., Yoda, K.: Protecting from stack-smashing attacks. Technical report, IBM Research Divison, Tokyo Research Laboratory (June 2000)
15. Mozilla Foundation. Firefox 3.5b4 (2009), http://developer.mozilla.org
16. Google. V8 Benchmark Suite - version 5, http://v8.googlecode.com
17. Intel. Intel architecture software developer's manual. vol. 2: Instruction set reference (2002)
18. E. C. M. A. International. ECMA-262: ECMAScript Language Specification. ECMA (European Association for Standardizing Information and Communication Systems), 3rd edn., Geneva, Switzerland (December 1999)
19. Jorendorff: Anatomy of a javascript object (2008), http://blog.mozilla.com/jorendorff/2008/11/17/anatomy-of-a-javascript-object
20. Kiriansky, V., Bruening, D., Amarasinghe, S.: Secure execution via program shepherding. In: Proceedings of the 11th USENIX Security Symposium, San Francisco, California, U.S.A., August 2002, USENIX Association (2002)
21. Krennmair, A.: ContraPolice: a libc extension for protecting applications from heap-smashing attacks (November 2003)
22. FireEye Malware Intelligence Lab. Heap spraying with actionscript (2009), http://blog.fireeye.com/research/2009/07/actionscript_heap_spray.html
23. Ratanaworabhan, P., Livshits, B., Zorn, B.: Nozzle: A defense against heap-spraying code injection attacks. Technical report, Microsoft Research (November 2008)
24. Robertson, W., Kruegel, C., Mutz, D., Valeur, F.: Run-time detection of heap-based overflows. In: Proceedings of the 17th Large Installation Systems Administrators Conference, San Diego, California, U.S.A., October 2003, pp. 51–60. USENIX Association (2003)
25. securiteam.com. Heap spraying: Exploiting internet explorer vml 0-day xp sp2 (2009), http://blogs.securiteam.com/index.php/archives/641
26. Securitylab. Adobe reader 0-day critical vulnerability exploited in the wild, cve-2009-0658 (2009), http://en.securitylab.ru/nvd/368655.php
27. skypher.com. Heap spraying (2007), http://skypher.com/wiki/index.php
28. Sotirov, A.: Heap feng shui in javascript (2007)
29. TMS. Data execution prevention, http://technet.microsoft.com/en-us/library/cc738483.aspx
30. Wagle, P., Cowan, C.: Stackguard: Simple stack smash protection for gcc. In: Proceedings of the GCC Developers Summit, Ottawa, Ontario, Canada, May 2003, pp. 243–256 (2003)

31. www2.webkit.org Sunspider javascript benchmark (2009), http://www2.webkit.org/perf/sunspider-0.9/sunspider.html
32. www.milw0rm.com Safari (arguments) array integer overflow poc (new heap spray) (2009), http://www.milw0rm.com/exploits/7673
33. www.packetstormsecurity.org 25bytes-execve (2009), http://www.packetstormsecurity.org/shellcode/25bytes-execve.txt
34. Xu, J., Kalbarczyk, Z., Iyer, R.K.: Transparent runtime randomization for security. In: 22nd International Symposium on Reliable Distributed Systems (SRDS 2003), Florence, Italy, October 2003, pp. 260–269. IEEE Computer Society, IEEE Press, Los Alamitos (2003)
35. Younan, Y., Joosen, W., Piessens, F.: Code injection in C and C++: A survey of vulnerabilities and countermeasures. Technical report, Departement Computerwetenschappen, Katholieke Universiteit Leuven (2004)
36. Younan, Y., Joosen, W., Piessens, F.: Efficient protection against heap-based buffer overflows without resorting to magic. In: Ning, P., Qing, S., Li, N. (eds.) ICICS 2006. LNCS, vol. 4307, pp. 379–398. Springer, Heidelberg (2006)
37. Younan, Y., Pozza, D., Piessens, F., Joosen, W.: Extended protection against stack smashing attacks without performance loss. In: Proceedings of the Twenty-Second Annual Computer Security Applications Conference (ACSAC 2006), pp. 429–438. IEEE Press, Los Alamitos (2006)