ChaffyScript: Vulnerability-Agnostic Defense of JavaScript Exploits via Memory Perturbation

Xunchao Hu[1,2], Brian Testa[2], and Heng Yin[3]

[1] DeepBits Technology LLC
[2] Syracuse University
[3] University of California, Riverside
bptesta@syr.edu, xchu@deepbitstech.com, heng@cs.ucr.edu

Abstract. JavaScript has been used to exploit binary vulnerabilities of host software that are otherwise difficult to exploit; they impose a severe threat to computer security. Although software vendors have deployed techniques like ASLR, sandbox, etc. to mitigate JavaScript exploits, hacking contests (e.g.,Pwn2Own, GeekPwn) have demonstrated that the latest software (e.g., Chrome, IE, Edge, Safari) can still be exploited. An ideal JavaScript exploit mitigation solution should be flexible and allow for deployment without requiring code changes. To this end, we propose ChaffyScript, a vulnerability-agnostic mitigation system that thwarts JavaScript exploits via undermining the memory preparation stage of exploits. We implement a prototype of ChaffyScript, and our evaluation shows that it defeats the 11 latest JavaScript exploits with minimal runtime and memory overhead. It incurs at most 5.88% runtime overhead for chrome and 12.96% for FireFox. The maximal memory overhead JS heap usage, observed using the Octane benchmark, was 8.2%. To demonstrate the deployment flexibility of ChaffyScript, we have integrated it into a web proxy.

1 Introduction

JavaScript has been a popular programing language for decades. It has been widely deployed in web browsers, servers, PDF processor, etc. JavaScript exploits, which take advantage of JavaScript's interactive features to exploit binary vulnerabilities (e.g., use-after-free, heap/buffer overflow) of host software and inject malicious code into victim's machine, have been imposing a severe threat to computer security due to the rise of exploit kits [36] and the sheer number of code execution vulnerabilities [8] reported in host software every year.

To mitigate JavaScript exploits, software vendors have deployed many mitigation techniques like Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP), control flow guard [7], sandbox [54], EMET [11], etc. These mitigation techniques increase the bar for exploitation. As a result, attackers have to combine complex memory preparation, memory disclosure, code reuse and other techniques to launch a successful exploit.

While these exploit mitigation techniques are constantly improving, hacking contests like Pwn2Own [19] and GeekPwn [12], consistently demonstrate that the latest versions of Chrome, Safari, Internet Explorer, and Edge can still be exploited. There are at least two reasons for this. First, most of the latest proposed mitigation techniques require changes on software or compiler tool chain and thus could not be deployed

promptly. For instance, ASLR-guard [43] is designed to thwart information disclosure attacks, but requires compiler changes and cannot be quickly deployed by software vendors. Second, the deployed mitigation techniques may fail due to newly invented exploitation techniques (e.g., sandbox bypass technique). We argue that *an ideal mitigation technique should be flexible to deploy without requiring code changes and should subvert inevitable exploitation stage(s).*

We observe that a typical JavaScript exploit adopts *memory preparation* to manipulate the memory states. This is an essential step for JavaScript exploits since attackers need to put deliberately chosen content (e.g., ROP chain, shellcode) into a known memory location prior to execution of that code. This offers an opportunity for defenders to stop the exploits by disabling this *memory preparation* step.

In this paper, we propose ChaffyScript, a vulnerability-agnostic mitigation system that *blocks JavaScript exploits via undermining the memory preparation stage.* Specifically, given suspicious JavaScript, ChaffyScript rewrites the code to insert chaff code, which aims to perturb the memory state while preserving the semantics of the original code. JavaScript exploits will inevitably fail as a result of unexpected memory states introduced by chaff code, while the benign JavaScript still behaves as expected since the memory perturbation code does not change the JavaScript's original semantics.

Compared with current mitigation techniques, ChaffyScript has the following advantages: (1) it requires no changes to the host software and thus is easily deployable; (2) it does not introduce false positive and can defeat zero-day (or previously unseen) attacks; (3) the implementation of ChaffyScript is simple and robust; and (4) it incurs low runtime and memory overhead, and can be deployed for on-line defense.

We have implemented a prototype of ChaffyScript, which consists of three main components: a memory allocation/de-allocation discovery module to identify the potential memory preparation operations, a lightweight type inference module to prune the unnecessary memory preparation candidates, and a chaff code generation module to insert chaff code alongside memory preparation operations. As a demonstration of the deployment flexibility afforded by our approach, we have integrated ChaffyScript into a web proxy to protect users against malicious HTML files. Our evaluation results show that: 1) the probability of guessing the correct memory states after ChaffyScript is extremely low (Section 6.1), 2) ChaffyScript can thwart the latest JavaScript exploits effectively (Section 6.2) and 3) it incurs runtime overhead 5.88% for chrome and 12.96% for FireFox at most, and the memory overhead is 6.1% for the minimal JS heap usage, and 8.2% for the maximal JS heap usage during runtime on Octane benchmark [14] (Section 6.3).

In summary, the contributions of this work are:

- We made a key observation that memory preparation is an essential stage in JavaScript exploits.
- Based on this observation, we proposed and designed ChaffyScript to disable the memory perturbation stage by inserting chaff code.
- We have implemented a prototype system, ChaffyScript, and our evaluation shows that ChaffyScript incurs acceptable runtime performance overhead and memory overhead on Octane benchmark and defeat all of the 11 JavaScript exploits.

– We demonstrated the deployment flexibility via integrating ChaffyScript into a web proxy to protect users against malicious HTML files.

2 Technical background and motivation

2.1 JavaScript Exploits

The interactive nature of JavaScript allows malicious JavaScript to take advantage of binary vulnerabilities (e.g., use-after-free, heap/buffer overflow) that are otherwise difficult to exploit. JavaScript provides attackers with a much easier means to conduct heap spraying [32], information leakage [49], and shellcode generation.

Figure 1 illustrates the high-level stages of JavaScript exploits [20]. In the pre-exploitation stage, malware fingerprints the victim machine to determine the OS version and target software and then launches the corresponding exploit. The exploitation stage triggers the vulnerability, bypassing exploit mitigation techniques (e.g., ASLR, EMET [11], and Control Flow Guard [7]) and diverts the code execution to the injected payload. The post-exploitation stage executes a Return-Oriented-Programming (ROP) payload to bypass DEP, drops the malicious payload while attempting to evade detection from endpoint security products.

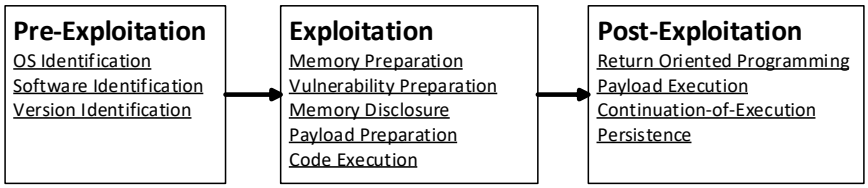

Fig. 1: The overall exploitation stages.

Defense against these exploits has evolved to react to the advances in exploitation techniques. Any defensive techniques that stop one of the exploitation stages can prevent the exploits from infecting victim machines. Some exemplar defenses include:

1) Cloaking the OS/software version during the pre-exploitation stage to stop attackers from launching the correct exploits.
2) Tools like BrowserShield [47] instrument the execution of JavaScript to match the predefined vulnerability feature and block the execution once a match is found.
3) Randomization-based techniques like Instruction Set Randomization [38] introduce uncertainty in the target.
4) Tools like ROPecker [28] exploit the Last Branch Record hardware feature to detect the execution of ROP chains.
5) Control Flow Integrity (CFI) [23] based techniques [52] are used to prevent the execution of injected payloads.

While these exploitation mitigation techniques are constantly improving, hacking contests like Pwn2Own [19] and GeekPwn [12], consistently demonstrate that the latest versions of Chrome, Safari, Internet Explorer, and Edge can still be exploited. The

reasons are two-fold: (1) most of the latest proposed mitigation techniques require software or compiler tool chain changes which may cause compatibility issues and thus cannot be deployed promptly; and (2) the deployed mitigation techniques may fail due to newly invented exploitation techniques. For instance, DEP mitigation can be defeated by ROP attacks. The JITSpray [50] makes the ROP defense useless since ROP is not needed anymore to bypass DEP in JITSpray based attack.

2.2 Memory Preparation

Based on our observations, the *memory preparation* stage is a critical stage for exploits.

Memory Management of the JavaScript Engine Before discussing memory preparation techniques, we first take V8 [6] as an example to present an overview of memory management within a JavaScript engine. JavaScript engines dynamically manage memory for running applications so that developers do not need to worry about memory management like coding in C/C++. V8 divides the heap into several different spaces: a young generation, an old generation, and a large object space. The young generation is divided into two contiguous spaces, called semi-space. The old generation is separated into a map space and an old object space. The map space exclusively contains all map objects while the rest of old objects go into the old space.

Each space is composed of a set of pages. A page is a contiguous chunk of memory, allocated from the operating system with system call (e.g., `mmap`). Pages are always 1 MB in size and 1 MB aligned, except in a separate large object space. This separate space stores objects larger than `Page::kMaxHeapObjectSize`, so that theses large objects are not moved during garbage collection process.

The allocated objects will be put into different spaces based on their size, type, and age. Garbage collection process is responsible for 1) scavenging young generation space by moving live objects to the other semi-space when semi-space becomes full; 2) major collection of the whole heap to free unreferenced objects and aggressive memory compaction to clean up fragmented memory. The other JavaScript engines like SpiderMonkey [21] and ChakraCore [5], JavaScriptCore [15] share the similar design of memory management. Attackers commonly abuse the memory management features to manipulate the memory states, known as memory preparation.

Memory Preparation Techniques Memory allocation and free operations are used to manipulate the memory. We can categorize these techniques based upon how they change the memory layout:

1) Emit data in a target address. This is usually implemented with a heap spray technique like Heap Fengshui [51] and its successors [3]. These techniques spray crafted objects into the heap. The size and type of the objects are carefully chosen to exploit the memory management of JavaScript engine. ❶
2) Emit adjacent objects. This is implemented by allocating two objects with the same type and size sequentially so the JavaScript engine will likely keep them adjacent in the heap; this technique is widely used by attackers. ❷

3) Create holes in memory. This is implemented by allocating adjacent objects first, then freeing one of them. A hole is created among those adjacent objects. ❸
4) De-fragment the heap. This is usually implemented via calling a garbage collection API provided by host software (e.g., `CollectGarbage()` in IE) or via a carefully crafted JavaScript snippet that forces the garbage collection process as discussed in Section 4.1.❹

In theory, attackers can use any JavaScript types to prepare memory. However, in practice `String` and `Array` are the best candidates because the implementations of these two types, especially `Typed Array`, are very close to native string/array in C/C++, making it easy for attackers to control them in memory. The implementation of other primitive types (`Boolean, Null, Number, Symbol`) and objects (e.g., `Math`) are quite different. They are stored either as references or complex tree structures in memory, and thus it is difficult to precisely manipulate them in memory. Realizing that memory preparation is an essential step towards successful JavaScript exploit, a natural question rises in our mind: "How can we disrupt this stage without changing the code of the host software?" The answer is: *memory perturbation*.

2.3 Memory Perturbation Techniques

Table 1: The overview of memory perturbation techniques.

Category	Approaches	Memory Change	Code Trans. Overhead
Storage	a. Split Variables	1, 2	High
	b. Change Encoding	1	High
	c. Promote scalars to objects	1, 2	High
	d. convert static data to procedure	1, 2	Medium
Aggregation	e. Merge scalar variables	1, 2	High
	f. Split, merge, fold, flatten arrays	2	High
Ordering	g. Reorder instance variables	2	Medium
	h. Reorder arrays	2	High
Inserting	i. Insert noise data allocation/free	2	Low
	j. Insert holes into arrays	2	High

1: content change 2: layout change

At a high level, memory perturbation stands for the semantically-equivalent transformation of an existing program, such that the transformed program exhibits a different and unpredictable memory layout. By nature, this technique shares some similarity with the data obfuscation technique used in code obfuscation [29] to defeat reverse engineering attacks because both of them transform the program without changing the program semantics. In the context of JavaScript-based exploit defense, our goal is to subvert the memory preparation stage through memory perturbation, so that exploits are defeated at runtime due to unpredictable memory states. While data obfuscation can make very aggressive obfuscation, memory perturbation technique in ChaffyScript has to be lightweight for performance concern. To the best of our knowledge, we are the first to propose lightweight memory perturbation technique for JavaScript exploit defense.

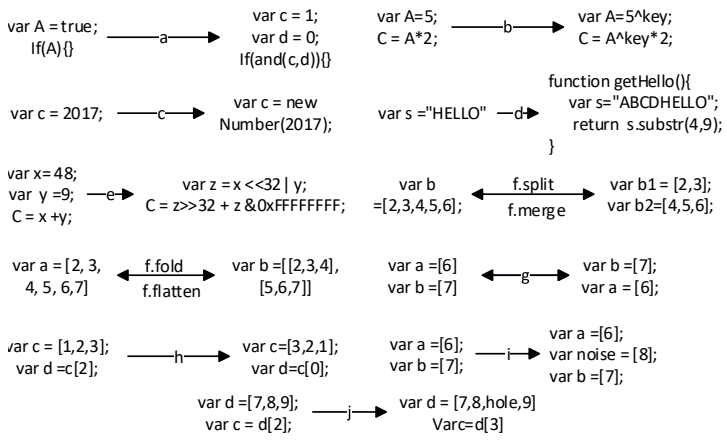

Fig. 2: Samples of memory perturbation techniques summarized in Table 1.

Table 1 provides an overview of memory perturbation techniques. In general, these techniques can be divided into 4 categories as affecting the storage, aggregation, ordering or inserting of the data in memory. Figure 2 presents sample code snippets for each of the approaches referenced in Table 1. While each of the approaches can induce similar memory changes regarding memory layout or content, the overhead associated with each of these approaches varies. For instance, Approach *i* only needs one statement insertion operation for the code transformation. It does not need any further program analysis to keep the semantics intact since the inserted statement does not affect the original code's data and control flow. However, Approach *f* requires additional program analysis to keep the program semantics intact. This is because after the array is restructured, a whole program def-use analysis has to be conducted to identify all the affected code and then update the code accordingly (update array index, array name, etc.). Column D of Table 1 presents the code transformation overhead for each approach.

2.4 Our Mitigation Solution

Based upon the previous discussion, we propose a vulnerability-agnostic defense approach for JavaScript exploits, ChaffyScript. Specifically, given suspicious JavaScript, ChaffyScript rewrites the code to insert chaff code to perturb memory states at runtime, while preserving the semantics of the original JavaScript code.

At first glance, our technique seems very similar to the diversification techniques summarized in [40], which also introduce randomness to stop exploits. These techniques diversify the host software at instruction level, basic block level, function level, program level, and system level, and require modifications to the source code or binary code of the host software. On the contrary, ChaffyScript diversifies the input (JavaScript is the input of the host software (e.g., Chrome, IE, etc.)) to stop exploits, and no code changes on the host software are required. Therefore, the deployment of ChaffyScript is *effortless*.

To summarize, compared with current mitigation techniques, ChaffyScript has the following advantages:

1) Vulnerability-agnostic nature. ChaffyScript does not rely on any specific vulnerability features as BrowserShield [47] does. Thus it is vulnerability-agnostic and can be used to defend against 0-day attacks.

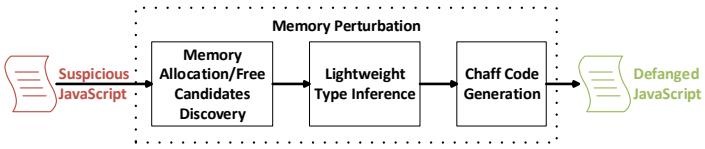

Fig. 3: The overall architecture of ChaffyScript.

2) Flexible deployment. JavaScript rewriting can be implemented without the change of host software (e.g., Chrome and IE). This makes the deployment of ChaffyScript very flexible. Users can disable or enable ChaffyScript promptly based upon their needs.
3) Stronger protection. As evaluated in Section 6.1, ChaffyScript provides much stronger protection than randomization-based approaches (e.g., ASLRGuard [43]).

In the following sections, we will elaborate details on the threat model and scope, design, and implementation of ChaffyScript.

3 Threat Model and Scope

To make sure our solution is practical, we define our threat model based on strong attack assumptions. We assume a commodity operating system with standard defense mechanisms, such as no-executable stack and heap, and ASLR. We assume attackers are remote, so they do not have physical access to the target system, nor do they have prior control over other local programs before a successful exploit.

We assume that the attacker use JavaScript to exploit the memory corruption vulnerabilities (use-after-free, memory disclosure, etc.), manipulate the memory layout, and divert the control flow to execute arbitrary code of his/her choice. These exploitations include but are not limited to control data attack, non-control data attack [55], and side channel attack [35]. With the deployed standard defense mechanisms, it is reasonable to assume that the attacker uses memory preparation to manipulate the memory layout since the bypass of the standard defense mechanisms needs precise memory layout preparation. This is true even when the attacker can exploit memory disclosure vulnerabilities because 1) most of the memory disclosure vulnerabilities needs precise memory preparation to trigger; 2) even in an extreme case that the attacker may read memory without memory preparation, the success of the other stages of JavaScript exploit (payload preparation, code execution, etc.) still rely on precise memory preparation. We impose no restrictions on the exploitation techniques.

Out-of-scope threats. Cross-site scripting (XSS) [39] and Cross-site forgery(CSRF) [25] are out of our scope since they do not target at memory corruption vulnerabilities.

4 Design

In theory, all of the memory perturbation techniques discussed in Section 2.3 could be used to sabotage the memory preparation stage in JavaScript exploits. In practice, an on-line defense approach should incur minimal code transformation overhead and thus cannot afford complex program analysis. With that in mind, we only apply Approach *i* in ChaffyScript for memory perturbation, which is good enough for defeating JavaScript exploits as demonstrated in Section 6. Specifically, ChaffyScript conducts JavaScript rewriting by inserting chaff code to 1) allocate random chunks of memory

along with existing memory allocation operations, and 2) disable memory free operations by adding an additional reference to freed objects. We leave the exploration of the rest memory perturbation techniques as future work.

Figure 3 demonstrates the overview of ChaffyScript. Given a suspicious JavaScript, it first traverses the code to discover memory allocation/de-allocation candidates. Then a lightweight type inference process is conducted on these candidates to identify the interesting memory allocation/free candidates that are usually used for memory preparation by attackers; this reduces unnecessary chaff code insertions and improves runtime performance. Finally, the chaff code is generated and inserted into the original JavaScript code to get a *transformed JavaScript*. At runtime, the chaff code will allocate random memory or disable memory free operations to destroy the memory preparation stage of JavaScript-based exploits. Benign JavaScript still executes normally since the chaff code does not change the original code's expected semantics.

4.1 Memory Allocation/De-Allocation Candidate Discovery

As discussed in Section 2, there are two kinds of operations that affect the memory state: object allocation and de-allocation. ChaffyScript inspects JavaScript code to identify the following memory manipulation candidates:

Memory Allocation Candidates As discussed in Section 2.2, *String* and *Array* are two common data types used by attackers to fill memory. ChaffyScript traverses JavaScript code to identify potential memory allocation candidates for *String* and *Array* operations. However, precise type inference for JavaScript is quite expensive [24]. So, to be simple, the *new* expression (e.g., *var c = new Array(5)*), value initialization expression (e.g., *var c = [3,7]*), and built-in function callsite (e.g., *var c = a.substr(0,10)*) are all considered as memory allocation candidates since they can trigger memory allocation in the heap. Note that the callee name can be dynamically generated in JavaScript. Thus, we cannot statically determine if it is a targeted built-in function callsite. To be complete, the callsites with dynamically generated callees are also considered as memory allocation candidates. In JavaScript *String* objects are immutable, so operations that change *String* objects (e.g., the '+' and '+=' operators) will also cause memory allocation. ChaffyScript also considers expressions with the '+' and '+=' operators as potential memory allocation candidates.

Memory De-allocation Candidates In JavaScript, there are three ways to explicitly free memory: assign *null* to the object, use the *delete* operator and explicitly trigger garbage collection. The first two methods remove the reference to the allocated object. For instance, *delete object.property* removes the reference to the *property* object. However, it does not directly free the *property* object in memory. When the *property* object is no longer referenced by any other objects, garbage collection process will eventually free it in memory. ChaffyScript still considers *null* assignment and *delete* calls as memory de-allocation candidates because attackers often use them to create holes in memory, so objects allocated later can fill these holes to trigger some vulnerabilities (e.g., Use-after-Free).

Explicit garbage collection (GC) calls are usually used by attackers to defragment the heap [32]. Some browsers like Internet Explorer and Opera provide public APIs

(e.g., *CollectGarbage()* for IE) to trigger garbage process. ChaffyScript can easily identify this kind of garbage collection process by matching the API name. However, the other browsers' garbage collection process can only be triggered when certain memory states are reached; in these cases there are no APIs to explicitly trigger the process. In these cases, attackers usually fill objects in memory to trigger the garbage collection process. For instance, the following code can fill up the 1MB semi-space page of V8 engine and force V8 to scavenge *NewSpace*.

```
for(var i=0; i<((1024*1024)/0x10); i++)
{
  var a = new String();
}
```

This kind of GC trigger is implicit and difficult to identify. Nevertheless, it includes memory allocation operations and will be considered as a *memory allocation candidate* and will still be captured by ChaffyScript. GC events using DOM objects instead of *String* and *Array* are not captured by ChaffyScript, but this is only one of the memory preparation techniques used by attackers. Furthermore, JavaScript exploits usually combine multiple memory preparation techniques, thus allowing ChaffyScript to be effective even when hybrid memory preparation methods are used.

The host software also provides APIs to allocate and free objects. For instance, on the browser, users can invoke the DOM API to add or remove node objects from the DOM tree. While in theory, it is possible for attackers to manipulate those APIs during memory preparation, as discussed in Section 2.2, it is challenging to do that since the layout and content of DOM objects are difficult to control. When a new memory preparation technique is discovered, we can update the memory allocation/de-allocation candidate discovery stage in ChaffyScript to block this new attack technique.

4.2 Lightweight Type Inference

The collection of memory allocation candidates is a superset of the memory allocation candidates of *String* and *Array* objects. If we insert chaff code along with all the candidates, the runtime performance would be unacceptable. To improve the runtime performance of rewritten JavaScript, we conduct lightweight type inference to prune the memory allocation candidates that are not related to *String* and *Array* objects. It is executed in two steps: *static type inference* and *dynamic type inference*.

Static type inference ChaffyScript only keeps the memory allocation candidates that operate on variables typed as *String*, *Array*, *ArrayBuffer*, *Int8Array*, *Uint8Array*, *Uint8ClampedArray*, *Int16Array*, *Uint16Array*, *Int32Array*, *Uint32Array*, *Float32Array*, or *Float64Array*. We can infer the types of the variables in expressions statically based upon how they are used. ChaffyScript uses the three following type inference rules:

1) The constructor of the *new* operator indicates the type of created object. For instance, the constructor *Int16Array* in *var b = new Int16Array(256)* indicates the type of *b* is *Int16Array*.

2) The return value of a built-in function indicates the type based on its description. For instance, *var c = s.split("a")* indicates that the type of *c* is *Array*.
3) For expressions with the *'+' or '+='* operators, if the type of the operands is *String*, then the result is also a *String*.

These three simple rules do not require complex program analysis and can be used to efficiently determine the variable types to filter out the memory allocation candidates. If static type inference cannot determine the types of all the variables used in memory allocation candidates, we conduct dynamic type inference to determine the variable types at runtime.

Dynamic type inference Static type inference does not always work for two reasons. First, since a function call's name can be dynamically generated, we cannot determine an object's type based upon the function's return value. Second, the three typing rules can be insufficient to determine the variable types in some cases. For instance, the three typing rules cannot be applied to candidates {`var d = a + b + c`}. It is possible to determine the type of *a, b and c* via backward analysis, but that is likely too expensive for our online defense system use case. Instead, we conduct *dynamic type inference* with the help of certain JavaScript features. In JavaScript, a variable's type can be extracted at runtime using the `instanceof` operator. With this operator, ChaffyScript inserts the dynamic type inference after the memory allocation candidate to check if it operates on one of the targeted types. While dynamic type inference incurs runtime performance overhead, it is less expensive than static type inference.

As a result, the combination of static and dynamic type inference makes it impossible for attackers to bypass our type inference process.

4.3 Chaff Code Generation

The goal of inserted chaff code is to affect the memory states at runtime. It achieves this goal via the following two methods.

Disable object de-allocations For object de-allocations using public APIs like *CollectGarbage()* in IE, ChaffyScript directly removes the API call from the original code. This does not change the semantics of original JavaScript code because garbage collection APIs does not have data or control dependency on the original code.

For object de-allocations using the `delete` operator or assigning a `null` value, the above method does not work because simply removing such code will change the semantics of the original code. The attackers' goal of freeing an object is to create holes in memory, so later allocated objects can occupy the freed memory. If we keep a reference to the object before the free operation is executed, the later allocated object can not occupy the position since the allocated memory still has a reference to it. Figure 4.(a,b) illustrates an example of such a transformation. In code snippet *[a]*, `x` is assigned the value `null`. In the transformed code *[b]*, ChaffyScript has added a new variable `4613335ea9901` to store a reference to `''abcdefgh''`. Although `x` is assigned to `null`, the object `''abcdefgh''` will not be scavenged since `4613335ea9901` keeps a reference to it.

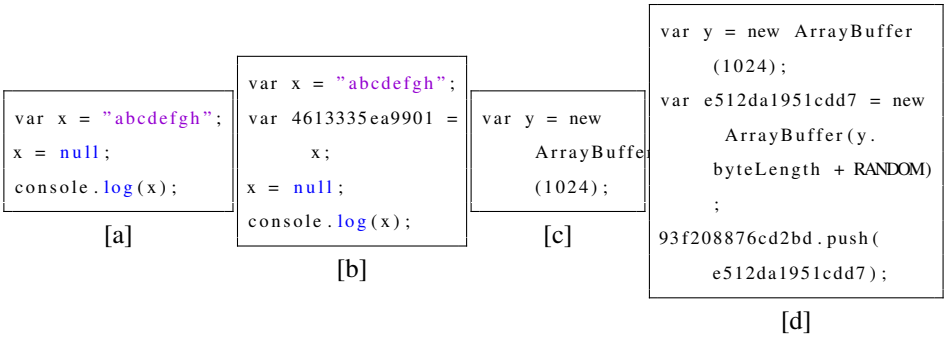

Fig. 4: Chaff code samples.

Insert chaff code after object allocations As discussed in Section 2, if two objects of the same type are allocated sequentially and with the same size, it is likely their positions on the heap are adjacent to each other. Attackers exploit this feature to manipulate the memory layout and content. ChaffyScript is also able to exploit this detail to defeat this exploit. After every memory allocation candidate, ChaffyScript inserts code to allocate additional memory with the same type, but with variable-length padding (*RANDOM*) at the end. Figure 4([c],[d]) presents an example of such code transformation. As you see in code snippet *[d]*, a new *ArrayBuffer* is allocated with size *(y.byteLength + RANDOM)*. With this randomness, it is almost impossible for attackers to guess the combinations of memory states as evaluated in Section 6.1.

To generate *RANDOM*, ChaffyScript provides two approaches to increase an attacker's uncertainty. The first one is at runtime; every time the inserted chaff code is executed, a new random number is generated for *RANDOM*. So if a given piece of chaff code is executed 1000 times, and the range of *RANDOM* is *0-50*, it will create 50^{1000} possible memory states at runtime. The second approach happens when we are inserting the chaff code into original JavaScript code; a random number is generated for *RANDOM*. This value is used to randomly increase the size of the memory chunk allocated by this piece of chaff code. For example, if 15 places are inserted by the chaff code, and the range of *RANDOM* is *0-50*, 50^{15} possible memory states will be created at runtime. Both approaches can defeat the JavaScript exploits with different security guarantees and performance overhead as discussed in Section 6.

The range of *RANDOM* cannot be too big. Otherwise, the objects allocated by chaff code might be allocated in a different location. Thus that fails to break adjacent arrays (❷). In our implementation, we set it as 0-50. This range works against memory preparation techniques while providing enough randomness as demonstrated in Section 6.1.

Since our inserted code is independent of the original code, attackers may abuse garbage collection to scavenge the allocated chaff memory and neutralize the effects of inserted chaff code. To avoid this, ChaffyScript keeps a reference to every allocated piece of chaff memory. This prevents scavenging of chaff memory by the garbage collection process because there is always at least one reference to the allocated chaff memory.

The variable names used in the chaff code are generated randomly. This prevents attackers from identifying memory allocated by the chaff code alongside their memory preparation code. Thus attackers cannot leverage variable naming conventions to

identify memory used as part of our countermeasures, thus making bypass of these countermeasures impossible.

5 Implementation

We implemented ChaffyScript using esprima [10]. It is a JavaScript parser used to generate Abstract Syntax Tree (AST) with full support for ECMAScript 6. Since it is written using JavaScript, it can be easily embedded into different documents like HTML or PDF. This makes the deployment very flexible. We use Estraverse [9] to traverse the AST, discover memory allocation/free candidates and perform lightweight type inference. The chaff code insertion is implemented via directly manipulating the original code with the offset information collected from AST generation process. We do not operate on AST directly to insert the chaff code because generating code from AST is more expensive than directly manipulating the original code. Section 6.3 evaluates the difference of rewriting performance for these two approaches.

The deployment of ChaffyScript is very flexible. It can be deployed as a browser extension, a web proxy, a standalone rewriting engine or one component of a JavaScript engine. In this paper, we demonstrated one deployment approach to protect users against malicious HTML files.

HTML Protector Ideally, it is most user-friendly to deploy ChaffyScript as a browser extension. Unfortunately, JavaScript rewrite, used to implement code transformation in ChaffyScript, is not natively supported in browser extensions. Instead, we deploy ChaffyScript as a web proxy. The downside is that a user needs to install an external program (and certificate) as opposed to only an extension. The benefit is that this proxy-based solution is browser-independent and can be easily deployed with minimal configuration.

We implemented the prototype in *Node.js* [17], using the *http-mitm-proxy* package [13]. ChaffyScript becomes an integral part of the Web proxy. We followed Dachshund [44]'s approach to handle dynamically generated code. Specifically, the dynamic code generation functions (e.g., *eval, SetTimeout, Function, SetInterval*) were hooked via new injected JavaScript code. To rewrite dynamically generated code, we used synchronous *XMLHttpRequest* requests from hooked JavaScript functions to the proxy. The response from the proxy contains the rewritten JavaScript code.

6 Evaluation

In this section, we present the evaluation of ChaffyScript. The evaluation tries to answer the following questions: First, How secure is ChaffyScript's approach in theory, compared to general randomization approaches? Second, how secure is ChaffyScript's approach against practical JavaScript-based exploits? Third, how much overhead does ChaffyScript impose, with respect to chaff code generation, runtime and memory overhead of the resulting chaff code?

Experimental setup. The performance overhead experiment was run under Chrome 57 and Firefox 54. All the experiments are conducted on a test machine equipped with intel Core i7-4790 CPU @ 3.60GHz x 8 with 16GB RAM.

Table 2: Experimental results of 11 latest JavaScript-based exploits using ChaffyScript

CVE	Environment Setup	Memory Preparation	M	N	Defeated?
CVE-2015-2419	IE11/32bit win7	❶ ❷ ❹	28	12594	Y
CVE-2015-1233	Chrome 41.0.2272.118/win1032bit	❶ ❷ ❹	9	8194	Y
CVE-2015-6086	IE11/32bit win7	❸ ❹	12	1280	Y
CVE-2015-6764	Chrome 46.0.2490.0/win10 32bit	❶ ❷ ❹	14	393224	Y
CVE-2016-9079	FireFox 50.0.1 32bit /Windows8.1	❶ ❷ ❹ (JITSpray)	13	20564	Y
		❶ ❷ ❹	28	17408	Y
CVE-2016-3202 (ms16-063)	IE11/Win7 32bit	❶ ❷ ❸ ❹	12	110005	Y
CVE-2016-1646	Chrome 46.0.2490.0/win10 32bit	❶ ❷ ❹	18	393226	Y
V8 OOB write	Chrome 60.0.3080.5 Linux14.04 64bit	❷	10	10	Y
X360_videoPlayerActiveX	IE10/VideoPlayerActiveX 2.6, win7 64bit	❶ ❷	9	352258	Y
CVE-2017-5400	Firefox 51.0.1 32bit/Win10 64bit	❶ ❷ ❹	7	2702	Y

M: # of inserted chaff code N: Executed times # of chaff code at runtime

6.1 Security Analysis

In this subsection, we present an analysis to determine the probability that an attacker could predict the memory layout after memory perturbation is introduced by ChaffyScript. The randomness introduced by ChaffyScript is determined by the following parameters:

1) *RANDOM* - the size variation range of created object by chaff code.
2) *M* - number of inserted chaff code.
3) *N* - executed times of chaff code at runtime.

The probability of guessing the correct memory states is defined as the following equation.

$$probability = \begin{cases} RANDOM^{-M} & predefine\ RANDOM \\ RANDOM^{-N} & Gen\,RANDOM\,at\,runtime \end{cases}$$

If *RANDOM* is predefined when ChaffyScript inserts the chaff code, the *probability* is $RANDOM^{-M}$. If a random number is generated for *RANDOM* every time the chaff code is executed, the *probability* is $RANDOM^{-N}$. If it is too big, the allocated objects by chaff code may not be adjacent to the objects allocated by original code. Thus it cannot break memory preparation ❷. In our implementation, we set *RANDOM* as 50 and it worked well on defeating JavaScript exploits as evaluated in Section 6.2.

The values of *M* and *N* vary case by case. Column 4 and Column 5 in Table 2 record the values of *M* and *N* for 11 exploits. The average of *M* was 15, and the average of *N* was 130876. The *probability* for JavaScript exploits in our implementation should be 50^{-15} and $50^{-130876}$. This provides much stronger randomness than ASLR [43] (2^{-64} at most). The highest *probability* of the 11 exploits was 50^{-10} which is still stronger than 2^{-56}. Through this analysis, we conclude that the probability for an attacker to predict the memory layout is extremely low after memory perturbation is introduced by ChaffyScript.

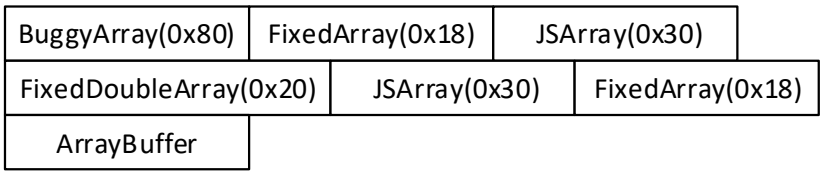

Fig. 5: Expected memory layout of sample chromev8_OOB_write

6.2 Effectiveness

Although in theory ChaffyScript can stop JavaScript exploits, we wanted to know how well it performed at defeating real JavaScript exploits without the knowledge of the targeted vulnerabilities. We went over browser-related PoCs in recent years from public resources like metasploit and technical blogs, collected around 40 of them since 2015, and eventually managed to successfully set up exploitation environments for 11 of them. This is our best effort, due to the scarcity of public exploits resources. These 11 exploits are representative of JavaScript exploits because:

1) The vulnerabilities targeted by these 11 exploits are quite new (from 2015 to 2017).
2) The target host software of these exploits covered the most popular web browsers (IE11, Chrome, and Firefox)
3) These 11 exploits used all of the memory preparation techniques presented in Column 3 in Table 2.
4) These 11 exploits covered the popular exploitation techniques - JITSpray and HeapSpray.
5) These 11 exploits not only targeted at vulnerabilities in host software, but also in the browser plugin (X360 _videoPlayerActiveX).

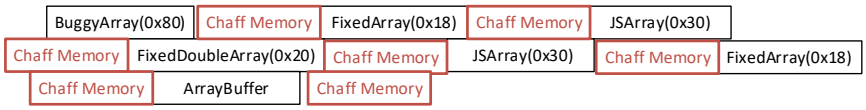

Fig. 6: Memory layout of sample chromev8_OOB_write after rewritten by ChaffyScript

For each exploit, we manually confirmed whether it could be stopped by ChaffyScript. Column 6 in Table 2 presents the result. As shown in the results, ChaffyScript defeated all 11 of the exploits without requiring knowledge of vulnerabilities targeted. This experiment demonstrated that ChaffyScript can effectively defeat JavaScript exploits without knowledge of the targeted vulnerability.

Note that the chaff codes inserted into the sample *chromev8_OOB_write* are only executed 10 times, but still thwart the exploit. This is not like the other samples, in which the chaff codes are executed thousands of times and change the memory states substantially. In fact, the memory preparation of this exploit expects the memory layout as shown in Figure 5. After the chaff code is inserted, the actual memory layout is close to the layout as shown in Figure 6. The chaff memory breaks adjacent array layout(❷).

Table 3: Rewriting Performance on well-known JavaScript libraries

JS library	jquery.mobile 1.4.2	angular 1.2.5	react 0.13.3
Modify Code string	99 ms	99 ms	100 ms
Modify AST	174ms	169 ms	155 ms

Therefore, the memory preparation of this exploit fails and this exploit is thwarted eventually. This case further demonstrates the effectiveness of memory perturbation used in ChaffyScript since it uses very few memory perturbation operations.

6.3 Performance

Rewriting Overhead In order to evaluate the rewriting overhead of ChaffyScript, we chose to measure the three popular and large JavaScript libraries - JQuery (mobile-1.4.2), AngularJS (1.2.5), and React (0.13.3). These libraries are commonly embedded in web pages and relatively large compared with other JavaScript applications (JQuery has 443KB, AngularJS has 702KB, React has 587KB). For the evaluation, we rewrote these libraries using ChaffyScript 1000 times. We measured the time required to rewrite these libraries, including all the steps required to generate chaff JavaScript.

We tested two code transformation approaches. The first approach modified the code directly based upon the offset information collected by the JavaScript parser. The second approach modified code within the AST. As demonstrated in Table 3. The time spent by the second approach is 1.67 times more than the first approach. ChaffyScript chose the first approach in the implementation. On average, It took $\tilde{1}$00ms to rewrite JQuery, AngularJS, and React. Note that rewriting is a one-time effort and we can further optimize performance by rewriting multiple scripts in parallel.

Table 4: Overall Overhead of ChaffyScript on Octane benchmark

Cofiguration	Chrome	Firefox
a. Runtime Generated *RANDOM* + GC escaper	5.88%	12.96%
b. Runtime Generated *RANDOM*	4.54%	7.98%
c. Predefined *RANDOM* + GC escaper	5.68%	11.27%
d. Predefined *RANDOM*	4.53%	6.60%

Runtime Overhead Next, we evaluated the runtime performance that is incurred on the client side due to the modified JavaScript code. We leverage Octane, a commonly-used benchmark for JavaScript engines [14]. For the evaluation, we ran the Octane benchmarks 5 times and used the mean scores as the final results.

Table 4 summarizes the overall overhead on the Octane benchmark under different ChaffyScript configurations. With the strongest protection, ChaffyScript incurs 5.88% overhead in Chrome, and 12.96% in FireFox. With the weakest protection, ChaffyScript incurs 4.53% in Chrome and 6.60% in Firefox. As discussed in Section 6.1, the weakest protection can still provide an acceptable randomness strength. Note that our threat model is only relevant to non-trusted and attacker-controlled JavaScript. Thus the overhead of popular JavaScript libraries can be eliminated by whitelisting trusted scripts. This overhead after the whitelisting should allow ChaffyScript to be deployed online to protect users against JavaScript exploits.

Table 5: Memory Overhead of Chrome on Octane benchmark

Usage	Original	a	b	c	d
Min(MB)	34.4	36.5 (6.10%)	36.5 (6.10%)	36.5 (6.10%)	36.4 (5.81%)
Max(MB)	609	659 (8.2%)	656 (7.71%)	640 (5.09%)	638 (4.76%)

a, b, c , d refers to the four ChaffyScript configurations described in Table 4

Memory Overhead In theory, the memory overhead should be around 2 times at most. This is because along with each object allocation, ChaffyScript could allocate another object with a similar size to disturb memory states. If all of the inserted objects by chaff code are not freed finally by Garbage Collector, the memory usage of the defanged JavaScript would be up to 2 times of the original JavaScript.

To evaluate the actual memory overhead, we ran Octane on Chrome and recorded the memory usage of JavaScript heap. Table 5 summarizes the results. *Min* refers to the observed minimal memory usage of JavaScript heap during the running of Octane, while *Max* refers to the observed maximal memory usage. As demonstrated in the table, for all four different ChaffyScript configurations, the memory overhead never exceeded 8.2%. This is not a big overhead since RAM has become very cheap and current personal computers are usually equipped with at least 8GB memory. Thus, ChaffyScript can be deployed by users without requiring upgraded hardware.

7 Discussion

Limitations of ChaffyScript First, attackers may find methods to bypass the JavaScript rewriting process. For instance, lexer confusing attacks [16] confuse the lexer causing executable code to be interpreted as the content of strings or comments, allowing an attacker to slip arbitrary unsafe code past a rewriter or verifier. The rewriter of ChaffyScript is vulnerable to this attack. In the future, we would like to adopt JaTE's [53] approach by considering all formats of JavaScript comments to gain resilience to this attack. It is also possible to attack the JavaScript parser *esprima* with crafted JavaScript. As a result, the code transformation cannot be finished. Fortunately, ChaffyScript can identify such attacks because it can detect *esprima* errors. ChaffyScript can alert security researchers for further analysis once such failures are observed.

Second, the JavaScript extraction approach used in deployment may undermine ChaffyScript. Attackers may hide JavaScript in an unusual way to escape from the extraction, thus preventing ChaffyScript from rewriting those portions. For instance, attackers may abuse PDF parsers to hide malicious JavaScript code [26]. This is not a ChaffyScript issue, but rather is a JavaScript extractor issue. Deployment of a state-of-art JavaScript extractor with ChaffyScript would reduce the risk of such attacks.

Third, ChaffyScript does not work on hybrid JavaScript exploits. Basically, such kind of exploits use JavaScript to trigger the vulnerability, and use other script language (e.g., ActionScript in Flash) to prepare the memory. This is quite common in recently years since vector-related rehabilitates in Flash are quite exploit-friendly, allowing construction of arbitrary memory read/write primitives [4]. However, it is possible to deploy the techniques used in ChaffyScript on ActionScript to stop such attacks as discussed in the following subsection.

Fourth, although extremely difficult, attackers may be able to find object types other than `String` and `Array` to prepare memory layout. This is only a limitation for our current implementation. Once these new memory preparation techniques are identified,

ChaffyScript just needs an update to its memory allocation/free candidate discovery process to reflect the new memory preparation technique.

Applicability on the other script-based exploits JavaScript is not the only script language that can be used to launch exploits; other script languages like VBScript [22] and ActionScript [1] are commonly used to launch exploits. These script-based exploits are widely used to create malicious Microsoft Documents (word, excel, powerpoint,etc.), flash files, web pages [2]. So the question rises in our mind - Can ChaffyScript be applied to stop the other script-based exploits?

The answer is yes because: 1) Script-based languages share a similar memory management approach. They all use some sort of garbage collector to recycle the memory and conduct automatic garbage collection at runtime. 2) Memory preparation is a general stage in exploits. Attackers require this stage to bypass mitigation techniques like ASLR, CFG [7] with crafted memory. This stage is also used by the other script-based exploits. 3) Other script-based languages also execute interpretively and can be rewritten as JavaScript. This allows ChaffyScript to provide the protection via rewriting.

To demonstrate this, we set up the exploitation environment for CVE-2016-0189 [18]. It is a VBScript-based exploit targeting a VBScript memory corruption in IE11. We manually applied ChaffyScript's rewriting process and insert the memory perturbation code. The test shows that this exploit is successfully blocked by memory perturbation. This demonstrates that ChaffyScript can also be applied to stop the other script-based exploits.

To apply ChaffyScript on the other script language, we need the corresponding script parser, and also need to adapt the lightweight typing rules on the new script language since different languages support different typing systems. These adaptations are feasible and can be implemented with engineering efforts.

8 Related Work

Malicious JavaScript Detection Malicious JavaScript detection has been a hot research topic. Several systems have focused on statically analyzing JavaScript code to identify malicious web pages [41][33][48][31].

Then dynamic analysis [30,42,37] is widely deployed to expose behaviors of obfuscated JavaScript code. These approaches often require complex code analysis and cannot be deployed online to protect users against malicious JavaScript. Therefore, they are used to generate IDS rules for runtime detection. In comparison, ChaffyScript can provide users real-time protection without any knowledge of the targeted vulnerabilities.

Exploit Mitigation Mitigation techniques (Control Flow Integrity [23,52,56,46], ROP Mitigation [27,28,11], Randomization [35,43,34] have been evolving with the advancement of exploitation techniques. However, the current mitigation techniques usually require changes to source code or binaries and thus cannot be deployed promptly. In addition, once deployed, they cannot be rapidly upgraded as exploitation techniques advance. In contrast, ChaffyScript's JavaScript rewriting approach allows for a more flexible deployment.

JavaScript Rewriting JavaScript rewriting has been used by researchers to meet various security requirements. BrowserShields [47] uses it to stop JavaScript exploits by matching predefined vulnerability features. It does not work for 0-day exploits, while ChaffyScript does. ConScript [45] rewrites JavaScript to specify and enforce fine-grained security policies for JavaScript in the browser. Dachshund [44] secures against blinded constants in JIT code via removing all the constants from JavaScript code.

9 Conclusion

In this paper, we made a key observation that memory preparation is an essential stage in JavaScript exploits and can be disturbed via a memory perturbation technique. Based on our observation, we proposed and designed ChaffyScript, a vulnerability-agnostic system to thwart JavaScript exploits by inserting memory perturbation with JavaScript rewriting. We have implemented a prototype system, ChaffyScript, and our evaluation shows that ChaffyScript incurs acceptable runtime performance overhead (5.88% for Chrome, 12.96% for FireFox) and memory overhead (6.1% for minimal JS heap usage, 8.2% for maximal JS heap usage) on Octane benchmark and defeat all of the 11 JavaScript exploits. We also demonstrated the deployment flexibility via integrating ChaffyScript into a web proxy to protect users against malicious HTML files.

References

1. Actionscript technology center. http://www.adobe.com/devnet/actionscript.html, 2017.
2. Akbuilder. https://nakedsecurity.sophos.com/2017/02/07/akbuilder-is-the-latest-exploit-kit-to-target-word-documents -spread-malware/, 2017.
3. The art of leaks: The return of fengshui. https://cansecwest.com/slides/2014/The%20Art%20of%20Leaks%20-%20read%20version%20-%20Yoyo.pdf, 2017.
4. Aslr bypass apocalypse in recent zero-day exploits. https://www.fireeye.com/blog/threat-research/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html, 2017.
5. Chakracore javascript engine. https://github.com/Microsoft/ChakraCore, 2017.
6. Chrome v8 engine. https://developers.google.com/v8/, 2017.
7. Control flow guard. https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx, 2017.
8. Cve details. http://www.cvedetails.com/, 2017.
9. Ecmascript js ast traversal functions. https://github.com/estools/estraverse, 2017.
10. Ecmascript parsing infrastructure for multipurpose analysis. http://esprima.org/, 2017.
11. The enhanced mitigation experience toolkit. https://support.microsoft.com/en-us/help/2458544/the-enhanced-mitigation-experience-toolkit, 2017.
12. Geekpwn. http://2017.geekpwn.org/1024/en/index.html, 2017.

13. Http mitm proxy. https://github.com/joeferner/node-http-mitm-proxy, 2017.
14. The javascript benchmark suite for the mordern web. https://developers.google.com/octane/, 2017.
15. Javascriptcore. https://trac.webkit.org/wiki/JavaScriptCore, 2017.
16. Lexer confusing attack. https://github.com/google/caja/wiki/JsControlFormatChars, 2017.
17. Node.js. https://nodejs.org/en/, 2017.
18. Proof-of-concept exploit for cve-2016-0189 (vbscript memory corruption in ie11). https://github.com/theori-io/cve-2016-0189, 2017.
19. Pwn2own. https://en.wikipedia.org/wiki/Pwn2Own, 2017.
20. Rop is dying and your exploit mitigations are on life support. https://www.endgame.com/blog/technical-blog/rop-dying-and-your-exploit-mitigations-are-life-support, 2017.
21. Spidermonkey javascript engine. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey, 2017.
22. Vbscript. https://en.wikipedia.org/wiki/VBScript, 2017.
23. ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security* (2005), ACM, pp. 340–353.
24. ANDERSON, C., GIANNINI, P., AND DROSSOPOULOU, S. Towards type inference for javascript. In *European conference on Object-oriented programming* (2005), Springer, pp. 428–452.
25. BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 75–88.
26. CARMONY, C., HU, X., YIN, H., BHASKAR, A. V., AND ZHANG, M. Extract me if you can: Abusing pdf parsers in malware detectors. In *NDSS* (2016).
27. CHEN, X., SLOWINSKA, A., ANDRIESSE, D., BOS, H., AND GIUFFRIDA, C. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS* (2015).
28. CHENG, Y., ZHOU, Z., MIAO, Y., DING, X., DENG, H., ET AL. Ropecker: A generic and practical approach for defending against rop attack.
29. COLLBERG, C., THOMBORSON, C., AND LOW, D. A taxonomy of obfuscating transformations. Tech. rep., Department of Computer Science, The University of Auckland, New Zealand, 1997.
30. COVA, M., KRUEGEL, C., AND VIGNA, G. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th International Conference on World Wide Web* (2010).
31. CURTSINGER, C., LIVSHITS, B., ZORN, B. G., AND SEIFERT, C. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security Symposium* (2011).
32. DANIEL, M., HONOROFF, J., AND MILLER, C. Engineering heap overflow exploits with javascript. *WOOT 8* (2008), 1–6.
33. FEINSTEIN, B., PECK, D., AND SECUREWORKS, I. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. *Black Hat USA* (2007).
34. GADALETA, F., YOUNAN, Y., AND JOOSEN, W. Bubble: A javascript engine level countermeasure against heap-spraying attacks. In *International Symposium on Engineering Secure Software and Systems* (2010), Springer, pp. 1–17.
35. GRAS, B., RAZAVI, K., BOSMAN, E., BOS, H., AND GIUFFRIDA, C. Aslr on the line: Practical cache attacks on the mmu. *NDSS (Feb. 2017)* (2017).

36. GRIER, C., BALLARD, L., CABALLERO, J., CHACHRA, N., DIETRICH, C. J., LEVCHENKO, K., MAVROMMATIS, P., MCCOY, D., NAPPA, A., PITSILLIDIS, A., ET AL. Manufacturing compromise: the emergence of exploit-as-a-service. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 821–832.
37. HARTSTEIN, B. Jsunpack: An automatic javascript unpacker. In *ShmooCon convention* (2009).
38. KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2003), CCS '03, ACM, pp. 272–280.
39. KIRDA, E., KRUEGEL, C., VIGNA, G., AND JOVANOVIC, N. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing* (2006), ACM, pp. 330–337.
40. LARSEN, P., HOMESCU, A., BRUNTHALER, S., AND FRANZ, M. Sok: Automated software diversity. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 276–291.
41. LIKARISH, P., JUNG, E., AND JO, I. Obfuscated malicious javascript detection using classification techniques. In *MALWARE* (2009), Citeseer, pp. 47–54.
42. LU, G., AND DEBRAY, S. Automatic simplification of obfuscated javascript code: A semantics-based approach. In *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability* (2012).
43. LU, K., SONG, C., LEE, B., CHUNG, S. P., KIM, T., AND LEE, W. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 280–291.
44. MAISURADZE, G., BACKES, M., AND ROSSOW, C. Dachshund: Digging for and securing against (non-) blinded constants in jit code.
45. MEYEROVICH, L. A., AND LIVSHITS, B. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 481–496.
46. PRAKASH, A., YIN, H., AND LIANG, Z. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security* (2013), ACM, pp. 311–322.
47. REIS, C., DUNAGAN, J., WANG, H. J., DUBROVSKY, O., AND ESMEIR, S. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Transactions on the Web (TWEB) 1*, 3 (2007), 11.
48. SEIFERT, C., WELCH, I., AND KOMISARCZUK, P. Identification of malicious web pages with static heuristics. In *Telecommunication Networks and Applications Conference, 2008. ATNAC 2008. Australasian* (2008), IEEE, pp. 91–96.
49. SERNA, F. J. The info leak era on software exploitation. *Black Hat USA* (2012).
50. SINTSOV, A. Writing jit-spray shellcode for fun and profit. *Writing* (2010).
51. SOTIROV, A. Heap feng shui in javascript. *Black Hat Europe* (2007).
52. TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, Ú., LOZANO, L., AND PIKE, G. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium* (2014), pp. 941–955.
53. TRAN, T., PELIZZI, R., AND SEKAR, R. Jate: Transparent and efficient javascript confinement. In *Proceedings of the 31st Annual Computer Security Applications Conference* (2015), ACM, pp. 151–160.
54. YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted

x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on* (2009), IEEE, pp. 79–93.

55. YU, Y. Write once, pwn anywhere. *BlackHat* (2014).
56. ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 559–573.