SoK: Make JIT-Spray Great Again

Robert Gawlik and Thorsten Holz
Ruhr-Universität Bochum

Abstract

Since the end of the 20th century, it has become clear that web browsers will play a crucial role in accessing Internet resources such as the World Wide Web. They evolved into complex software suites that are able to process a multitude of data formats. *Just-In-Time (JIT)* compilation was incorporated to speed up the execution of script code, but is also used besides web browsers for performance reasons. Attackers happily welcomed JIT in their own way, and until today, JIT compilers are an important target of various attacks. This includes for example *JIT-Spray*, *JIT-based* code-reuse attacks and *JIT-specific* flaws to circumvent mitigation techniques in order to simplify the exploitation of memory-corruption vulnerabilities. Furthermore, JIT compilers are complex and provide a large attack surface, which is visible in the steady stream of critical bugs appearing in them.

In this paper, we survey and systematize the jungle of JIT compilers of major (client-side) programs, and provide a categorization of offensive techniques for abusing JIT compilation. Thereby, we present techniques used in academic as well as in non-academic works which try to break various defenses against memory-corruption vulnerabilities. Additionally, we discuss what mitigations arouse to harden JIT compilers to impede exploitation by skilled attackers wanting to abuse Just-In-Time compilers.

1 Introduction

Since it became clear that memory bugs, especially stack-based buffer overflows, can be used to execute arbitrary attacker-controlled code [51], a plentitude of attacks and defenses were proposed over the years [77]. Comparable to an arms race, new defenses popped up and were broken shortly afterwards with novel attacks that led to new defenses again. Especially web browsers became an attractive target for attacks given their practical importance and wide-spread use, in addition to their complexity.

Attacks against client-side programs such as browsers were at first tackled with a non-executable stack to prevent execution of data on the stack and also with a non-executable heap to stop heap sprays of data being later executed as code. This defense became widely known as $W \oplus X$ (**W**ritable **xor** e**X**ecutable) or *Data Execution Prevention (DEP)* to make any data region non-executable in 2003 [45, 54]. To counter DEP, attackers started to perform *code reuse* such as *Return-Oriented Programming (ROP)* and many variants [10, 11, 32, 68]. In general, if an adversary knows the location of static code in the address space of the vulnerable target, she can prepare a fake stack with addresses of these *gadgets*. As soon as control of the instruction pointer is gained, these gadgets execute in a chained manner and perform the desired actions. To prevent code-reuse and similar types of attacks, *Address Space Layout Randomization* (ASLR) was proposed in 2003 [55]. It randomizes the address layout making it difficult to find code snippets to reuse [66, 78].

JIT-Spray came in handy here: If expressions with constant values of a high-level language are *Just-In-Time (JIT)* compiled into native code, they can be abused to embed malicious code bytes at run time. This bypasses DEP because data is (indirectly) injected as code. Additionally, if the adversary manages to create many regions of this code, their locations become predictable. Hence, by *spraying* many code regions, she can predict the address of one region to bypass ASLR. Finally, only control over the instruction pointer is needed to redirect the control flow to the injected code. Thereby, a *use-after-free*, *type confusion* or *heap-buffer overflow* vulnerability is sufficient. We provide an overview of JIT compilers in Section 2, and an in-depth look at JIT-Spray in Section 3.

JIT-Spray enabled the creation of complete self-sustained payloads such as arbitrary shellcode that executes continuously. Moreover, there are techniques to spray small snippets (so called *JIT gadgets*) which have to be chained together in an exploit. If their addresses are predictable, we still count it as JIT-Spray. However, if a memory disclosure vulnerability is necessary to locate them, we do *not* count it as JIT-Spray, but as *JIT-based* code-reuse attack. The main reason for this distinction is that memory disclosures are usually more difficult to achieve than control over the instruction pointer only. Hence, the adversary needs more control and has to invest more resources than with JIT-Spray alone. An overview of JIT-based code-reuse attacks is provided in Section 4.

Note that JIT-Spray and JIT-based code reuse have in common that injected code is *unintended* and is usually in the *middle* of an instruction stream the JIT compiler intentionally emitted. A defense which does not only prevent execution of unintended JIT code but also unintended static code such as ROP gadgets is *Control-Flow Integrity (CFI)* [1]. It assures that only predefined code entries are valid targets for branches in static as well as JIT code. A prominent implementation of CFI is Microsoft's *Control-Flow Guard (MS-CFG)* which also protects JIT-code regions in Microsoft Edge [41]. More recently, *LLVM-CFI* appeared in Google Chrome as an effective CFI solution [27, 80].

Another general mitigation against code injection attacks is *Arbitrary Code Guard (ACG)* [43]. If enabled, it prevents making code pages writable and writable pages executable. This poses a challenge for JIT compilers as they need to write code first and change the permissions to executable afterwards. Hence, Microsoft Edge uses an out-of-process JIT server which maps dynamic code into a memory region shared with the browser process. In Section 5, we describe additional security problems which may arise from JIT compilers.

There are also specific mitigations against JIT-Spray and JIT-related flaws. We shortly introduce some of them needed to provide a basic understanding for the rest of the paper. In Section 6, we provide a more in depth and complete picture of mitigations against attacks abusing JIT compilers. One mitigation is a convenient compiler optimization named *constant folding*: if the (JIT) compiler is able to calculate operations at compile time before generating native code, then only the remaining operations/results will end up in native code. Hence, constants do not appear in which an attacker would have been able to embed malicious code bytes. A more specific defense against embedding code in constants is *constant blinding*. For that, an immediate value is xored with a random key value before the JIT compiler emits it into the native instruction. As soon as it is used it is "unxored", but the immediate value does not appear in native code. Hence, it directly hinders code injection through JIT-Spray. We explain JIT-related defenses in more detail in Section 6.

Overall, we make the following contributions:

- We provide an overview and survey of JIT-Spray on the x86 and ARM architecture including academic work as well as non-academic attacks, and describe the offensive techniques in detail. The most recent JIT-Spray technique appeared in *ASM.JS* in Mozilla Firefox, which we illustrate in more detail.
- We distinguish *JIT-Spray* from *JIT-based* code-reuse attacks and explain the landscape of JIT-compiler based code-reuse attacks.
- We summarize mitigation bypasses that were possible with the help of JIT compilers and exhibit defenses which emerged over the years to protect against various JIT-related flaws.

The remaining sections of the paper are categorized in Table 1. It provides JIT-Spray and JIT-based code reuse attacks based on the achieved exploit goal in the affected targets. Various defenses were proposed as hardening mechanisms.

2 Just-In-Time Compilation

As noted before, the most popular client-side programs for everyday users are undoubtly web browsers. Major browsers such as *Mozilla Firefox*, *Internet Explorer*, *Microsoft Edge*, *Google Chrome* and *Apple Safari* contain a *JavaScript* run-time environment. JavaScript is a dynamic scripting language and allows convenient manipulation of web content, a crucial aspect of the modern Web. While JavaScript engines run as interpreters, they embed *Just-In-Time* (JIT) compilers as well. The benefits of JIT compilation compared to the interpretation of JavaScript (bytecode) are huge performance gains: instead of executing JavaScript code in a virtual-machine-like manner, *native code* is emitted by the JIT compilers for the CPU the browser runs on. If a function runs *hot*, i. e. is executed frequently, Just-In-Time compilation kicks in and transforms the function into machine code for the architecture the browser runs on.

There exist several Just-In-Time compilers and optimization layers. Broadly speaking, JavaScript in web browsers share the same design [17]. There is an interpreter and one or more Just-In-Time compilers with different optimization levels. JavaScriptCore (JSC) of *WebKit*, for example, which is the base for Apple Safari, uses a *four-tier* JavaScript optimization strategy of which three tiers are JIT optimizations (*LLint, Baseline, DFG, FTL*) [57]. While the first tier interprets JavaScript bytecode, the second to fourth JIT stages kick in on an increasing number of executions of functions and statements. Thereby, the optimizations become more and more aggressive to improve the performance of native code emitted by the second to fourth tier optimization.

Similarly, *ChakraCore*, the JavaScript engine of Microsoft Edge, has multiple tiers. It contains an interpreter and a two-tier JIT compiler [40]. If hardware resources

Table 1: Defenses bypassed by JIT-Spray and JIT-based code reuse attacks and proposed mitigations.

Attack Flavor	Exploit Goal	Targets (see § 2)	Target Architecture	Bypassed Mitigations	Proposed Defenses (see § 6)
JIT-Spray (see § 3)	Code execution of continuous payload	ActionScript JIT [9]	x86	W⊕X, ASLR	Constant folding, Constant blinding, Random nop insertion, JIT allocation restriction
		Apple Safari (JSC) [69]			
		JVM [14]			
		{Jaeger|Trace}Monkey [64, 65]			
		Mozilla Firefox (ASM.JS) [25]			
		Linux kernel (eBPF) [39, 60]	x86	SMEP, KERNEXEC	CFI (RAP)
		ActionScript JIT [8]	ARM	W⊕X, ASLR	Constant blinding
		JSC [34]			
		SpiderMonkey [34]			
	Code execution of JIT gadgets	ActionScript JIT [67]	x86	W⊕X, ASLR, Random nops	Constant blinding
		{Jaeger|Trace}Monkey [64, 65]	x86	W⊕X, ASLR	Constant blinding
		LLVM JIT [65]			
		Internet Explorer (WARP JIT) [42]	x86	W⊕X, ASLR, MS-CFG	Enforce MS-CFG for WARP JIT code
		Microsoft Edge (WARP JIT) [42]	x64		
		JSC [33]	ARM	W⊕X, ASLR	JIT allocation randomization
		V8 [34]			
JIT-based code reuse (see § 4)	Code execution of JIT gadgets	SpiderMonkey [6]	x86	Gadget-free static code	Constant blinding
		Internet Explorer (Chakra) [6]	x64		
		SpiderMonkey [37]	x64	Execute-only memory	Remove implicit constants from native code
		Internet Explorer (Chakra) [37]	x86		
		Google Chrome (V8) [37]	x86/x64		

are available, JIT compilation may be split into parallel background threads. Mozilla Firefox currently uses *IonMonkey* as Just-In-Time compiler, which is designed in a cross-architectural manner to simplify run-time compilation to various CPUs such as x86 and ARM [4]. *TurboFan* is the JIT compiler of *V8*, the JavaScript engine of Google Chrome, and implements its own set of aggressive optimizations, while reducing complexity of previous JIT compilers [26, 29].

Besides browsers, also the *Java Virtual Machine (JVM)* uses an interpreter and JIT compiler (Oracle HotSpot). This way, Java compiled binaries remain portable featuring a universal bytecode, which is interpreted and JIT-compiled to the underlying architecture on which the program has to be executed [35]. For the sake of completeness, we mention that Microsoft's *dotNet* framework features a JIT compiler *(RyuJIT)* [20] as well, and even the Linux kernel uses JIT compilation for *extended Berkeley Packet Filters (eBPF)* [21]. For the popular server-side language PHP, a JIT compiler named *HHVM* exists and the language Lua uses *LuaJIT* for dynamic native code generation [52, 73]. Interestingly, as we explain in the next section, the most prominent attack against a JIT compiler was *JIT-Spray* [9], which targeted the *ActionScript Virtual Machine (AVM)* of Adobe *Flash* [3].

3 JIT-Spray

JIT-Spray is an elegant exploitation technique to bypass both DEP and ASLR. While it is not as generic as ROP—because a JIT compiler is necessary—it significantly simplifies the exploitation of memory bugs.

3.1 JIT-Spray on x86

The instruction length of the x86 architecture is variable. Hence, within a code bytestream, every byte-offset is a potential beginning of an instruction given that the bytes at every offset are decodable by x86. From an exploiter's perspective, this can be abused to inject code: if an adversary has enough control over a JIT compiler, she can force it to emit instructions containing immediate values, while these contain valid instruction bytes themselves. In 2010, Blazakis found that *ActionScript* of Adobe Flash directly emits script-level constants into native machine code [9]. Consider a long `XOR` expression in ActionScript as shown in Listing 1.

```
var y = (
    0x3c909090 ^
    0x3c909090 ^
    0x3c909090 ^
    ...
)
```

Listing 1: ActionScript statement containing a long `XOR` sequence.

The ActionScript JIT compiler generates native machine code containing the instructions shown in Listing 2. While these instructions represent the high-level calculation, different instructions are executed if execution starts at the first offset (see Listing 3). It shows in a school-book manner how a nop-sled can be created by injecting `NOP` (`0x90`) instructions. As the ActionScript constants are fully controllable by the attacker, arbitrary payload instructions less or equal than three bytes in size can be injected. The fourth byte (`0x3C`) serves the purpose to mask the legitimate operation represented by the opcode `0x35`

(⊕), and results in a semantic nop-like instruction (`cmp al, 0x35`). It also prevents instruction *resynchronization*.

```
0x00:   B8 9090903C    mov eax, 0x3c909090
0x05:   35 9090903C    xor eax, 0x3c909090
0x0a:   35 9090903C    xor eax, 0x3c909090
...
```

Listing 2: Intended native code emitted by the ActionScript JIT compiler.

The adversary then forces the JIT compiler to generate enough copies of native machine code such that their addresses in memory become predictable (on 32-bit systems). Then she can redirect the control flow to a predicted address and execute the injected code. The chance is 80% that the nop-sled will be hit. In one of five cases (20%), the legitimate instruction stream will be hit and the exploit fails.

```
0x01:   90      nop
0x02:   90      nop
0x03:   90      nop
0x04:   3C35    cmp al, 0x35
0x06:   90      nop
0x07:   90      nop
0x08:   90      nop
0x09:   3C35    cmp al, 0x35
0x0b:   90      nop
0x0c:   90      nop
0x0d:   90      nop
...
```

Listing 3: Injected code not intended by the ActionScript JIT compiler.

This was the birth of JIT-Spray and further attacks were not long to be coming. In 2010, Sintsov showed how to automate and write shellcode for JIT-Spray attacks [70]. Instructions larger than three bytes in size are a problem, but most of them can be transformed into semantically equivalent instructions less or equal than three bytes. For example, "`MOV EAX, 0x41424344`" results in a five-byte instruction. However, it can be split into three instructions performing the same operation: "`MOV EAX, 0x41424yyzz`" is emitted by controlling three bytes and letting the JIT compiler mangle two bytes (`yyzz`). These are set separately with two instructions both two bytes in size: "`MOV AH, 0x43`" and "`MOV AL, 0x44`". Another nifty trick was to use `0x6A` instead of `0x3C` as a masking byte. This way, instead of creating a semantic nop which tampers with CPU flags (`3C35 cmp al, 0x35`), a push instruction emerged (`6A35 push 0x35`). This allowed adversaries to use conditional jumps (i. e., `JNZ`) afterwards.

JIT-Spray was also possible in the Windows version of JavaScript of Apple Safari in 2010 [69]. As the JIT compiler emitted much code between the controlled JavaScript constants, the author used two bytes from the constant as payload bytes, and the two other bytes as an unconditional jump to the next constant. Consider the constant `0x14EB9090` in a JavaScript operation. Apple Safari's JavaScriptCore baseline-JIT compiler generated code which could be abused in the following way (see Listing 4):

```
0x01:   90      nop
0x02:   90      nop
0x03:   eb14    jmp 0x19
...
0x19:   90      nop
0x1a:   90      nop
0x1b:   eb14    jmp 0x31
...
```

Listing 4: Abusing JavaScript constants to connect two payload bytes with short jumps.

While only two bytes are usable effectively for malicious purposes, Sintsov showed that arbitrary operations are still possible. This included writing a malicious payload to a writable and executable JIT page, and then jumping to it.

In 2010 and 2011, an in-depth investigation was performed on especially JIT compilers of LLVM and Mozilla Firefox [64, 65]. The authors demonstrated that former Mozilla Firefox JIT engines such as *JaegerMonkey* and *TraceMonkey* [24] were prone to JIT-Spray. In addition, they showed that floating point values such as `-6.828527034422786e-229` are usable for JIT-Spray, as the value's hexadecimal representation of eight `0x90` bytes was directly emitted into executable code regions. For TraceMonkey and LLVM, they were able to force the JIT compiler to emit little code snippets usable for code-reuse attacks which they named *gaJITs*. Additionally, the authors researched the mitigations in various JIT engines and found that most of them did not apply enough protections against JIT-Spray (see also Section 6).

In a code-reuse manner, Serna showed in 2013 that it was still possible to let the JIT compiler of Flash ActionScript emit small code snippets to predicable addresses [67]. While a full payload was not possible due to the mitigations Adobe incorporated at that time (i.e., *random nops*, see Section 6), these small JIT snippets were used to leak return addresses from the stack.

JIT-Spray affected the JVM as well: In 2013, it was shown that constants in `XOR` operations in Java were emitted into executable code. Similar to Listing 3, three bytes were usable to inject code [14]. Generating multiple classes and functions containing the operations triggered code generation to predictable addresses. Hence, DEP and ASLR were bypassed by controlling the instruction pointer with a memory corruption vulnerability.

One of the more recent JIT-Spray attacks affecting JIT compilers was published in 2016: WebGL *shaders* were usable inside JavaScript of Internet Explorer and Microsoft Edge [76]. The WARP JIT compiler produced native code not protected by MS-CFG. Thus, the authors were able to inject code to predictable addresses with the *Windows Advanced Rasterization Platform (WARP) Shader* JIT compiler [42].

JIT-Spray is also possible in the Linux kernel if *extended Berkeley Packet Filters (eBPF)* are available (which are switched off by default). However, this technique bypasses defenses which forbid the kernel to execute code provided by userspace such as SMEP and KERNEXEC [39, 60]. It has been shown that building a BPF program and creating many sockets with attached (BPF) filters leads to a JIT-Spray inside the kernel. This way, an attacker-controlled payload can be executed such as spawning a root shell when the control flow is hijacked.

3.2 Case Study: ASM.JS

We published a new JIT-Spray attack in 2017. What makes a difference is that it does not target a JIT compiler but an *Ahead-Of-Time (AOT)* compiler in Mozilla Firefox 32-bit on Windows [25]. Technically speaking, AOT compilers do not generate and optimize code *if* and *after* certain high-level code was already executed several times, but *before* it is executed the first time. ASM.JS is an AOT compiler using a subset of JavaScript [46]. It obeys a certain syntax and appeared in 2013 in Mozilla Firefox [81].

```
function asm_js_module(){
    "use asm"
    function asm_js_function(){
        var val = 0xc1c2c3c4;
        return val|0;
    }
    return asm_js_function
}
```

Listing 5: Simple ASM.JS module with a function returning a 32-bit integer.

A simple ASM.JS module which is compiled ahead of time *without* having been executed is shown in Listing 5. Loading the web page containing the code is sufficient to trigger AOT. This module was requested many times and we found that several native code copies were emitted to predictable addresses (see Listing 6).

```
modules = []
for (i=0; i<=0x2000; i++){
        modules[i] = asm_js_module()
}
```

Listing 6: Requesting a ASM.JS module several times to spray many code copies to predictable addresses.

As the constants were not blinded, they appeared as immediate values in native code operations and were usable as the perfect target to hide attacker-controlled payload bytes. Listing 7 shows the native code which the AOT compiler generated and demonstrates that the constant `0xc1c2c3c4` appears directly in executable code regions. Additionally, these regions are located several times at predictable addresses turning ASLR ineffective. Armed with that possibility, arbitrary code injection was possible.

```
****0023:  b8c4c3c2c1   mov   eax, 0xc1c2c3c4
****0028:  6690         xchg  ax,ax
****002a:  83c404       add   esp,4
****002d:  c3           ret
```

Listing 7: Four-byte constant within native code of a simple ASM.JS code copy, of which many are emitted to predictable addresses.

We were able to abuse several operations to spray malicous code to predictable addresses. An overview is presented in Figure 1. Amongst others, we identified that arithmetic operations, setting array elements, and passing parameters to foreign function calls served well in embedding hidden code bytes. The most interesting method is, however, related to the first JIT-Spray technique on ARM (see Section 3.3): when floating point values are used as parameters in a function call within an ASM.JS module, they do not appear directly in the native code. Instead, instructions are emitted referencing these parameters indirectly (see Listing 8).

```
val = +ffi_func(
    2261634.5098039214,      // 0x4141414141414141
    156842099844.51764,      // 0x4242424242424242
    1.0843961455707782e+16,  // 0x4343434343434343
    7.477080264543605e+20    // 0x4444444444444444
)
```

```
0x00:  movsd xmm1, mmword [****0530]
0x08:  movsd xmm3, mmword [****0538]
0x10:  movsd xmm2, mmword [****0540]
0x18:  movsd xmm0, mmword [****0548]
...
****0530:
41414141 41414141 42424242 42424242
****0540:
43434343 43434343 44444444 44444444
...
```

Listing 8: Function call in ASM.JS with double float parameters and disassembly of generated native code referencing constants in the same code region.

However, the constants reside in the same executable region and are continuous in memory. This is very convenient for the adversary, because she can use all eight bytes of a double float value as payload and inject continuous shellcode without being distrupted by other opcodes. It was demonstrated with various exploits [61–63] that this technique is feasible and simplifies the exploitation

`v = (v + 0xa8909090)|0;`	`0: add eax, 0xa8909090`

`array[X] = 0x06eb9090;`	`0: mov eax, 0x06eb9090`
	`1: mov dword [X], eax`

`val = ffi_func(`
`0xa9909090|0,`
`0xa9909090|0,`
`0xa9909090|0,`
`)|0;`
	`...`
	`0: mov dword [esp], 0xa9909090`
	`1: mov dword [esp+4], 0xa9909090`
	`2: mov dword [esp+8], 0xa9909090`
	`...`

Figure 1: ASM.JS operation and corresponding emitted native code embedding attacker controlled code bytes in immediate values.

of memory corruption vulnerabilities drastically. As adversaries can refrain from memory disclosures and code-reuse, only control of the instruction pointer is needed. We also developed a tool to transform payloads into its ASM.JS form, which then at run time is emitted to native code by the AOT compiler. While only two or three bytes from a high-level constant are used as payload bytes, it is still possible to execute arbitrary code. E. g., a *stage0* code will resolve and call the Windows API function `VirtualAlloc()`, then copy a bigger payload to it (stage1) and execute it.

3.3 JIT-Spray on ARM

A fundamental property making JIT-Spray possible on x86 is not available on the ARM architecture: native instructions have a fixed size of either 32-bit and are aligned to four-byte addresses (*ARM* mode), or have a size of 16-bit and are aligned to two-byte boundaries (*Thumb* mode). Additionally, the *Thumb-2* mode added new 32-bit instructions to Thumb and allowed mixing 16-bit and 32-bit instructions [33]. Hence, it is much more difficult to inject arbitrary code compared to x86. However, the first type of JIT-Spray on ARM we are aware of was using float constants in ActionScript [8]. The tested ActionScript JIT engine generated code with `PC`-relative references to the constants. Additionally, the float values resided in the same page as code and were continuous in memory. This allowed for continuous shellcode without other disrupting opcodes.

```
function readGadget(x) {
    return x ^ 0x11111610;
}
```

Listing 9: JavaScript function used in JavaScriptCore on ARMv7-A [33].

Lian et al. investigated the idea of JIT-Spray on ARMv7-A in more depth [33]. The JavaScriptCore JIT compiler (of WebKit) generated under certain circumstances attacker-influenced gadgets to predictable addresses. A JavaScript snippet getting compiled down into code containing a gadget is shown in Listing 9. The DFG JIT generated useful instructions (see Listing 10). Note that most of the gadget's instructions were intended instructions, only the first instruction was unintended: instead of executing the intended 32-bit instruction aligned to a four-byte boundary, the execution started with a branch to the second half of it. Then, the execution resynchronized with the intended instruction stream. Listing 11 shows the gadget instructions for the `readGadget` function. Execution starts in the middle of an instruction at offset `0x38` and resynchronizes at offset `0x3a`. Nonetheless, the gadget provides the adversary with the capability to read memory and return content as 32-bit integers into JavaScript.

```
0x00: mov    r2, lr
0x02: str.w  r2, [r5, #-16]   ;save return address
...
0x32: ldr.w  r0, [r5, #-64]   ;load argument
0x36: movw   r12, #5648       ;0x1610
0x3a: movt   r12, #4369       ;0x1111
0x3e: eor.w  r0, r0, r12
0x42: mov.w  r1, #4294967295  ;0xffffffff
0x46: ldr.w  r2, [r5, #-16]   ;load return address
0x4a: ldr.w  r5, [r5, #-40]   ;restore frame ptr
0x4e: mov    lr, r2
0x50: bx     lr               ;return
...
```

Listing 10: Native code emitted for `readGadget` JavaScript code.

More details can be found in the original paper [33]. This and similar gadgets were called by high-level JavaScript code to perform *chained* operations. This way, the authors achieved to write shellcode to a writable code page and execute it by having control over the program counter only.

```
0x38: ldr    r0, [r2, #64]    ;read memory from r2+#64
0x3a: movt   r12, #4369       ;0x1111
0x3e: eor.w  r0, r0, r12
0x42: mov.w  r1, #4294967295  ;0xffffffff
0x46: ldr.w  r2, [r5, #-16]   ;load return address
0x4a: ldr.w  r5, [r5, #-40]   ;restore frame ptr
0x4e: mov    lr, r2
0x50: bx     lr               ;return
...
```

Listing 11: Unintended instruction stream for `readGadget` JavaScript code, able to read memory pointed to by the `R2` register.

In follow-up work, Lian et al. targeted Mozilla Firefox' IonMonkey JIT compiler of its SpiderMonkey JavaScript engine on ARM [34]. They were able to build a complete self-sustaining JIT payload at predictable adresses without relying on gadgets. They forced emitting 32-bit ARM `AND` instructions while having 20 bits of control over each from the JavaScript context. They managed to get this instruction interpreted as two 16-bit Thumb-2 instructions. Armed with that possibility, the first instruction is used to perform useful operations for the attacker. The second instruction is used as an unconditional `PC`-relative forward branch to the next to-be-executed unintended instruction. Additionally, it prevents switching back to the compiler-intended instruction stream. Overall, this full JIT-Spray on ARM was the first of its kind, and the authors disproved the belief that JIT-Spray is not feasible on RISC architectures with fixed instruction lengths and fixed instruction boundaries.

4 JIT-Based Code Reuse

The first (academic) work which used run time compiled gadgets from a JIT compiler arouse from the need to bypass code-reuse protections in 2015 [6]. If static code of a program is *gadget-free*, then code-reuse is usually not an option [50]. However, if gadgets are produced by the JIT compiler, code-reuse becomes feasible again. Athanasakis et al. targeted IonMonkey on 32-bit Linux and Chakra of 64-bit Internet Explorer 9 on Windows [6]. In general, they provoked the JIT compiler to emit gadgets containing only a few instructions and were using two-byte JavaScript constants. This bypassed constant blinding in Internet Explorer and various other JIT-related defenses which were incorporated at that time. However, note that the authors needed memory disclosures to locate the gadgets in memory. Hence, we do not count it as JIT-Spray, because JIT-Spray does not require info leaks but only control over the instruction pointer to redirect control flow to a predetermined address containing the JIT-compiled attacker code.

In 2015, other flaws in the Chakra JavaScript engine of Internet Explorer related to JIT-code reuse were found. While Chakra applies constant blinding, divison expressions in JavaScript with 32-bit integers resulted in non-blinded four-byte constants containing injected code [84]. Additionally, the authors showed that two 16-bit constants were emitted directly into one x86 instructions when the first was used as array index, and the second as array element. To be able to jump to this injected code, they used a "`JMP ECX`" instruction of the JavaScript engine itself which was not protected by MS-CFG. However, both the injected code and the jump instruction were only locatable with memory disclosures. Hence, we do not count it as JIT-Spray, but JIT-code reuse.

We also want to distinguish the term *JIT-ROP* to the offensive techniques presented in this paper. It is **not** related to JIT compilers [71]. It merely describes the technique to repeatedly *locate*, *read* and *disassembly static* code with e. g. memory disclosures in JavaScript. Then, a code-reuse payload can be build *just-in-time*. This is necessary if fine-grained code randomization is applied to the target binaries, as it hides not only the base addresses of modules in the address space, but also function entries, basic blocks, and addresses of instructions.

`m = i ? 0x12345678 :`
`0x23456789`
	`0: test rax, rax`
	`1: je 2`
	`2: mov ebx, 0x23456789`
	`3: jmp 5`
	`4: mov ebx, 0x12345678`

`switch(j){`
`case 0x23232323: m++;`
`}`
	`0: mov rdx, [rbp+20]`
	`1: cmp edx, 0x23232323`
	`2: jne X`

`0x34343434[j]`	`0: mov rdx, 0x3434343400000000`
	`1: ;set other parameters`
	`2: call GetProperty`

`m = j ^ 0x45454545`	`0: mov rax, [rbp+20]`
	`1: xor eax, 0x45454545`

`globarr[i] = 0x67676767`	`0: mov [rdx+X], 0x67676767`
`return 0x12121212`	`0: mov rax, 0x1212121200000000`

Figure 2: JavaScript code and corresponding emitted JIT code missing to blind four-byte constants in Google Chrome (found by Dachshund [38]).

Nonetheless, defenses against JIT-ROP were weakened with JIT-compiled gadgets: *Execute-Only* memory is one of such defenses [7]. It forbits the reading of code, but the addresses of JIT-gadgets can still be found via subsequently leaking readable data object until pointers to the gadgets are discovered. Maisuradze et al. [37] were able to force the JIT compilers of Internet Explorer, Google Chrome and Mozilla Firefox to hide code within *branches*. Their carefully constructed JavaScript code resulted in control flow instructions such as conditional jumps and direct calls. Their target addresses and offsets had code bytes hidden themselves, and thus, were usable as gadgets for code-reuse attacks. As a defense, constant folding or blinding is not an option, as the implicit constants are within (relative) calls/jumps. Hence, they proposed to eliminate all implicit constants by replacing them with indirect control flow instructions.

As we mentioned in the introduction and described in mitigations (see Section 6), long JIT-Spray payloads and gadgets are prevented with constant blinding. However, Maisuradze et al. were also digging for *unblinded* four-byte constants *despite* four-byte constant blinding in modern web browsers [38]. They utilized fuzzing to generate JavaScript code containing constants and searched the target's memory for bytes representing the constants. Overall, the succeeded in finding several JavaScript operations with constants the JIT compiler encodes into memory in Google Chrome and Microsoft Edge. Figure 2 presents their findings in Google Chrome.

5 Abusing JIT-Compiler Flaws

The offensive techniques discussed in the previous two sections relied on the inner workings of JIT compilers and went into the direction of exploit-mitigation bypasses. In the following, we present attacks which can be considered to be based on flaws and bugs of JIT compilers, but please note that the distinction can sometimes be fuzzy.

5.1 More Mitigation Bypasses

A very popular mitigation is DEP. While JIT-Spray bypasses DEP, there was also the possiblity to *overwrite* emitted code as long as the permissions of the code pages are writable. Nowadays, neither static code nor JIT-compiled code should be executable and writable simultaneously. Currently, this does not hold for Google Chrome [59]. Nonetheless, even if $W \oplus X$ is enabled for JIT regions, they have to be written first and be executable afterwards. In 2015, Song et al. abused this small time window to overwrite code caches in multi-threaded code using web workers [72]. Hence, they achieved to inject code despite $W \oplus X$ in dynamic code regions.

In 2016, Chakra of Internet Explorer was attacked in a similar way. The authors proceeded in three steps [79]: First, they triggered the JIT compiler to encode a large code region. This created a time window in which a background thread was working on a temporary writable code buffer. In the second step, this buffer was located with memory disclosures and overwritten with malicious code in the third step. With this attack, they were able to bypass both DEP and MS-CFG.

In 2017, Frassetto et al. demonstrated a *data-only* attack on the *intermediate represention (IR)* of Chakra in Microsoft Edge [22]. Instead of creating/modifying code or code pointers, they crafted malicious C++ object representing IR statements with the prerequisite of a read/write primitive from within JavaScript. As the JIT compiler uses these object to generate native code, the authors were able to create and execute their code of choice.

Fratric researched Microsoft Edge's JIT compiler in depth in 2018 [23]. Amongst other flaws, he was able to inject his code of choice to bypass ACG by abusing the JIT server architecture. Microsoft Edge uses a separate JIT server to generate dynamic code for the browser (content) process. This process shares a memory region with the browser process and maps dynamic code into that shared region. The JIT server accesses the region with write permissions, while the browser process has an executable view of the region. This way, the browser process obeys to ACG: it cannot modify or create code pages by itself, it can only execute them. The author found that the JIT server can be tricked into making attacker-controlled memory in the browser process executable: if the browser process is compromised, the executable view of the shared region can be unmapped. Next, the attacker allocates writable memory on the same address and writes a payload to it. The JIT process will happily use this address next and will mark it as executable without changing the payload. This way, ACG was bypassed without directly tampering with content of code pages. A security update fixed the issue.

5.2 JIT-Compiler Vulnerabilities

JIT compilers are complex software systems as other compilers are as well. Hence, it is natural that they contain security-critical bugs. While there are vulnerabilities found constantly, we want to briefly summarize prominent ones at the time of writing in an exemplary manner.

Apple Safari's DFG JIT fell victim to optimization bugs during the Pwn2Own contests in 2017 and 2018 [15, 16, 75]. In both years, these were stepping stones for the contestants to further increase their privilege level with additional exploitation methods.

One interesting JIT-optimization bug in Google Chrome's V8 gave the attacker very powerful primitives such as leaking arbitrary memory and creating arbitrary JavaScript objects. Code execution was achieved by writing code into a JIT page. For more details, the reader is refered to the original research [58, 59].

The JIT compiler in Chakra is also prone to vulnerabilities. For example, CVE-2018-0953 allowed to create a type confusion by setting an array element with a magic value [36]. This was possible because the JIT compiler missed to emit a check into the dynamic code.

6 Mitigations

Before diving into more general mitigations against JIT-Spray and other JIT-related defenses, we take a look at the fixes against ASM.JS JIT-Spray from Section 3.2. Mozilla assigned CVE-2017-5375 and CVE-2017-5400 to that mitigation bypass technique [47, 48]. There were two CVEs assigned because the first patch was insufficient: randomization for code allocations was increased, but under certain circumstances, the old unpatched code responsible to emit ASM.JS regions to predictable addresses might still trigger. The follow-up patch redesigned the allocation scheme: At startup, when the address space is nearly empty, a random address range is reserved for ASM.JS code allocations. Hence, the location of that range is difficult to predict for an attacker. Later at run time, as soon as the ASM.JS modules are requested, regions from this set of pages are comitted and released upon need. As the source code is shared between ASM.JS and *WebAssembly*, this defense scheme (also known as *allocation randomization*) is also used for WebAssembly code allocations in Mozilla Firefox. The inital address range is additionally constrained to a certain number of pages to prevent address-space exhaustion during spraying (also known as *allocation restriction* [2, 65]).

We already explained *constant folding* in the introduction. Adobe incorporated constant folding into the Flash ActionScript JIT compiler to counter against the original JIT-Spray attacks [67]. This prevented immediate attacker-controlled constants in JIT code. Nonetheless, it was shortly bypassed by using the ActionScript `IN` operator [53]. It was sufficient to use one `IN` operation in an operation otherwise containing `OR`s to trick the JIT compiler and get attacker-controlled immediate values again. The ActionScript code "`0x3c909090 IN 0x3c909090 | 0x3c909090 | ...`" yielded a native nop-sled again, when executed from an unindented offset.

With *constant blinding*, a defense was developed and incorporated which prevents attacker-controlled immediate values in JIT code. Therefore, the compiler xors the immediate value at compile time with a random key unknown to the attacker. Before the value is used in an operation in native code, it is xored with that key at run time. This way, all operations in native code remain valid,

Table 2: Features and their impact on JIT-Spray (and JIT-code reuse) in modern web browsers in 2018.

Feature	Impact on	Mozilla Firefox	Google Chrome	Microsoft Edge	Internet Explorer
64-bit address space	Predictable location of injected code	✓	✓	✓	✗/✓[*]
ASLR	Predictable location of injected code	✓	✓	✓	✓
CFI	Execution of injected code	✗	✓[°]	✓[†]	✓[†‡]
Random nop insertion	Predictable location of injected code	✗	✓	✓	✓
Constant folding	Code injection	✓	✓	✓	✓
Constant blinding of constants $\geqslant$ four bytes	Code injection	✗	✓	✓	✓
$W \oplus X$ JIT regions	JIT-region overwrite	✓	✗	✓	✓
JIT-base offset and/or randomization	Predictable location of injected code	✓	✓	✓	✓
JIT allocation restriction	Predictable location of injected code	✓	✓	✓	✓
Guard pages	JIT-region overwrite	✓[#]	✓	✓[#]	✓[#]

[*] the default is 32-bit, switching to 64-bit requires a change in the registry (non-default)
[°] enforced by LLVM-CFI
[†] enforced by MS-CFG
[‡] partial implementation of MS-CFG
[#] not necessary for executable JIT regions

but are not easily predictable by adversaries, and hence, hiding code bytes is not possible. As the performance is impacted when blinding constants, usually only four-byte or larger constants are blinded. The impact is too drastic for two byte or smaller constants. As a result, this leads to the ability to hide code and let the JIT compiler create gadgets again [6].

Another possiblity to make hidden code in immediates less predictable are *random nops*. Various nop-like instructions with various sizes such as "`LEA ESP, [ESP]`," "`XCHG EDX, EDX`" or "`MOV ECX, ECX`" can intermingle intended JIT-compiled operations. If these are unknown to the attacker, jumping to a nop sled by controlling the instruction pointer may fail as the random nops may get hit. Adobe incorporated this technique in 2011 into the JIT compiler of ActionScript, but the emission frequency of random nops was too low to break small JIT gadgets [67]. However, it prevented complete self-sustained JIT payloads. Similarly, shifting the intended JIT code with a *random base offset* in code regions adds unpredictability to the location of injected code bytes.

While not directly defending JIT-Spray or JIT-based code-reuse attacks, *secure permissions* or *guard pages* harden against JIT-related flaws. JIT-code pages missing the write flag cannot be overwritten and guard pages prevent e. g. heap-buffer overflows to reach JIT-code pages (in case they are writable and adjacent to heap objects) [65]. For example, JIT-code regions in Mozilla Firefox are not writable anymore since the end of 2015 [19]. In 2011, *JitDefender* already employed mechanisms to retrofit $W \oplus X$ into JIT code of ActionScript and JavaScript [12]. The approach kept JIT-code regions not executable and switched to executable-only if a legitimate call to JIT regions was made. This prevented illegitimate executions of JIT-sprayed payloads, as jumps to it landed in code which was not executable.

Librando, a JIT-hardening framework, can apply some of these protections to COTS JIT compilers in a black box manner. Thereby, the operating system allocation functions are intercepted to analyze, diversify and rewrite the to-be just-in-time compiled code. Allocation randomization, constant blinding, random nop insertion, and various optimizations are applied to the JIT code [30]. The authors were able to harden Java and V8 with a slowdown of 1.15 to 3.5 times. RIM, JITSafe and INSeRT use obfuscation, diversification and randomization in a similar way to break hidden code in immediate values and hinder JIT-Spray [13, 82, 83].

While control-flow integrity became very popular for static code [1, 44, 56, 80, 86, 87] there is little CFI for dynamic code. *RockJIT* is one system aiming at providing control-flow integrity for native code dynamically [49]. It enforces policies on the JIT code and the JIT compiler itself. Amongst other techniques, after an analysis pass on source code, checks are used to allow only valid code entries to be targets of indirect branches. This thwarts JIT-Spray because unindented code bytes in the middle of an intended instruction are invalid code targets. The authors applied RockJIT to V8 with an performance hit of 14.6%. *Native Client (NaCl)*, the formally native code engine in Google Chrome, is similar to RockJIT. It applies software fault isolation (SFI) in the browser context [5]. Code is aligned to 32-byte chunks, and indirect branches are only allowed to jump to the chunk-beginnings. This is accomplished by checking for address masks before an indirect branch is issued.

Follow-up work tried to improve CFI for JIT code [85]. *JITScope* works with the LLVM infrastructure and is able to not only harden JIT code and the JIT compiler, but also the application as a whole with CFI. The authors applied JITScope to TraceMonkey with a performance impact of less than 10%. Nowadays, CFI is enabled for JIT code in Microsoft Edge with MS-CFG and LLVM-CFI is incorporated in Google Chrome. Grsecurity features *RAP* [28], a forward and backward-edge CFI solution for Linux preventing execution of unintended instructions [74].

Similar to the out-of process JIT separation in Microsoft Edge, *Lobotomy* proposed a two process model in 2014 [31]. The browser process has an executable view of a shared memory region, while a JIT process has a writable view to that region. This way, only the JIT process can create and manipulate JIT code and vulnerabilities in the browser process cannot tamper with or overwrite JIT regions.

Several other defenses against JIT-related flaws such as *JITSec* [18] were proposed. JITSec applies a monitor to JIT code to forbit system calls. This helps to break self-sustained JIT-Spray payloads which issue system calls, when the authors assumption holds that intended JIT code does not execute system calls. Another sophisticated defense called *JITGuard* uses the Intel SGX architecture and various other hardening techniques to protect against code-injection, code-reuse and data-only attacks [22]. Critical parts of the JIT compiler are isolated into SGX enclaves, a randomization layer is added for JIT code, and memory disclosures are tackled with a layer of indirection (i.e., trampolines).

A major change which can be seen as an implicit mitigation is the shift from 32-bit to 64-bit architectures. All major web browsers except Internet Explorer run as native 64-bit x86 applications nowadays. Similary, the ARM platform has a 64-bit architecture named *AArch64* which supports the 64-bit address space. As the 64-bit address space is larger than 32-bit, JIT-Spray seems to be impossible, as spraying many code regions to hit a predictable address is infeasible. Nonetheless, JIT-code reuse may still be valuable if memory disclosures can be conducted in 64-bit targets.

Table 2 summarizes the impact of (security) features on JIT-Spray and JIT-code reuse. If emission of unintended code bytes (*code injection*), spraying to *predictable locations*, and *execution* of unintended code is perfectly prevented, then JIT-Spray is infeasible. Having a combination of constant blinding, a 64-bit address space, high-entropy ASLR, and CFI achieves this goal if everything is correctly implemented.

7 Conclusion

JIT-Spray is an offensive technique that conveniently simplified the exploitation of memory corruption vulnerabilities as it bypasses both DEP and ASLR. At a time when exploit mitigations began to rise, hijacking the control flow was still enough to get arbitrary (remote) code execution. Since 2010, the year JIT-Spray appeared first, many JIT compilers were vulnerable to it. In this paper, we reviewed this technique in depth and showed what methods are used to hide code bytes within constants of high-level languages. Moreover, we surveyed the affected targets on the x86 and ARM architecture and established a connection to code-reuse attacks which abuse JIT compilers. While we provided on overview of additional flaws which may arise in JIT engines in an exemplary manner, we took a closer look at the latest JIT-Spray flaw which affected the AOT compiler ASM.JS in 2017 in Firefox.

JIT-compiler based attacks and defenses is still an ongoing and lively field of research, as the multitude of (academic) mitigations shows. The shift of browsers to 64-bit and an address space layout randomization with high entropy seems to make traditional JIT-Spray infeasible. However, in the light of memory disclosures, an imperfect CFI implementation, and incomplete (i.e., two-byte) constant blinding, JIT-code reuse is still a valuable asset for attackers. Moreover, flaws in JIT compilers (e.g., vulnerabilities) remain attractive targets. While the landscape of exploit mitigations increases and exploiting memory corruption vulnerabilities is becoming harder and harder, we do not want to exclude the possibility of JIT-Spray appearing again in future targets. Of course, defenses are getting more sophisticated, but so do attackers.

Acknowledgements

We would like to thank Matt Miller for shepherding this paper and his fruitful suggestions and comfortable cooperation. Moreover, we thank our anonymous reviewers for the constructive feedback which helped us to improve the paper. This work was supported by the European Commission through project H2020 DS-07-2017 “REACT” under Grant Agreement No. 786669.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] Accuvant. Browser Security Comparison. `https://accuvantstorage.blob.core.windows.net/web/files/AccuvantBrowserSecCompar_FINAL.pdf`, 2011.

[3] Adobe. ActionScript Virtual Machine. `https://github.com/adobe-flash/avmplus`, 2013.

[4] David Anderson. IonMonkey in Firefox 18. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals`, 2012.

[5] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L Schuff, David Sehr, Cliff L Biffle, and Bennet Yee. Language-independent Sandboxing of Just-In-Time Compilation and Self-modifying Code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[6] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[7] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[8] Pete Beck. JIT Spraying on ARM. `https://prezi.com/ih3ypfivoeeq/jit-spraying-on-arm/`, 2011.

[9] Dionysus Blazakis. Interpreter Exploitation. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2010.

[10] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[11] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming without Returns. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[12] Ping Chen, Yi Fang, Bing Mao, and Li Xie. JITDefender: A Defense Against JIT Spraying Attacks. In *IFIP Information Security Conference and Privacy Conference*, 2011.

[13] Ping Chen, Rui Wu, and Bing Mao. JITSafe: A Framework Against Just-In-Time Spraying Attacks. *IET Information Security*, 2013.

[14] Yuki Chen. Exploit Your Java Native Vulnerabilities on Win7/JRE7 in One Minute. `https://bit.ly/2KZlSr5`, 2013.

[15] Dustin Childs. The Results - Pwn2Own 2017 Day One. `https://blog.trendmicro.com/results-pwn2own-2017-day-one/`, 2017.

[16] Dustin Childs. Pwn2Own 2018: Results from Day One. `https://www.thezdi.com/blog/2018/3/14/pwn2own-2018-results-from-day-one`, 2018.

[17] Lin Clark. A Crash Course In Just-In-Time (JIT) Compilers. `https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/`, 2017.

[18] Willem De Groef, Nick Nikiforakis, Yves Younan, and Frank Piessens. JITSec: Just-In-Time Security for Code Injection Attacks. In *Benelux Workshop on Information and System Security, WisSec*, 2010.

[19] Jan de Mooij. W^X JIT-Code Enabled in Firefox. `https://jandemooij.nl/blog/2015/12/29/wx-jit-code-enabled-in-firefox/`, 2015.

[20] dotNet. Ryujit - Overview. `https://github.com/dotnet/coreclr/blob/master/Documentation/botr/ryujit-overview.md`, 2017.

[21] Ferris Ellis. eBPF, Part 1: Past, Present, and Future. `https://ferrisellis.com/posts/ebpf_past_present_future/`, 2017.

[22] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. JITGuard: Hardening Just-In-Time Compilers with SGX. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[23] Ivan Fratric. Bypassing Mitigations by Attacking JIT Server in Microsoft Edge. https://googleprojectzero.blogspot.de/2018/05/bypassing-mitigations-by-attacking-jit.html, 2016.

[24] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based Just-In-Time Type Specialization for Dynamic Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[25] Robert Gawlik. From Assembly to JavaScript and Back. https://github.com/rh0dev/slides/blob/master/OffensiveCon2018_From_Assembly_to_JavaScript_and_back.pdf, 2018.

[26] Google. TurboFan. https://github.com/v8/v8/wiki/TurboFan, 2017.

[27] Google. Control-Flow Integrity. https://www.chromium.org/developers/testing/control-flow-integrity, 2018.

[28] Grsecurity. Frequently Asked Questions About RAP. https://grsecurity.net/rap_faq.php, 2018.

[29] Michael Hablich. Digging into the TurboFan JIT. https://v8project.blogspot.de/2015/07/digging-into-turbofan-jit.html, 2015.

[30] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Librando: Transparent Code Randomization for Just-In-Time Compilers. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[31] Martin Jauernig, Matthias Neugschwandtner, Christian Platzer, and Paolo Milani Comparetti. Lobotomy: An Architecture for JIT Spraying Mitigation. In *Availability, Reliability and Security (ARES)*, 2014.

[32] Sebastian Krahmer. x86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique. http://users.suse.com/~krahmer/no-nx.pdf, 2005.

[33] Wilson Lian, Hovav Shacham, and Stefan Savage. Too LeJIT to Quit: Extending JIT Spraying to ARM. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[34] Wilson Lian, Hovav Shacham, and Stefan Savage. A Call to ARMs: Understanding the Costs and Benefits of JIT Spraying Mitigations. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[35] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Pearson Education, 2014.

[36] Lokihardt. Microsoft Edge: Chakra: JIT: Magic Value Can Cause Type Confusion. https://bugs.chromium.org/p/project-zero/issues/detail?id=1531, 2017.

[37] Giorgi Maisuradze, Michael Backes, and Christian Rossow. What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In *USENIX Security Symposium*, 2016.

[38] Giorgi Maisuradze, Michael Backes, and Christian Rossow. Dachshund: Digging for and Securing Against (Non-) Blinded Constants in JIT Code. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[39] Keegan McAllister. Attacking Hardened Linux Systems with Kernel JIT Spraying. http://mainisusuallyafunction.blogspot.de/2012/11/attacking-hardened-linux-systems-with.html, 2012.

[40] Microsoft. ChakraCore - Architecture Overview. https://github.com/Microsoft/ChakraCore/wiki/Architecture-Overview, 2017.

[41] Microsoft. Control Flow Guard. https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx, 2018.

[42] Microsoft. Windows Advanced Rasterization Platform (WARP) Guide. https://sites.google.com/site/bingsunsec/WARPJIT, 2018.

[43] Matt Miller. Mitigating Arbitrary Native Code Execution in Microsoft Edge. https://blogs.windows.com/msedgedev/2017/02/23/mitigating-arbitrary-native-code-execution/, 2017.

[44] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. Opaque Control-Flow Integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[45] Ingo Molnar. Exec Shield, New Linux Security Feature. *News-Forge, May*, 2003.

[46] Mozilla. ASM.JS Working Draft. http://asmjs.org/spec/latest/, 2014.

[47] Mozilla. CVE-2017-5375: Excessive JIT code allocation allows bypass of ASLR and DEP. https://www.mozilla.org/en-US/security/advisories/mfsa2017-01/#CVE-2017-5375, 2017.

[48] Mozilla. CVE-2017-5400: ASM.JS JIT-Spray Bypass of ASLR and DEP. https://www.mozilla.org/en-US/security/advisories/mfsa2017-05/#CVE-2017-5400, 2017.

[49] Ben Niu and Gang Tan. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[50] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries. In *Annual Computer Security Applications Conference (ACSAC)*, 2010.

[51] Aleph One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 1996.

[52] Michael Pall. The LuaJIT Project. https://luajit.org/, 2018.

[53] Ming-chieh Pan and Sung-ting Tsai. Weapons of Targeted Attack. http://media.blackhat.com/bh-us-11/Tsai/BH_US_11_TsaiPan_Weapons_Targeted_Attack_Slides.pdf, 2011.

[54] PaX Team. Pageexec. https://pax.grsecurity.net/docs/pageexec.txt, 2001.

[55] PaX Team. Documentation for the PaX Project. https://https://pax.grsecurity.net/docs/index.html, 2015.

[56] Jannik Pewny and Thorsten Holz. Control-Flow Restrictor: Compiler-Based CFI for iOS. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[57] Filip Pizlo. Introducing the WebKit FTL JIT. https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/, 2014.

[58] Jordan Rabet. V8 JIT Escape Analysis Bug. https://bugs.chromium.org/p/chromium/issues/detail?id=765433, 2017.

[59] Jordan Rabet. Browser Security Beyond Sandboxing. http://www.bluehatil.com/files/browser%20security%20beyond%20sandboxing.pdf, 2018.

[60] Elena Reshetova, Filippo Bonazzi, and N. Asokan. Randomization can't Stop BPF JIT Spray. https://www.blackhat.com/docs/eu-16/materials/eu-16-Reshetova-Randomization-Can't-Stop-BPF-JIT-Spray-wp.pdf, 2016.

[61] Rh0. Firefox 50.0.1 - ASM.JS JIT-Spray Remote Code Execution. https://www.exploit-db.com/exploits/42327/, 2017.

[62] Rh0. Firefox 44.0.2 - ASM.JS JIT-Spray Remote Code Execution. https://www.exploit-db.com/exploits/44294/, 2018.

[63] Rh0. Firefox 46.0.1 - ASM.JS JIT-Spray Remote Code Execution. https://www.exploit-db.com/exploits/44293/, 2018.

[64] Chris Rohlf. It's 2010 and Your Browser Has an Assembler. http://em386.blogspot.com/2010/06/its-2010-and-your-browser-has-assembler.html, 2010.

[65] Chris Rohlf and Yan Ivnitskiy. Attacking Clientside JIT Compilers. *BlackHat USA*, 2011.

[66] Mark Russinovich, David Solomon, and Alex Ionescu. *Windows Internals, Part 2*. Microsoft Press, 2012.

[67] Fermín J. Serna. Flash JIT-Spraying for Info Leak Gadgets. http://zhodiac.hispahack.com/my-stuff/security/Flash_Jit_InfoLeak_Gadgets.pdf, 2013.

[68] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (On the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[69] Alexey Sintsov. JIT-Spray Attacks & Advanced Shellcode. https://bit.ly/2rMAR0p, 2010.

[70] Alexey Sintsov. Writing JIT-Spray Shellcode for Fun and Profit. https://dl.packetstormsecurity.net/papers/shellcode/Writing-JIT-Spray-Shellcode.pdf, 2010.

[71] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy (SP)*, 2013.

[72] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. Exploiting and Protecting Dynamic Code Generation. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[73] Facebook Open Source. Moving Fast with High Performance Hack and PHP. https://hhvm.com/, 2018.

[74] Brad Spengler. Linux Kernel BPF JIT Spraying. https://forums.grsecurity.net/viewtopic.php?f=7&t=4463, 2016.

[75] Jason Spielman. Deconstructing a Winning WebKit Pwn2Own Entry. https://www.thezdi.com/blog/2017/8/24/deconstructing-a-winning-webkit-pwn2own-entry, 2017.

[76] Bing Sun and Chong Xu. JIT Spraying Never Dies - Bypass CFG By Leveraging WARP Shader JIT Spraying. https://sites.google.com/site/bingsunsec/WARPJIT, 2016.

[77] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (SP)*, 2013.

[78] PaX Team. Address Space Layout Randomization. https://pax.grsecurity.net/docs/aslr.txt, 2001.

[79] Theori. Chakra JIT CFG Bypass. https://theori.io/research/chakra-jit-cfg-bypass, 2016.

[80] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security Symposium*, 2014.

[81] Luke Wagner. ASM.JS in Firefox Nightly. https://blog.mozilla.org/luke/2013/03/21/asm-js-in-firefox-nightly/, 2013.

[82] Tao Wei, Tielei Wang, Lei Duan, and Jing Luo. INSeRT: Protect Dynamic Code Generation Against Spraying. In *2011 International Conference on Information Science and Technology (ICIST)*, 2011.

[83] Rui Wu, Ping Chen, Bing Mao, and Li Xie. Rim: A Method to Defend from JIT Spraying Sttack. In *Availability, Reliability and Security (ARES)*, 2012.

[84] Yang Yu. Bypass DEP and CFG Using JIT Compiler in Chakra Engine. https://xlab.tencent.com/en/2015/12/09/bypass-dep-and-cfg-using-jit-compiler-in-chakra-engine/, 2015.

[85] Chao Zhang, Mehrdad Niknami, Kevin Zhijie Chen, Chengyu Song, Zhaofeng Chen, and Dawn Song. JITScope: Protecting Web Users from Control-Flow Hijacking Attacks. In *IEEE Conference on Computer Communications (INFOCOM)*, 2015.

[86] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control-Flow Integrity and Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy (SP)*, 2013.

[87] Mingwei Zhang and R. Sekar. Control-Flow Integrity for COTS Binaries. In *USENIX Security Symposium*, 2013.