The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines

Michalis Athanasakis
FORTH, Greece
michath@ics.forth.gr

Elias Athanasopoulos
FORTH, Greece
elathan@ics.forth.gr

Michalis Polychronakis
Stony Brook University
mikepo@cs.stonybrook.edu

Georgios Portokalidis
Stevens Institute of Tech.
gportoka@stevens.edu

Sotiris Ioannidis
FORTH, Greece
sotiris@ics.forth.gr

***Abstract*—Return-oriented programming (ROP) has become the dominant form of vulnerability exploitation in both user and kernel space. Many defenses against ROP exploits exist, which can significantly raise the bar against attackers. Although protecting existing code, such as applications and the kernel, might be possible, taking countermeasures against dynamic code, i.e., code that is generated only at run-time, is much harder. Attackers have already started exploiting Just-in-Time (JIT) engines, available in all modern browsers, to introduce their (shell)code (either native code or re-usable gadgets) during JIT compilation, and then taking advantage of it.**

Recognizing this immediate threat, browser vendors started employing defenses for hardening their JIT engines. In this paper, we show that—no matter the employed defenses—JIT engines are still exploitable using solely dynamically generated gadgets. We demonstrate that dynamic ROP payload construction is possible in two modern web browsers without using any of the available gadgets contained in the browser binary or linked libraries. First, we exploit an open source JIT engine (Mozilla Firefox) by feeding it malicious JavaScript, which once processed generates all required gadgets for running any shellcode successfully. Second, we exploit a proprietary JIT engine, the one in the 64-bit Microsoft Internet Explorer, which employs many *undocumented, specially crafted* defenses against JIT exploitation. We manage to bypass all of them and create the required gadgets for running any shellcode successfully. All defensive techniques are documented in this paper to assist other researchers. Furthermore, besides showing how to construct ROP gadgets on-the-fly, we also show how to *discover* them on-the-fly, rendering current randomization schemes ineffective. Finally, we perform an analysis of the most important defense currently employed, namely *constant blinding*, which shields all three-byte or larger immediate values in the JIT buffer for hindering the construction of ROP gadgets. Our analysis suggests that extending constant blinding to all immediate values (i.e., shielding 1-byte and 2-byte constants) dramatically decreases the JIT engine's performance, introducing up to 80% additional instructions.

I. Introduction

Web browsers are undoubtedly omnipresent. They are found on PCs, smartphones, tablets, smart TVs, gaming consoles, and elsewhere. Most Internet users probably use a browser every day. Even users that prefer apps, instead of a general purpose browser, unknowingly interact with browser components frequently used by app developers [1]. Their popularity is probably one of the reasons that they are such an attractive target for attackers and security researchers alike [2]–[4].

A. Problem Statement

Attacks against browsers continue despite the fact that compromising binary software using buffer overflows and control-hijacking attacks is much harder today. Modern operating systems (OSs) include features like stack canaries [5], non-executable pages [6], and address-space layout randomization (ASLR) [7], which severely hinder exploitation. Even code-reuse techniques such as return-oriented programming (ROP) [8] are not straightforward, since they require information-leak bugs to reveal the randomized location of code [9], [10] or legacy code and libraries that ASLR cannot randomize.

Recent works on control-flow integrity (CFI) [11], [12], fine-grained code randomization [13]–[15], and run-time behavioral monitoring [16], [17] promise to protect software from ROP-like attacks, but unfortunately they have been also shown to be vulnerable to niche attacks [9], [18], [19]. Other approaches that require application source code, such as modular fine-grained CFI [20] and G-Free [21], offer greater guarantees again control-flow hijacking attacks and ROP. So far, there are no documented attacks against these defenses, so, in principle, they could protect our precious browsers in the future, even though one cannot make strong predictions.

Unfortunately, even defenses that may be effective for conventional software are not always so for browsers. Modern browsers dynamically generate code through just-in-time (JIT) compilation to accelerate the execution of JavaScript (JS) code at run time. Although defenses like the ones discussed above can be efficient in protecting existing code, code generation is frequently not handled and it is outside their threat model.

Attacks exploiting the JIT engines of browsers are not new. Figure 1 depicts the evolution of attacks and defenses against them. Originally, code and data were not separated by the code-generation engine, so both the generated native code and the data it was operating on was placed on the same executable memory pages. It was enough for the attacker to place shellcode in a JavaScript array and then redirect the program's control-flow to his shellcode in memory. Because

NDSS '15, 8-11 February 2015, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2015.23209

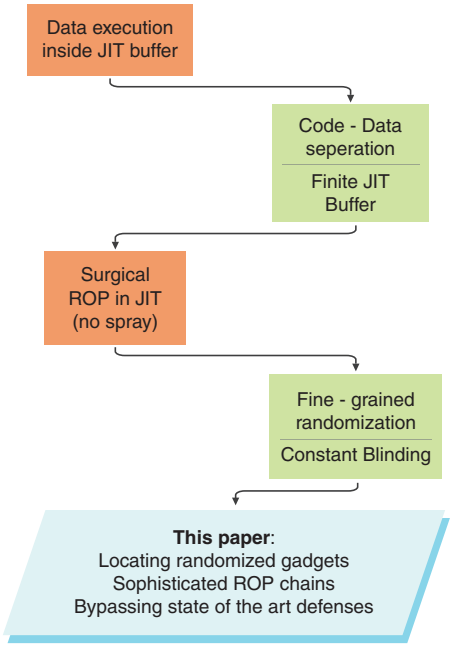

Fig. 1. Evolution of attacks and defenses in browser JIT engines.

ASLR was already in place, attackers created many copies of the data array, which resulted in one of the JIT buffers being allocated in a relatively predictable memory location, where they could transfer control. This technique is commonly referred to as JIT spraying [22].

JIT spraying attacks were countered by adopting two simple strategies, which can be deployed independently. First, JavaScript data and code is separated, placing the first in non-executable memory pages. Second, the amount of memory dedicated to generated code is limited (i.e., the JIT buffer is finite) to prevent the placement of buffers at predictable offsets. In response, attackers invented new techniques, building ROP payloads solely with code generated by the JIT engine [23]. Briefly, this is achieved by using four or eight byte constants within JavaScript code, which are emitted as immediates in the generated code. The attacker can carefully select these constants to build small instruction segments, called gadgets, and link them in ROP fashion.

The most recent countermeasures against JIT exploitation have been adopted by Internet Explorer's Chakra engine. First, it interleaves `NOP` instructions in random locations in the generated code to randomize the location of any gadgets. Second, it applies *constant blinding* on constant values larger than two bytes. This means, that a constant value will never appear in the JIT buffer as is, but as the `XOR` product of itself with a random value. Of course, this does not come for free, but requires emitting two additional machine instructions for *unblinding* constants during execution, i.e., to restore the original value of the constant. These two countermeasures along with some others, which for brevity we do not discuss here but in Sec. V, make JIT exploitation *significantly* harder.

It is crucial to question: are the above defenses sufficient for preventing browser exploitation through the JIT engine?

B. Our Approach

This paper performs a security analysis of the latest JIT engines used by Mozilla and Internet Explorer, attempting to answer the above question. Our findings indicate that in both cases it is possible to introduce *new* gadgets within the browsers, and we can use these gadgets to construct useful payloads without using any gadgets from the browsers and their libraries. This means that despite what defenses are employed to prevent exploitation of the browser, the JIT engine is still its Achilles heel.

We demonstrate our findings by building ROP payloads in Mozilla on 32-bit Linux and Internet Explorer (IE) on 64-bit Microsoft Windows 8. In the first case, we confirm that separating JavaScript data from code and using finite memory for JIT buffers is vulnerable to ROP attacks. In the latter, we first reverse engineer all the JIT-related defenses used by IE, and then compose a payload that bypasses all imposed restrictions. To utilize our payloads, we inject vulnerabilities in the two browsers based on an older Internet Explorer vulnerability [2], since at the moment of writing there are no publicly available exploits for the browsers we used. Using this realistic vulnerability, we show how a determined attacker can locate the JIT buffer with high accuracy, overcoming randomization-based defenses. Finally, we propose various modifications that could severely impede the attacker's options when exploiting JIT engines.

This paper makes the following contributions:

- We leverage the JIT engines in two of the most popular web browsers, Mozilla Firefox and Internet Explorer (perhaps, the most attacked one), for constructing ROP gadgets inside the JIT buffer. Using these gadgets we can successfully change the access permissions and make the page holding the shellcode executable.
- Contrary to previous attacks that use JIT spraying [22], we do not place a shellcode in JavaScript data, as the new JIT engines place all data into non-executable pages. Instead, we introduce ROP gadgets using more sophisticated techniques.
- We conducted the first extensive analysis of the undocumented defense mechanisms against the construction of ROP gadgets in the JIT buffer, employed by the IE JIT engine, Chakra. Not only did we discover and document these defense mechanisms to assist other researchers that want to understand the Chakra engine, but we also proposed and implemented ways to successfully bypass them.
- The attacks we introduce in this paper cannot be stopped using code diversification (e.g., using Librando [24]). Based on published research [9], we can discover the JIT buffer even if it is *randomized*, and progressively discover

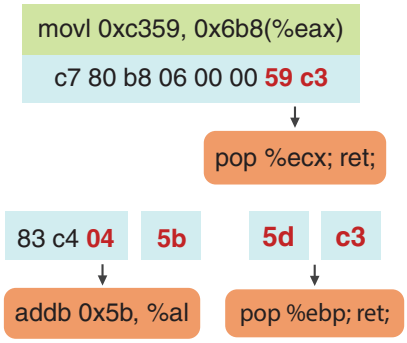

Fig. 2. Example of small code sequences (gadgets) that can be chained for performing computations controlled by an adversary.

the constructed gadgets on-the-fly before launching the actual attack.

C. Organization

The rest of the paper is organized as follows. In Section II we present all essential background information required for understanding this paper. We explain what gadgets and a JIT compiler are. In Section III we discuss the assumptions we make for this attack to happen, as well as the setup where this attack is really important. We demonstrate the attack in two popular JIT engines, the one incorporated in Mozilla Firefox and the one incorporated in Internet Explorer (see Sections IV and V). We then discuss how gadgets are located in Section VI and possible defenses in Section VII. We present prior work in Section VIII and conclude in Section IX.

II. Background

In this section we discuss some background information. The reader needs to become familiar with two things: gadgets and JIT compilers.

A. ROP Gadgets

Simple code injections cannot easily be used in attacks nowadays as most operating systems and architectures support non-executable data. Therefore, the only avenue for exploitation is *code-reuse*. This means that the attacker must take advantage of existing functionality available in the vulnerable program to exploit it. A popular technique to do so is return-oriented programming (ROP). We assume that the attacker has program control, for example they can control the contents of a function pointer (or return address which is essentially equivalent), but they cannot redirect control to the shellcode (the part that does the actual compromising, like opening a root shell). Instead, the attacker finds small sequences of useful code, which usually end with a `ret` instruction and chains them together. The `ret` instruction is important, as the attacker needs to execute a few of these sequences in a row using a virtual stack, but we know that `ret` can be also simulated using jumps [25], [26].

These small sequences are called *gadgets* and can be found *anywhere* in a program. In fact, in CISC architectures such as x86, where instructions have variable length, the attacker can jump in-between two instructions and form a new (overlapping) instruction. Gadgets can do small tasks, such as load a register with a particular value, but *if* combined correctly they can perform any computation. In fact, it has been shown that ROP is Turing Complete [8]. In Figure 2 we depict a few example gadgets.

B. JIT Compilation

Scripts expressed in high-level languages usually run inside an interpreter. For example, a browser embeds a JavaScript interpreter for evaluating and running JavaScript code. This is significantly slower than executing native code. Since JavaScript programs have become complex and large, it often makes sense to try to compile them into native code, at least the parts that conduct heavy computation. Compiling does not happen ahead of time, like it happens when programs are compiled, but *just-in-time* (JIT), exactly when the JavaScript interpreter decides that a particular part of code will be executed repeatedly. The program that compiles a part of JavaScript into native code is called a JIT compiler.

JIT compilers are found in many applications. In this paper we use browsers as examples, since they are prime targets for attackers. Our techniques however, are general and should be applicable against many JIT architectures.

III. Threat Model

We now define the conditions under which the attack is possible and our assumptions regarding the attacker's capabilities. Both are on par with recent advances in attacks and defenses [9], [11], [16], [18].

A. Active Defenses

We assume that popular defenses, already present in most popular modern systems, are in effect. Data execution prevention (DEP) [6] is active, preventing the direct injection and execution of native code into vulnerable applications. DEP essentially ensures that no page is both writable and executable, otherwise referred to as preserving $W \oplus X$ semantics. Address-space layout randomization (ASLR) [7] is also enabled, ensuring that the main binary image and the shared libraries used by the vulnerable application are loaded at a different, randomly selected virtual address every time it executes. The goal of ASLR is to introduce uncertainty and turn the target application into a moving target for the attacker, who can no longer make assumptions on where a particular library, and consequently a function or code block, resides within the address space of an application. Stack smashing protection may also be present in the form of stack canaries or function cookies [5], as well as security toolkits like EMET [27].

Moreover, we assume that the vulnerable software has been hardened against code-reuse attacks [13], [16], [21]. For example, we assume that the binary and its libraries have been compiled with G-free [21], a compiler framework that produces binaries without gadgets, which enable powerful code-reuse attacks such as the ones based on return-oriented programming (ROP) [8]. In other words, composing a ROP or other type of core-reuse payload [26] for the vulnerable application using its code alone is *hard*. Note that such defenses have been proposed in literature, but have not been broadly adopted (yet).

Browser-specific defenses against code-reuse attacks using the code generated by the JIT compiler are also active. For instance, the JIT engine included in Internet Explorer (IE) since version 9, codenamed Chakra, includes certain counter-measures, aiming at countering attacks like the one introduced in this paper.

We discuss Chakra and how we bypass it later in this paper, after presenting details on IE's JIT engine and the attack itself. Librando [24] also provides many defense schemes and shares many common defenses with Chakra. We consider the common defenses offered by these two to be equivalent.

B. Attacker Capabilities

Our assumption is that the attacker is skilled and can launch complex attacks against the vulnerable application, such as the ones already demonstrated by researchers and security analysts [3], [4], [28]. Consequently, the attacker can bypass stack canaries, EMET, and similar defenses to alter the vulnerable program's control flow by controlling an indirect control-flow instruction. Most importantly, ASLR can be defeated, either through a memory disclosure bug [29], or by forcing the vulnerable application to place attacker data (the generated code by the JavaScript JIT engine in our case) in predictable locations. The latter is comparable with heap spraying [30], where the attacker allocates many copies of their data in an attempt to ensure that one of the copies lands at a predictable memory address in the vulnerable program's heap. While this assumption may appear as overreaching, recent prominent work [9] has shown that such memory disclosure attacks are both powerful and feasible, and they can overcome even highly dynamic randomization schemes [15].

C. Is this attack unstoppable?

The attack presented in this paper mainly targets binaries hardened with G-free [21]. To this end, we assume that the only effective defense mechanism enabled in the system is realized through gadget-free binaries. However, the fact that we (i) introduce and (ii) discover the gadgets at run-time automatically nullifies many other protections which are either based on off-line code analysis (nullified by (i)) or software diversification (nullified by (ii)). Therefore, all CFI-based techniques [11], [12] can be bypassed as well, unless the JITed code is also analyzed and CFI is applied to it. The latter needs further exploration as the performance penalty introduced by the analysis might nullify the gain from JIT compilation. Randomization schemes [7], [13], [14], [24] can also be bypassed as we discover the gadgets at run-time. Finally, there is the category of run-time monitoring tools, which include ROPecker [17] and kBouncer [16]. ROPecker needs off-line analysis of the code, which makes it weak against our attack. The only mitigation technique that can potentially detect the attack is kBouncer. Even though it has been shown that kBouncer can be bypassed [31], [32], it remains unclear if the attack presented here can be expressed with the particular gadget types that evade the tool. This requires further research, which we leave as future work.

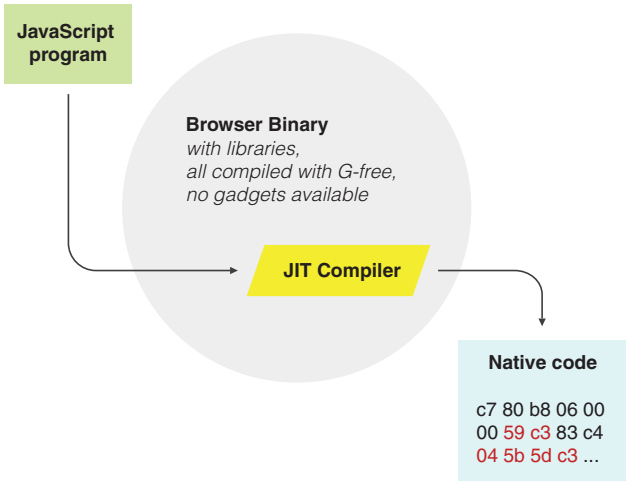

Fig. 3. Illustration of the attack presented in this paper. A browser, which we assume contains no ROP gadgets, is forced to JIT compile a JavaScript program. The JavaScript is carefully written in a way that, once compiled, ROP gadgets will manifest in the JIT buffer.

/.

IV. Exploiting Mozilla Firefox

```
1 pop %ebx; ret;
2 pop %ecx; ret;
3 xor %eax, %eax; ret;
4 mov 0x7d, %al; ret;
5 xor %edx, %edx; ret;
6 mov 0x7, %dl; ret;
7 int 0x80; ret;
```

Listing 1. Required gadgets for calling `mprotect` in Linux (32-bit).

```
1  var g1 = 0;
2  ...
3  var g7 = 0;
4
5  for (var i=0; i<100000; ++i) {
6          g1 = 50011;      \\ pop ebx; ret;
7          g2 = 50009;      \\ pop ecx; ret;
8          g3 = 12828721;   \\ xor eax, eax; ret;
9          g4 = 12811696;   \\ mov 0x7d, al; ret;
10         g5 = 12833329;   \\ xor edx, edx; ret;
11         g6 = 12781490;   \\ mov 0x7,  dl; ret;
12         g7 = 12812493;   \\ int 0x80; ret;
13 }
```

Listing 2. The JavaScript program which once compiled will produce the needed gadgets in the JIT buffer.

In this section we present how we exploit a vulnerable version of Mozilla Firefox in Linux (32-bit) without using any of the available gadgets contained in its binary or shared libraries. For this particular study we use SpiderMonkey, the JavaScript engine of Mozilla Firefox, which incorporates IonMonkey, the JIT engine, version 1.85.

Figure 3 shows a high-level overview of the attack. Briefly, the steps required to launch this attack are:

1) The browser renders a malicious web page.
2) JavaScript contained in this page, once compiled, produces a series of gadgets in the JIT buffer.
3) A memory-disclosure bug reveals the locations of the gadgets [9].
4) A ROP chain is built with the just constructed gadgets and control is transferred to it.
5) The ROP chain calls `VirtualProtect` (or `mprotect`, depending on the platform) for making a data page (where the shellcode is stored) executable.
6) Control is transferred to the shellcode and the browser is compromised.

A. Preparation

The goal is to force the vulnerable web browser to load a JavaScript program, which, once JIT-compiled, will produce a series of gadgets that will eventually call `mprotect` making a data page hosting a shellcode executable. The gadgets needed for this purpose are presented in Listing 1. We must control four registers to invoke `mprotect`. To do so, we first store `0xb`, the system call number of `mprotect`, in `%eax`. We then store the address of the page we want to change the permission of, in `%ebx`. The length of the region is stored in `%ecx`. Lastly, the desired access rights, in our case `0x7` for read, write and execute, are stored in `%edx`. To accomplish this we need a ROP chain which will load the four registers with the required values and then a gadget which will call `mprotect`. In Linux for x86 architectures, system calls are performed by using the `int 0x80` instruction.

Loading a register with the required value can be done in two ways. The first is to place the desired value on the stack and use `pop` with the register name, and the second is to use a `mov`-like command to *copy* the value to the register. As we will show later in this section, the gadgets are constructed via the emitted code from JITed JavaScript. This approach significantly constrains us in the construction of opcodes for `mov`-based gadgets with large values. Therefore we used the stack and the `pop` instruction when storing a large value to a register, and a `mov` instruction when loading a register with a small value. In the second case, where a `mov` is used, it affects only part of the register and therefore we need to zero the register before moving the value, which we do by using an `xor` gadget.

Based on the above, the ROP chain for calling `mprotect` (see Listing 1) includes seven gadgets which work as follows. The first two gadgets (lines 1 and 2) are two `pop` gadgets for storing the page address and region length in `%ebx` and `%ecx` respectively. The following two gadgets (lines 3 and 4) are zeroing `%eax` and copying the value of `0x7d` (the system call number of `mprotect`) to it, and the next two (lines 5 and 6) copy the value `0x7`, for enabling permissions to read, write, and execute a page, to `%edx`. Finally, the last gadget (line 7) calls `mprotect`.

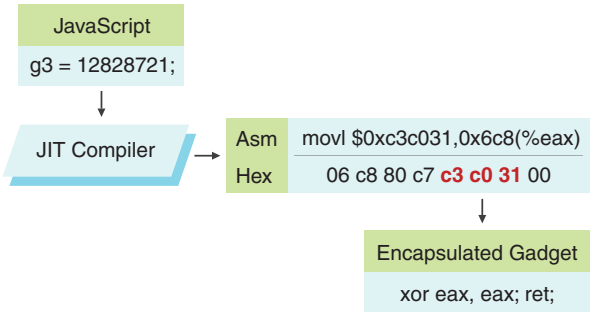

Fig. 4. JavaScript snippet which once compiled a particular assembly instruction sequence manifests on memory. This sequence can later be used in a ROP chain for exploiting the browser.

B. Exploit Implementation

Now that we have presented the required gadgets for executing a shellcode, we will discuss how we create them in a gadget-free environment. Recall that we assume that the binary and all shared libraries contain no gadgets. This means that none of the gadgets belonging to the ROP chain of Listing 1 can be located in existing code, even if a sophisticated gadget finder [33] is employed. Notice also that typically ROP exploitation needs some form of memory disclosure because of available defenses based on randomization [7], [13], [14]. However, these requirements are orthogonal to the techniques presented in this paper. First, it has been shown that even fine-grained randomization schemes can be defeated if the vulnerability allows arbitrarily reading process memory [9], and second, in our setup the attacker must overcome an even stronger defense mechanism: a gadget-free environment. Thus, the attacker needs to *first create* the gadgets and *locate* them inside the *JIT buffer* before creating the ROP chain and executing it.

To introduce the ROP chain of Listing 1 in an executable page, we leverage the browser's JIT engine. We specially craft a JavaScript program which triggers the JIT engine and once compiled the desired gadgets will appear in an executable page. Accomplishing this requires two things: (i) a way to trigger the JIT compiler, and (ii) a way to influence the output of the JIT compiler so that the desired gadgets will be created in memory. As far as (i) is concerned, we use JavaScript loops to increase the compute load and therefore trigger the JIT engine. To accomplish (ii) we use variable initializations with specially crafted immediate values which encapsulate the opcodes of the desired gadgets. Once these immediate values appear in the JIT buffer we can jump on them and execute the

encapsulated gadget.

Listing 2 shows a JavaScript program, which once executed, generates the gadgets required to compromise Mozilla Firefox. To do this, we first declare seven variables (lines 1-3). Each variable is carefully initialized to *host* a gadget. The initialization takes place inside a long loop to trigger the JIT engine. In Figure 4 we show how we influence the JIT output by assigning particular immediate values to JavaScript variables. For example, assigning the value `12728721` to a variable will introduce the following assembly code once compiled:

```
movl $0xc3c031, 0x6c8(%eax)
```

In hex this has the value of `0x06c8080c7c3c03100` which includes `0xc3c031`, which is a gadget for zeroing %eax:

```
xor %eax, %eax; ret;
```

In the same fashion we can construct all of the gadgets contained in the ROP chain of Listing 1 and eventually call `mprotect` for making the shellcode executable.

V. Exploiting Internet Explorer

```
1 pop %r8; ret;
2 pop %r9; ret;
3 pop %rcx; ret;
4 pop %rdx; ret;
5 pop %rax; ret;
```

Listing 3. Required gadgets for calling `VirtualProtect` in Windows (64-bit).

In this section we present how we exploit a vulnerable Internet Explorer (IE) in Microsoft Windows (64-bit) without using any of the available gadgets contained in the binary or DLLs used by the browser.

A. Why Internet Explorer is different

Version 9 of IE has started employing a JavaScript JIT engine called Chakra [34]. As IE is proprietary, little is known about its internals, and in particular how the JIT engine works. There are several issues that make carrying out an attack as the one presented in Section IV for IE significantly harder. First, lack of source code makes understanding how the JIT engine is triggered, where the JIT buffer is located, and other related detail important for exploiting the engine, very difficult. Second, Chakra employs a series of defenses specifically introduced for preventing the generation of gadgets in the JIT buffer. Third, we want to exploit the 64 bit version of IE, which changes things in terms of calling conventions, as *fastcall* is used, and the first function arguments are passed through registers and not through the stack. In the rest of the section we describe how we overcome these difficulties.

B. Preparation

As before, to exploit IE we must make the page that holds the shellcode executable. This means that we need to call `VirtualProtect` with the appropriate arguments, and to accomplish this we must use the gadgets that will be introduced in the JIT buffer, once a properly crafted JavaScript program is compiled. The calling convention of `VirtualProtect` in Windows is the following. The function takes 4 arguments using the `%rcx`, `%rdx`, `%r8`, and `%r9` registers. Therefore, assuming we control the stack, we need to introduce the gadgets shown in Listing 3. In Listing 3 we include an additional gadget, which `pops %rax`. This gadget is not needed for calling `VirtualProtect` but for breaking a defense mechanism employed by Chakra as we will discuss later.

Apart from the gadgets we need for calling `VirtualProtect`, we also need an additional gadget for adjusting the stack. Usually, the vulnerability is related to the heap, therefore we need to adjust the real stack to the fake stack controlled by us, something that we commonly call *stack pivoting*. We avoided discussing the stack-pivoting gadget in Section IV, since in the case of Mozilla this gadget can be constructed trivially. Constructing the stack-pivoting in IE is usually based on exchanging a register the attacker controls with `%rsp`, so that the stack pointer points to the attacker's fake stack. This exchange can be done using `xchg`, which unfortunately is 2 bytes long, and with the additional `ret` instruction becomes a 3-byte gadget. As we show later in this section 3-byte gadgets cannot be constructed trivially (see "Long gadgets" later in this section). For constructing the stack-pivoting gadget we need an additional requirement: having control over `%al`. The reason is discussed later in this section.

C. Exploit Design Considerations

Similarly to the approach we took in Section IV, we started with a compute-heavy loop to trigger the JavaScript JIT compiler and a series of variable initializations, to introduce the desired gadgets in the JIT buffer once the loop is compiled. However, IE is very different from Mozilla and such an approach failed. IE's JIT engine, Chakra, employs a number of defenses which makes introducing gadgets in the JIT buffer through immediate values in the JavaScript source impossible. To make our attack work we had to reverse engineer Chakra's defenses. We will discuss some of these defenses here and how we were able to circumvent them.

a) Constant Blinding: Any immediate value less than 2 bytes long is never emitted as is in the JIT buffer. Instead, it is XORed with a random value and then XORed again when it is actually used. For example, assume the following JavaScript code:

```
var gadget = 0xc35841;
```

Once it is compiled, we would normally expect to see the following code in the JIT buffer:

```
mov %rcx, 1000000c35841h
mov qword ptr [rax+48h], %rcx
```

This code essentially puts the (immediate) value `0xc35841` in `%rcx`, which we assume is the register that holds the value of the JavaScript variable `gadget`. This reflects the example we discussed in Figure 4, where an immediate value (in our

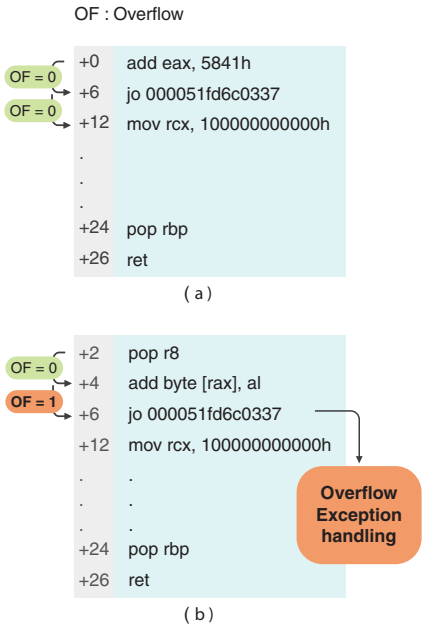

Fig. 5. JavaScript functions once compiled by Chakra embed a conditional jump in case the overflow bit is on. If an attacker tries to jump inside the function arguments, which as immediate values may encapsulate gadgets (see Figure 4), the overflow bit will be set and thus the flow will follow the conditional jump (the program will be terminate).

case `0xc35841`) encapsulates a hidden gadget (in our case the gadget `pop %r8;ret`). This is prevented in Chakra by never emitting the immediate as is, but instead producing code would generate the value using the boolean expression `XOR`:

```
mov %rcx, 3BF43B1820E7ED7Dh
mov %rdx, 3BF53B182024B53Ch
xor %rcx, %rdx
mov qword ptr [%rax+48h], %rcx
```

Notice, how `0xc35841` is never present in the JITed code, instead Chakra does the following. First, it places `0xc35841` XORed with a random value to register %rcx, then it places the random value to %rdx, and finally it XORs %rcx and %rdx, to generate the initial immediate value. As the value never appears in the JIT buffer as is, our encapsulated gadget is useless. This means that *only* gadgets with opcodes of 2-byte length can be constructed in the fashion we described in Section IV. All other gadgets must be constructed using a different technique.

b) Long Gadgets: Since gadgets longer than 2 bytes cannot be encapsulated in immediate values, the gadget must be broken into two parts: the `pop` part which loads the register with the desired value and the `ret` part (1 byte). Breaking the gadget in two means that we are emitting the first part encapsulated in an immediate and we expect that the flow—if it starts executing the gadget—reaches a `ret` instruction. A possible avenue is a JavaScript function, which, if called with the right arguments, can emit immediate values that encapsulate gadgets of maximum 2-byte length in the JIT

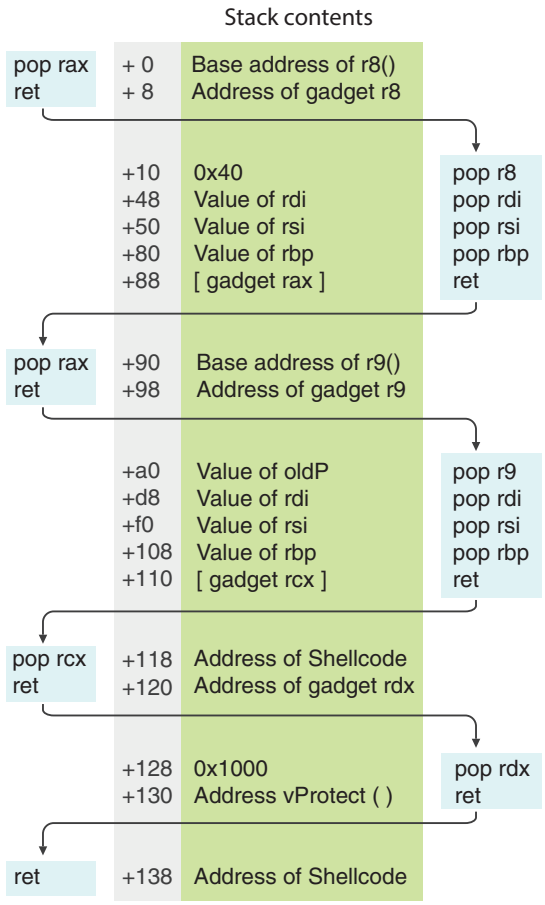

Fig. 6. The stack during the attack against IE.

buffer and can eventually reach a `ret` instruction, since the function will return. However, such an approach is far from trivial.

Consider for example the following JavaScript code:

```
function f(addr) {
    return addr + 0x5841;
}
f(0);
```

We show the compiled version of this code in Figure 5. First, note that the immediate value (`0x5841`) which holds an encapsulated gadget is added to `%eax`. Initially, this seems promising, since by jumping two bytes further we can start executing from the immediate value `0x5841` which translates to `pop %r8` (one of the critical gadgets for calling `VirtualProtect`). We show how the code looks like if we start executing from two bytes further in Figure 5. Then, note that the addition of:

```
add %eax, 5841h
```

has been replaced with:

```
pop %r8
add byte [%rax], %al
```

The important part is that Chakra has placed a conditional jump which is followed if the overflow bit is set:

```
jo 000051fd6c0037
```

Notice, that the addition following the `pop` instruction sets the overflow bit and thus the flow follows the conditional jump which executes an access violation handler. To overcome this we need to achieve two things:

- Make sure that the code between the partially emitted gadget (the `pop`-part) *does not* alter the exploit's logic (i.e., does not modify any of the registers from Listing 3).
- Somehow unset the overflow bit before the conditional jump.

In one special case, for constructing the stack-pivoting gadget (which is a 3-byte gadget), it is sufficient to control `%rax` (specifically guarantee that its low part, `%al`, has a zero value), and thus avoid raising the overflow flag. This is why we need to control `%rax` for exploiting IE, as we described in the beginning of this section.

c) Code Diversification: Chakra adds another diversification layer in the JIT buffer by emitting a random number of `nop` instructions. These instructions perform no useful computation, however they change the layout of the JIT buffer, and therefore, all important gadgets have a different location every time they are generated. This particular technique, inserting random `nop` instructions, has been also used for diversifying the Linux kernel layout [35]. Software diversification [36] has been a promising defense mechanism against exploitation, and we have seen it applied with many different strategies [7], [13], [14], as well as used for preventing the attacks we discuss in this paper [24]. Unfortunately, recently we have seen at least two sophisticated techniques [9], [10], that can bypass fine-grained randomization methods by exploiting information-leak bugs. Our work here is *about generating the gadgets in a heavily defended environment, such as Chakra, and not on techniques for discovering the process layout*. Recall that with the wide adoption of ASLR all exploits need *at least* one information-leak bug for discovering the position of the needed gadgets. The amount of information leakage depends of course on the nature of the vulnerability.

D. Exploit Implementation

```
1  function r8(addr) {
2      return addr + 0x5841;
3  }
4
5  function r9(addr) {
6      return addr + 0x5941;
7  }
8
9  function emit_gadgets() {
10     for (i = 0; i < 0xc35841; i++) {
11         rax = 0xc358;
12         rcx = 0xc359;
13         rdx = 0xc35a;
14         r8(0);
15         r9(0);
16     }
17     return 0;
18 }
19
20 emit_gadgets();
```

Listing 4. The JavaScript program which once compiled will produce the needed gadgets in the JIT buffer of IE.

Now that we have presented the defenses employed by Chakra, we will discuss how we introduce the needed gadgets in the JIT buffer for running the exploit. As already mentioned we need to create four gadgets for loading `%rcx`, `%rdx`, `%r8`, and `%r9`, with the correct values for calling `VirtualProtect` (see Listing 3). Two of the four gadgets (the ones for `%rcx` an `%rdx`) are only 2 bytes in length and thus they can be created with the techniques we analyzed in Section IV (see lines 12 and 13 in Listing 4). The challenging part is to create the other two for loading `%r8` and `%r9`, which are longer than 2 bytes.

These gadgets are emitted in the JIT buffer using JavaScript functions. Observe lines 1–7 in Listing 4. We implemented two JavaScript functions, `r8()` and `r9()`, which simply return a fixed value added to their single argument input. These functions, once compiled, produce the following code (for example `r9()`):

```
add %eax, 5941h
jo  000000D71F8F0132
mov %rcx, 1000000000000h
or  %rax, %rcx
add %rsp, 30h
pop %rbx
pop %rsi
mov %rsp, %rbp
pop %rbp
ret
```

Now, if execution starts from the address of the immediate value (`0x5941`), a `pop %r9` will be executed and control flow will eventually reach the `ret` instruction where the (compiled) JavaScript function returns. The only problem is the conditional jump for the overflow bit which will be set. To overcome this we use an additional gadget which sets `%rax` (line 11 in Listing 4). The complete JavaScript source for introducing all needed gadgets in the JIT buffer is shown in Listing 4 and the stack, along with the way the individual gadgets are chained, is depicted in Figure 6.

VI. Discovering the Gadgets

In Sections IV and V we demonstrated how someone can introduce ROP gadgets in the JIT buffer of Mozilla Firefox and IE. However, for a successful attack, the adversary has to locate the position of each gadget in order to form the ROP chain, which will eventually compromise the vulnerable program. In this section we investigate how this can be carried out successfully. Notice, that we assume that a fine-grained randomization scheme has been enabled, like Librando [24] or Chakra.

A. How Information Leaks Work

All randomization schemes have an Achilles' heel: information leaks. An attacker can read the contents of a part of

memory and infer things about the process layout. Once this happens, the attacker can launch the actual attack, i.e. form a ROP chain based on the *discovered* gadget locations. This might sound as a complicated process, and many times it is, but it has been demonstrated that it can be achieved in practice and it can really give an attacker full control of a vulnerable process, which is randomized in fine-grain (not just shifted in memory) [9].

Let us briefly discuss the attack presented in [9], which compromises a randomized [13] but vulnerable IE. The vulnerability of IE is based on a heap overwrite. By arbitrarily increasing the length of a JavaScript string object, located on the heap, and by reading the string's contents, the attacker can read past the end of the string. Now, the attacker is able to place another JavaScript object, let's say object X, next to the string object whose length can be arbitrarily extended using the heap overwrite. Essentially, the attacker by reading the string's contents, discovers the memory layout of object X. By doing this, the attacker can discover a code pointer which points to X's *vtable*, i.e., a pointer which points to the `.text` segment. Once the attacker knows the address of a code pointer, they can write a large value to the string's length (2^{32}) and transform the heap overwrite to a generic `DiscloseByte` interface. So, the attacker can start *disclosing* memory of the `.text` segment, since they already have a pointer there. By further disclosing code, and by disassembling it at runtime, following jumps and calls, the attacker eventually discovers the entire process layout. At this point, the attacker can use a ROP compiler [33], discover gadgets, and use them to launch the attack.

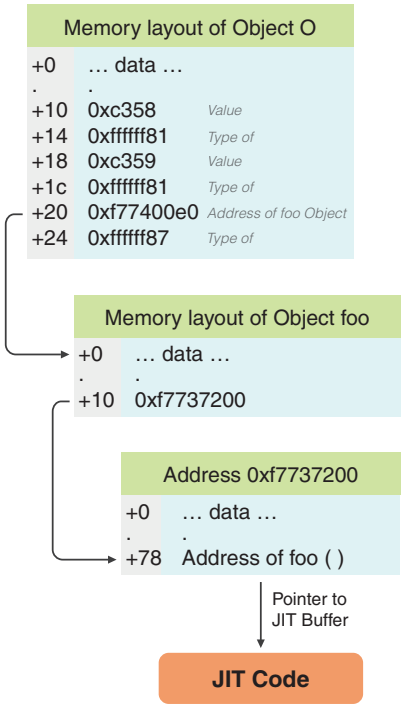

Fig. 7. The memory layout of `Object O` and `Object foo`, along with the necessary steps for locating a (code) pointer inside the JIT buffer. Once an address of the JIT buffer is known, all ROP gadgets can be discovered and the attacker can build the needed ROP chain for exploiting the browser.

B. Leaking the JIT Buffer

As we have just described above, it is possible to transform a heap overwrite to a powerful memory disclosure interface. More details on how an actual exploit works can be found here [3]. It is essential to understand that leaking the first code pointer is very important. In our case, we assume that the code segment has no useful gadgets, therefore leaking a pointer in text is of no use to us. Instead, we need to leak a pointer inside the JIT buffer. Once we do this, then the attack proceeds in the same fashion, but instead of revealing the code segment, we reveal the JIT buffer which contains the artificially constructed gadgets. Unfortunately, the particular version of IE which has the information leak bug contains no JIT engine (Chakra was introduced later) therefore we did not try to port the particular exploit [9] using our technique. However, it is hard to give assurances that new versions of IE, the ones incorporating Chakra, will not eventually suffer from such type of bugs.

Nevertheless, we provide an exploit, which is based on an information leakage vulnerability that constructs a memory disclosure interface in Mozilla Firefox. This memory disclosure interface eventually provides a code pointer which points inside the JIT buffer. The needed code is presented in Listing 5. Lines 1 through 3 create an empty JavaScript `Object O` and later fill it by creating two properties, `O.g1` and `O.g2`. Line 4 creates a simple `function foo`. In Line 5 `function foo` is assigned to a property of `O` named `func`.

Now, we have to discover the memory layout of `Object O` in order to find a way to the JIT buffer. Figure 6 shows the layout of `Object O` and how each property is aligned in memory. The simple arithmetic values are stored directly inside the object followed by a value showing their type (`0xffffff81`). The function pointer `func(+20)` points to the location of the function object. For discovering the location of the JIT buffer, we can follow the pointer and therefore land inside `Object foo`. At a specific offset from the start of the object there is a pointer that points to another data address inside the object. To that address and at a specific offset, in this example `+78`, there is a pointer that finally points to the JIT buffer. Therefore, by following three pointers, we have managed to disclosure an address inside the JIT buffer. More precisely the final disclosed (code) pointer is the starting address of `function foo`. Once an address inside the JIT buffer is revealed, the rest of the gadgets can be easily located by searching in the rest of the JIT buffer. Recall that *we* have constructed the gadgets and, therefore, we know in advance for what we are searching for inside the JIT buffer.

Now it is possible to combine the exploit we described in Section IV and the JIT-disclosing technique presented in this

section in order to create a fully working exploit. The exploit creates *all* needed ROP gadgets in the JIT buffer, it locates them one by one using an information-leakage vulnerability, it builds the ROP chain, which once executed it makes a page hosting the shellcode executable, and, finally, compromises the browser.

```
1 O = new Object();
2 O.g1 = 0xc358;
3 O.g2 = 0xc359;
4 function foo(x) { return 0x5841; }
5 O.func = foo;
```

Listing 5. JavaScript code that generates a Object which memory layout is described in Figure 7.

VII. Defenses

In this section we discuss defenses. We first discuss existing defenses and their applicability and later we propose new countermeasures based on our experience from building the attacks presented in this paper.

A. Existing Defenses

So far, there are two ways to defend against the attacks we described: (i) preventing the construction of gadgets using techniques such as constant blinding (see Section V), and (ii) diversifying the JIT buffer so that the created gadgets cannot be located. Both these strategies are used in IE's JIT engine (Chakra) and Librando [24]. We have serious concerns that these strategies may not actually constrain sophisticated and determined attackers.

As far as strategy (i) is concerned, we demonstrated its weaknesses by realizing an actual attack on Chakra which bypasses constant blinding by constructing gadgets in short immediate values of 1 and 2 byte. One could argue that by applying constant blinding in *all* immediate values, no matter the size, could, in theory, stop the attack. This is correct, however, enforcing constant blinding in all immediate values does not come for free. We perform our evaluation using the SunSpider benchmarks suite. We log all the JIT instructions that were *actually* executed in each test. We count how many instructions involve an immediate value (of 1 or 2 bytes) and the respectively required CPU cycles. We extract this information from Intel's manual, by matching instructions and corresponding cycles. Essentially, there are three families of instructions that may involve an immediate value, the distribution of which we depict in Figure 9. Note that in all tests the instructions involving an immediate value comprise a significant percentage, ranging between 18–52% of all executed instructions. Therefore applying constant blinding to all immediate values is quite costly, introducing an estimated overhead of 15% to 80%, as shown in Figure 8. We assume that the JIT compiler emits (at least) two or six more instructions for each instruction involving an immediate value, depending on whether the instruction has one or two immediates. We match these additional instructions to corresponding cycles and calculate the overhead as additional cycles. Notice that this estimation is quite conservative, since we do not account for additional code that will be executed for preparing the blinding (i.e., calls to `rand()`, code analysis, and so on).

Moreover, strategy (ii) is based on simply hiding the gadgets. This strategy has been adopted by many proposals for countering software exploitation. Unfortunately, all these strategies can be defeated either through a memory disclosure bug [29], or by forcing the vulnerable application to place attacker data, that is the generated code by the JavaScript JIT compiler in our case, in predictable locations. The latter is comparable with heap spraying [30], where the attacker allocates many copies of his data in an attempt to ensure that one of the copies lands at a predictable memory address in the vulnerable program's heap. This might sound improbable but recent work in this field has shown that such memory disclosure attacks are both powerful and realizable, and they can bypass even highly dynamic randomization schemes [15]. In fact, in this paper we have demonstrated a similar technique for leaking the location of the JIT buffer (see Section VI), and discovering all constructed ROP gadgets, rendering all randomization schemes ineffective.

One possible direction for mitigating ROP in general, and thus the attacks presented in this paper, is Control-Flow Integrity (CFI). [37] This was initially a very promising technique against code-reuse attacks, which quickly drove to implementations [11], [12], [16], [17] that support legacy code and impose negligible overhead. Unfortunately, there are many concerns about the validity of these approaches [18], [19], [31], [32], [38], therefore making the applicability of CFI, especially the coarse-grained version, questionable. Nevertheless, there are still efforts for applying fine-grained CFI in dynamic-code generation [20], which is essentially very similar to JIT compilation, and possibly could be a practical solution—as long as the overhead is reasonable—for countering the attacks presented in this paper.

B. Proposed Defenses

Based on our experience while developing the attacks presented here we propose two defense mechanisms. Both, require code analysis. Realizing these techniques is beyond the scope of this paper and we believe further research is needed for implementing them. Both techniques introduce overhead which may eventually nullify the gains from JIT compilation. This is the reason why we believe that the attacks presented in this paper cannot be easily addressed.

d) JIT Analysis: The most obvious defense mechanism is to enhance the JIT compiler with the techniques proposed by G-Free [21] for eliminating all gadgets. This has as a major advantage that the produced code is safe and gadget-free, however this does not come for free. The code has to be further processed for eliminating the gadgets, and the produced native code will experience overheads compared to the non gadget-free code. Last but not least, it is unclear if it actually easy to apply G-Free techniques in code that is generated partially and on-the-fly.

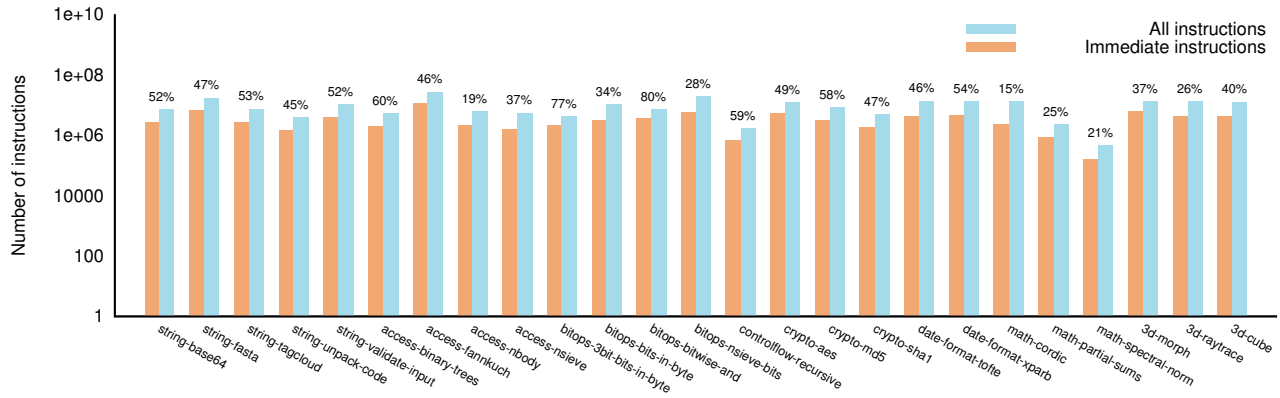

Fig. 8. The cost of constant blinding. In this histogram we present the number of all the executed instructions and immediate related ones. In percent above each test we present the overhead for introducing constant blinding on 1 or 2 byte immediate.

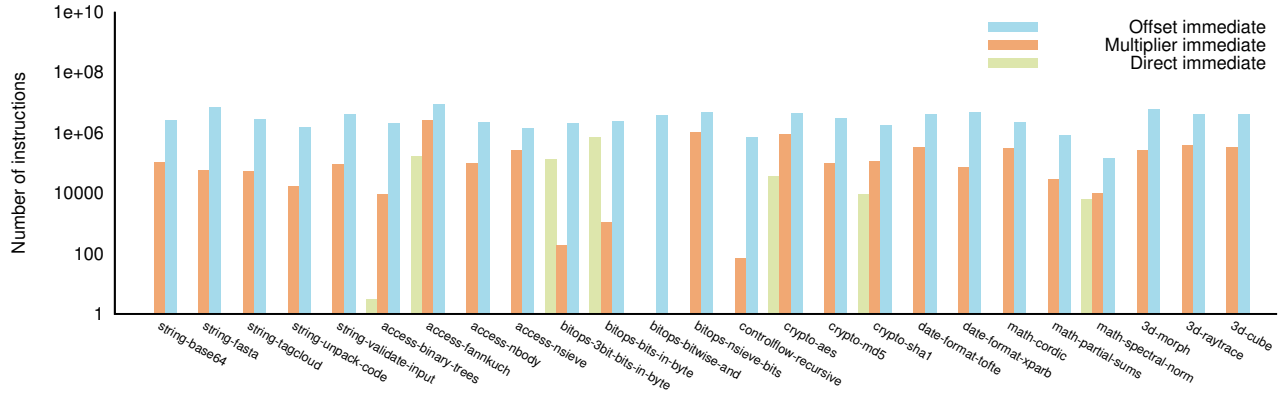

Fig. 9. Distribution of instructions emitted in the JIT buffer that affect an immediate value. There are basically three types of instructions that may involve an immediate value: *offset immediate* (e.g., `mov rcx,[rcx+0x8]`), *multiplier immediate* (e.g., `mov [r8+rdi*8+0x20], 0xfff88000`), and *direct immediate* (e.g., `shl r13,3`).

TABLE I
OFFSET IMMEDIATE INSTRUCTIONS

Length of *offset immediate*	Ratio
1 byte	43.5%
2 byte	43%
>2 byte	9.5%

e) JavaScript Analysis: As we can observe from the JavaScript payloads used for attacking Mozilla Firefox and IE (see Listing 2 and 4), special patterns, such as constant values that can be interpreted as x86 opcodes, are utilized. Therefore, source analysis could infer if a JavaScript program is targeting the JIT engine. However, modern obfuscation techniques can make such analysis hard.

VIII. Related Work

Software exploitation has evolved over the last decade. Initially, about a decade ago, a simple buffer overflow could corrupt the stack, re-write the return address and make it point to a buffer holding a shellcode, and eventually compromise the program [39]. Today, many practical mitigations available in all commodity operating systems make such a technique infeasible. First, DEP [6] prevents execution of data, making it impossible to store the shellcode on a data buffer and jump to it, and, second the stack is protected by canaries, i.e., random values that are placed near the return address and checked at run-time for their value [5]. An attacker that overflows a buffer aiming at re-writing the return address will eventually re-write the canary, too. Furthermore, there are many academic proposals for protecting the stack and the return address, as well as counter for buffer overflows [40]–[42].

Therefore, attackers today can only use existing code from

the vulnerable program in an unintended behavior [8], [25], [26], [43], [44]. So far, the most practical defense mechanism for defeating code-reuse attacks is ASLR [7], which simply randomizes the process layout in the virtual space, so that the attacker cannot locate the existing code. Researchers have managed to bypass ASRL when the entropy is not enough or using information leaks [45]–[48]. Following the practice of ASLR, researchers developed more fine-grained randomization schemes [13], [14], but again they were defeated by sophisticated exploitation techniques based on information leaks [9] or on brute forcing crash-resistant processes [10].

It seems that the most promising direction for countering code-reuse attacks is to eliminate the feasibility of *code-reuse* itself. This can be done either by either re-writing the binary [11], [12] to respect its call-graph [37], either monitoring at run-time [16], [17], or by re-compiling it for eliminating all code-reuse paths (gadgets) [21]. Although attacks for such systems have been demonstrated [18], [31], [32], we believe that the bar for exploiting a binary has been significantly raised by the community and that attackers have to discover *new* avenues for exploitation. One of this, is the one presented in this paper: exploiting a program in an environment which is gadget free.

In parallel with this work, Song et al [49] show that JIT buffers can be exploited through code cache injection techniques. This is possible if the JIT buffer is both writable and executable or even temporarily writable at times. This threat is more realistic if the generated code is multi-threaded, because the switch between writable and executable leaves a time window for exploitation. They propose a new dynamic code generation architecture which utilizes a separate process and shared memory to prevent such exploits.

IX. Conclusion

In this paper we introduced and demonstrated a method to attack gadget-free binaries. We demonstrated our attack on Mozilla Firefox and Microsoft Internet Explorer, two of the most widely used applications. Our starting assumption was that the binaries and shared libraries contain no gadgets that can be exploited. Our attack manages to introduce useful gadgets by utilizing the JIT engine present in both browsers, but also present in other applications as well. Using the JIT engine, we can create the required gadgets at run-time, *inside* the JIT buffer.

Furthermore, we modified a technique based on already published work [9] for discovering the gadgets at run-time by leaking the address of the JIT buffer. Our attack is powerful in the sense that it allows the execution of *any* shellcode, since it can change the access permissions of the data page holding the shellcode. Our techniques are able to exploit the JIT engine of IE (Chakra), which incorporates a series of defense mechanisms designed specifically to thwart such attacks. Finally, we performed an extensive analysis and present details about the *undocumented* defensive techniques of Chakra.

X. Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work was supported in part by the FP7-PEOPLE-2010-IOF project XHUNTER No. 273765 and by the US Air Force through Contract AFRL-FA8650-10-C-7024. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, or the Air Force.

References

[1] Google, "WebView," Android Developers API Reference, https://developer.android.com/reference/android/webkit/WebView.html.

[2] N. Joly, "Advanced exploitation of Internet Explorer 10 / Windows 8 overflow (Pwn2Own 2013)," VUPEN Vulnerability Research Team (VRT) Blog, May 2013, http://www.vupen.com/blog/20130522.Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php.

[3] A. Pelletier, "Advanced exploitation of Internet Explorer heap overflow (Pwn2Own 2012 exploit)," VUPEN Vulnerability Research Team (VRT) Blog, July 2012, http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php.

[4] J. L. Obes and J. Schuh, "A tale of two pwnies," The Chromium Blog, May 2012, http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html.

[5] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang *et al.*, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th USENIX Security Symposium*, vol. 81, 1998, pp. 346–355.

[6] S. Andersen and V. Abella, "Changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, Data Execution Prevention," Microsoft TechNet Library, September 2004, http://technet.microsoft.com/en-us/library/bb457155.aspx.

[7] PaX Team, "Address Space Layout Randomization (ASLR)," 2003, http://pax.grsecurity.net/docs/aslr.txt.

[8] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and Communications security*, October 2007, pp. 552–61.

[9] K. Z. Snow, L. Davi, A. Dmitrienko, C. Liebchen, F. Monrose, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, May 2013.

[10] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking blind," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, vol. 14, 2014.

[11] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 1013 Security and Privacy Symposium*, 2013, pp. 559–573.

[12] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *22nd USENIX Security Symposium*, 2013.

[13] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012, pp. 601–615.

[14] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 157–168.

[15] A. K. Cristiano Giuffrida and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012, pp. 40–55.

[16] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *Proceedings of the 22nd USENIX Security Symposium*, 2013, pp. 447–462.

[17] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," in *Proceedings of the 2014 Network and Distributed System Security (NDSS) Symposium*, February 2014.

[18] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, May 2014.

[19] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *23rd USENIX Security Symposium*, 2014.

[20] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th PLDI*, 2014, pp. 577–587. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594295

[21] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: Defeating return-oriented programming through gadget-less binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010, pp. 49–58.

[22] D. Blazakis, "Interpreter exploitation," in *Proceedings of the 4th USENIX Conference on Offensive Technologies*, ser. WOOT'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=1925004.1925011

[23] C. Rohlf and Y. Ivnitskiy, "Attacking clientside jit compilers," *Black Hat USA*, 2011.

[24] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Librando: transparent code randomization for just-in-time compilers," in *ACM Conference on Computer and Communications Security*, A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM, 2013, pp. 993–1004.

[25] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and Communications Security*, October 2010, pp. 559–72.

[26] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ASIACCS*, March 2011, pp. 30–40.

[27] Microsoft, "The enhanced mitigation experience toolkit," http://support.microsoft.com/kb/2458544.

[28] A. Portnoy, "Bypassing all of the things," Exodus Intelligence, https://www.exodusintel.com/files/Aaron_Portnoy-Bypassing_All_Of_The_Things.pdf.

[29] F. J. Serna, "CVE-2012-0769, the case of the perfect info leak," http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf.

[30] DarkReading, "Heap spraying: Attackers' latest weapon of choice," http://www.darkreading.com/security/vulnerabilities/showArticle.jhtml?articleID=221901428, November 2009.

[31] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehman, and Fabian Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd USENIX Security Symposium*, 2014.

[32] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maa, Martin Steegmanns, Moritz Contag, and Thorsten Holz, "Evaluating the effectiveness of current anti-rop defenses," in *Proceedings of the Internation Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2014.

[33] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *Proceedings of the USENIX Security Symposium*, 2011.

[34] Microsoft MSDN, "Advances in javascript performance in ie10 and windows 8," http://blogs.msdn.com/b/ie/archive/2012/06/13/advances-in-javascript-performance-in-ie10-and-windows-8.aspx.

[35] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kguard: Lightweight kernel protection against return-to-user attacks," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 39–39. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362832

[36] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, May 2014.

[37] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and Communications Security*, 2005, pp. 340–353.

[38] Nicholas Carlini and David Wagner, "Rop is still dangerous: Breaking modern defenses," in *23rd USENIX Security Symposium*, 2014.

[39] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, p. 365, 1996.

[40] M. Frantzen and M. Shuey, "StackGhost: Hardware facilitated stack protection," in *Proceedings of the 10th USENIX Security Symposium*, August 2001, pp. 55–66.

[41] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with wit," in *IEEE Symposium on Security and Privacy*, 2008, pp. 263–277.

[42] A. Slowinska, T. Stancescu, and H. Bos, "Body armor for binaries: preventing buffer overflows without recompilation," in *Proceedings of USENIX Annual Technical Conference*, Boston, MA, June 2012.

[43] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection*, 2011, pp. 121–141.

[44] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *Proceedings of CCS 2008*, P. Syverson and S. Jha, Eds. ACM Press, Oct. 2008, pp. 27–38.

[45] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and Communications Security*, 2004, pp. 298–307.

[46] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proceedings of the Second European Workshop on System Security*, ser. EUROSEC '09. New York, NY, USA: ACM, 2009, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1145/1519144.1519145

[47] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 191–205.

[48] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib (c)," in *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 2009, pp. 60–69.

[49] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski, "Exploiting and protecting dynamic code generation," in *Proceedings of the 2015 Network and Distributed System Security (NDSS) Symposium*, February 2015.