Bypassing Defenses of Just-In-Time Compilers in Modern Browsers

Michail Athanasakis

Thesis submitted in partial fulfillment of the requirements for the

Master of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Knossou Av., P.O. Box 2208, Heraklion, GR-71409, Greece

Thesis Advisors: Prof. Evangelos Markatos, Dr. Sotirios Ioannidis

This work was partially supported by Institute of Computer Science, Foundation of Research and Technology Hellas
Bypassing Defenses of Just-In-Time Compilers in Modern Browsers

Thesis submitted by
Michail Athanasakis
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author:
Michail Athanasakis

Committee approvals:
Evangelos Markatos
Professor, Thesis Supervisor

Antonios Savvidis
Professor

Sotirios Ioannidis
Principal Researcher

Elias Athanasopoulos
Assistant Professor

Departmental approval:
Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, January 2016
Abstract

Return-oriented programming (ROP) has become the dominant form of vulnerability exploitation in both user and kernel space. Many defenses against ROP during run-time make it 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 thesis, 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 thesis 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.

Thesis supervisor: Prof. Evangelos Markatos
Περίληψη

Οι επιθέσεις με την χρήση επιστρεφόμενου προγραμματισμού (ROP) είναι η πιο διαδεδομένη μορφή επιθέσεων σε επίπεδο χρήστη και λειτουργικών συστημάτων. Η δημιουργία αμυντικών τεχνικών έχει κάνει την επίθεση πιο δύσκολη. Ηδη οι επιτιθέμενοι έχουν αρχίσει να εκμεταλλεύονται μεταγλωττιστές δυναμικής παραγωγής κώδικα (JIT), διαθέσιμοι σε όλους τους γνωστούς περιηγητές διαδικτύου, για την δημιουργία κακόβουλου πηγαίου κώδικα. Ο κακόβουλος κώδικας δημιουργείται με την μορφή πηγαίου κώδικα ή gadgets και μετά το τέλος της JIT μεταγλώττισης οι επιτιθέμενοι μπορούν να τον εκμεταλλευτούν.

Αναγνωρίζοντας την απειλή οι δημιουργοί περιηγητών έχουν αρχίσει να εισάγουν μηχανισμούς άμυνας στις JIT μηχανές μεταγλώττισης κώδικα. Σε αυτή τη διατριβή δείχνουμε ότι παρόλο που όλες αυτές οι άμυνες είναι διαθέσιμες σήμερα δεν είναι ακόμη αποτελεσματικές σήμερα. Αναφερόμαστε σε δύο διαδεδομένους διαδικτυακούς περιηγητές: Mozilla Firefox και Internet Explorer 64-bit ως περιπτώσεις διαδικτυακών περιηγητών. Δείχνουμε ότι η δημιουργία gadgets είναι δυνατή σε δύο διαδεδομένους διαδικτυακούς περιηγητές: Mozilla Firefox και Internet Explorer 64-bit συνολικά. Επίσης, ο Internet Explorer 64-bit διαθέτει κλειδιά μηχανισμούς αμυνόμενος τελευταίας γενιάς, κόσινους μη-καταγραμμένους τούτων τέρω. Παράλληλα και δείχνουμε ότι δεν είναι αξιολογήσιμοι για την εξάσκηση του λογισμικού. Καταγράφουμε όλες τις μηχανισμούς τεχνικές με σκοπό να βοηθήσουμε άλλους ερευνητές, Επιπροσθέτως, εκτός από την παρουσίαση τεχνικών για την δημιουργία ROP gadgets δείχνουμε δυνατότητα μετατροπής των 3-bytes σε 80% παραπάνω εντολών.

Επόπτης μεταπτυχιακής εργασίας: Καθηγητής Ευάγγελος Μαρκάτος
Acknowledgements

I would like to thank my supervisor, Professor Evangelos Markatos for his advice and guidance during my studies. I am especially grateful to Dr. Sotiris Ioannidis for his help and guidance. I would like to thank him for his trust, the life lessons and all the opportunities he gave me. Special thanks to Dr. Elias Athanasopoulos for his guidance, patience, sharing his knowledge and introducing me to the world of research during these last three years. This work could not be completed without their valuable help and pushing toward. Also, my thanks to Dr. Georgios Portokalidis and Dr. Michalis Polychronakis for their advice and help on this work. I would like to thank all the past and current members of Distributed Computing Systems Labatory at FORTH. They made my stay fun, educational and were proven great companions during our journeys.

My thanks to all my friends for their friendship, love, inspiration and for sharing all these years with me. Special thanks to Giannis Apostolidhs, Achilleas Tsiolkas, Spyrogianiannis Vlachos, Marios Rouggas Soundias and Giannis Vogiatzis for their friendship, the adventures, the days and nights together and making the last few years the best of my life.

I would like to thank my family for their love and support all these years. My heartfelt gratitude to my mother for her unconditional love, patience and guidance all my life, she made this work possible. My father for his love, the knowledge he shared with me and showing me the way to lead this life. My brother for being awesome, challenging me and sharing with me all our journeys.
The real meaning of enlightenment is to gaze with undimmed eyes on all darkness.

Nikos Kazantzakis
An early report on this work appeared in the proceedings of the 21st Annual Network & Distributed System Security Symposium (NDSS), February 2015 [1].
# Contents

1 Introduction ................................................. 1
   1.1 Problem Statement ..................................... 1
   1.2 Our Approach ......................................... 3
   1.3 Organization ......................................... 4

2 Related work ............................................... 5
   2.1 Buffer overflows ...................................... 5
   2.2 DEP & ASLR ........................................... 5
   2.3 Fine-grained defenses ................................. 6
   2.4 Stopping code-reuse attacks .......................... 7
   2.5 Control-Flow Integrity ............................... 8
   2.6 EMET ................................................ 9
   2.7 Parallel to this work ................................ 9

3 Background ................................................. 11
   3.1 ROP Gadgets .......................................... 11
   3.2 JIT Compilation ..................................... 13

4 Threat Model ............................................... 15
   4.1 Active Defenses ...................................... 15
   4.2 Attacker Capabilities ................................. 16
   4.3 Is this attack unstoppable? ......................... 16

5 Exploiting Mozilla Firefox ............................... 17
   5.1 Preparation .......................................... 18
   5.2 Exploit Implementation .............................. 20

6 Exploiting Internet Explorer ............................. 21
   6.1 Why Internet Explorer is different ................. 21
   6.2 Preparation .......................................... 21
   6.3 Exploit Design Considerations ..................... 22
   6.4 Exploit Implementation .............................. 25
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Discovering the Gadgets</td>
<td>29</td>
</tr>
<tr>
<td>7.1</td>
<td>How Information Leaks Work</td>
<td>29</td>
</tr>
<tr>
<td>7.2</td>
<td>Leaking the JIT Buffer</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>Defenses</td>
<td>33</td>
</tr>
<tr>
<td>8.1</td>
<td>Existing Defenses</td>
<td>33</td>
</tr>
<tr>
<td>8.2</td>
<td>Proposed Defenses</td>
<td>35</td>
</tr>
<tr>
<td>8.2.1</td>
<td>JIT Analysis</td>
<td>36</td>
</tr>
<tr>
<td>8.2.2</td>
<td>JavaScript Analysis</td>
<td>36</td>
</tr>
<tr>
<td>9</td>
<td>Conclusion</td>
<td>37</td>
</tr>
<tr>
<td>10</td>
<td>Future Work</td>
<td>39</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Evolution of attacks and defenses in browser JIT engines. ............. 2

2.1 Each time the OS boots the location of the process memory segment is different. The same randomization happens to all system libraries making attacker’s work much harder. ................................. 6

2.2 The 1st line shows an example of instruction transformation, destroying the gadgets based on the immediate number that would be moved to %eax. The 2nd example shows that again the immediate value is altered by offsetting the jmp instruction by 1. ........................................... 7

2.3 The 1st line of binary contains a misaligned gadget. In order to destroy it G-Free inserts a number of nops it achieves the separation of the binary stream that would be needed to create the gadget. .................. 8

3.1 Example of small code sequences (gadgets) that can be chained for performing computations controlled by an adversary. ................. 12

5.1 Illustration of the attack presented in this thesis. 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. ......................... 17

5.2 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. ............... 18

6.1 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 5.2), the overflow bit will be set and thus the flow will follow the conditional jump (the program will be terminate). ............... 23

6.2 The stack during the attack against IE. ................................. 26

7.1 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. 30
8.1 The cost of constant blinding. In this histogram we present the JIT instructions that were actually executed in a session where the complete suite of SunSpider benchmarks run. In addition we plot the volume of JIT instructions that involve an immediate value (of 1 or 2 bytes), and therefore may be used potentially for creating ROP gadgets. Notice that in all tests the instructions involving an immediate value comprise a significant percentage, between 18–52% of all executed instructions. Therefore applying constant blinding to all immediate values is quite costly, introducing an estimated overhead of 40% to 82%. The overhead is estimated as follows. We assume that the JIT compiler will emit (at least) two more instructions for each instruction involving an immediate value (i.e., 18% of instructions that should be blinded introduce an overhead of 40%). 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 \texttt{rand()}, code analysis, and so on).

8.2 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: \textit{offset immediate} (e.g., \texttt{mov rcx, [rcx+0x8]}), \textit{multiplier immediate} (e.g., \texttt{mov [r8+rdi*8+0x20], 0xfff88000}), and \textit{direct immediate} (e.g., \texttt{shl r13,3}).
2.1 EMET’s mitigations are categorized based on what level they apply. System-wide defenses are basic but provide a good level of protection. The Application-based mitigations offer a per-application policy, offering system’s administrator better control.

3.1 This table shows commonly used gadgets that if chained together could possibly exploit a x86 Linux Operating System. Since the x86 architecture is heavily populated with instructions it is easy to find other similar gadgets to accomplish the same goal.

3.2 From the 10 mostly-used languages on Github all languages except C++/C have available JIT compilers. This shows the adaptation of JIT compilation on almost every programming environment.

8.1 offset immediate instructions
Chapter 1

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 [2]. Their popularity is probably one of the reasons that they are such an attractive target for attackers and security researchers alike [3–5]. In parallel, one of the main reasons web content is becoming increasingly interactive is JavaScript. A high-level, dynamic, untyped language that its script can be found in almost every site. Each time you visit a website the browser downloads JavaScript libraries and scripts to increase the functionality of the site. Every popular browser contains a JavaScript interpreter, and one of it’s key components is a Just-In-Time (JIT) compiler that transforms JavaScript to native code in order to improve performance. The introduction of dynamically generated code introduces a new set of security risks. This work takes advantage of those vulnerabilities and design choices and shows a methodology on how to bypass existing defenses in order to execute malicious code.

1.1 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 [6], non-executable pages [7], and address-space layout randomization (ASLR) [8], which severely hinder exploitation. Even code-reuse techniques such as return-oriented programming (ROP) [9] are not straightforward, since they require information-leak bugs to reveal the randomized location of code [10, 11] or legacy code and libraries that ASLR cannot randomize.

Recent works on control-flow integrity (CFI) [12, 13], fine-grained code randomization [14–16], and run-time behavioral monitoring [17, 18] promise to protect software from ROP-like attacks, but unfortunately they have been also shown to be vulnerable to niche attacks [10, 19, 20]. Other approaches that require application source code, such as modular fine-grained CFI [21] and G-Free [22], 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 1.1 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-
1.2. OUR APPROACH

flow to his shellcode in memory. Because 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 [23].

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 [24]. 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 \texttt{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 \texttt{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. 6, make JIT exploitation significantly harder.

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

1.2 Our Approach

This thesis 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 [3], 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 work 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 [23], 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 thesis cannot be stopped using code diversification (e.g., using Librando [25]). Based on published research [10], we can discover the JIT buffer even if it is randomized, and progressively discover the constructed gadgets on-the-fly before launching the actual attack.

1.3 Organization

The rest of the thesis is organized as follows. We present prior and related work under Chapter 2. In Chapter 3 we present all essential background information required for understanding this thesis. We explain what gadgets and a JIT compiler are. In Chapter 4 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 Chapters 5 and 6). We then discuss how gadgets are located in Chapter 7 and possible defenses in Chapter 8. We conclude in Chapter 9.
Chapter 2

Related work

2.1 Buffer overflows

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 [26]. Today, many practical mitigations available in all commodity operating systems make such a technique infeasible. First, Data Execution Prevention (DEP) [7] marks the pages of memory as executable or non-executable. Code resides in readable and executable memory pages and data at readable and writable pages. This prevents the execution of data, making it impossible to store the shellcode on a data buffer and jump to it. In the beginning DEP was implemented by software but today modern CPUs carry an extra bit called NX-bit to enforce DEP from the hardware layer. Another defense mechanism is canaries [6]. Canaries are random values that are placed near each return address on the stack. Usually, buffer overflows aim to overwrite the return address of a function to jump to its shellcode. While overwriting the return address they also overwrite the canary value. Using a simple run-time check it is easy to determine if a return value has been compromised and stop the execution of the program or driver. This prevents the execution of the code. Other proposed defenses such as [27] encrypt the return addresses stored in the stack. Another solution [28] was to implement a shadow return stack and check during run-time to ensure the integrity of all return addresses or function exits. Furthermore, there are many academic proposals for protecting the stack and the return address, as well as counter for buffer overflows [29, 30].

2.2 DEP & ASLR

Therefore, attackers today can only use existing code from the vulnerable program, introducing a new type of attack called code-reuse attacks [31]. Attacks of this sort do not require from the attacker the injection of harmful code but rather utilize existing code from system libraries or the exploitable program. The manifestation of code-reuse attacks is Return Oriented Programming (ROP) [9] or Jump Oriented Programming (JOP) [32, 33].
CHAPTER 2. RELATED WORK

These attacks use gadgets, already introduced at chapter 3, to achieve the unintended behavior. So far, the most practical defense mechanism for defeating code-reuse attacks is Address Space Layout Randomization (ASLR) [8], which simply randomizes the process layout in the virtual space, so that the attacker cannot locate the existing code. Figure 2.1 shows how each time the OS boots or a program starts the location of each system library and program’s memory segments are located at a different address at the process’s virtual address space. This forces the adversary to require a memory leak to know the location of the memory pages containing the exploitable code. Researchers have managed to bypass ASLR when the entropy is not enough in 32-bit operating systems [34]. Other proposed bypasses of ASLR utilized side-channel attacks /citekaslr-bypass or information leaks, either by using brute force /citeaslr-bypass2 or a surgical approach /citeair. During this period new frameworks /citeq,mona were presented by researchers to increase the feasibility of an automated attack.

2.3 Fine-grained defenses

Following the practice of ASLR, researchers developed more fine-grained randomization schemes. Smashing the gadgets [14] mitigation scheme is based on in-place code randomization. Its method uses various narrow-scope code transformations that can be applied statically into the binary without altering the location of basic blocks. This allows the safe randomization of binaries stripped of debug information even with partial disassembly coverage. Since the addition of code is minimal the performance overhead is small. Another proposed defense mechanism [15] enabled the binary to stir each time it launches. Again, it could work without debug information and it can be applied even when full disassembly is possible. It comes with an overhead of 1.6%, making it a viable defense. As expected attackers also fine-grained their technique and introduced JIT code reuse [10]. The attack worked by repeatedly abusing a memory disclosure bug to

Figure 2.1: Each time the OS boots the location of the process memory segment is different. The same randomization happens to all system libraries making attacker’s work much harder.
map each memory page of an application on-the-fly. This mapping also included necessary system API memory addresses, such as `VirtualProtect`, and other system wide gadgets. Another attack [11] manages to remotely find the gadgets and performs a write system call and transfers the vulnerable binary over the network, after which an exploit can be completed using known techniques.

2.4 Stopping code-reuse attacks

It seems that the most promising direction for countering code-reuse attacks is to eliminate the feasibility of code-reuse itself. A proposed defense [12] is based on collecting all the valid indirect control-transfer instructions, puts them into a dedicated "Springboard section" in a random order, and then limits indirect transfers to flow only to them. In a similar pattern [13] proposes a solution by developing specific techniques of disassembly, static analysis and finally the transformation of large binaries. Control-Flow Graphs where used by [35] to determine ahead of time all the possible paths a binary could take.

In Section 2.5 we explain more about this defense scheme and its issues. Another solution, G-Free [22] is based on re-compiling the binary to eliminate all the possible code-reuse paths or gadgets. This defense implements several measures to completely eliminate all possible gadgets from the program. In a similar way it can be applied to system libraries with minimal overhead. The modification of the binary happens by adding the proper plug-in into the compiler. Proposed defense mechanism include alignment sleds, return address protection, frame cookies, register reallocation, instruction transformations, jump offset adjustment and immediate and displacement reconstruction. Some of these defense schemes are better shown at Figure 2.2 and 2.3.

Figure 2.2: The 1st line shows a example of instruction transformation, destroying the gadgets based on the immediate number that would be moved to %eax. The 2nd example shows that again the immediate value is altered by offsetting the jmp instruction by 1.
Figure 2.3: The 1st line of binary contains a misaligned gadget. In order to destroy it G-Free inserts a number of nops it achieves the separation of the binary stream that would be needed to create the gadget.

2.5 Control-Flow Integrity

As explained in Chapter 2.4 Control-Flow Integrity (CFI) prevents the execution of arbitrary code by mapping all the normally expected control-flow paths. Based on that concept, we have a lot of proposed defenses. kbouncer [17] utilized hardware features pf Intel CPUs to detect abnormal control transfers before the execution of sensitive system calls. An improved solution of the previous defense scheme [18] is using a kernel module to further ensure the early and proper detection of ROP attacks in an efficient way. As it is expected several attacks [19, 36, 37] have demonstrated that these defenses can be bypassed, although the difficulty bar has been raised. Even more fine-grained defense schemes have been introduced such as ShrinkWrap [38] and others [39]. One issue with these defenses is that they require recompilation, something not always possible for the required software and libraries. RockJIT [40] is another proposed solution, developed almost in parallel with this work, that introduces CFI for JIT engines with an overhead of 14.6%. Their solution protects not only the JITed code but the JIT compiler itself. This work introduces an attack to a binary without CFI. In the case of CFI-enabled web browser the task would be much harder and potentially really hard to achieve. The CFI would introduce another problem. Even if you manage to introduce gadgets inside the JIT code buffer it would be hard to hijack the control flow of the browser due to the CFI. If the browser’s binary is CFI protected it was shown that it can be bypassed, although it is unclear if it possible to redirect the control flow to a random address inside the JIT code buffer. This part depends on how the CFI handles the cases where the control flow is transferred from and to the JIT code buffer. If both the JIT compiler and the JIT code buffer is CFI protected the challenge is even bigger. For this work, we bypass already existing defenses incorporated in the JIT engines but we determine that further exploration is required regarding the efficiency of CFI defenses and if are able to stop a determined attacker.
2.6 EMET

Enhanced Mitigation Experience Toolkit (EMET) [41] is a freeware security toolkit provided by Microsoft for the operating system Windows. It provides an interface to enable and manage fine-grained security features of Windows. The goal of this toolkit is to provide additional protection to un-patched software and protect from common and targeted exploits. EMET provides a series of system-wide mitigations such as DEP, ASLR and Structured Exception Handler Overwrite Protection (SEHOP). Other defenses schemes include implementations of fine-grained mitigations already introduced by research. A more thorough presentation of the available defenses of EMET is presented in Table 2.1 based on [42].

<table>
<thead>
<tr>
<th>System-wide Mitigations</th>
<th>DEP</th>
<th>ASLR</th>
<th>SEHOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced Mitigations</td>
<td>Deep hooks</td>
<td>Anti detours</td>
<td>Banned functions</td>
</tr>
<tr>
<td></td>
<td>DEP</td>
<td>SEHOP</td>
<td>Null Page</td>
</tr>
<tr>
<td>Application Specific</td>
<td>Heap Spray</td>
<td>Mandatory ASLR</td>
<td>EAF</td>
</tr>
<tr>
<td>Mitigations</td>
<td>Bottom-Up</td>
<td>Load Library</td>
<td>Memory Protection</td>
</tr>
<tr>
<td></td>
<td>Caller Checks</td>
<td>Check</td>
<td>Check</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Simulate Execution Flow</td>
<td>Stack Pivot</td>
</tr>
</tbody>
</table>

Table 2.1: EMET’s mitigations are categorized based on what level they apply. System-wide defenses are basic but provide a good level of protection. The Application-based mitigations offer a per-application policy, offering system’s administrator better control.

2.7 Parallel to this work

In parallel with this work, Song et al [43] show that buffers generated from JIT compilers are prone to attacks that utilize code-cache injections. It is possible a JIT buffer can have both writable and executable rights temporarily based on the nature of JIT code generation. The threat is more realistic in multi-threaded environments were many threads generate and execute code from the same code buffer in memory. Their proposed solution is a new dynamic code generation architecture that invokes a different process for the generation and the execution of the code, while sharing common memory. They provide evidence that some types of attacks can be hindered this way.

Isomeron [44] is another solution against JIT-ROP attacks. After creating a novel attack to bypass fine-grained defenses in JIT environments they present a new hybrid defense approach. It is a combination of code randomization with execution path randomization in order to mitigate ROP and JIT-ROP attacks. The solution includes cloning program image using dynamic binary instrumentation. The framework breaks down the program into basic blocks and after diversifying the actual code it emits then multiple times in different memory addresses. During the run-time a *coin-flip* decides what instance of the block to execute.
Chapter 3

Background

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

3.1 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 [32, 33].

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 order for the sequence of gadgets to work it is necessary to have a special type of gadget called stack pivoting gadget. This gadget is not a specific instruction but almost always accomplishes the same task. It transfers the stack pointer to point to the location of the virtual stack. The virtual stack is filled with the addresses of each gadget in a way that with each `ret` instruction we move into our ROP chain. This allows to execute a proper exploit. In fact, it has been shown that ROP is Turing Complete [9]. In Figure 3.1 and listings 3.1, 3.2, 3.3 we depict a few example gadgets and how they can be extracted by a using a misaligned instruction pointer.
CHAPTER 3. BACKGROUND

Table 3.1: This table shows commonly used gadgets that if chained together could possibly exploit a x86 Linux Operating System. Since the x86 architecture is heavily populated with instructions it is easy to find other similar gadgets to accomplish the same goal.

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Hexcode</th>
<th>Common use</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop %ebx; ret;</td>
<td>5b c3</td>
<td>Move a stack value to %ebx register</td>
</tr>
<tr>
<td>pop %ecx; ret;</td>
<td>59 c3</td>
<td>Move a stack value to %ecx register</td>
</tr>
<tr>
<td>xor %eax, %eax; ret</td>
<td>31 c0 c3</td>
<td>Clear previous value of register</td>
</tr>
<tr>
<td>mov $0x7, %al; ret</td>
<td>b0 7d c3</td>
<td>Load immediate value to register</td>
</tr>
<tr>
<td>int $0x80</td>
<td>80 c3</td>
<td>Do a linux system call</td>
</tr>
</tbody>
</table>

Listing 3.1: This stack pivot is executed repeatedly until %rsp has the desirable value.

Listing 3.2: The stack pivot moves the value from the already compromised %rax to %rsp.

Listing 3.3: A longer sequence of instructions of a stack pivoting gadget. This gadget assumes that %eax is compromised and with the 1st instruction it is transferred to the stack. The second instruction moves the value to %esp to achieve the stack pivoting. The remaining instructions until ret are not useful but they have to be accounted when the virtual stack is created.

Figure 3.1: Example of small code sequences (gadgets) that can be chained for performing computations controlled by an adversary.
3.2 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.

<table>
<thead>
<tr>
<th>Language</th>
<th>Is a JIT compiler available?</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript</td>
<td>Yes, multiple available engines</td>
</tr>
<tr>
<td>Java</td>
<td>Yes, multiple available JVMs</td>
</tr>
<tr>
<td>Python</td>
<td>Yes, available from PyPy</td>
</tr>
<tr>
<td>CSS</td>
<td>Yes, available from WebKit</td>
</tr>
<tr>
<td>PHP</td>
<td>Yes, available from HHVM</td>
</tr>
<tr>
<td>Ruby</td>
<td>Yes, multiple available engines</td>
</tr>
<tr>
<td>C++</td>
<td>No</td>
</tr>
<tr>
<td>C</td>
<td>No</td>
</tr>
<tr>
<td>Shell</td>
<td>Yes, available with libgccjit</td>
</tr>
<tr>
<td>C#</td>
<td>Yes, available from .NET</td>
</tr>
</tbody>
</table>

Table 3.2: From the 10 mostly-used languages on Github all languages except C++/C have available JIT compilers. This shows the adaptation of JIT compilation on almost every programming environment.

In the beginning Just In Time compilation was used as a tool to improve the performance of slow interpreters. Nowadays, JIT code exists almost in every JavaScript commercial engine but it is not limited to clients or browsers. node.js, an event-driven JavaScript runtime built based on Chrome’s V8, is commonly used to today. node.js is mainly used for websites and other web services but has many more utilizations. There is even a project that uses node.js to gain the remote control of a car. Another popular development environment offered by Microsoft is .NET. It also offers JIT compilation [45] for C# and other languages. Actually, not only offers JIT compilation but also provides different types of JIT compilations, each one with each advantages to better suit the needs of the environment it executes. Based on the data of Github [46] JIT compilation adaptation on popular languages seems very successful as shown at Table 3.2.

We conclude that JIT compilers are found in many applications. In this thesis 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. This creates a wide potential surface of attacks.
Chapter 4

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 [10, 12, 17, 19].

4.1 Active Defenses

We assume that popular defenses, already present in most popular modern systems, are in effect. Data execution prevention (DEP) [7] 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) [8] 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 [6], as well as security toolkits like EMET [41].

Moreover, we assume that the vulnerable software has been hardened against code-reuse attacks [14, 17, 22]. For example, we assume that the binary and its libraries have been compiled with G-free [22], a compiler framework that produces binaries without gadgets, which enable powerful code-reuse attacks such as the ones based on return-oriented programming (ROP) [9]. In other words, composing a ROP or other type of core-reuse payload [33] 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 countermeasures, aiming at countering attacks like the one introduced in this thesis.
We discuss Chakra and how we bypass it later in this thesis, after presenting details on IE’s JIT engine and the attack itself. Librando [25] also provides many defense schemes and shares many common defenses with Chakra. We consider the common defenses offered by these two to be equivalent.

4.2 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 [4, 5, 47]. 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 [48], 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 [49], 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 over-reaching, recent prominent work [10] has shown that such memory disclosure attacks are both powerful and feasible, and they can overcome even highly dynamic randomization schemes [16]. We further explore how this is possible in JIT environments in Chapter 7 where we locate the JIT Code Buffer memory address.

4.3 Is this attack unstoppable?

The attack presented in this thesis mainly targets binaries hardened with G-free [22]. 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 [12, 13] 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 [8, 14, 15, 25] 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 [18] and kBouncer [17]. 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 [36, 37], 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.
Chapter 5

Exploiting Mozilla Firefox

Figure 5.1: Illustration of the attack presented in this thesis. 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.

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 5.1 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 [10].

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.

![Figure 5.2: 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.]

### 5.1 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 5.1. 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
5.1. PREPARATION

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.

```
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 5.1: Required gadgets for calling `mprotect` in Linux (32-bit).

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.

```
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 5.2: The JavaScript program which once compiled will produce the needed gadgets in the JIT buffer.

Based on the above, the ROP chain for calling `mprotect` (see Listing 5.1) 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`. 
5.2 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 5.1 can be located in existing code, even if a sophisticated gadget finder [50] is employed. Notice also that typically ROP exploitation needs some form of memory disclosure because of available defenses based on randomization [8,14,15]. However, these requirements are orthogonal to the techniques presented in this thesis. First, it has been shown that even fine-grained randomization schemes can be defeated if the vulnerability allows arbitrarily reading process memory [10], 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 5.1 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 5.2 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 5.2 we show how we influence the JIT output by assigning particular immediate values to JavaScript variables. For example, assigning the value \texttt{12728721} to a variable will introduce the following assembly code once compiled:

\begin{verbatim}
movl 0xc3c031, 0x6c8(%eax)
\end{verbatim}

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

\begin{verbatim}
xor %eax, %eax; ret;
\end{verbatim}

In the same fashion we can construct all of the gadgets contained in the ROP chain of Listing 5.1 and eventually call \texttt{mprotect} for making the shellcode executable.
Chapter 6

Exploiting Internet Explorer

Listing 6.1: Required gadgets for calling \texttt{Virtual Protect} in Windows (64-bit).

In this chapter 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.

6.1 Why Internet Explorer is different

Version 9 of IE has started employing a JavaScript JIT engine called Chakra \cite{51}. 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 5 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 \texttt{fastcall} is used, and the first function arguments are passed through registers and not through the stack. In the rest of the chapter we describe how we overcome these difficulties.

6.2 Preparation

As before, to exploit IE we must make the page that holds the shellcode executable. This means that we need to call \texttt{Virtual Protect} 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 \texttt{Virtual Protect}
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 6.1. In Listing 6.1 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 5, 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 chapter 3-byte gadgets cannot be constructed trivially (see “Long gadgets” later in this chapter). For constructing the stack-pivoting gadget we need an additional requirement: having control over %al. The reason is discussed later in this chapter.

6.3 Exploit Design Considerations

Similarly to the approach we took in Section 5, 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.

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:

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

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

```assembly
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 5.2, where an immediate value (in our 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:
6.3. EXPLOIT DESIGN CONSIDERATIONS

Figure 6.1: 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 5.2), the overflow bit will be set and thus the flow will follow the conditional jump (the program will be terminate).

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 5. All other gadgets must be constructed using a different technique.
**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 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:

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

We show the compiled version of this code in Figure 6.1. 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 6.1. Then, note that the addition of:

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

has been replaced with:

```javascript
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:

```javascript
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 6.1).

- 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 chapter.
6.4 EXPLOIT IMPLEMENTATION

**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 [52]. Software diversification [53] has been a promising defense mechanism against exploitation, and we have seen it applied with many different strategies [8, 14, 15], as well as used for preventing the attacks we discuss in this thesis [25]. Unfortunately, recently we have seen at least two sophisticated techniques [10, 11], 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.

### 6.4 Exploit Implementation

Listing 6.2: The JavaScript program which once compiled will produce the needed gadgets in the JIT buffer of IE.

```javascript
function r8(addr) {
    return addr + 0x5841;
}

function r9(addr) {
    return addr + 0x5941;
}

function emit_gadgets() {
    for (i = 0; i < 0xc35841; i++) {
        rax = 0xc358;
        rcx = 0xc359;
        rdx = 0xc35a;
        r8(0);
        r9(0);
    }
    return 0;
}
emit_gadgets();
```

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 6.1). Two of the four gadgets (the ones for `%rcx` and `%rdx`) are only 2 bytes in length and thus they can be created with the techniques we analyzed in Section 5 (see lines 12 and 13 in Listing 6.2). 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 6.2. We implemented two JavaScript functions, \texttt{r8()} and \texttt{r9()}, which simply return a fixed value added to their single argument input. These functions, once compiled, produce the following code (for example \texttt{r9()}):

\begin{verbatim}
add %eax, 5941h
jo 00000D71F8F0132
or %rax, %rcx
add %rsp, 30h
pop %rbx
pop %rsi
mov %rsp, %rbp
pop %rbp
ret
\end{verbatim}
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 6.2). The complete JavaScript source for introducing all needed gadgets in the JIT buffer is shown in Listing 6.2 and the stack, along with the way the individual gadgets are chained, is depicted in Figure 6.2.
Chapter 7

Discovering the Gadgets

In Sections 5 and 6 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 chapter we investigate how this can be carried out successfully. Notice, that we assume that a fine-grained randomization scheme has been enabled, like Librando [25] or Chakra.

7.1 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) [10].

Let us briefly discuss the attack presented in [10], which compromises a randomized [14] 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 [50], discover gadgets, and use them to launch the attack.

7.2 Leaking the JIT Buffer

Figure 7.1: 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.

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 [4]. 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 [10] 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 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 (0xffffffff81). 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 disclose 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 5 and the JIT-disclosing technique presented in this chapter 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.

Listing 7.1: JavaScript code that generates a Object which memory layout is described in Figure 7.1.
Chapter 8

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 thesis.

8.1 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 6), 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 [25]. 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 8.2. 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 8.1.

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 conserva-
tive, 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).

Figure 8.1: The cost of constant blinding. In this histogram we present the JIT instructions that were actually executed in a session where the complete suite of SunSpider benchmarks run. In addition we plot the volume of JIT instructions that involve an immediate value (of 1 or 2 bytes), and therefore may be used potentially for creating ROP gadgets. Notice that in all tests the instructions involving an immediate value comprise a significant percentage, between 18–52% of all executed instructions. Therefore applying constant blinding to all immediate values is quite costly, introducing an estimated overhead of 40% to 82%. The overhead is estimated as follows. We assume that the JIT compiler will emit (at least) two more instructions for each instruction involving an immediate value (i.e., 18% of instructions that should be blinded introduce an overhead of 40%). 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 [48], 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 [49], 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
8.2. 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 thesis 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 thesis cannot be easily addressed.

Figure 8.2: 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).

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 [16]. In fact, in this thesis we have demonstrated a similar technique for leaking the location of the JIT buffer (see Section 7), 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 thesis, is Control-Flow Integrity (CFI). [35] This was initially a very promising technique against code-reuse attacks, which quickly drove to implementations [12, 13, 17, 18] that support legacy code and impose negligible overhead. Unfortunately, there are many concerns about the validity of these approaches [19, 20, 36, 37, 54], 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 [21], 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 thesis.

8.2 Proposed Defenses
8.2.1 JIT Analysis

The most obvious defense mechanism is to enhance the JIT compiler with the techniques proposed by G-Free [22] 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.

8.2.2 JavaScript Analysis

As we can observe from the JavaScript payloads used for attacking Mozilla Firefox and IE (see Listing 5.2 and 6.2), 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.
Chapter 9

Conclusion

In this thesis 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 [10] 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.
Chapter 10

Future Work

This work reveals that common defenses against ROP attacks have not been adapted properly for JIT environments. In the case of Internet Explorer there are deployed fine-grained defenses but an attacker can still slip through the cracks and perform an attack. For Mozilla Firefox there is not even the standard defenses such as DEP for JIT buffers. We believe that JIT compilers should incorporate proposed defenses for standard ROP attacks. Constant blinding is one of the strongest defenses available that hinders the attack by making the required gadgets through constants unavailable. IE implements constant blinding but it fails because it assumes that gadgets have to be at least 2 byte.

In the future we will explore the idea of conditional constant blinding. Our evaluation shows that the implementation of constant blinding to all constants is not efficient enough. The concept of conditional blinding is that only the constants that propose a security risk by introducing gadgets should be blinded. This way the overhead of generating a large part of constant blinding code is eliminated. Also, the run-time overhead of running the same code without a reason is minimised only to the cases it is required. This idea requires a way to detect potential gadgets in the existing code or in misaligned code near constants. Already we started implementing a version of constant blinding in Mozilla Firefox that even in version 38 does not have any defense scheme. It would be more appropriate to implement or extend the constant blinding of IE but until now Chakra is not open-source. Preliminary evaluation in Firefox shows that conditional constant blinding introduces minimal overhead and always lower that constant blinding for values over 2 bytes.
Bibliography


BIBLIOGRAPHY


