Abstract—Modern browsers have just-in-time (JIT) compilers that compile JavaScript programs into native binary code on the fly. Since recent JIT compilers put JavaScript data into non-executable memory regions, simply putting shell code as JavaScript data does not work. To overcome this protection, recent attacks exploit constants in JavaScript programs that are compiled into instructions placed in code regions, and use them as small pieces of code, called gadgets, to chain them by return-oriented programming (ROP). To counter this attack, recent browsers introduce constant blinding that encrypts JavaScript constants with a secret key and decrypts them at run time, preventing attackers from inserting arbitrary gadgets. Unfortunately, current browsers (including Firefox, Google Chrome, and Microsoft Edge) only blind constants larger than two bytes for the performance reason, allowing attackers to emit one and two byte gadgets that are known to be sufficient to mount ROP attacks. This paper proposes a high-performance and secure constant blinding technique for JIT compilers. In this technique, we decide whether to blind a constant based on the value of the constant. If a constant includes a value that can be interpreted as a control flow instruction (e.g. `ret` and `jmp`), we blind that constant even if it is two bytes or less. Otherwise, we do not blind the constant because it cannot be used as a gadget. This technique effectively reduces the overhead of constant blinding by reducing the number of constants that must be blinded, while improving the security by eliminating the possibility that even small constants are exploited as gadgets. We implemented this technique in ChakraCore, the JIT Engine of Microsoft Edge, running on x64 systems and measured the performance of the JIT engine. Experimental results confirmed that our technique improved performance by maximum of 2.85% compared with blinding all constants.

Index Terms—Code-reuse attacks, JIT compiler, return-oriented programming, constant blinding

I. INTRODUCTION

Web browsers are found on PCs, tablets, smartphones, smart TVs, gaming consoles, and so forth. Almost everybody surfs the Internet every day. Thanks to the prevalence of the Internet, web browser developers are struggling to pursue the performance or processing speed. Modern browsers such as Google Chrome, Microsoft Edge, Mozilla Firefox, and Apple Safari have their own just-in-time (JIT) compilers. JavaScript programs are compiled to intermediate representations in advance. Then they are re-compiled to native code at runtime by JIT compilers. This mechanism contributes to high speed code execution.

These familiar browsers, however, are often targeted by software exploitations. For example, attackers declare numerous constants in JavaScript programs. Then, after JIT compilation, the attacker can use JITed code as shellcode, if there exists some vulnerabilities in JIT compilers about manipulating the instruction pointer. Constant blinding is one of the defenses against such attacks. It blinds constants in JavaScript programs by XORing with random value.

Besides constant blinding, there are many research on protecting JITed code layouts. INSeRT [41] and librando [26] randomize the JITed code by randomly inserting either illegal instructions (INSRT) or NOP instructions (librando) into the code. Libmask [27] transforms constants into global variables and marks the memory pages for these global variables as read only.

Current constant blinding, however, targets all constants larger than two bytes. Further, due to performance impacts, constants less than two bytes are ignored. Still, recent study showed that you can succeed in exploiting by using only one-byte and two-byte constants.

In this paper, we propose secure and high-performance constant blinding. It only blinds constants that contain values such that they are translated into control flow changing instructions (ret, for example) by JIT compilation.

We implemented the proposal in ChakraCore, the open-source software of Microsoft Edge’s JIT compiler Chakra, and ran several benchmarks. The results showed that the performance was improved by the maximum of 2.85% and indicated the effectiveness of our proposal.

II. BACKGROUND

Memory disclosure attacks have been main attacks against computer systems. They have the ability to execute arbitrary code on remote target systems after hijacking the control flow. Memory disclosures which leak code pointers enable attackers to bypass defense schemes like ASLR and exploit binaries.
ing mitigates indirect JIT-ROP by hiding code pointer destinations and replacing code pointers in readable memory with trampoline layers in execute-only memory.

JIT Spraying [8] [35] (and subspecies of that [2]) is different from aforementioned attacks. Attackers prepare JavaScript programs containing numerous constant values which can be erroneously executed as attacker-controlled instruction sequences by JIT compilation. The attackers use some vulnerabilities such as manipulating the EIP to redirect code flow to the native code. This fact means that the attacker can execute hidden shellcode.

III. CONSTANT BLINDING

To tackle with JIT exploitations, Chakra, the JIT engine of Microsoft Edge, deploys defense mechanism called constant blinding. It masks constants in JavaScript programs by XORing with randomly generated value during JIT compilation. For example, considering that RND_KEY = 0x2511663F, following instruction

\[
\text{mov eax, } 0x3C909090
\]

will be transformed into two instruction sequences by constant blinding. Although constant blinding brings additional overhead (increases instructions), It makes JIT engine safe because the attacker can no longer use the constants he prepared.

\[
\text{mov eax, } 0x1981F6AF \\
\text{xor eax, } 0x2511663F
\]

Existing constant blinding, however, has two problems. First, constants under two bytes are ignored, because of performance overheads. This design is unfavorable because certain research have proposed to exploit JIT compilers using only one or two-byte constants [2]. Second, constant blinding is applied to whatever constants larger than two bytes. This problem causes JIT engines high performance overhead.

IV. DESIGN AND IMPLEMENTATION

Therefore we aim to propose secure and high-performance constant blinding. We concentrate on values which change the control flow of the program. Our proposed constant blinding only blinds constants which contain those values, regardless of their sizes.

For example, "RET" transfers program control flow to a return address located on the top of the stack. Actually, return-oriented programming (ROP), one of the famous exploit techniques [37], uses instruction sequences which end with RET instruction (called gadgets).

We decided the constants to be targeted in our proposal. They are described in TABLE IV [17].

V. EXPERIMENTS AND RESULTS

We use the version 2.0.0.0 of ChakraCore with Clang 5.0.0 and Cmake 3.9.0. As for comparison, we use four different source code: no-modified ("Default"), constant blinding to all constants ("All-Blind"), disable constant blinding ("No-Blind"), and the proposal ("Propose"). For each, we build
ChakraCore and run four benchmarks: JetStream, Octane, Kraken, and SunSpider, all of them are JavaScript benchmarks for web browsers.

The evaluation was performed on a PC with Intel Core i7-3630QM processor, 16.0GB RAM running Ubuntu 16.04 (x64). Also we disabled Intel TurboBoost Technology, Intel HyperThreading Technology, and Intel SpeedStep Technology.

Fig.1, Fig.2, Fig.5, and Fig.6 represents the results of running JetStream, Octane, Kraken, and SunSpider for each four kinds of source code. According to them, for almost all programs (x axis), the values of "Propose" is between "No-Blind", the best performance case, and "All-Blind", the worst performance case.

Fig.3, Fig.4, Fig.7, and Fig.8 represents the improvements of "Propose" over "Default". The key point of this comparison is that both of them are under the same security level. According to them, the benchmark results indicate that "Propose" is superior to "All-Blind". Considering qualitatively, our proposal has benefit of better performance because the target constants to be constant-blinded is less. However, we cannot observe this benefit significantly than we expected. This is because the influence of CPU is stronger than that of our proposal. So we are now conducting on evaluations for ARM CPU, most of IoT devices or mobile devices run under ARM CPUs and therefore this try will be meaningful.

TABLE.II describes performance improvements of "Propose" among four benchmarks. Be careful that both JetStream and Octane measure program processing speed per unit of time, so higher value is better. On the contrary, both Kraken and SunSpider measure execution time, so lower value is better. Based on these benchmark characteristics, you can find that all values go better in the column of "Propose/All-Blind". These results indicate the effectiveness of our proposal.

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>RET</td>
<td>C3, CB, C2, CA</td>
</tr>
<tr>
<td>JMP</td>
<td>E8, FF, 9A</td>
</tr>
<tr>
<td>CALL</td>
<td>EB, E9, EA</td>
</tr>
<tr>
<td>SYSCALL</td>
<td>0F 05</td>
</tr>
<tr>
<td>INT n, INTO, INT 3</td>
<td>CD, CE, CC</td>
</tr>
</tbody>
</table>

**TABLE I**
The list of specific constants which our proposal targets

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Propose/No-Blind</th>
<th>Propose/Default</th>
<th>Propose/All-Blind</th>
</tr>
</thead>
<tbody>
<tr>
<td>JetStream</td>
<td>99.21%</td>
<td>101.24%</td>
<td>102.85%</td>
</tr>
<tr>
<td>Octane</td>
<td>99.61%</td>
<td>100.70%</td>
<td>100.60%</td>
</tr>
<tr>
<td>Kraken</td>
<td>100.15%</td>
<td>98.88%</td>
<td>99.04%</td>
</tr>
<tr>
<td>SunSpider</td>
<td>99.07%</td>
<td>100.13%</td>
<td>98.37%</td>
</tr>
</tbody>
</table>

**TABLE II**
Comparison of relative values for each four benchmarks
Fig. 1. The Results of running JetStream benchmark

![Graph showing the results of running JetStream benchmark](image)

Fig. 2. The Results of running Octane benchmark

![Graph showing the results of running Octane benchmark](image)

Fig. 3. The improvements of "Propose" over "All-Blind" in JetStream benchmark

![Graph showing the improvements of Propose over All-Blind in JetStream benchmark](image)

Fig. 4. The improvements of "Propose" over "All-Blind" in Octane benchmark

![Graph showing the improvements of Propose over All-Blind in Octane benchmark](image)
Fig. 5. The Results of running Kraken benchmark

Fig. 6. The Results of running SunSpider benchmark

Fig. 7. The improvements of "Propose" over "All-Blind" in Kraken benchmark

Fig. 8. The improvements of "Propose" over "All-Blind" in SunSpider benchmark
VI. CONCLUSIONS AND FUTURE WORKS

As the complete eradication of memory disclosure vulnerabilities remains a challenging task, defenses against their exploits is necessary. In particular, web browsers are the most targeted applications due to their popularity and property of marking JITed code with RWX. To defend against such attacks, modern browsers deploy a mitigating method called constant blinding. It prevents JIT Spraying and JIT-ROP, and related attacks by masking integer constants by XORing with a random value to generate the obfuscated constant. However, constant blinding is applied only constants larger than two bytes and therefore bypassed by using only one-byte or two-byte constants.

In this paper, we introduced a modified constant blinding scheme. It utilizes constant blinding to only constants which contain some control flow changing words, regardless of their sizes. Then we demonstrated the performance evaluation on ChakraCore. The results showed that our proposal performs 2.85% performance improvements at most.

You still have rooms to further improve secure constant blinding. It is known that constant blinding is incomplete in that constants in some JavaScript writing manners survive constant blinding [31].

We will work on evaluating our proposal from various points of view to prove its effectiveness objectively in real environment. Also, as there are a lot of cyber attacks targeting IoT devices, we will introduce our proposal to ARM architecture and survey the difference of benchmark results with those of Intel CPU.

REFERENCES


