JITDefender: A Defense against JIT Spraying Attacks

Ping Chen, Yi Fang, Bing Mao, Li Xie

HAL Id: hal-01567598
https://hal.inria.fr/hal-01567598

Submitted on 24 Jul 2017

JITDefender: A Defense Against JIT Spraying Attacks

Ping Chen, Yi Fang, Bing Mao, Li Xie

State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University, Nanjing 210093
{chenping,fangyi,maobing,xieli}@nju.edu.cn

Abstract. JIT spraying is a new code-reuse technique to attack virtual machines based on JIT (Just-in-time) compilation. It has proven to be capable of circumventing the defenses such as data execution prevention (DEP) and address space layout randomization(ASLR), which are effective for preventing the traditional code injection attacks. In this paper, we describe JITDefender, an enhancement of standard JIT-based VMs, which can prevent the attacker from executing arbitrary JIT compiled code on the VM. Thereby JITDefender can block JIT spraying attacks. We prove the effectiveness of JITDefender by demonstrating that it can successfully prevent existing JIT spraying exploits. JITDefender reports no false positives when run over benign actionscript/javascript programs. In addition, we show that the performance overhead of JITDefender is low.

1 Introduction

In recent years, attackers have resorted to code-reuse techniques instead of injecting their own malicious code. Typical techniques are Return-oriented Programming (ROP) [21], BCR [12] and Inspector [11]. Different code-reuse attacks launch the attack based on different codebases, including the application, shared libraries or even the kernel. However, all the techniques need to find useful instruction sequence in the codebase, and the task is tedious and costly in practice.

Recently, a new code-reuse attack named JIT (Just-In-Time) spraying was proposed by Blazakis [10]. It reuses JIT-compiled code on the Flash VM's heap in accordance with the attacker's wish to construct the attack. Later, Sintsov published several real-world JIT spraying attacks on Flash VM [26] and further proposed advanced shellcode which leverages the code on Safari's Javascript engine [25]. JIT spraying can circumvent the techniques such as data execution prevention (DEP) [6] and address space layout randomization (ASLR) [6, 9], which are effective for preventing the traditional code injection attacks. The JIT compiler improves the runtime performance of the VM by translating bytecodes into native machine code. A JIT spraying attack is the process that it coerces the JIT compiler to generate native code with the malicious code on the heap of VM, then it exploits the bugs in the browser or its plug-ins to hijack the control and uses the injected malicious code to achieve the attack. JIT spraying attack have become a big threat to Web Security, because the browser often enables dynamic languages (e.g., actionscript and javascript). JIT spraying attacks have two advantages compared with other code-reuse techniques: First, the malicious code is generated by the JIT compiler, so the attacker needs not to piece up useful code snippets for constructing the attack in the codebase. Second, malicious code is generated on the heap.

This can be combined with heap spraying techniques [29] to increase the probability of the attack.

In this paper, we describe JITDefender, a defense of JIT spraying attacks, which enforces the JIT-code execution control policy for the VM. To defend against JIT spraying attacks, JITDefender changes the VM in the following way: the VM's heap is generally set non-executable so that $W \oplus X$ protection applies. If a specific part of JIT-code has to be executed on the VM based on the definition of the program, the heap is set to be executable, before the code is executed, and immediately set to non-executable afterwards. This paper makes the following contributions:

- We propose JITDefender, an effective technique for defending JIT spraying attacks by controlling the execution of the JIT-code. This techniques distinguishes the benign usage of JIT-code from malicious usage of the attacker.
- We implement and evaluate JITDefender on several commonly used VMs that use JIT compilation (Tamarin flash VM, the V8 javascript engine and Safari's javascript engine). We show that JITDefender can defend against JIT spraying attacks on VMs based on JIT compilation, although the performance overhead of JITDefender is less than 1%.
- We also show that JIT spraying attacks are not only available on flash/javascript VMs, but also available on other VMs based on a JIT compiler, such as QEMU. Our technique is effective for defending against JIT spraying on arbitrary VMs based on JIT compilation.

2 Background: JIT Spraying

JIT Spraying is a code-reuse technique that uses the code generated by the VM based on JIT compilation to launch the attack. By writing the objects using dynamic languages (e.g., javascript, actionscript), the attacker coerces the JIT compiler to generate the malicious code on the heap of VM, and hijacks the control flow to the malicious code snippet. However, different from the existing code reuse techniques, the malicious code does not need to be found within an existing codebase (e.g., libraries, kernel), instead, the attacker predictably defines the objects which are intended to be compiled into the malicious code on the heap of VM, and uses the heap spraying technique to populate the heap with a large number of objects containing the malicious code. Therefore, when the attacker drives the control flow to arbitrary addresses on the heap, with high probability the jump will land inside one of malicious code snippets.

Figure 1 illustrates the JIT spraying attack on Flash VM. First, the attacker defines the actionscript object (`ret` in Figure 1) which contains many uniform statements (`XOR` in Figure 1) with dedicated constructed integers (`0x3c909090` in Figure 1) in the source code. Then the JIT compiler will dynamically translate the source code and generate the native code. In this example, the flash VM will translate the `XOR` as `0x35` and the integer as the original value in the native code. If the attacker carefully constructs the integer value, the native code may be transferred into the malicious code with one byte offset. Suppose the attacker lays out many such objects on the heap, and the attacker can turn the control flow to the malicious snippet (e.g., through a buffer overflow attack), finally this results in the JIT spraying attack.

```
var ret=(0x3C909090^
         0x3C909090^
         0x3C909090^
         0x3C909090^
              …);

         Flash Object
```

```
90 NOP
90 NOP
90 NOP
3C35 CMP AL, 35
      …

Native Code with one byte offset
```

```
359090903C XOR EAX, 3C909090
359090903C XOR EAX, 3C909090
359090903C XOR EAX, 3C909090
359090903C XOR EAX, 3C909090


         Native Code
```

```
pad              pad
shellcode    pad    shellcode
           shellcode

   pad          pad
 shellcode    shellcode

      The heap of VM
```
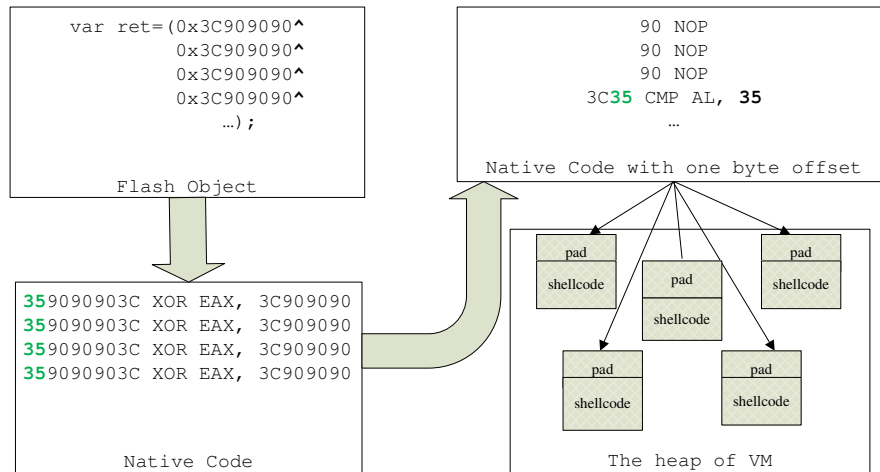
Fig. 1: JIT Spraying Attack

The JIT spraying example illustrated in Figure 1 shows that the code translated by the JIT compiler can be leveraged to launch the JIT Spraying attack. In fact, many other VMs based on a JIT compiler have the same problem. Furthermore, browsers such as IE8 and Chrome are particularly vulnerable to JIT spraying because actionscript/-javascript programs embedded in a web page greatly simplify such attacks.

3   Overview of JITDefender

VMs based on JIT compilation (e.g., Flash VM, Javascript VM) often dynamically translate the source code into the native code on the heap. They necessarily have to mark the page containing the JIT-code as executable and reuse the code repeatedly without re-compiling or interpreting in order to boost the performance. This mechanism implicitly turns the $W \oplus X$ protection off and gives the attacker the opportunity to launch JIT spraying attacks which were illustrated in Section 2. In this paper, we describe the idea of JITDefender, a method to prevent the compiled code from being executed by the attacker. The main idea is to re-enforce $W \oplus X$ protection within the VM. More precisely, we generally mark the native code pages as non-executable. When the VM executes JIT-code, we change the corresponding code pages to executable, after executing the code, the code pages are reset to non-executable again.

When designing JITDefender, we need to identify two points in the the code base of the JIT execution: (1) the *code compilation point*, i.e., the point when the JIT compiler generates the native code, and (2) the *code execution point*, the point when the VM executes the native code. The workflow of JITDefender is that we mark the code pages as non-executable at the first point. Shortly before the second point we mark the pages as executable, and shortly after we mark the pages as non-executable again. This mechanism can be applied to arbitrary VMs based on JIT compiler. Under this protection of

```
┌─────────────────┐          ┌──────────────────────────────────────────┐
│ .abc/.swf parser │          │              JIT Compiler                 │
└─────────────────┘          │     ┌──────────────────────────────┐      │
        │                    │     │      MIR Code Generator        │      │
        ▼                    │     └──────────────────────────────┘      │
┌─────────────────┐          │                   │                        │
│Bytecode Verifier │─────────────▶    ┌──────────────────────────────┐    │
└─────────────────┘          │     │      MD Code Generator          │    │
        │                    │     └──────────────────────────────┘    │
        ▼                    │                   │                        │
┌─────────────────┐          │     ┌──────────────────────────────┐    │
│   Interpreter    │          │     │        Native Code              │    │
└─────────────────┘          │     │    Execution Controller         │    │
        │                    │     └──────────────────────────────┘    │
        │                    └──────────────────────────────────────────┘
        │                                       │
        ▼                                       ▼
        ┌──────────────────────────────────────────┐
        │              Runtime System                │
        └──────────────────────────────────────────┘
                            │
                            ▼
        ┌──────────────────────────────────────────┐
        │    Memory Manager / Garbage Collector     │
        └──────────────────────────────────────────┘
```
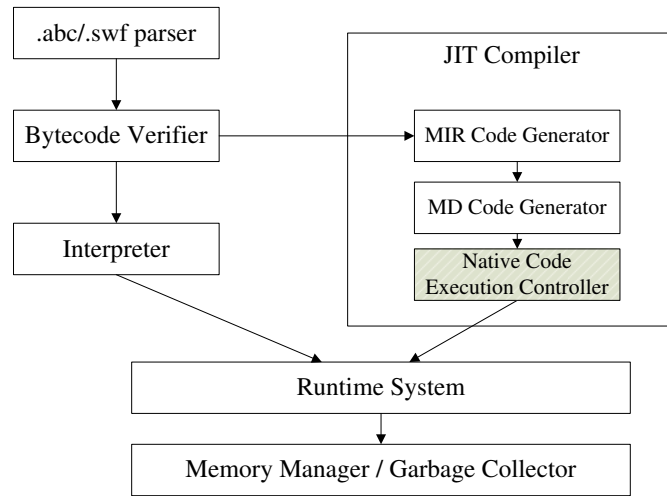
Fig. 2: Tamarin Flash Engine

JITDefender, if the attacker hijacks the control flow to the code snippet on the heap for JIT spraying attack the access will be blocked because the VM keeps the code pages non-executable. In fact, JITDefender provides different views of the compiled code for the VM and the attacker with the native code execution control policy.

4 Design and Implementation

In this section, we firstly take the Flash VM as an example to illustrate our method. We identify code parts in the VM that define code compilation and code execution points. Then we demonstrate that our method can be applied to Javascript VM.

4.1 Introduction of the Flash Engine

The source code of flash is written in the actionscript language. Through the different actionscript compilers, the source code is translated into the actionscript byte code (ABC) or the ShockWave File (SWF). The code can be JIT compiled or interpreted on the flash engine. Flash engine handles the ABC or SWF file in the following steps as shown in Figure 2: first it passes through the ABC or SWF files, translates them into the objects which are stored in the object pool. Then it steps into the compiling phase or interpreting phase. In this paper, we only focus on the compiling phase where the JIT compiler fetches the object from the pool, then compiles it into the native code. This is the code compilation point defined above. JIT compiling can be divided into MIR code generation and Machine Code (MD) generation. The native code is stored in the newly allocated heap memory of flash VM. Third, the flash VM provides the loop monitoring whether there is the compiled code on the heap. If so, it will execute the native code.

This is the code execution point defined above. In order to prevent JIT spraying attack, we manipulate the *Native Code Execution Controller* on VMs, which is between the MD code generation and Runtime code execution. In the following subsection, we will illustrate the mechanisms implemented in Flash/Javascript VMs in details.

4.2 Adapting the Flash Engine

In Flash Engine Tamarin, when an ABC or SWF file is loaded, the JIT compiler will translate the source code into native code at the unit of a function. More specifically, Tamarin uses the class `MethodInfo` to store the information of the functions that can be executed by the VM, including user-defined functions, native functions and so on. Another key class named `CodeMgr` is used to manage memory for compiled code, including the code itself (in a `nanojit::CodeAlloc`), and any data with code lifetime (in a `nanojit::Allocator`), such as debugging info and inline caches. In order to set the attributes of the code pages, we should get the the compiled code information at the unit of function. Therefore, we add the variable `CodeMgr* mgr` in the `MethodInfo` class. It will provide us the convenience for setting attributes of the code pages, because we can get the information of the function including the native code generated by this function. We now give more details how to set the page attributes.

First, we need to get related information at the code compilation point when native code has been generated. We find that, after Tamarin generates the compiled code for one function, it will store the compiled code information in the `codeMgr` which is the member variable of the class `PoolObject`, and `PoolObject` is a container for the pool of resources decoded from an ABC file. Therefore, we will transfer the compiled code information into the new variable `mgr` we defined in `MethodInfo`. Then we extract the start and end address of the compiled code memory, and invoke the function named `VMPI_setPageProtection` to set the related pages as non-executable.

Second, we need to find the code execution point. As mentioned in Section 3, we should set the related code pages as executable before executing the compiled code, and after executing the compiled code, we set the compiled code pages as non-executable again. In the Tamarin flash engine, the function `coerceEnter` is used to execute the compiled code, at the end of this function, the specific handler function is invoked by `endCoerce(argc, ap, ms)`. As such, we leverage the function `VMPI_setPageProtection` to set the related pages as executable, and after executing the function, we use the same function to set the related pages as non-executable. Note that we calculate the related pages based on the recorded information of `mgr` in the `MethodInfo`.

The method mentioned above can successfully protect the compiled code on the heap of the VM. Because the compiled code can be only executed by the VM itself, but can not be executed by the attacker.

4.3 Javascript Engine

Sintsov [25] shows that JIT Spraying can be also mounted on the Javascript Engine in Safari [4]. Similar to the flash engine, the attacker can construct the Javascript object using the `XOR` operation with the specific integer operands. Then the Javascript Engine

will compile the Javascript objects into the native code which contains the malicious code. We find that the same problem occurs in the V8 Javascript engine – the javascript engine of Chrome [18]. We could leverage a known buffer overflow [1] to launch a JIT spraying attack. In this section, we demonstrate that JITDefender can be implemented on both Javascript engines.

V8 Javascript Engine As we mentioned in Section 3, we need to identify the code compilation and the code execution point. The V8 Javascript engine provides two API function for compilation and execution respectively. The `Compile` function is used to compile the Javascript program into the native code on the heap, and the `Run` function is used to execute the compiled native code.

In the compilation phase, the V8 Javascript engine parses the Javascript files and divides the code into two parts, one is the specific function such as the `eval`, the global function, and other functions are regarded as the shared function. Then it compiles the Javascript code into the native code according to the function categories. And the native code is stored as `SharedFunctionInfo` at the unit of the function. `SharedFunctionInfo` is the child class of `HeapObject`, which maintains the information of the Javascript objects.

In the execution phase, V8 gets the compiled code by using the `Code::GetCode` method, and finds the entry to the code by the `Code::entry` method, then it jumps to the code entry to execute the code on the heap using `Execution::Call`. Note that the class `Code` is the child class of `HeapObject`, it contains the native code generated at the compilation point. Similar with the flash engine, the compiled code is laid out on the heap and its page is marked as executable, which is independent of whether the code is executed by the Javascript engine or not.

We applied our code control policy to the V8 javascript engine. First, at the end of `Compiler::Compile`, we use the Windows API function `VirtualProtect` to set the compiled code pages as non-executable. Then before the `Execution::Call`, we use `VirtualProtect` to set the code pages as executable, and after executing the compiled code, we set the code pages as non-executable again.

Safari's Javascript Engine Similar to the V8 Javascript Engine, at the code compilation point, Safari's Javascript Engine compiles the Javascript code into the native code according to the function definition. The native code is saved in the structure of `JITcode`. At the code execution point, Safari Javascript Engine will query the native code base, invoke the entry to the current executed function and then execute it. We modify Safari's Javascript Engine (JavascriptCore) at these two points: First, Safari's Javascript Engine gets the JITed code by the method `JIT::compile` in memory space in the form of `JITcode`. Then we set the code pages as non-executable. Second, the Javascript Engine executes the JITed code by using the method `JITStubCall::call`. Before we invoke this function to execute the JIT-code, we first set the code pages as executable. After we executed the code, we reset the code page as no-executable again.

5 Evaluation

In this section, we describe the experimental evaluation of our JITDefender prototype. First, we test JITDefender's ability to dynamically defend the JIT spraying attack. Second, we measure the performance overhead of JITDefender. The evaluation is performed on an Intel Pentium Dual E2180 2.00GHz machine with 2GB memory and Windows 7. Tested programs are compiled by Microsoft Visual Studio 2008.

Table 1: JIT Spraying Attacks Tested on JITDefender

VMs	JIT Spraying Attacks	LOC(K)	Description	JITDefender
Tamarin	SAP-Logon7-System [24]	3K	SaveViewToSessionFi ActiveX Buffer Overflow	✓
	QuikSoft-STAGE0 [23]	3K	SubmitToExpress ActiveX Buffer Overflow	✓
V8	SaveAs-JITSpray	3K	"SaveAs" Buffer Overflow in Chrome	✓
Safari's JS Engine	Safari_parent_close_sintsov [22]	3K	Safari 4.0.5 parent.close() (memory corruption)	✓

5.1 Effectiveness

Since JIT spraying is a new attack, there are little attack samples published. Therefore we used existing JIT spraying attacks published by Sintsov for Tamarin flash engine and Safari's Javascript engine and wrote samples for the V8 javascript engine by ourselves. More specifically, we chose two JIT spraying attack samples written by Sintsov to evaluate Tamarin flash engine: "SAP-Logon7-System" [24] and "QuikSoft-STAGE0" [23]. The two JIT spraying attacks leverage the buffer overflow vulnerability in SAPGUI 7.10 ActiveX and Oracle Document Capture (EasyMail Objects EMSMTP.DLL 6.0.1) ActiveX Control respectively, and can successfully launch attacks on IE8. Note the two original JIT spraying attacks leverage Flash Player 10's flash engine [26]. In our experiment, we use the Tamarin Flash engine instead. For Safari, we chose the JIT Spraying attack "Safari_parent_close_sintsov" published by Sintsov [22]. It exploits the vulnerabilities in Safari 4.0.5 `parent.close()` to launch attacks. In addition, in order to test the effectiveness of our tool on the V8 Javascript engine, we wrote one JIT spraying code by leveraging the buffer overflow "SaveAs" in Chrome [1]. We named the attack as "SaveAs-JITSpray". With all the attacks mentioned above, we tested the effectiveness of JITDefender for detecting the JIT spraying attacks, including Tamarin, V8, and Safari's javascript engine. Experimental results in Table 1 show that JITDefender can successfully defend JIT spraying in VMs based on JIT compilation. We also tested JITDefender on benign code, namely the performance benchmarks embedded in the JIT VMs. The actionscript/javascript programs are listed in Table 2. Overall, we found no false positives.

5.2 Performance Overhead

We also measured the performance overhead of JITDefender. Because JITDefender modifies the JIT-code page attributes of the VM at runtime, it will bring some overhead

Table 2: Performance Overhead of the JIT VMs under JITDefender

VMs	Benchmarks	LOC(K)	Original VM	JITDefender	Performance Overhead
Tamarin	SOR.as	3	72.844s	72.999s	0.2%
	Heapsort.as	3	9.980s	10.325s	3.5%
	SparseMatmult.as	4	0.958s	0.983s	2.6%
	FFT.as	5	10.334s	10.552s	2.1%
	Series.as	6	4.661s	4.763s	2.2%
	LUFact.as	12	27.827s	27.989s	0.6%
	Moldyn.as	14	6.101s	6.276s	2.9%
	Crypt.as	15	0.314s	0.322s	2.5%
	RayTracer.as	22	1.566s	1.573s	0.4%
	Euler.as	81	0.246s	0.249s	1.2%
	Average		13.483s	13.603s	0.9%
V8	crypto.js	48	5.189s	5.196s	0.1%
	richards.js	16	2.107s	2.112s	0.2%
	deltablue.js	26	2.100s	2.113s	0.6%
	raytrace.js	28	2.129s	2.134s	0.2%
	earley-boyer.js	195	5.198s	5.213s	0.3%
	regexp.js	105	4.274s	4.287s	0.3%
	splay.js	11	3.575s	3.629s	1.5%
	Average		3.510s	3.526s	0.5%
Safari's JS Engine	crypto.js	48	5.336s	5.419s	1.6%
	richards.js	16	2.277s	2.288	0.5%
	deltablue.js	26	2.246s	2.301s	2.4%
	raytrace.js	28	4.309s	4.339s	0.7%
	earley-boyer.js	195	12.496s	12.792s	2.4%
	regexp.js	105	15.944s	15.959s	0.1%
	splay.js	11	4.911s	4.918s	0.1%
	Average		6.788s	6.859s	0.1%

to the VMs. In this section, we chose the three VMs based on JIT compilation to evaluate the performance overhead of JITDefender, including Tamarin Flash Engine, Safari's Javascript Engine and V8 Javascript Engine. Table 2 shows the performance overhead of JITDefender when it is applied to the VMs when executing the actionscript/javascript programs. For each VM, we run the benchmark of it, and compare the time costs when the tested program running on the original VMs and the modified VMs. By comparison, we can see that JITDefender has less than 1% costs on VMs. Generally speaking, the performance overhead is proportional to the number of function chunks in the program. This is because the more function chunks of the program exist, the more frequently JIT VMs will transfer the JIT objects into different functions' native code and execute these codes individually. Since we modify the code pages' attributes, the more function chunks of the programs there are, the more performance overhead will be introduced to program's execution. Take the experimental results in Table 2 for instance, the test case "Heapsort.as" uses recursion techniques to sort arrays. It frequently invokes the small function `NumHeapSort`, which performs a heap sort on an array. Therefore, when the Tamarin flash engine compiles "Heapsort.as" into native code, the code pages will be

marked as "non-executable" or "executable" alternatively. This is the reason why the performance overhead of executing "Heapsort.as" under JITDefender is 3.5%, which is more than the average.

6 Discussion

6.1 JITDefender on Other VMs

QEMU [3] is the commonly used CPU-emulator, which leverages the JIT techniques. We discovered that the programs running in QEMU will cast their code on the heap and mark it as executable. We note that JIT spraying attacks may be constructed on it too. To perform such an attack, we need to install a VM on QEMU. Within the VM, we need to construct a malicious program, which contains the shellcode. Then we fork several processes for running the program that keep on spraying malicious code on the heap of QEMU. In order to construct the attack, we need to leverage one of the bugs in QEMU and drive control to the malicious code [28]. Since QEMU is used as CPU-emulator in KVM [2], this effectively means that the approach can be leveraged to construct an attack on Cloud Computing. So JIT spraying may threaten the security of the Clouds. Similarly with to flash and javascript engine, it should be possible to implement JITDefender on QEMU. In fact, we believe JITDefender can be applied to arbitrary VMs based on a JIT compiler to defend against JIT Spraying exploits.

6.2 Circumventing JITDefender

The method proposed in this paper addresses the problem that most Just-In-Time compilers leave a large window of opportunity for exploiting code "sprayed" into the compiled code. We reduce the window to the minimum time so that only the VM based on JIT compilation can set the compiled code as executable at page granularity. People may argue that is it possible that the attacker exploits the vulnerability within the window which spans from VM setting the page as executable to VM executing the code on the page. We think this potential JIT spraying attack is feasible in theory but we did not find it in practice. The potential JIT spraying attack is that the malicious code is on the same page with the code that is executing on VM when the attacker exploits. In that case, JITDefender will keep the page as executable, as such, JIT spraying will make JITDefender ineffective. Although we did not find such attacks till now, we consider the attack is possible. For example, the attacker introduces the delay method (e.g., loop) in certain function to keep the code executing and the page containing the malicious code. Even worse, if the program is asynchronous, for example, it uses multi-threaded methods to load multiple objects on the heap, and keep all the pages executable by the delay method, it will give a big opportunity for the attacker to circumvent JITDefender. Although nobody has proposed the attack till now, in theory, it will be a threat to JITDefender. To counter the attack, we consider to add some optimization to the VM, for example, we can pre-calculate the `XOR` operation, and get the result directly without translating its operands on the heap.

7 Related Work

7.1 Heap Spraying Defenses

Heap spraying [29] is a technique that will increase the probability to land on the desired memory address. The act of spraying simplifies the JIT spraying attack and increases its likelihood of success. There are several defenses specifically designed against heap-spraying attacks [14, 16, 20]. Nozzle [20] is the countermeasure specifically designed against heap-spraying attacks by the analysis of the contents of any object allocated by the Web browser. Similarly, Egele et al. [14] propose an emulation method to detect heap spraying attacks with drive-by downloads. Based on libemu [5], it emulates the code download and checks whether there is malicious code. Bubble [16] is the Javascript engine level countermeasure against Heap-spraying attacks. It introduces the diversity of the heap by inserting special interrupting values in strings at random positions when the string is stored in memory and removing them when the string is used by the application. All the existing Heap spraying defenses all rely on the assumption that the malicious code is introduced from the outside (network, keyboard), therefore they may be circumvented by JIT spraying attack.

7.2 JIT Spraying Mitigation

Concurrently and independently, Bania [8] proposed a heuristic detection method, based on the assumption that JIT spraying attacks use arithmetic operations. They detect JIT spraying by calculating the number of bad instructions. However, JIT spraying may not use the arithmetic operations to generate the malicious code. Tao et al. [27] propose the code randomization techniques on VMs based on JIT compilation. However, it has 5.9% space and 5% runtime overhead. In addition, they currently implemented the prototype on V8 engine only. Our techniques are quite different compared to theirs. First, we do not assume any particular form of JIT spraying attack (e.g., using XOR). Second, our method can prevent JIT spraying with less performance overhead (less than 1%). Third, our method has been implemented and tested on different VMs, and proved to be easily deployed on other VMs in practice. Most recently, de Groef et al. [17] proposed a kernel patch JITsec that defeats JIT spraying based on testing certain restrictions when invoking a system call. Different from JITDefender, JITsec can only defeat the malicious code that uses the system call not general code.

7.3 Other Defenses

Most recently, Payer [19] summarized the different attack types and defenses. Generally speaking, all the attacks leverage software bugs and maliciously craft the control or non-control objects to achieve malicious behavior. To defeat these attacks, researchers proposed CFI [7] and DFI [13] that protect against sensitive objects. XFI [15], a kernel module, leverages static analysis for the guard and checks the jump's target, in addition, it uses two stacks to guarantee the return address. There are intrinsic differences between XFI and JITDefender. First, JITDefender prevents the code snippet in the function from being reused from outside, while XFI controls the function callsites and

prevents it being maliciously invoked. Second, JITDefender prevents the JIT spraying attack, while XFI prevents the code injection attacks.

8 Conclusions

In this paper, we present the design, implementation, and evaluation of JITDefender, a tool for defeating JIT spraying attacks. JITDefender applies code execution control on the VMs, and to the best of our knowledge, it is the comprehensive defense for JIT spraying. The evaluation of JITDefender shows that it has no false positives, and the performance overhead is low.

9 Acknowledgements

References

1. Google chrome 0.2.149.27 'saveas' function buffer overflow vulnerability. http://seclists.org/bugtraq/2008/Sep/70
2. KVM. www.linux-kvm.org/
3. QEMU. http://wiki.qemu.org/Main_Page
4. The Webkit open source project. webkit.org/
5. x86 shellcode detection and emulation. http://libemu.mwcollect.org/
6. The Pax project (2004). http://pax.grsecurity.net/
7. Abadi, M., Budiu, M., Ligatti, J.: Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS). pp. 340–353. ACM, New York, NY, USA (2005)
8. Bania, P.: JIT spraying and mitigations. (2010). http://arxiv.org/abs/1009.1038
9. Bhatkar, E., Duvarney, D.C., Sekar, R.: Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In: Proceedings of the 12th USENIX Security Symposium. pp. 105–120 (2003)
10. Blazakis, D.: Interpreter exploitation. In: Proceedings of tth USENIX Workshop on Offensive Technologies (WOOT'10). pp. 1–9 (2010)
11. Kolbitsch, C., Holz, T., Kruegel, C., Kirda, E.: Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. Proceedings of the 30th IEEE Symposium on Security and Privacy. pp 29-44 (2010)
12. Caballero, J., Johnson, N.M., McCamant, S., Song, D.: Binary code extraction and interface identification for security applications. Proceedings of the 17th Annual Network and Distributed System Security Symposium (2010)
13. Castro, M., Costa, M., Harris, T.: Securing software by enforcing data-flow integrity. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7. pp. 11–11. USENIX Association, Berkeley, CA, USA (2006)

14. Egele, M., Wurzinger, P., Kruegel, C., Kirda, E.: Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In: Proceedings of Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA). pp. 88–106 (2009)
15. Erlingsson, U., Valley, S., Abadi, M., Vrable, M., Budiu, M., Necula, G.C.: XFI: Software guards for system address spaces. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7. pp. 6–6. USENIX Association, Berkeley, CA, USA (2006)
16. Francesco, G., Yves, Y., Wouter, J.: Bubble: A javascript engine level countermeasure against heap-spraying attacks. In: Proceedings of the 2010 ESSoS (2010)
17. de Groef, W., Nikiforakis, N., Younan, Y., Piessens, F.: Jitsec: Just-in-time security for code injection attacks. In: Benelux Workshop on Information and System Security (WISSEC 2010) pp. 1–15. (2010)
18. Google Inc.: V8 javascript engine. `code.google.com/apis/v8/intro.html`
19. Payer, M.: I control your code attack vectors through the eyes of software-based fault isolation. In: 27C3 (2010)
20. Ratanaworabhan, P., Livshits, B., Zorn, B.: Nozzle: A defense against heap-spraying code injection attacks. In: Proceedings of 18th USENIX Security Symposium (2009)
21. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security(CCS). pp. 552–561. ACM, New York, NY, USA (2007)
22. Sintsov, A.: JIT spraying attack on safari. `http://www.exploit-db.com/exploits/12614/`
23. Sintsov, A.: Oracle document capture (easymail objects emsmtp.dll 6.0.1) activex control bof - JIT-spray exploit. `http://dsecrg.com/files/exploits/QuikSoft-reverse.zip`
24. Sintsov, A.: SAP GUI 7.10 webviewer3d Activex - JIT-spray exploit. `http://dsecrg.com/files/exploits/SAP-Logon7-System.zip`
25. Sintsov, A.: JIT-spray attacks & advanced shellcode (2010). http://dsecrg.com/files/pub/pdf/HITB%20-%20JIT-Spray%20Attacks%20and%20Advanced%20Shellcode.pdf
26. Sintsov, A.: Writing JIT-spray shellcode for fun and profit. In: Technical Report of Digital Security (2010)
27. Tao, W., Tielei, W., Lei, D., Jing, L.: Secure dynamic code generation against spraying. In: CCS '10 poster. pp. 738-740. ACM, New York, NY, USA (2010)
28. Wang, T.: Integer overflow on QEMU. http://lists.nongnu.org/archive/html/qemu-devel/2008-08/msg01052.html
29. Wikipedia: Heap spraying (2010), `http://en.wikipedia.org/wiki/Heap_spraying`