SoK: JIT ROP Protection

Koyena Pal, *Brown University*

Abstract

After the introduction of JIT-ROP as an attack technique against ASLR, there have been various mitigations proposed as well as other sophisticated JIT-ROP attack instances invented that break the defense mechanisms available in systems to prevent JIT-ROP attacks. Due to the vast availability of scripting environments built into modern software, like JavaScript in web browsers, programs can still be vulnerable to JIT-ROP attacks.

Hence, this paper systematically explores JIT-ROP related well-known defenses to categorize different ways researchers have approached this issue as well as point to various trade-offs and challenges such as metrics used to evaluate defenses.

1. Introduction

Over the years, various protection mechanisms have been used by programs as an attempt to mitigate JIT-ROP attacks, a novel class of code-reuse attacks. However, it is evident from the large number (about 1000) of memory disclosure vulnerabilities discovered per year over the last five years [1] that programs are still vulnerable to code-reuse attacks. When a memory disclosure bug is present, a typical code reuse attack would entail finding usable code gadgets, building a payload, and executing it in order to hijack the control flow of the vulnerable program. With the JIT-ROP attack technique, attackers can leverage such a bug along with a scripting environment, which is available in all current browsers, in order to read a program's code, even after it may have been randomized.

While the obvious solution might be to ensure that attackers do not get access to the codebase and cannot run out-of-order instruction sequences of any part of the program's code, it is known to be fundamentally hard to do so within a reasonable cost and performance overhead. Over the last decade, a set of defenses have been developed wherein some created a custom hypervisor [5, 6, 7, 9, 12], some modified the compiler [4, 11, 15], OS kernel [5, 9] or runtime [4, 11], and the rest created general defenses against most known types of ROP through continuous re-randomization [10]. However, these defense mechanisms were either not used in practice or were compromised by other attack vectors. If they were not used in practice, it was mainly because of one or more of the following factors: the *performance or memory overhead* hindered more than the benefit gained by the protection, the approach required *publicly unavailable tools or required huge changes* in the system and *may not have been compatible* with them as a result. It could also be that the offered protection was incomplete.

With such a variety of proposed protections and attacks, it can be difficult to understand the efficiency, effectiveness, approach, and challenges that different solutions have. Hence, this paper's motivation is to systematize and evaluate previously suggested solutions. The systematization is done by categorizing various types of JIT-ROP attacks as well as classifying defenses based on the step/phase of JIT-ROP exploit they are trying to prevent. We then evaluate these defenses' performance, compatibility, and robustness. With this, we set general criteria that can be used as a measurement to satisfy for successful deployment of a proposed JIT-ROP-related defense. The paper does not cover every single proposed solution, but it rather analyzes promising proposals and indicates fundamental issues and challenges.

Therefore, we make the following contributions:

- classify different JIT-ROP attacks and defenses and create a general threat model;
- identify, evaluate and compare proposed defenses for robustness, performance, and compatibility;
- discuss the inadaptability of suggested solutions and pinpoint challenges and criteria to be met by new solutions.

The paper is organized as follows. Section 2 describes and classifies known variations of JIT-ROP attacks. Section 3 discusses and classifies popularly known protections and their main weaknesses. Section 4 sets up the evaluation criteria, which is used as the basis of our analysis and discussion in Section 5. Section 6 concludes the paper.

2. Taxonomy of JIT-ROP Attacks

While there is a large spectrum of attacks that can take advantage of memory disclosure vulnerability, the following categories are most relevant for JIT-ROP Attacks.

2.1. Direct JIT-ROP

In this category [2], the attacker is assumed to know a valid code address. With this knowledge, they can employ the memory disclosure recursively. By doing so, they can harvest code pages and find gadgets to conduct a ROP

attack. After executing the attack payload, they have hijacked control.

2.2. Indirect JIT-ROP

In this category, the attack is dependent on leaking code pointers [5]. By harvesting code pointers from data pages, attackers can infer the location of the gadgets without ever reading them.

To have a generalized threat model, we can assume that the target system has code-injection protection mechanisms to prevent code injection attacks. Furthermore, the attacker is assumed to have access to a memory disclosure vulnerability that can be invoked a limited number of times. Lastly, any memory read or write that violates memory permissions causes a detectable crash. Such assumptions are consistent with the threat models described in the JIT-ROP offense and defense papers.

3. Taxonomy of Defenses

Because of the various types of attacks described in Section 2, there have been a bunch of defenses tackling certain parts of the attacking process. We separate these approaches into the following categories:

3.1. Pointer Indirection

A couple of the notable defenses under this approach are Oxymoron [3] and CPI [4]. Oxymoron, for instance, is a defense that aims at hiding direct code and data references by generating randomized code so that there are no direct references to code pages. This is the solution proposed after Snow et al. [2] revealed the first type of JIT-ROP attack. While it was effective in terms of blocking the attack described in [2], JIT-ROP with indirect disclosure attacks were still possible because adversaries would use indirect code references.

3.2. Memory permissions

The defenses that come under this category include XnR [9], HideM [7], Readactor [5], etc. The common goal of these defenses was to prevent the attacker from using the memory disclosure vulnerability to follow the code pointers in memory and read code pages. By preventing this part of the attack scheme, the defenses essentially stop the attack from dynamically searching for gadgets in the code sections that are non-readable.

XnR mark code pages as “non-present” and check permissions inside a custom page fault handler [9]. HideM also performs permission checks on memory accesses to do the same. Unlike XnR, Readactor uses hardware support to realize their goal. It utilizes Extended Page Table (EPT) permissions to implement execute-only permission and adds trampolines (a layer of indirection) to prevent indirect disclosure.

Even though these implementations mark JIT-compiled code as not readable, they allow JIT code pointers to be leaked via JIT-compiler. Maisuradze et al. [14] show that an attacker that controls a JavaScript code can run code-reuse attacks on the code generated by JIT compilers. Another instance where such defenses fall short is explored in the paper written by Rudd et al. [13]. The paper demonstrates that even though such techniques prevent leakage, they are fundamentally unable to stop code reuse. In response to such vulnerabilities, leakage resilient diversification [11] was proposed. It combines execute-only memory with fine-grained ASLR and function trampolines. By doing so, code pages cannot be read and inferred. However, these are huge system modifications, which makes it a bit incompatible to deploy.

3.3. Destructive code read

Heisenbyte [6] and NEAR [12] are a couple of the notable defenses that come under this category. Such defenses prevent executable region leakages by ensuring that any code-area read is destructive, i.e., the code is garbled right after it is read. By doing so, the attacker would not be able to leverage the memory disclosure bug in both static and dynamically generated JIT code. The limitations for such defenses are usually that they cannot handle code that read or write to itself (for instance, self-modifying code). Heisenbyte, for instance, can’t detect Blind-ROP. However, as we are only focusing on JIT-ROP, this is out of scope.

3.2. Continuous re-randomization

Defenses such as Remix [15] and Shuffler [10] fall under this category. At this stage, researchers started realizing that continuous re-randomization can be used to defend against JIT-ROP. The idea behind this defense is to re-randomize code between the time it is leaked and the time when a gadget chain is invoked. By re-randomizing during that time, the attack gadget would no longer be valid as it does not exist.

Remix implements this idea by continuously re-randomizing the basic block ordering within functions. However, this approach is vulnerable to attacks that use function locations or reuse function pointers. Shuffler, on the other hand, continuously re-randomizes code locations on the order of milliseconds. This introduces a real-time deadline for the attacker. However, such an approach can have an unpredictable shuffling latency under load.

4. Evaluation Criteria

In this section, we define requirements and set some targets that a prospective defense would want to fulfill in order to have a higher chance of being deployed. This is consistent with the SoK paper written by Szekeres et al. [18].

4.1. Protection

The protection's strength is determined by what policy it enforces. Its practicality is determined by the attacks it can protect against. For JIT-ROP specific defenses, if the defense can protect against JIT-ROP with direct and indirect disclosure for static and dynamic code, that would be considered a really strong defense. However, its accuracy is just as important. This is determined by the number of false positives and negatives that this defense creates.

False-positive. False-positive occurs when a situation is considered as an attack when in reality, it is not. Deployed defenses must not, under any circumstance, lead to a false positive as this causes unnecessary and potentially harmful disruptions in the production environment. Hence, when it comes to the prospective defense's accuracy status, it should not have any false-positive cases.

False-negative. False-negative occurs when a situation is considered a normal operation when in reality, it is an attack. When it comes to a probabilistic defense (which is what randomization defenses are), the probability of a successful attack is greater than 0. For deterministic approaches, the probability is greater than or equal to 0. A deployable defense either has no false negatives or a negligible amount (in the case of probabilistic).

4.2. Cost

Performance overhead. The primary cost that protection mechanisms need to care about is the performance overhead as speed is almost as important as security.

Most deployed defenses have used CPU-bound and I/O bound benchmarks to measure performance. In the next section, we see that all the mentioned defenses have used SPEC [17]. Based on the results and knowing what defenses were deployed, an acceptable performance overhead usually ranges from 0% to 10%.

Memory overhead. Due to codebase modifications or the addition of code and data, significant memory overhead can be introduced. This overhead is usually measured by comparing the sizes of the codebase compiled by the original version and the modified version, which is equipped with the defense mechanism. Although it is great to have similar ranges of acceptable performance overhead, having a higher memory overhead creates a lesser issue than performance overhead.

4.3. Compatibility

For compatibility, the more compatible a defense has with the current technology, the more likely it can be deployed.

Source compatible. We consider a defense to be source compatible if source code is not required in order for the defense to protect the application. With a source-compatible defense, companies would not have to worry about providing public access or have a human intervention to modify their source code.

Binary compatible. Binary compatibility means that the defense can still work without modifying binary modules. This allows for legacy (backward) compatibility, which implies that protection can be extended to legacy libraries. In addition, it protects the program from another binary that might be unprotected and hence, exploitable.

Module compatible. Module compatibility implies that individual modules can be handled by themselves. For instance, a compiler-based defense should be able to support different module compilations.

5. Discussion

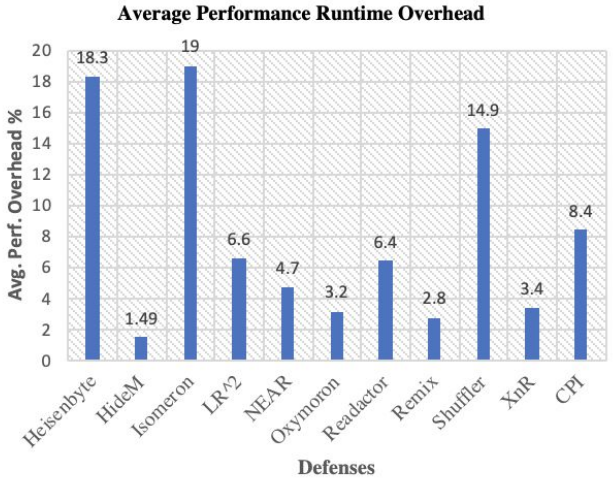

Figure 1: A bar graph illustrates the average performance runtime overhead that JIT-ROP related defenses have. The x-axis refers to the defenses and the y-axis refers to the value of the avg. performance overhead which is in terms of percentage. Heisenbyte [6] - 18.3%, HideM [7] - 1.49%, Isomeron [8] - 19%, LR^2 [11]- 6.6%, NEAR [12] - 4.7%, Oxymoron [3] - 3.2%, Readactor [5]- 6.4%, Remix [15] - 2.8%, Shuffler [10] - 14.9%, XnR [9]- 3.4%, CPI [4] - 8.4%.

Figure 1 represents the average performance overhead of the defenses mentioned in Section 3. The numbers are taken from the respective defense papers. While they all used SPEC CPU 2006, all these measurements were taken in different configurations and environments. Hence, Figure 1

only provides a roughly estimated comparison. From this graph, it seems that HideM [7] is the fastest defense. However, HideM is incompatible with modern hardware. In addition, it requires human intervention in order to separate data and code, which implies that it is also Source incompatible. Therefore, even though it has great performance, its incompatibility with current technology has made it hard for it to be deployable. Hence, from this instance, we gather that when we have to decide between high performance (speed) and compatibility, compatibility takes precedence.

The second-lowest performance overhead is the Remix defense. Remix [15] is one of the general defenses that use a re-randomization technique to defend against code reuse attacks. Based on their paper, it seems that they have the potential to stop JIT-ROP attacks. However, it is not always effective. Despite that, because it is generalized, source compatible, and has less cost, it is highly preferable for it to be deployed.

Although Shuffler [10] has a much higher performance overhead than Remix, this defense does not require source, kernel, compiler, or hardware modifications. It is also effective in preventing known JIT-ROP attacks. Therefore, it is an effective compatible protection mechanism. This is another instance where compatibility takes precedence over the likelihood of deployment.

With generalized re-randomization schemes, it is hard to access and compare the "protection" aspect of the evaluation criteria using just test cases. It would be ideal if we could quantify re-randomization security. This year, Ahmed et al. [16] published a paper that solves this issue by providing measurement and guidelines to standardize re-randomization defense evaluation. Therefore, future defense mechanisms can refer to the paper in order to have a better understanding of the defense they are using to protect programs from JIT-ROP.

6. Conclusion

Numerous papers have demonstrated various approaches to tackle JIT-ROP attacks, which is a popular type of code reuse attack that exploits memory disclosure bugs of programs. Some of these approaches include protecting memory permissions, destructive code read pointer indirection and re-randomization. Each type of approach has got its own perks and drawbacks in terms of its ability to protect programs with a certain level of compatibility and cost. The popular approach seems to be re-randomization schemes that allow defenses to be general, highly compatible, and less costly. Therefore, we would need stronger defenses for this approach, which also implies that we need to use better-standardized test suites and metric schemes. Keeping that in mind, we hope that this systematization of knowledge will help researchers to better evaluate their JIT-ROP defenses and make progress in this sector.

Acknowledgments

We thank Prof. Vasilis for providing interesting reads throughout the semester that have been used as part of this paper's reference.

References

[1] CVEDETAILS. Vulnerability distribution of CVE security vulnerabilities by types. https://www.cvedetails.com/vulnerabilities-by-types.php, 2020.

[2] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A.-R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In Proc. of IEEE SOSP (2013).

[3] BACKES, M., AND NÜRNBERGER, S. Oxymoron: Making fine grained memory randomization practical by allowing code sharing. In Proc. of USENIX Security (2014), pp. 433–447.

[4] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In Proc. of USENIX OSDI (2014), pp. 147–163.

[5] CRANE, S., LIEBCHEN, C., HOMESCU, A., DAVI, L., LARSEN, P., SADEGHI, A.-R., BRUNTHALER, S., AND FRANZ, M. Readactor: Practical code randomization resilient to memory disclosure. In Proc. of IEEE S&P (2015), pp. 763–780.

[6] TANG, A., SETHUMADHAVAN, S., AND STOLFO, S. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In Proc. of ACM SIGSAC (2015), pp. 256–267

[7] GIONTA, J., ENCK, W., AND NING, P. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In Proc. of ACM CODASPY (2015), pp. 325–336.

[8] DAVI, L., LIEBCHEN, C., SADEGHI, A.-R., SNOW, K. Z., AND MONROSE, F. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In Proc. of NDSS (2015)

[9] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nurnberger, and J. Pewny. You can run but you ¨ can't read: Preventing disclosure exploits in executable code. In Proceedings of the 2014 ACM

SIGSAC Conference on Computer and Communications Security, CCS '14, pages 1342–1353, New York, NY, USA, 2014. ACM.

[10] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and deployable continuous code re-randomization. In *Proc. USENIX OSDI*. 367–382.

[11] Braden, Kjell, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz and Per Larsen. "Leakage-Resilient Layout Randomization for Mobile Devices." *NDSS* (2016).

[12] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z. Snow, Fabian Monrose, and Michalis Polychronakis. 2016. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16). Association for Computing Machinery, New York, NY, USA, 35–46.

[13] Rudd, Robert, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, Ahmad-Reza Sadeghi and Hamed Okhravi. "Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity." *NDSS* (2017).

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

[15] CHEN, Y., WANG, Z., WHALLEY, D., AND LU, L. Remix: Ondemand live randomization. In Proc. of ACM CODASPY (2016), pp. 50–61.

[16] Ahmed, Md Salman, Ya Xiao, Gang Tan, Kevin Z. Snow, Fabian Monrose and Danfeng Yao. "Methodologies for Quantifying (Re-)randomization Security and Timing under JIT-ROP." *arXiv: Cryptography and Security* (2020): n. pag.

[17] C. D. Spradling, "SPEC CPU2006 benchmark tools," SIGARCH Comp. Arch. News'07.

[18] Szekeres, Laszlo, Mathias Payer, Tao Wei and Dawn Xiaodong Song. "SoK: Eternal War in Memory." *2013 IEEE Symposium on Security and Privacy* (2013): 48-62.

[19] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis. Return to the zombie gadgets: Undermining destructive code reads via code inference attacks. In 37th IEEE Symposium on Security and Privacy, 2016.