A Study on Defense Technique Against Use-After-Free Attacks Using MTE

Yunseong Hwang*, Junseung You* Yunheung Paek*
*Dept. of Electrical and Computer Engineering and Inter-University Semiconductor Research Center(ISRC), Seoul National University
yshwang@sor.snu.ac.kr, jsyou@sor.snu.ac.kr, ypaek@snu.ac.kr

MTE 를 활용한 사용 후 해제 공격 방어기법 연구

황윤성*, 유준승*, 백윤흥*
*서울대학교 전기정보공학부, 반도체공동연구소

Abstraction

The Use-after-free (UAF) bug is a long-standing temporal memory safety issue. To prevent UAF attacks, two commonly used approaches are lock-and-key and pointer nullification. Recently, ARM architecture supports the Memory Tagging Extension (MTE) that implemented a lock-and-key mechanism using a 4-bit tag during memory access. Previous research proposed a virtual address tagging scheme utilizing MTE to prevent UAF attacks. In this paper, we aimed to measure a simplified version of the previously proposed virtual address tagging scheme on real machines supporting MTE by implementing a simple module and conducting experiments.

1. Introduction

Use-after-free (UAF) bug is a temporal memory safety violation. If a pointer whose referent chunk have been freed still exists, such pointer is referred to dangling pointer. Dereferencing dangling pointers is used to perform erroneous memory accesses and causes UAF bugs. Many UAF bugs [1, 2, 3] exists though the mitigation endeavors [4, 5, 6, 7, 8].

Lock-and-key and pointer nullification are two conventional approaches to prevent UAF attacks. For the former [11, 12], the lock is set on the object, while the key is assigned to the pointer. For each object access, a lock-and-key match process occurs, where the key provided by the pointer is compared with the lock on the referent object. The latter [6, 7] is employed to eliminate the possibility of dereferencing dangling pointers by removing their links to freed chunks.

Recent ARM architectures, starting from ARMv8.5 [9], introduced the Memory Tagging Extension (MTE) as a hardware feature. MTE involves setting a 4-bit tag on both memory and pointers, implementing a lock and key access mechanism. During memory access, a tag match procedure is performed, where the tag of the pointer is compared with the tag of the memory. If the tags match, access is granted; otherwise, it is considered as malicious memory access and an exception is raised.

In a previous study by [10], a proposed virtual address tagging scheme leveraged MTE for preventing UAF attacks and an emulation is conducted.

This paper implements a simple module on real MTE supporting machine to measure the simplified version of the aforementioned virtual address tagging scheme. Through the addition of MTE-related operations to the malloc and free functions, a method for UAF prevention utilizing MTE was implemented.

Experimental result demonstrated successful prevention of a simple UAF attack by dereferencing dangling pointers. Furthermore, we evaluated the performance overhead induced by the additional MTE-related operations on gcc and mcf benchmarks from SPEC2006. For gcc, there was a -0.5% overhead, while for mcf, there was a 3.9% overhead.

2. Backgrounds

2.1 Use-After-Free (UAF) bug

Use-After-Free bugs are vulnerabilities that exploit arbitrary memory access and control flow hijacking via dereferencing dangling pointers. Dangling pointers refer to the pointers still pointing to a freed heap chunk. If the freed chunk holds sensitive data or a critical function pointer (e.g., a function pointer that invokes a sensitive system call), an attacker can perform malicious memory access and intercept control flow to execute the attacker's arbitrary code.

2.2 Existing UAF prevention approach

2.2.1 Lock-and-Key

In the lock and key scheme, a lock is assigned to every memory allocation, and a key is assigned to a corresponding valid pointer. These lock and key pairs form the basis for verifying any potentially malicious attempts during pointer dereferencing for memory access. If the key provided by the pointer does not match the lock assigned to the memory, it is deemed invalid, leading to the generation of an exception. This mechanism ensures that only valid pointers with matching keys can access the corresponding memory regions, preventing unauthorized or malicious memory accesses.

2.2.2 Pointer Nullification

Pointer nullification is an intuitive method for mitigating the risks associated with dangling pointers. Use-After-Free (UAF) attacks occur when dangling pointers are dereferenced. However, by nullifying dangling pointers, any subsequent attempt to dereference them would result in a segmentation fault due to null pointer dereferencing, thereby preventing the exploitation of dangling pointers.

2.3 Memory Tagging Extension (MTE)

ARM introduced the Memory Tagging Extension (MTE) as part of the Armv8.5 architecture. MTE aims to improve the security by detecting and mitigating memory-related vulnerabilities.

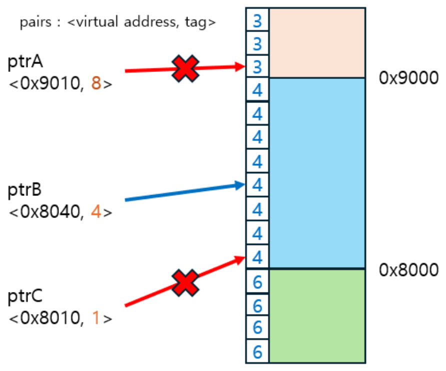

Figure 1. memory access mechanism of ARM memory tagging extension

As shown in Figure 1, MTE supports two types of tags: pointer tags and memory tags. Pointer tags are assigned to 56~59 bits of each pointer's virtual address. Memory tags are assigned to each 16B (or 32B) memory block and stored separately. MTE uses an underlying lock and key mechanism to access memory. If memory tag(lock) and pointer tag(key) do not match, a memory access violation occurs and an error is raised. MTE adds special instructions (e.g., irg, stg, ldg, addg, etc.) to explicitly perform tag-related operations. Furthermore, the pointer tag is implicitly propagated through pointer arithmetic to other pointers which referencing the same object.

MTE provides two operation modes: synchronous (SYNC) mode and asynchronous (ASYNC) mode. In SYNC mode, a mismatch between the tag in the pointer and the tag in memory causes a synchronous exception. SYNC mode prioritizes the accuracy of error detection and endures performance overhead. In ASYNC mode, the processor continues execution despite a tag mismatch. Opposed to SYNC mode, ASYNC mode is optimized for performance over the accuracy of error detection. A recent linux kernel [13] supports MTE by generating SIGSEGV. This procedure uses a code, SEGV_MTESERR (i.e., synchronous error) at the SYNC mode or SEGV_MTEAERR (i.e., asynchronous error) at the ASYNC mode.

3. Threat Model

We assume that a program running with our design has UAF vulnerabilities. The attacker can exploit those by leveraging dangling pointers. Other memory attacks, such as buffer overflow and type confusion, are out of scope. So, our module is guaranteed not to be modified by attackers.

4. Design

We implemented a simple module that intercepts malloc and free from application and performs some tag-related operations.

4.1 Overview

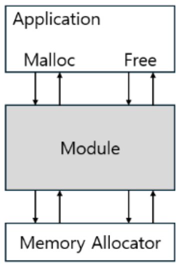

Figure 2. Overall design

Our overall design is depicted in Figure 2. Simple module is placed between the application and the memory allocator. When an application sends malloc or free request to the allocator, the module hooks the request and performs tag-related operations for each request. For malloc, the module first invokes the real memory allocator's malloc function. Then, the module assigns initial random tag to the returned pointer with irg instruction and sets the same tag number to its referent memory chunk with stg instruction. By doing so, the pointer and its referent memory chunk have the same random tag number. After setting the tag for the pointer and memory chunk, module returns the pointer to the application. For free call, similar to malloc call, the module intercepts the request. Then, the module increments the referent memory chunk by 1 and performs the real memory allocator's free function sequentially.

4.2 UAF prevention through MTE

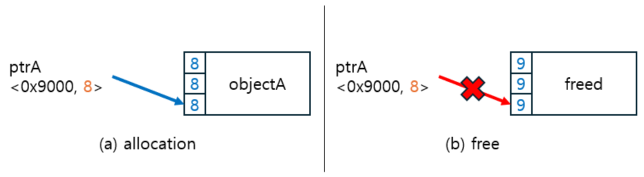

Figure 3. UAF prevention by tag matching. For memory allocation request, the random tag number is assigned. The tag number is incremented when freeing the object.

As shown in Figure 3, our simple module utilizing MTE prevents UAF attack by tag matching. As discussed in Section 4.1, the module performs some tag-related operations for malloc and free request of the application. When the application calls malloc to request an object, the random tag number is assigned to both the returned pointer and its referent memory chunk. The tag of the memory chunk is incremented by 1 for free call. After free, if a dangling pointer tries to access the freed chunk, a tag mismatch occurs since the freed memory chunk's tag incremented while the dangling pointer's tag is unchanged. As a result, an exception is raised.

5. Experiment

The experiment was conducted on a Pixel 8 Android device that supports MTE. Our module was implemented on the device. Simple UAF attack that dereferences a dangling pointer after freeing was examined and performance measurements were conducted on the gcc and mcf benchmarks from SPEC2006.

An exception was raised when we conducted the simple UAF attack. We observed that our module could prevent the possibility of dereferencing the dangling pointer after freeing by incrementing the tag number of the freed memory chunk.

For gcc, there was approximately a -0.5% overhead, while for mcf, there was around a 3.9% overhead. Considering that the operation of the module is the insertion of one irg and stg instruction for malloc and one stg instruction for free call, the performance overhead attributed to this was not significant in the overall application performance and other factors could have influenced the results.

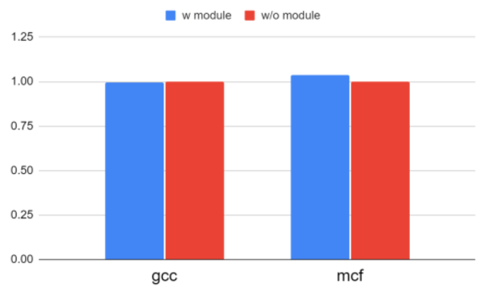

Figure 4. Performance measurement on gcc and mcf benchmarks

6. Limitation

Tagging a pointer with a random tag can introduce the possibility of a UAF attack. If an attacker's randomly assigned pointer tag coincidentally matches the memory tag of a freed chunk containing secret data, the attacker could manipulate the pointing address of the pointer to point to the freed chunk containing the secret data. Consequently, the attacker could pass tag matching procedure and access the secret data.

7. Conclusion

We implemented a simplified version of the virtual address tagging scheme proposed in previous research on a real machine supporting MTE. We examined its effectiveness in preventing simple UAF attack and evaluated its performance on several benchmarks of SPEC2006. However, simply setting a random tag had some security pitfalls, and it was deemed necessary to have a more sophisticated tag setting and management system. Through this, research on a module capable of preventing complex UAF attacks should continue, as it remains crucial.

8. Acknowledgement

This work was supported by the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2024. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) under the artificial intelligence semiconductor support program to nurture the best talents (IITP-2023-RS-2023-00256081) grant funded by the Korea government(MSIT). This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2023-00277326). This work was supported by Inter-University Semiconductor Research Center (ISRC).

References

[1] CVE-2024-31083 https://nvd.nist.gov/vuln/detail/CVE-2024-31083

[2] CVE-2024-3299 https://nvd.nist.gov/vuln/detail/CVE-2024-3299

[3] CVE-2024-26801 https://nvd.nist.gov/vuln/detail/CVE-2024-26801

[4] Van Der Kouwe, E., Nigade, V., & Giuffrida, C. (2017, April). Dangsan: Scalable use-after-free detection. In Proceedings of the Twelfth European Conference on Computer Systems (pp. 405-419).

[5] Caballero, J., Grieco, G., Marron, M., & Nappa, A. (2012, July). Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In Proceedings of the 2012 International Symposium on Software Testing and Analysis (pp. 133-143).

[6] Lee, B., Song, C., Jang, Y., Wang, T., Kim, T., Lu, L., & Lee, W. (2015, February). Preventing Use-after-free with Dangling Pointers Nullification. In NDSS.

[7 9] Ainsworth, S., & Jones, T. M. (2020, May). MarkUs: Drop-in use-after-free prevention for low-level languages. In 2020 IEEE Symposium on Security and Privacy (SP) (pp. 578-591). IEEE.

[8] Erdős, M., Ainsworth, S., & Jones, T. M. (2022, February). MineSweeper: a "clean sweep" for drop-in use-after-free prevention. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 212-225).

[9] ARM Limited, "Armv8.5-A memory tagging extension," White Paper, 2021.

[10 15] Bang, I., Kayondo, M., You, J., Kwon, D., Cho, Y., & Paek, Y. (2023). Enhancing a Lock-and-key Scheme with MTE to Mitigate Use-After-Frees. IEEE Access.

[11] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. dancewic, "CETS: Compiler enforced temporal safety for C," in Proc. Int. Symp. Memory Manage., Jun. 2010, pp. 31–40.

[12] T. H. Y. Dang, P. Maniatis, and D. Wagner, "Oscar: A practical pagepermissions-based scheme for thwarting dangling pointers," in Proc. 26th USENIX Secur. Symp., 2017, pp. 815–832.

[13] Memory Tagging Extension User-Space Support, 2020. [Online]. Available: https://lore.kernel.org/linux-arm-kernel/20200703153718.16973-1-catalin.marinas@arm.com