Defending Use-After-Free via Relationship Between Memory and Pointer

Guangquan Xu[1(✉)], Miao Li[1], Xiaotong Li[1], Kai Chen[2], Ran Wang[3], Wei Wang[4], Kaitai Liang[5], Qiang Tang[6], and Shaoying Liu[7]

[1] College of Intelligence and Computing, Tianjin University, Tianjin, China
{losin,lixiaotong}@tju.edu.cn, limiao_tju@163.com
[2] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
chenkai@iie.ac.cn
[3] Security Center, JD.com, Beijing, China
wangran8088@gmail.com
[4] Beijing Jiaotong University, Beijing, China
wangwei1@bjtu.edu.cn
[5] Surrey Centre of Cyber Security, University of Surrey, Guildford, UK
k.liang@surrey.ac.uk
[6] New Jersey Institute of Technology, Newark, USA
qiang@njit.edu
[7] Hosei Univesity, Tokyo, Japan
sliu@hosei.ac.jp

Abstract. Existing approaches to defending Use-After-Free (UAF) exploits are usually done using static or dynamic analysis. However, both static and dynamic analysis suffer from intrinsic deficiencies. The existing static analysis is limited in handling loops, optimization of memory representation. The existing dynamic analysis, which is characterized by lacking the maintenance of pointer information, may lead to flaws that the relationships between pointers and memory cannot be precisely identified.

In this work, we propose a new method called UAF-GUARD without the above barriers, in the aim to defending against UAF exploits using fine-grained memory permission management. In particular, we design a key data structure to support the fine-grained memory permission management, which can maintain more information to capture the relationship between pointers and memory. Moreover, we design code instrumentation to enable UAF-GUARD to precisely locate the position of UAF vulnerabilities to further terminate malicious programs when anomalies are detected.

We implement UAF-GUARD on a 64-bit Linux system. We carry out experiments to compare UAF-GUARD with the main existing approaches. The experimental results demonstrate that UAF-GUARD is able to effectively and efficiently defend against three types of UAF exploits with acceptable space overhead and time overhead.

H. Gao et al. (Eds.): CollaborateCom 2020, LNICST 349, pp. 583–597, 2021.
https://doi.org/10.1007/978-3-030-67537-0_35

Keywords: Use-after-free vulnerability · Fine-grained memory permission management · Static instrumentation

1 Introduction

As a memory corruption flaw, Use-After-Free (UAF) vulnerability is defined by Common Weakness Enumeration (CWE) as an attack that "referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code" [1]. A single UAF vulnerability cannot be exploited in isolation, and it must be exploited along with other heap memory exploitation techniques (e.g., Heap Spray [2]). As a result, UAF vulnerability exploitation can often be discovered from most heap vulnerabilities. The exploit of UAF vulnerabilities may result in horrifying system corruption, such as arbitrary read, write-back, and malicious code execution. Arbitrary read may lead to information leakage, including the system information, user sensitive data, and process memory layout. Once attackers get the process information, they can bypass system security guards, e.g., Canary [3], PIE [4], ASLR [5], and further combine with other vulnerabilities to launch attacks to computer systems. Arbitrary write-back and code execution may make control flow be easily hijacked in such a way that attackers may execute preset instructions, potentially get shell and obtain all the permissions of a system.

There have been some proposed techniques in the literature to address the UAF vulnerability exploit problem.

Static Detection and Defense. The static analysis is defined to work as follows: a) converting binary programs into an uniform Intermediate Representation (IR), in which the conversion methods use small instruction sets, e.g., REIL IL [6], Bincoa [7] and BAP [8]); b) analyzing the code of IR with symbolic execution [9] and abstract interpretation [10]; c) building the Control Flow Graph (CFG) to describe the relationships inter and intra functions. Specifically, using the analysis results of the famous commercial reverse software IDA Pro [11] to avoid the repeated analysis for a given function; d) extracting the UAF vulnerabilities from the analysis results.

Traditional dynamic solutions, such as [12–14], choose to nullify the corresponding dangling pointer - a pointer which points to a freed memory - after freeing an object. However, it may easily imperil the original program logic and be extremely difficult to detect the vulnerability where the related pointer is fulfilled by calculating relative offsets.

Memory error detector, such as [15–17], can also be used to capture UAF vulnerabilities at runtime. By maintaining the allocated memory status, it may identify if a given memory is used after being freed. However, if attackers can control the heap memory allocation process, for instance, they make use of Heap Spray [2] to force the program to reallocate memory on a freed memory, this approach may be prone to be bypassed.

```c
#include <stdlib.h>
#include <string.h>
    using namespace std;

    int main(){
        char *ptr = (char *) malloc(40);
        free(ptr);
        strcpy(ptr, "Use-After-Free");
        return 0;
    }
```

Fig. 1. An example of Type I UAF vulnerability.

```c
#include <stdlib.h>
#include <string.h>
    using namespace std;

    int main(){
        char *ptr = (char *) malloc(40);
        char *ptr2 = ptr;
        _int64 val = (_int64) ptr;
        free(ptr);
        ptr = 0;
        strcpy(ptr2, "Use-After-Free");
        strcpy((char *)(val + 0x8), "Use-After-Free");
        return 0;
    }
```

Fig. 2. An example of Type II and III UAF vulnerabilities.

We make the following contributions:

- We propose an effective Use-After-Free (UAF) vulnerabilities detection and defense approach called UAF-GUARD. It can prevent pointer abuse and further defend the exploits of all types of UAF vulnerabilities (we define in this work) through checking the permission of the pointer-to-memory at runtime.
- We design a key data structures, which can maintain more information between pointer and memory, to concisely represent the relationship of them.
- We implement UAF-GUARD and evaluate its performance. The evaluation result shows that our UAF-GUARD approach can efficiently detect all the defined vulnerabilities with 5.7% space on average and 22% time overhead, respectively.

2 Background

In this section, we introduce the principle and exploitation of UAF vulnerabilities, and we define three types of UAF vulnerabilities to describe the characteristics of UAF and the difficulty of defending.

2.1 Use-After-Free Exploits

When an operating system allocates dynamically memory to a process through the `brk` or `mmap` system call (in Linux), the memory allocated to the process should be managed by the memory management mechanism in order to get rid of memory fragmentation. But we have to state that after being freed by these mechanisms, the allocated memory will yield UAF exploits if there is a way to access this memory.

2.2 Three Types of UAF Vulnerabilities

To understand the UAF vulnerability more clearly, we categorize the UAF vulnerabilities into three types as shown in Table 1 according to the source of exploited pointers and the difficulty of exploits.

Table 1. Three types of UAF vulnerability.

UAF vulnerability Type	Attack Means
Type I	original allocated pointer
Type II	fulfilled by pointer propagation
Type III	fulfilled by calculating relative offset

Type I allows attacker to directly leverage the pointer that is allocated with the corresponding memory space, to run read/write/execute operations. This type of vulnerability is due to the lack of carrying out the nullifying of the pointer in time, which results in dangling pointer. We state that Type I is the most common UAF vulnerability that may be easily exploited, and we show an example of the vulnerability of this type in Fig. 1.

In Type II, a pointer is obtained by pointer propagation (see `ptr2` in line 7 of Fig. 2). It has to rely on an existing pointer which points to a freed memory. For example, the pointer variable `ptr` is returned when memory space is allocated, but the value of the variable `ptr2` is made identical to `ptr` by assignment. The UAF vulnerabilities caused by `ptr2` should belong to Type II (please refer to an example in Fig. 2, at line 11).

Type III enables the pointer to be obtained by directly calculating offset, rather than by using the existing pointer which has already pointed to a freed memory. Type III vulnerability is typically exploited by combining multiple processing logic (please refer to Fig. 2, at line 12). Its exploitation may require the combination of other vulnerabilities, such as integer overflow and stack segment overflow.

3 UAF-GUARD Design

In this section, we present an overview of UAF-GUARD. To improve the performance of defending against the UAF vulnerabilities, we propose two key data structures before proceeding to the design of UAF-GUARD. We then introduce our technical details w.r.t. static instrumentation and runtime defending. Finally, we propose a target instruction filtering algorithm to optimize the detection overhead.

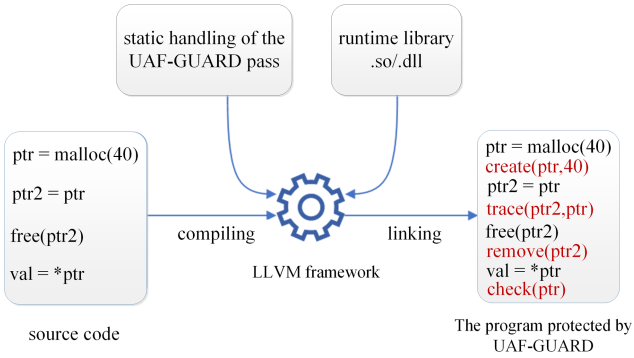

Fig. 3. The overall framework of UAF-GUARD.

3.1 Overview

To detect the UAF vulnerabilities, UAF-GUARD checks the permission of pointer to memory for each pointer operation at runtime and strictly limits the usage of pointer in user space.

As described in Sect. 2.1, a successful exploitation over a UAF vulnerability requires the following needs: a) the memory management mechanism frees the allocated memory, b) attackers read/write the freed memory or execute code, and c) proceed to subsequent exploitations, including reallocating the freed memory, information leakage and control flow hijack.

An overview of our design is illustrated in Fig. 3. The overview is based on LLVM [18] compiler framework/system. The "UAF-GUARD pass" in Fig. 3 is realized by the LLVM PASS (which is a module of code optimization in LLVM). UAF-GUARD participates in the compilation and linking process of the source code so that it can a) insert the instrumentation code in the compiled target program, b) link the compiled function library with the binary program during the linking process, and c) implement the instrumentation function to produce the protected binary program.

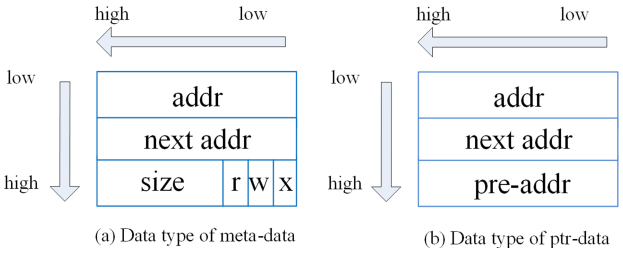

Fig. 4. Data types of meta-data and ptr-data.

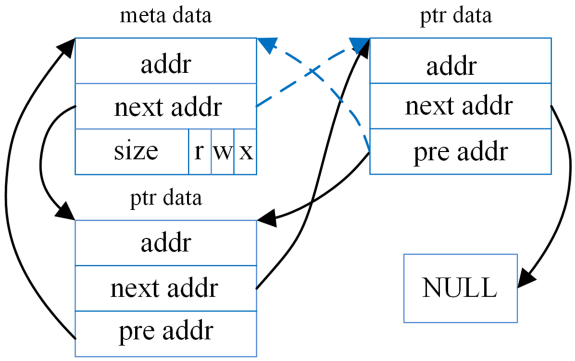

Fig. 5. Data structure of the relationship between memory and pointer.

3.2 Key Data Structures

To verify the permission of pointer-to-memory in a convenient way, we first design a key data structures: data structure for memory, pointer and their relationship.

Data Structure for Memory, Pointer and Their Relationship. We here design "special" data structures to record the information of memory, pointer and their relationship, which will be used for subsequent maintenance of the permission of pointer-to-memory.

Figure 4(a) shows the data type of the meta-data which describes the memory block. The meta-data includes three fields: `addr`, `next-addr`, and `size|rwx`. `addr` indicates the starting position of the data in memory chunk. `next-addr` represents the next data structure address in the linked list associated with the memory, which contains two types of address: a) the data structure address describing the pointer information (which is the permission of the pointer to the memory), and b) 0xFFFFFFFF (taking 32-bit system for example) indicating that there is no pointer pointing to the memory. The last field `size|rwx` consists of two parts, in which `size` is the size of the memory. Since the memory block allocated by `ptmalloc2` is 8 byte aligned in 32-bit operation system - i.e. the size

of this field is a multiple of 8, and thus we can utilize the last three bits of this field (`rwx`) to store the permissions (i.e., read/write/execute) of memory. Such permission information will be checked in our later instrumentation realization (see Sect. 3.3).

Figure 4(b) illustrates the data type ptr-data which describes the pointer information. ptr-data has three fields: `addr`, `next-addr`, and `pre-addr`, in which `addr` indicates the address of pointer variable, `next-addr` shares the same definition as in meta-data, and `pre-addr` indicates the previous data structure address in the linked list associated with the pointer. The last field contains two types of address: a) the data structure type address describing the pointer information which has permission on the same memory, and b) the address of the memory meta-data which referred by the pointer, indicating that the pointer has the permission to the memory.

Figure 5 presents the data structure of the relationship between memory and pointers. Memory and pointer metadata are stored in a red-black tree structure. In addition, UAF-GUARD needs a doubly linked list for describing the permission of pointers to memory. This is done by utilizing the fields including pointer information in meta-data data type and ptr-data data type. The dotted line indicates the indirect pointing relationship. The operation in the doubly linked list is relatively simple through the pointer operations, including inserting and deleting elements.

The data types proposed in this section can be used to describe the information of memory and pointer at bit-level granularity, which means that our memory permission tracking provides a bit-level granularity. Traditional process memory permissions are coarse-grained, usually divided by "segment" or further by "section", which owns the consistent access permissions. Our UAF-GUARD divides memory permissions more properly. The minimum unit of memory permission is 64 bit. Bit-level granularity makes more precise and flexible control of memory permissions, and hence it performs better in preventing pointer abuse. In addition, the doubly linked list is sufficient to describe the many-to-one relationship between pointers and memory.

In real-world applications, the number of the pointers linked to the same memory is relatively small (since `m` is small), therefore the complexity of the traversal operation should be acceptable.

3.3 Technical Details

In this section, we introduce the technical details of our UAF-GUARD, including the mechanism of static instrumentation and the design of the runtime library for achieving the goal of runtime defenses.

Static Instrumentation. Static instrumentation for our UAF-GUARD is implemented in the LLVM compiler framework/system, therefore the source code can be compiled into an intermediate representation based on LLVM IR. The UAF-GUARD aims to achieve the following goals based on LLVM: a) generate the corresponding meta-data for each memory allocation, and b) for each

pointer operations, propagate the associated memory information or updating the memory permissions of the pointer, so as to protect all the heap memory blocks.

In the compilation phase, we traverse each instruction in each function, parse all related instructions with the pointer operations using LLVM IR, and save the corresponding parameters and opcodes into the preset variables. We classify the parsed instructions obtained from previous step according to the LLVM IR instruction set, and insert the corresponding instrumentation functions according to each instruction.

The handling details for each type of instruction in LLVM IR are described as follows.

a) Memory allocation, reading and writing instructions.

- Memory allocation/free instructions. Since the `call` instruction sends request to the heap for memory space, it should be invoked. Taking C++ in Ubuntu Linux x86 system as an example, we use `call@malloc(k)` as the memory request function that returns is `(i8*)ptr`, where `k` is a positive integer. In the UAF-GUARD, we insert an instrumentation function `create(ptr, k)` to record the allocated memory block after this instruction. If the memory block is successfully allocated, the record is inserted into the data structure. Freeing the memory, we need to use `call@free(ptr)`, therefore, we should insert an instrumentation function `remove(ptr)` to delete the record of the memory block before the instruction. If the record is successfully deleted, the instruction will be executed.
- Memory reading instruction. Since the load instruction leverages the pointer to read process memory, which is equivalent to using the read permission of the pointer, we insert an instrumentation function `check(ptr)` before the load instruction. The load instruction usually uses `%val = load i32, i32* %ptr`. If `check(ptr)` doesn't throw an exception, the instruction will be executed (but not the other way around). We insert an instrumentation function `trace(ptr, val)` before the instruction if `val` is a pointer type. If `ptr` is a valid heap memory address after parsing, `val` will get the permission of the memory, which is called "permission propagation". Otherwise, if `ptr` is an invalid address, the instruction cannot be executed.
- Memory writing instruction. The write instruction usually uses `store i32 3, i32* %ptr`, before which we insert an instrumentation function `check(ptr)`. Since the instruction uses the pointer to write into the process memory, which is equivalent to making use of the write permission of the pointer. If `check(ptr)` doesn't throw an exception, the instruction will be executed, and otherwise, it will not be executed.

b) Arithmetic operation and bit operation instructions, including `add/fadd`, `sub/fsub`, `mul/fmul`, `udiv/sdiv/fdiv`, `urem/srem/frem`, `shl`, `lshr`, `ashr`, `and`, `or`, `xor`. UAF-GUARD focuses on the instructions associated with the heap memory pointer. When the arithmetic result includes pointer `ptr`, we insert an instrumentation function `trace(val, ptr)` before the instruction (if the instruction directly assigns the result to the pointer `ptr`). Namely, the right value `val`

calculated by this instruction is assigned to the pointer variable `ptr`, so as to change the address pointed by the pointer. If the value is a valid address, the `ptr` will get the permission of the memory and be recorded into the UAF-GUARD data structure.

Runtime Defending. According to the preset instrumentation functions in the compilation process of the static instructions, we design a runtime library for UAF-GUARD to fulfil the task of runtime defending. There are four main instrumentation functions, known as `create()`, `trace()`, `check()`, and `remove()`.

`create()`. The instrumentation function `create(ptr, k)` is to build the permission relationship between pointer and memory. The `ptr` is a pointer storing the address of the memory that is allocated by the heap memory management mechanism, and `k` is the size of the heap memory space that requested by users. The design of `create()` is described as follows.

a) If the value of `ptr` is null, which means that `malloc()` has not successfully allocated the memory, then we stop building the permission relationship.
b) Otherwise, we build the meta-data of memory block and ptr-data of pointers according to the value of `ptr` and `k`, and establish the relationship of meta-data and ptr-data in doubly linked list. Moreover, we insert the meta-data and ptr-data into the enhanced red-black tree to accelerate the querying process. We use the memory address and pointer address as the keys of meta-data and ptr-data, respectively.

`check()`. The instrumentation function `check(ptr)` is to check the permission of pointer to memory, where the `ptr` is the pointer to be checked. The process of `check()` is described as follows.

a) Get the address and value of the pointer from `ptr`.
b) Search the information of the pointer in the structure. If the ptr-data of this pointer cannot be found, it indicates that the pointer is abused. Otherwise, if we get the information of the ptr-data, we can further trace the meta-data of the memory associated by the pointer in the doubly linked list, and calculate the scope of the memory block to determine if 1) the value of `ptr` is within the scope so as to prevent the pointer being obtained by assigning or calculating offsets; and 2) the pointer has the corresponding permissions in memory read, write, and execute. If the `ptr` can satisfy the above two conditions, the `ptr` will pass the detection.

`trace()`. The instrumentation function `trace(ptr1, ptr2)` is to transfer the memory permission between pointers, where the `ptr1` is a heap memory address or a pointer that points to a heap memory block, the `ptr2` is the pointer to be assigned, and `trace()` is an overloaded function to handle different parameters. The process of `trace()` works as follows.

a) If `ptr1` is a pointer, we determine if `ptr1` is legal by checking its address and value (that is, if the memory which `ptr1` points to is an allocated available memory).
b) If `ptr1` is a heap memory address, we determine if the address is a valid and allocated heap memory address.
c) If `ptr1` is illegal, the status information will be recorded and the program will throw an exception.
d) If `ptr1` is legal, we check if the information of `ptr2` has been built. If not, we build the ptr-data of `ptr2`, which will be inserted into the tree structure. Lastly, the permission transferring is accomplished after the `ptr2` is inserted into the doubly linked list of `ptr1`.

`remove()`. The instrumentation function `remove(ptr)` is to remove the permission of pointer to memory. The process of `remove()` works as follows.

a) If `ptr` is a heap memory address, we check if the address is valid. If not, it indicates that the memory to be freed is the pointer's unprivileged memory, unallocated memory or freed memory. Otherwise, we maintain the data structure to find the pointer information ptr-data of all associated pointers and meta-data of the corresponding memory block. After that, these elements from the tree structure are removed.
b) If `ptr` is pointer, removing the permission indicates the end of the pointer life cycle. We further decide if the `ptr` exists in the tree structure. If yes, delete them from the doubly linked list and the tree structure. Otherwise, it indicates that the pointer is not pointing to a memory block.

4 Experimental

To verify the effectiveness and efficiency of our design, we present a prototype implemented for our UAF-GUARD and compare it with the related ones. Our experiments include the security and performance tests. Note that the source code of our UAF-GUARD is publicly accessible: https://github.com/UAF-GUARD/sourcecode.

4.1 Experimental Environment

First, we evaluate the defending effectiveness of the UAF-GUARD over all types of UAF vulnerabilities, and further test the well-known CVE vulnerabilities in multiple versions of programs including Chrome, Wireshark, and OpenSSL. We further calculate the runtime overhead of the UAF-GUARD in the compiling phase. To bring simplicity and fairness in the comparison, we use the same pending programs across the related works, e.g., bzip2, and gcc. Finally, we analyze the runtime overhead of the UAF-GUARD based on the compilation results, and prove that UAF-GUARD can ensure the security of program at runtime.

All the experiments in this paper are accomplished on a PC with the configuration: eight-core Intel Core i7-6700HQ CPU @ 2.60 GHz, 16 GB RAM and 500 GB SSD, with 64-bit Ubuntu 16.04 (Linux Kernel 4.16).

Table 2. Comparison between UAF-GUARD and DANGNULL.

Vulnerabilities	Incidence	Position Of Vulnerabilities	Type	Detection Result	
				UAF-GUARD	DANGNULL
CVE-2010-2939	OpenSSL 1.0.0a, 0.9.8,0.9.7	0x800000000 22ba510	II	SIGSEGV	SIGSEGV
CVE-2016-4077	Wireshark 2.0.0-2.0.3	-	II	SIGSEGV	SIGSEGV
CVE-2013-2909	Google Chrome 30.0.1599.66	0x1bfc9901ece1	II	SIGSEGV	NORMAL
CVE-2013-2909	Google Chrome 30.0.1599.66	0x7f2f57260968	II	SIGSEGV	NORMAL
CVE-2013-2918	Google Chrome 30.0.1599.66	0x490341400000	III	SIGSEGV	TRIGGER
CVE-2013-2922	Google Chrome 30.0.1599.66	0x60b000006da4	II	SIGSEGV	NORMAL
CVE-2013-6625	Google Chrome 31.0.1650.48	0x897ccce6951	II	SIGSEGV	SIGSEGV
CVE-2012-5137	Google Chrome 23.0.1271.95	0x612000046c18	II	SIGSEGV	ASSERTION

4.2 Effectiveness of Detection

Our experiment uses the same vulnerabilities included in the three vulnerable Chromium versions in DANGNULL. To make the experimental results convincing, we leverage two mainstream software, known as OpenSSL and Wireshark, to exam their open vulnerabilities.

Table 2 is the comparison results between UAF-GUARD and DANGNULL. By "SIGSEGV", "TRIGGER", "NORMAL" and "ASSERTION" we mean "throw a exception", "trigger the vulnerability", "run in a normal way", and "the security assertion of Chrome", respectively. Compared to DANGNULL, UAF-GUARD is able to detect and terminate the execution of instruction in time. It is worth mentioning that DANGNULL does not throw SIGSEGV exception in some cases, such as CVE-2013-2909, and CVE-2013-2922. The reason behind is that the program can enter other execution paths through the branch structure after nullifying the pointer, and hence the program can re-execute normally. As a result, the position of the vulnerabilities cannot be reported as UAF-GUARD does. The ASSERTION result for CVE-2012-5137 is caused by the security assertions in Chrome, rather than by DANGNULL itself.

Through the above analysis based on real-world vulnerabilities, we can conclude that UAF-GUARD is highly competitive with the other runtime detection methods. Although DANGNULL is slightly more lightweight, the insufficient pointer information makes it suffer from the potential risk of bypassing.

4.3 Performance

In the performance tests we test the changes of the program at the static compiling and linking, which may include both the file size and the instrumentation functions, and we further test the extra overhead generated by UAF-GUARD at runtime.

Overhead at Compiling and Linking Process. We measure the changes of the programs by counting the difference between the file size before and after compiling and linking, and gathering statistics for the instrumentation functions. The experiment counts the related information at compiling-linking process for 16 programs, including 11 C programs and 5 C++ programs. We implement the test for UAF-GUARD based on the LLVM Compiler project, and the experimental results are shown in Table 3.

Table 3. The space overhead of UAF-GUARD in compiling and linking. Note *Before Compilation* represents the size of program before compiling, *Increase* indicates the increased size of program for detection after compiling and linking, *Percentage* shows the size of fixed dynamic library without calculation, and *Instrumentation Function* represents the number of instrumentation functions inserted in detection.

Name	Language	File Size				
		Before	Increase			
			DANGNULL	Percentage	UAF-GUARD	Percentage
gcc	C	8380KB	768KB	4.7%	618KB	2.7%
soplex	C++	4292KB	453KB	1.9%	428KB	0.9%
povray	C++	3383KB	513KB	4.2%	479KB	2.6%
h264ref	C	1225KB	420KB	4.1%	423KB	2.7%
gobmk	C	5594KB	416KB	0.8%	426KB	0.6%
chromium	C++	1858MB	10MB	0.5%	7M B	0.3%
sjeng	C	276KB	386KB	5.8%	396KB	2.2%
namd	C++	1182KB	382KB	1.1%	417KB	2.3%
hmmer	C	814KB	396KB	3.2%	420KB	3.7%
sphinx3	C	541KB	389KB	3.5%	422KB	5.9%
milc	C	351KB	386KB	4.6%	419KB	8.3%
astar	C++	195KB	378KB	4.1%	410KB	10.2%
bzip	C	172KB	378KB	4.7%	399KB	5.2%
mcf	C	53KB	376KB	11.3%	404KB	26.4%
libquantum	C	106KB	378KB	7.5%	400KB	9.4%
lbm	C	37KB	374KB	10.8%	393KB	8.1%
Average		117.7MB	1.1MB	4.55%	0.8MB	5.7%

Table 3 shows that for the small programs, the space overhead of UAF-GUARD is quite close to that of DANGNULL, but for the larger programs,

the space overhead of the program is less than that of DANGNULL, with a decrease from 0.2% to 2% (the first 6 rows in Table 3).

Since the size of the different programs differs, the amount of instrumentation function should be different accordingly. This leads to a fact that the changes of the file size are also inconsistent. UAF-GUARD requires the support from the runtime library with a size of about 390KB. But we state that this extra overhead is necessary for recording the pointer status so that we can guarantee the precise detection and defense over the UAF vulnerability exploits.

Runtime Overhead. We design and implement a chrome plug-in for UAF-GUARD, which records the results of accessing the Alexa top 100 websites. The average increased time overhead is 22%. Due to space limit, we here only list the results for the top 8 (out of the 100) websites in Table 4. Each group of the results represents the average time taken by the website from the loading to finishing rendering of the web pages in the 1000 repeated experiments. In this way, we try to precisely reflect the user's experience after deploying UAF-GUARD.

Table 4. The rendering time of accessing web pages by the chromium. Note that *Requests* and *DOM Nodes* represent the number of requests and the number of DOM Nodes from loading to finishing rendering the pages, respectively. *Rendering Time* shows the rendering time after the chromium protected by UAF-GUARD and DANGNULL, in which the time is the mean of the results from one thousand experiments.

Website	Complexity of Page		Rendering Time				
	Requests	DOM Nodes	Original Time(s)	DANG-NULL(s)	Increase(%)	UAF-GUARD(s)	Increase(%)
qq.com	92	2604	0.53	0.75	41.5	0.69	30.2
youtube.com	54	2397	3.11	4.13	32.8	3.90	25.4
baidu.com	21	142	0.21	0.25	19.0	0.25	19.0
taobao.com	80	1069	0.31	0.38	22.6	0.38	22.6
google.com	24	360	1.11	1.35	21.6	1.36	22.5
amazon.com	216	1508	2.28	2.66	16.7	2.67	17.1
gmail.com	52	240	1.82	2.23	22.5	2.24	23.1
twitter.com	18	668	3.45	3.81	10.4	3.97	15.1
Average	80	1124	1.73	1.95	21.7	1.94	21.9

In Table 4, the average time overheads of UAF-GUARD and DANGNULL are increased by 21.9% and 21.7% respectively, which shows that both methods share similar performance w.r.t. the average time overhead. The rendering time of a normal website is within about 2 s, and the extra time overhead is about 0.5 s, which is almost "imperceptible" to website clients. With the increase of DOM node number, our UAF-GUARD performs better than DANGNULL (please refer to the first 2 rows in Table 4).

5 Conclusion

In this work, we have proposed a novel approach called UAF-GUARD, based on a design over fine-grained memory permission management, with the aim to identify and locate UAF vulnerabilities. We have also conducted experiments in which we implement UAF-GUARD on a 64-bit Linux system and employ it to transform a program into a practical version. The experimental results demonstrate that our UAF-GUARD outperforms other existing approaches in terms of detection accuracy and overhead. It also shows that our UAF-GUARD is able to effectively and efficiently defend all the three types of UAF exploits.

In future work, we plan to design an automated UAF vulnerability discovery system for defending exploiting heap memory vulnerabilities with more description information of pointers.

References

1. CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html. Accessed 11 Oct 2019
2. Ratanaworabhan, P., Livshits, V.B., Zorn, B.G.: NOZZLE: a defense against heap-spraying code injection attacks. In: Proceedings of the 18th Conference on USENIX Security Symposium (SSYM 2009), Berkeley, CA, USA, pp. 169–186 (2009)
3. Canary (buffer overflow). http://www.cbi.umn.edu/securitywiki/CBI_Computer Security/MechanismCanary.html. Accessed 11 Oct 2019
4. Position Independent Executables (PIE). https://access.redhat.com/blogs/766093/posts/1975793. Accessed 11 Oct 2019
5. Address space layout randomization (ASLR). https://searchsecurity.techtarget.com/definition/address-space-layout-randomization-ASLR. Accessed 11 Oct 2019
6. Dullien, T., Porst, S.: REIL: a platform-independent intermediate representation of disassembled code for static code analysis. In: Proceedings of Cansecwest, Vancouver (2009)
7. Bardin, S., Herrmann, P., Leroux, J., Ly, O., Tabary, R., Vincent, A.: The BINCOA framework for binary code analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 165–170. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_13
8. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: a binary analysis platform. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 463–469. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_37
9. Ye, J., Zhang, C., Han, X.: POSTER: UAFChecker: scalable static detection of use-after-free vulnerabilities, New York, NY, USA, pp. 1529–1531 (2014)
10. Dolan-Gavitt, B., Hulin, P., et al.: LAVA: large-scale automated vulnerability addition. In: 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, pp. 110–121 (2016)
11. Hex-Rays. https://www.hex-rays.com/. Accessed 11 Oct 2019
12. Lee, B., Song, C., Jand, Y., et al.: Preventing use-after-free with dangling pointers nullification. In: Symposium on Network and Distributed System Security (NDSS), San Diego, CA, USA, pp. 8–11 (2015)

13. Caballero, J., Grieco, G., Marron, M., Nappa, A.: 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, New York, NY, USA, pp. 133–143 (2012)
14. Kouwe, E., Nigade, V., Giuffrida, C.: DangSan: scalable use-after-free detection. In: Proceedings of the Twelfth European Conference on Computer Systems, New York, NY, USA, pp. 405–419 (2017)
15. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: a fast address sanity checker. In: Proceedings of the 2012 USENIX Conference on Annual Technical Conference, Berkeley, CA, USA, p. 28 (2012)
16. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, pp. 89–100 (2007)
17. Nagarakatte, S., Zhao, J., Martin, Milo M.K., Zdancewic, S.: CETS: compiler enforced temporal safety for C. In: Proceedings of the 2010 International Symposium on Memory Management, New York, NY, USA, pp. 31–40 (2010)
18. The LLVM Compiler Infrastructure. https://llvm.org/. Accessed 11 Oct 2019