HEAPSENTRY

DETECTING HEAP-BASED BUFFER OVERFLOWS

May 5, 2019

Group Name:	RuntimeError
Group Members:	Sangtian Wang
	Xiao Zheng
	Yifan Zhai
Professor:	Nick Nikiforakis

Contents

1 Introduction

This project, the HeapSentry, aims to detect and prevent malicious heap overflows. To achieve this goal, we add canary at the end of each heap object and check it in the kernel before system calls are executed. The HeapSentry prevents systems from both Control-data attacks and Non-control-data attacks. The implementation of HeapSentry can be divided into two components: HeapSentry-U and HeapSentry-K.

HeapSentry-U works in user space and intercepts memory allocation and deallocation functions. It adds a unique random value called "canary" in each dynamically allocated memory block, and passes information of canaries to HeapSentry-K. The kernel component, HeapSentry-K runs in kernel space and stores all the information of canaries for each protected process. It is a Loadable Kernel Module that checks the intactness of stored canaries each time the process invokes a system call or frees a memory block. Once some current canary value cannot match the originally registered one, an overflow is detected and HeapSentry-K will terminate the process for the sake of operating system security. Therefore, the HeapSentry renders us an effective way of stopping attackers exploiting vulnerabilities of heap overflows regardless of what object they try to overflow and how they execute malicious code.

2 Methodology

This section will present methods we used to implement HeapSentry in both user and kernel sides. For the user space part, we overrode memory allocation functions and free function and append random-value canaries to heap objects. As for the kernel space, we implemented a hash table with two levels to store values and locations of canaries pulled from user space. Moreover, we rewrote the system call table in the memory to hijack original system calls and used unimplemented syscall numbers to define functions that control communications between kernel and user spaces within HeapSentry. All details will be explained in the following content.

2.1 Basic Design

2.1.1 User Space

In user space, we can override library functions by setting LD_PRELOAD environment variable. It allows the loader to load our code before C runtime library (*libc.so*) so that we can intercept functions in "*stdlib.h*". To be more specific, we overwrote memory allocation functions (i.e., "malloc", "calloc" and "realloc") as well as "free" function. Every time a process tries to allocate memory space in the heap, HeapSentry-U hijacks it and returns a block with 4 bytes larger than the requested size to hold the canary. Then it generates a random integer which will be added to the end of this memory block as a canary. Once the process incurs a heap overflow, the canary value will be changed and thus can be detected. Addresses and values of canaries will be registered in the hash table in kernel space. When this process calls the free function, HeapSentry-U invokes a system call to request a check of the corresponding canary in the kernel part. If the original value stored in kernel space is different from the current one, HeapSentry-K will terminate the calling process since an overflow happens. Otherwise, HeapSentry-U will take the control back from kernel and return the block to the underlying memory allocator. A high level view of heap canaries in user space is shown in figure 1.

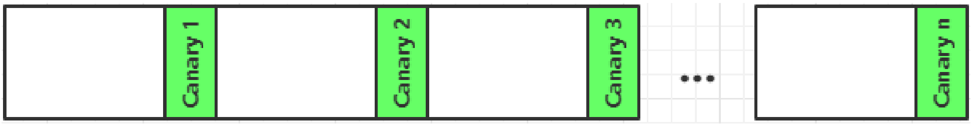

Figure 1: High-level view of the heap canaries

2.1.2 Kernel Space

In the kernel space, in order to hijack system calls we need, we redefined original system calls and redirected them to our newly-defined functions in system call table in kernel memory space. Besides, we made use of unimplemented system call numbers to build communication functions between user space and kernel. For example, when a process calls some memory allocation function in user space, a new canary will be generated in the meantime; then HeapSentry-U will invoke a system call we defined in HeapSentry-K, telling kernel to receive the canary information and store it into the internal data structure. Calling free function works in a similar way, except that the syscall here requests for a canary value check.

We manage canaries in HeapSentry-K according to their process ID. To improve search efficiency, we use a hash table with two levels for the canary information storage. The hash key of the first level is the process ID, and the value is a struct which contains all information of canaries in current process. And there is a second level hash table in the struct mentioned above. The key of second level hash table is canary address, and the value is canary's value. In other words, we create hash table for each process to store all of its canaries' information. When HeapSentry-K checks a canary value, it will first find which second-level hash table this canary locates in according to the process ID; then based on the address received from user space, it will get the original canary value stored in the hash table. Now, since HeapSentry-K knows the canary address, it can get access to the user space memory and read current canary value from it. If these two values match, the kernel then returns the control back to HeapSentry-U; otherwise, the calling process will be killed. In our basic design, "check" operations will be invoked right after system calls or free functions are executed. A high level view of hash table structure in kernel is presented in tables below.

Process ID Table	
Process ID (key)	Canary Hash Table
PID 1	canary_hlist_1
PID 2	canary_hlist_2
...	...
PID n	canary_hlist_n

(a) First-level hash table

canary_hlist_1	
Canary Address (key)	Canary Value
Location 1	Value 1
Location 2	Value 2
...	...
Location n	Value n

(b) Second-level hash table

Table 1: Hash table structure in kernel

2.2 Optimization

In our basic design, HeapSentry performance could be negatively affected due to the following reasons:

1. Each time a process calls a memory allocation function or free(), HeapSentry-U will either push a new canary or request a "canary value check" to kernel through system calls. Thus, the time cost is unacceptable when the heap is heavily used.

2. Whatever system call is invoked, the HeapSentry-K will always check the whole canary list of the calling process, even if the syscall will not benefit attackers and jeopardize the system at all.

Above cases will result in frequent system call invocations and continuous hash table look-ups so that the application efficiency can be severely decreased. Therefore, we referred to the professor's paper [1] and improved our design.

First, we created two buffers in the user space for each process: one for allocation functions, the other for free operation. If *malloc* (or *calloc*, *realloc*) occurs, the newly generated canary will be stored in the allocation buffer temporarily instead of being pushed to kernel space immediately. Only when the buffer is full (buffer size is user-configurable), HeapSentry-U performs a system call and pushes all of the canaries' information into the kernel. Similarly, when some memory block is deallocated, HeapSentry-U should first check whether the canary of this block exists in the current allocation buffer: if so, do the canary value check, then directly remove the canary from the buffer and free the memory block if no heap overflow detected; otherwise, store it in the "free buffer" and inform HeapSentry-K of receiving the new set through a system call when the buffer fills-up. An important point is that real free operation should be executed only after "canary value check" is done in kernel to avoid false-positive. In addition to passively waiting for syscalls from user space, HeapSentry-K can also actively "pull" information when it is necessary, i.e., whenever any one of *High-Risk* system calls occurs (see details in next paragraph). Obviously, kernel interruptions can be cut down sharply with the help of buffers.

Second, based on the study in the paper, not all system calls can be exploited to attack the system. The author categorized system calls by the likelihood that they are requested by attackers. For example, `fork`, `execve`, `chmod` and `open` have the highest risk. Therefore, only after an invocation of such *High-Risk* syscall, HeapSentry-K will check all canaries registered in kernel space of the calling process. Furthermore, there may be some "pending " canaries and pointers in user space buffers that have not been pushed to kernel yet when such syscalls are detected. In this situation, HeapSentry-K should pull all of the unreported information, store it into the internal hash table and check the intactness of canaries. Hence, hash table look-up times can be effectively decreased by system call categorization.

2.3 Communication between HeapSentry-U and HeapSentry-K

The communication channel between kernel and user space of HeapSentry is based on our newly defined system calls. System call numbers larger than 354 are available in the operating system we use. Table 2 shows the syscalls we defined and their functions.
The work flow of HeapSentry after optimization is demonstrated in the Figure 2 below.

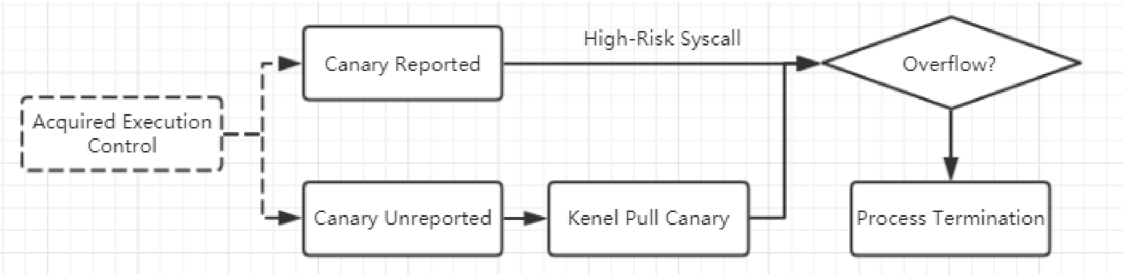

Figure 2: Overall working process of the HeapSentry

Syscall #	Name	Function
361	accept_alloc_canary_buf_addr	Receive allocation buffer information from user space when the buffer is full and store it in the hash table.
362	accept_free_canary_buf_addr	Receive free buffer information from user space when the buffer is full and check canaries' values.
369	pull_and_check_alloc_canary_buf	Pull allocation buffer information from user space when *High-Risk* syscalls are invoked, store it in the hash table and check canaries' values.
370	pull_and_check_free_canary_buf	Pull pointers' information from user space free buffer when *High-Risk* syscalls are invoked and check canaries' values.

Table 2: Newly defined system calls

3 Set Up & Test

3.1 Set Up

Our project was developed under Ubuntu 14.04.6 LTS 32-bit system, and the kernel version is Linux 4.4.0. Download link:

`http://releases.ubuntu.com/14.04/ubuntu-14.04.6-desktop-i386.iso`

After deploying the operating system, pull/download this repo. Then open a new terminal and run following commad to display kernel messages.

```
tail -f /var/log/{messages,kernel,dmesg,syslog}
```

We recommand you step into and run Demo (folder) first. Codes in Demo set more print functions, which will help you understand how user and kernel space work.

```
cd Demo
```

Next, compile kernel space code and install HeapSentry-K module with following commands. If the module is successfully installed, the system call table address will be printed.

```
cd kernel_space/
sudo make
sudo insmod kernel_space.ko
```

Next, prepare the user space code and compile the test file. Make sure you use "LD_PRELOAD" every time you run a test program. Possible commands are listed below:

```
cd ../user_space
make
make test_p
LD_PRELOAD=./user_space.so ./test_p
```

Remove the kernel module with the following command after use:

```
sudo rmmod kernel_space
```

3.2 Test Cases

Our test cases includes multiple calls of "*malloc*", "*calloc*", "*realloc*" and "*free*" functions and invocations of *High-Risk* system call functions, including "`fork`", "`execve`", "`chmod`", "`open`".In the test files, we use "alloc" functions combined with "strycp" to make a heap overflow condition. Several functions are designed in the test file to comprehensively test whether the kernel is able to kill the process when each *High-Risk* system call is executed and heap overflow occurs.

"test_p" takes the number of allocations/deallocation functions you want to test and give the run time of running them under HeapSentry. "test1" tests a simple overflow function which will be detected when the free buffer is full in the user space. "test2", "test3", "test4" and "test5" invoke high-risk system calls "`open`", "`chmod`", "`fork`" and "`execve`" separately after a heap overflow. In our test experiment, the process will be killed by HeapSentry-K, and the warning messages of heap overflow can be displayed in terminal.

```
make test_p
LD_PRELOAD=./user_space.so ./test_p
make test1
LD_PRELOAD=./user_space.so ./test1
make test2
LD_PRELOAD=./user_space.so ./test2
make test3
LD_PRELOAD=./user_space.so ./test3
make test4
LD_PRELOAD=./user_space.so ./test4
make test5
LD_PRELOAD=./user_space.so ./test5
```

4 Performance

To test the performance of HeapSentry, we call allocation and free functions iteratively, we compare the run time using built-in functions versus our modified memory allocation and free functions (table 2).

By plotting the data into figure 3, we can tell that the running time of processes with HeapSentry are significantly longer than that of processes without as the number of commands increasing. The delay is mainly caused by the communication between user space and kernel space. Pushing canary values and locations to kernel when buffers are full requires system call invocations; besides, copying and searching information through buffers are time-consuming as well. Therefore, the canary buffer size in user space is designed to be user-configurable so that it can be adjusted based on performance. Since user space buffers in our design are simple linear lists (i.e., the time complexity of search is $O(n)$), there is a trade-off between buffer size and number of system call invocations. In figure 3, based on our experiment so far, larger canary buffer size (e.g., 1000) performs much better than smaller one (e.g., 100) as number of heap objects increases.

# of malloc/free calls	runtime without HS	HS Buffer Size 100	HS Buffer Size 1000
10	0.000033	0.00022	0.000064
100	0.000034	0.000301	0.000095
1000	0.000105	0.002842	0.002816
5000	0.000432	0.017667	0.013337
10000	0.00061	0.044138	0.027095
20000	0.001375	0.132875	0.064031
30000	0.00207	0.257288	0.103453
40000	0.002962	0.534146	0.157886
50000	0.004182	0.866863	0.208458

Table 3: Performance comparison between tests with and without HeapSentry installed (seconds)

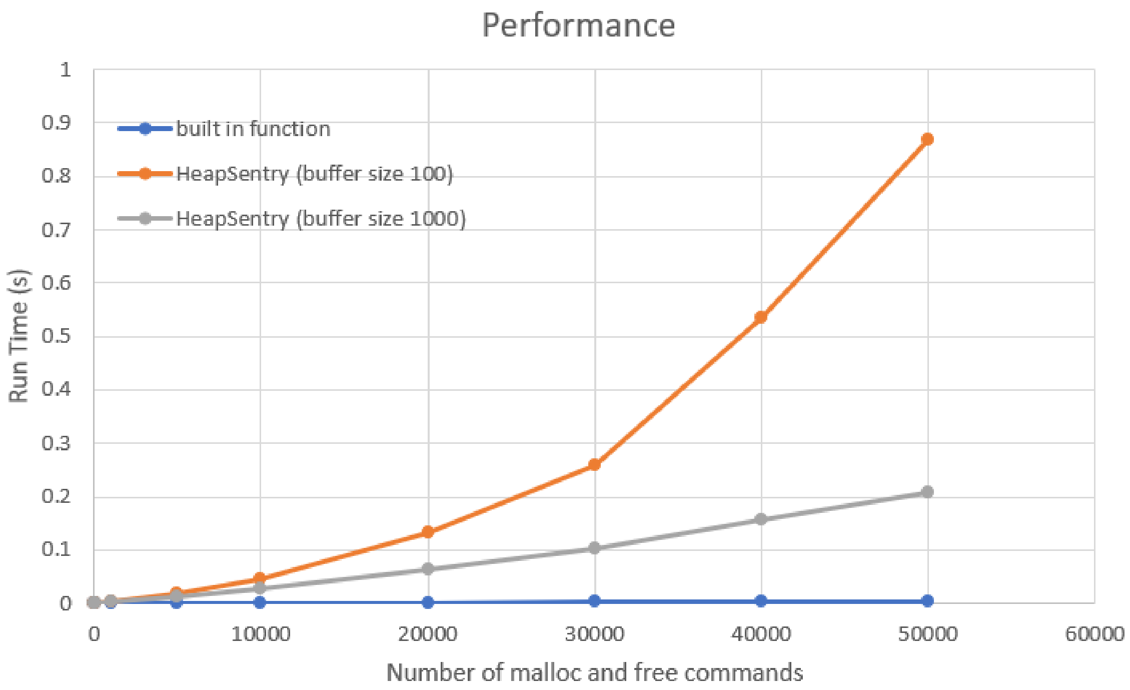

Figure 3: Performance comparison between tests with and without HeapSentry installed

5 Future Work

Even though we have implemented most of functions of HeapSentry, there are still some improvements need to be done in the future.

1. Guard Pages: Since canary buffers locates in user space memory, they are still vulnerable to attackers who try to overwrite canaries directly in buffers. Therefore, in order to prevent attackers from tampering canaries, we can set two memory pages without write permission as guard pages and place the page that holds buffers in the middle of them. Once attackers attempt to overwrite middle page, the calling process should be killed.

2. Medium-Risk Calls: Currently we have not dealt with *Medium-Risk* system calls such as "`read`", "`write`" and "`mount`". In further steps, we can add functions to check a subset of live canaries in kernel space before *Medium-Risk* system calls are executed. Checking part of canaries for such system calls is time-saving compared to checking the entire canary list. The rationale of doing this is explained in paper [1].

3. User_Space Buffer Data Structure: As we mentioned before, both the allocation buffer and free buffer are linear lists. Therefore, the time complexity of buffer search is linear. A possible optimization solution we think of is to use hash tables as buffer data structure.

6 Conclusion

In this project, we implemented HeapSentry, an application that can detect and prevent heap overflows with the cooperation of the memory allocation library and the kernel of an operating system. Our tests and experiment results prove that our project realizes all of the functions of HeapSentry in the blueprint, including memory allocation/deallocation functions interception, generation of canary for each heap object, storage of canaries in hash table in kernel space, *High-Risk* system calls detection and canary value check, etc. HeapSentry is a reasonable trade-off between operating system security and efficiency. Though the word we have done so far is not perfect, the HeapSentry we developed is still an effective countermeasure against most of the heap-based overflow attacks.

7 Team Members & Work Division

Our team name is "RuntimeError", which represents our goal is to terminate all the malicious heap-based buffer overflows. Our work distribution:
Sangtian Wang, graduating master student in the Department of Computer Science, focuses on the coding part of HeapSentry-K and communication channel design as well as repo management on Github.
Xiao Zheng, graduating master student in the Department of Computer Science, is in charge of the coding task of HeapSentry-U, control logic design and report revision and embellishment.
Yifan Zhai, first-year master student in the Department of Technology & Society, is responsible for debugging and testing both user space and kernel components, writing project report draft and preparing demonstration.

8 References

1. Nikiforakis, N., Piessens, F. & Joosen, W. (2013). HeapSentry: Kernel-assisted Protection against Heap Overflows.

2. Github repo: https://github.com/maK-/Syscall-table-hijack-LKM

3. Github repo: https://github.com/danielmaker/linux_study/tree/master/list_example

4. Github repo: https://github.com/ex0dus-0x/hijack

5. Blog: https://catonmat.net/simple-ld-preload-tutorial-part-two