HeapSentry: Kernel-assisted Protection against Heap Overflows

Nick Nikiforakis, Frank Piessens, and Wouter Joosen

iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be

Abstract

The last twenty years have witnessed the constant reaction of the security community to memory corruption attacks and the evolution of attacking techniques in order to circumvent the newly-deployed countermeasures. In this evolution, the heap of a process received little attention and thus today, the problem of heap overflows is largely unsolved.

In this paper we present *HeapSentry*, a system designed to detect and stop heap overflow attacks through the cooperation of the memory allocation library of a program and the operating system's kernel. HeapSentry places unique random canaries at the end of each heap object which are later checked by the kernel, before system calls are allowed to proceed. HeapSentry operates on binaries (no source code needed) and has, by design, no false-positives. At the same time, the active involvement of the kernel provides stronger security guarantees than the current state of the art in heap protection mechanisms for a modest performance overhead.

1 Introduction

Over two decades have passed since the release of the first well-known computer worm, the Morris worm, which used a buffer overflow vulnerability as its main spreading mechanism [35] and attracted the world's attention to buffer overflows and to the potential resulting from their exploitation. Despite the significant amount of research conducted in the area of buffer overflows and memory corruption attacks, modern software still suffers from such vulnerabilities. The last years have been a showcase for memory corruption attacks where high-profile companies like Google, Yahoo, Symantec and RSA were attacked by zero-day memory corruption exploits targeting major software products [1, 24, 30]. The National Vulnerability Database [26] reports 307 buffer overflow vulnerabilities for 2012 that allow an attacker to execute arbitrary code on a victim machine.

Even though modern operating systems ship with a set of orthogonal runtime and compile-time protection techniques which together harden processes against memory corruption attacks, the aforementioned cases and statistics show that the problem is still not fully resolved. The three most popular and complementing countermeasures present in all modern operating systems are: *(i)* Address Space Layout Randomization (ASLR) [8, 27], *(ii)* non-executable stack and

heap [34] and *(iii)* probabilistic protection of stack frames (e.g. StackGuard [13]). Compared to the stack, the heap of a process has received much less attention by the security community. Today, depending on the operating system and the underlying memory allocator, a heap overflow is either never detected or detected at the deallocation time of the overflowed object. The detection at the time of deallocation is sufficient to stop traditional attacks against the inline metadata of a heap implementation but cannot stop attacks against adjacent control data (e.g. function pointers or entries in the virtual table of a COM object [4, 28]), adjacent non-control data [11] or even legitimate executable code created by a Just-in-Time compiler and present on writable memory [17].

To address this lack of security we present *HeapSentry*, a system protecting against malicious heap overflows through the cooperation of the memory allocation library and the kernel of the operating system. HeapSentry is not a new memory allocator but a defense layer on top of existing memory allocators making it compatible with all allocators of modern operating systems as well as the ones described in literature. Our system intercepts all calls to dynamic memory allocation functions and appends to each allocated object a unique random value that serves as a "canary" for that heap object. The location and value of each canary are propagated to the kernel component of HeapSentry which holds a complete list of the heap canaries of the protected process. The kernel component of HeapSentry is a Loadable Kernel Module which checks the intactness of the registered canaries every time that the process requests a system call from the operating system. If the current value of one of the canaries is different from its original value, the process wrote past the boundaries of that specific heap object. Since such an overflow could be the result of an attacker exploiting a vulnerability in the program, the process is terminated. This approach enables HeapSentry to accurately detect and stop attacks regardless of the overflowed object (e.g. heap meta-data, function pointers and non-control-data) and regardless of the attacker's method of executing malicious code (e.g. injected shellcode in memory pages, *return2libc* and *return-oriented* programming).

While canary-based heap protection systems have been proposed in the past, HeapSentry's characteristics make it more secure and resilient against sophisticated attackers who are aware that a protection system is in-place. Contrary to previous work, the canaries placed by HeapSentry at the end of heap blocks are uniquely random (no system-wide or process-wide canaries [31]) and are not reconstructable as the canaries of previously proposed systems [41]. The location and original value of each canary are stored in the kernel space, out of the process' and the attacker's reach. Instead of performing a health check of each heap block at its deallocation time, HeapSentry checks the health of the protected process' heap right before the execution of system calls, thus effectively denying the final and necessary element of all related attacks, i.e., access to kernel resources. The canary-check itself, is enforced and performed in kernel space where it cannot be bypassed by any user space process. Lastly, our system operates on existing binaries and does not need access to the source code of applications [2,

9, 15, 19] or kernel recompilation [7] enabling its effortless adoption in desktop and server environments. The contributions of this paper are:

- Design of a novel OS-independent cooperative system between a memory allocation library and the operating system's kernel to protect against heap overflows
- Accurate detection of heap overflows without the need of an application's source code and regardless of the contents of the overflowed object and the attacker's method of executing malicious code
- Implementation of an optimized HeapSentry prototype for the Linux operating system with an average performance overhead of less than 12% over the SPEC CPU2006 Integer benchmark suite
- Security evaluation using RIPE [38], showing the benefits of a HeapSentry-protected system both independently as well as cooperatively with popular countermeasures

2 Attacker model

In this work we assume that a heap overflow vulnerability exists in a running process that will allow a local or remote attacker to overflow from one heap object to another target heap object. A heap object is the chunk of memory obtained through the call of one memory allocation function. Unlike previous work, we allow for the worst-case scenario where the attacker is free to overflow an arbitrary number of bytes and not just a small number of them [6] e.g. through a `memcpy()` operation with an attacker-controlled source and number of bytes to copy instead of an *off-by-one* vulnerability. This target heap object may contain one or more variables that are used by the program at a later time either to explicitly transfer the control-flow of the application (control-data attacks) or as part of a sensitive operation (non-control-data attacks). Whenever a heap overflow is detected, HeapSentry terminates the offending process thus, in general, Denial-of-Service attacks against vulnerable user space applications are not considered in scope for our system.

Control-data attacks In the case of control-data attacks, the target heap object may contain a value that is normally used by the program to redirect execution to a location calculated at run-time (e.g. a function pointer or an entry in a virtual function table). In free-list based memory allocators (common in Windows and Linux systems) the inline heap metadata can also be abused by an attacker to redirect the execution flow, thus they are also part of our model. When the execution-flow of the process reaches the overflowed variable it will be redirected to an attacker-controlled memory location. It is important to point out that we do not make any assumptions about the attacker's methodology of executing malicious code. Thus, in our model, the attacker can utilize all the known ways of executing malicious code, i.e., injecting malicious shellcode in the process' address space [3, 12], *return2libc* attacks [14, 32] and *return-oriented* programming [10, 33].

Non-control-data attacks Chen et al. [11] have shown that non-control-data attacks can be as dangerous as control-data attacks. In a non-control-data attack, attackers no longer try to redirect the execution-flow of the vulnerable program to malicious code but rather attempt to change the values of data structures that can lead them to privileged operations (such as changing the contents of a variable containing a file-path or an integer variable indicating the application-specific privilege level of the current user). Due to their severity, non-control-data attacks are also part of HeapSentry's attacker model.

3 HeapSentry Design

HeapSentry is a system designed to protect against malicious heap overflows through the cooperation of the dynamic memory allocation library (user space component) of any given process and the kernel (kernel space component). HeapSentry intercepts all calls to memory allocation functions and appends each allocated object with a random value that serves as a canary for that heap object. The locations of all canaries and their original values are communicated to the kernel component of HeapSentry where they will be checked when the program requests a system call. In order to differentiate between the two HeapSentry components in the later sections, *HeapSentry-U* will be used to denote the user space component and *HeapSentry-K* to denote the kernel space component. Although, in general, this paper focuses on the Linux OS, the techniques and design of HeapSentry are, in principle, applicable to all modern OSs.

3.1 Interception of memory allocation functions

In the user space, a process is dynamically-linked when parts of the code necessary to execute are resolved and linked to the address space of the process at runtime. The most commonly-used library that virtually all executables link to is `libc`. Among the functionality existing in `libc`, is the ability to dynamically allocate and deallocate memory, through functions such as `malloc` and `free`. HeapSentry-U is added to the run-time link chain of a process in a way that allows us to intercept all calls towards the memory allocation functions (for implementation details see Sec. 3.5). Depending on the allocation function requested, HeapSentry-U performs different operations:

void *malloc(size_t size): `malloc` is called by a program when it requires a new chunk of memory of a specific size. HeapSentry-U intercepts every call to `malloc` where it adds to the requested size, the size of an integer (4 bytes in 32-bit systems) before calling the actual memory allocation library. When the call returns, HeapSentry-U generates a "fresh" random integer which it writes in the last 4 bytes of the allocated block. HeapSentry-U generates a new random value for each allocated object in order to stop attackers that may attempt to infer the value of an overflowed canary by observing neighboring ones.

In our basic design, the canaries are communicated to the kernel space component before the allocation function returns; therefore, HeapSentry-U then invokes

a system call passing the address of the canary and its value as arguments. We use unimplemented system call numbers to pass information to HeapSentry-K which are then ignored by the rest of the kernel. This allows us to transfer information to our module without the need of adding new system calls to the kernel and thus without the need for kernel recompilation. The process now pauses and the kernel wakes-up to handle the interrupt for the system call. When HeapSentry-K is loaded, it hijacks the execution flow of the kernel, just before the dispatch to each individual system call. HeapSentry-K recognizes the system call as part of its protocol and adds the new canary (location and original value) to its internal structures. Once the addition is complete, HeapSentry-K returns the control to HeapSentry-U. HeapSentry-U then returns the pointer to the allocated object to its caller and the execution continues.

void free(void *ptr): A program calls the `free` function once it is done with a memory block and wishes to return it to the allocation library, so that it may be used in later allocations. Once HeapSentry-U intercepts the call, it uses the data structures already in place by the memory allocation library to detect the size of the current memory block. In this way, the user space component of HeapSentry can find the location of canaries, (`ptr + sizeof(block) - sizeof(canary)`), without holding explicit location information about them. Once the canary is located, HeapSentry-U contacts HeapSentry-K and requests a check of the canary. HeapSentry-K locates the original canary value and compares it with the current one. If the two values are different, this means that the program overflowed past the boundary of that specific heap object. In this case, HeapSentry-K terminates the calling process. If the canary is untampered, HeapSentry-K removes it from its internal lists and gives control back to HeapSentry-U which returns the block to the underlying memory allocator.

All other dynamic memory allocation functions, e.g., `realloc` and `calloc`, are implemented based on the two aforementioned ones.

3.2 Detection and Termination

When a program is protected with HeapSentry each heap object contains a random canary and HeapSentry-K has a list of all canary locations and their original values - see Fig. 1. After a successful heap overflow, an attacker with control of the execution flow of a program will eventually need to access kernel resources, through the means of a system call. Checking the liveness of canaries at the point of system call invocation at the kernel-level provides a desired balance between security and performance for the following reasons:

- The attacker cannot normally do any long-lasting damage to the system without the use of system calls [7, 20, 22, 29] since they are necessary to perform all operations outside of the process' environment e.g. write and read files and launch new processes. Even in non-control-data attacks, an attacker seeks to abuse a program's existing system calls.
- Hardware-level isolation does not allow an attacker to bypass or tamper with the detection routines if they are situated in kernel-level memory. Any

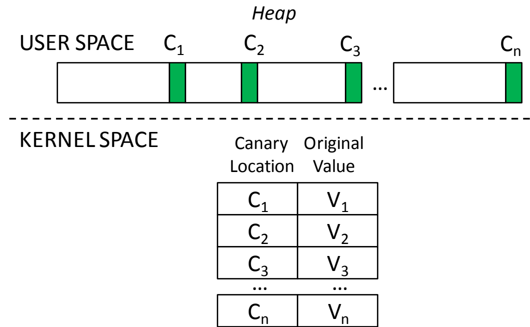

Fig. 1. High-level view of the heap canaries and kernel-level structures of *HeapSentry*

  attempts to access memory from Ring 3 to Ring 0 will cause an interrupt and immediate termination of the offending process
– System calls occur less frequently than normal function calls (e.g. `malloc`).

As mentioned earlier, when HeapSentry-K is loaded to the kernel of the operating system, it hijacks the program flow of the system call code path just before the dispatching of each specific system call. This is an advantageous point since the kernel has not yet called any specific system call and using the `eax` register, HeapSentry-K is able to unambiguously detect the system call requested by the calling process.

A requested system call may be either one that is part of the program (legitimately or maliciously invoked) or one of the unimplemented system calls that are used by HeapSentry-U to communicate information to HeapSentry-K. If one of the latter is detected, HeapSentry-K either adds a new canary location and value to its internal structures or checks the value of an existing one. HeapSentry-K stores the canary locations and their original values by utilizing a combination of a hash table and double-link tail-based lists for handling hash collisions.

When HeapSentry-K detects the execution of a system call not belonging to HeapSentry, it scans the heap of the calling process for modified canaries, by comparing the current values of all canary locations, with the original values stored in its internal structures. In case of a mismatch, HeapSentry-K needs to terminate the process before the execution of the system call (since it may be already malicious). In order to cleanly terminate the process, HeapSentry-K substitutes the original value of `eax` (containing the number corresponding to the requested system call) with the number of the `exit` system call. When the control is given back to the original system call handling code, the Linux kernel will recognize the requested system call as `exit` and will in turn terminate the process instead of calling the originally-requested system call.

3.3 Protecting the kernel

Since all the information about canaries in HeapSentry-K are stored on the kernel's heap, it is necessary to protect the kernel of the operating system from

Denial-of-Service attacks where an attacker would add enough canaries to exhaust the kernel's heap. This scenario is different from DoS attacks against the vulnerable user space application which are not included in our attacker model.

In order to stop such an attack, HeapSentry-K allows up to a user-configurable maximum number of tracked canaries. If that number is reached, HeapSentry-K checks the entire set of canaries for overflows, and if all the canaries are untampered, it empties the kernel-level hash table and returns the memory to the kernel's heap. The entire set of canaries is scanned so as to protect the system from a possible attacker who is attempting to evade detection by forcing HeapSentry-K to "forget" the location of the canary he modified during the overflow that provided him with control of the execution flow. The flushing behavior of HeapSentry-K can be abused by an attacker only if there is a heap object that was allocated before the flushing of the HeapSentry-K tables and is reachable and overflowable by vulnerable code after the flushing. Both of these conditions rarely occur in tandem, since individual heap allocations are by nature temporal and due to HeapSentry-K's small memory footprint (see Section 4.3), our system can keep track of millions of allocated objects without the need of flushing.

3.4 Optimizations

While the previously described design of HeapSentry is able to detect and stop all heap overflows listed in our attacker model, its frequent use of system calls and the continuous check of all canaries could negatively affect the performance of applications which make heavy use of the heap. In this section we describe two optimizations over HeapSentry's basic design that greatly improve its performance without sacrificing its security contributions.

System call Categorization In the previous sections we discussed how an attacker needs to perform a system call in order to do anything of value. Accordingly, HeapSentry exploits the attacker's dependency of the kernel to check the liveness of its heap canaries and terminate the attacked process if it detects an overflow. In heap-intensive programs, the check of all heap canaries at every system call invocation could degrade the overall performance of the application.

To avoid this behavior, we categorized each system call based on the likelihood that it is requested by an attacker, as part of an on-going attack. We did this, by carefully examining and recording the behavior of existing attacks against well-known vulnerabilities. For instance, in drive-by download attacks against browsers, a user's vulnerable browser starts downloading and executing, without the user's consent, malicious binaries from the Internet [16]. Thus in these attacks, the attacker would have to execute the necessary system calls for the creation and execution of new files, as well as the retrieval of data from remote hosts.

Our categorization resulted in three groups, namely *High-Risk*, *Medium-Risk* and *No-Risk* – see Table 1. *High Risk*, are the system calls that attackers traditionally use when exploiting a system, e.g. the `execve` system call that executes a

Category	Name	Description
High-Risk	`fork`	create a child process
	`execve`	execute program
	`chmod`	change file access permissions
	`open`	open a file or device
Medium-Risk	`read`	read from file descriptor
	`write`	write to file descriptor
	`mount`	mount file system
No-Risk	`getpid`	get process identifier
	`chdir`	change working directory
	`brk`	change data segment size

Table 1. Sample from the categorization of Linux system calls according to their risk/usefulness for an attacker

requested program. An invocation of such a system call could be the result of an attacked process and thus, when a *High-Risk* system call is detected, HeapSentry checks all of the active canaries in the process' address space to ensure that no heap overflows predate the system call. The *Medium Risk* group, contains system calls that can be advantageous for an attacker but, unlike the *High-Risk* ones, not in isolation. In this case, HeapSentry checks a subset of the active canaries, expressed as a percentage of the total live canaries, before allowing the system call to proceed. The rationale behind this strategy, is that while the overflowed object may not be detected at the first *Medium-Risk* system call, the attacker would be detected before completing his attack. In Section 4.3 we investigate how the ratio of canaries that are checked at every system call affects the performance of our system.

Lastly, *No-Risk* system calls are system calls that are either not advantageous to be used as part of an attack, or can be used only after a *High-Risk* system call has been used. A typical example, is the `brk` system call which occurs very frequently in memory-intensive programs. This system call is usually initiated by the memory allocator which requests from the kernel the expansion of the process' heap. While this is very useful for a process, it is of no value to an attacker. Consequently, when HeapSentry detects a *No-Risk* system call it allows it to proceed without checking any canaries. The system-call classification is encoded and enforced in the kernel-part of HeapSentry and thus cannot be tampered-with or bypassed by a user space attacker. A security evaluation of our classification, using real-life attack code is presented in Section 4.2.

Grouping operations In the basic design of HeapSentry, each time a new object is allocated or deallocated, this information must be propagated to the kernel (adding a new canary to the list of active canaries or checking and removing an existing one). In this scenario, HeapSentry-U (the user space component of our system), would need to perform a system call at each of these operations.

In order to avoid frequent system calls, HeapSentry-U reports to the kernel in groups. When a `malloc` occurs, HeapSentry-U generates and appends a

new random canary to the allocated block but does not report it directly to HeapSentry-K. Instead, the canary's location and value are stored temporarily in a buffer in the memory allocation library. When this buffer fills-up (the size of the buffer depends on the user's configuration of our system), HeapSentry-U then performs a system call which informs HeapSentry-K of the new set of canaries. In addition to HeapSentry-U "pushing" information to the kernel, HeapSentry-K "pulls" information when it deems it necessary. More precisely, when a *High-Risk* system call occurs, the kernel part of HeapSentry reads the buffers from user space and adds any "pending" canaries to its internal list. This is done to ensure that no overflows have occurred in blocks that are not yet reported.

Similarly, when a `free` occurs, HeapSentry-U adds the canary to a separate buffer. An important difference between the batch operations done for `malloc` and `free`, is that in the case of `free` operations, HeapSentry-U does not actually free the allocated objects, until after it informs HeapSentry-K about them. This is done due to the fact that the actual memory allocator could coalesce the deallocated object with neighboring free objects and then return this new block to a future request for a larger memory block. In the new block, the old canary of HeapSentry would likely be overwritten (since it is at a position that is now a legitimate part of the requested size), resulting in a false-positive. In our case, the memory blocks will only be available for re-use after HeapSentry-K checks their canaries and subsequently removes them from its internal lists.

While the performance benefits of executing less system calls are obvious, one may think that this grouping may open up HeapSentry to attackers who can abuse the canaries that are not yet reported to the kernel. We address these concerns with the following example: consider a process that is allocation-intensive and has, on average, 100,000 allocated objects on the heap. The default size of HeapSentry's batch buffers is 50 entries, i.e., the memory allocator will transfer information to the kernel, once every 50 allocations/deallocations. In our example, at any point in time, the locations and values of the canaries of 99,950 allocated objects (99.95%) will be already safely-stored in the kernel away from the process' reach. This leaves a maximum of 50 canaries (0.05%) that an attacker could attempt to modify. Now, assume that a heap overflow occurs and the attacker achieves control of the execution flow (Fig. 2). If the overflow happened in one of 99.95% of the canaries already stored in the kernel, the attacker has no way of tampering with the canaries' original values. Additionally, since our system generates a new random canary per allocated block, the attacker cannot infer the value of the overflowed object, based on values of neighboring canaries. Thus, HeapSentry will detect the overflow at the invocation of a system call and terminate the program and the attack.

If the heap overflow happened in one of the unreported heap canaries (0.05%), the attacker will have to locate the temporary buffers, find the entry of the heap object he overflowed and remove it, or read it and restore the overwritten canary. If the attacker skips this step and attempts to perform a *High-Risk* system call, the kernel-part of HeapSentry will check for any pending canaries at the user

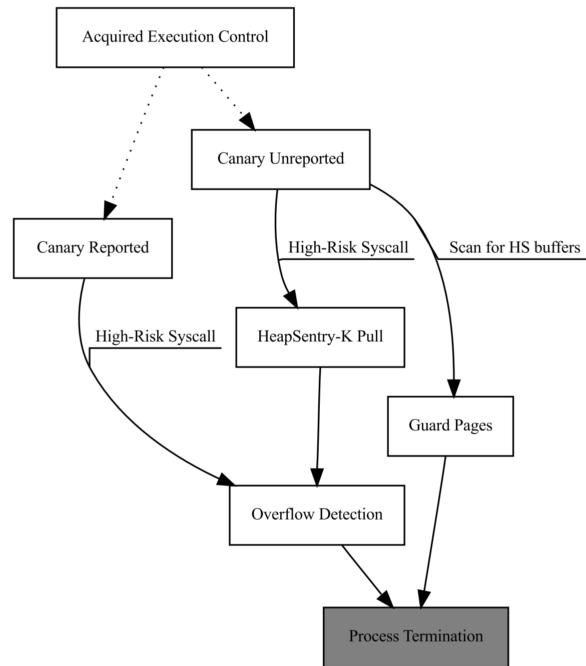

Fig. 2. CFG of an attacker achieving control of a process' execution flow and the reaction of HeapSentry to actions initiated by the attacker. The handling of Medium-Risk system calls is not shown, in order to maintain the overall readability of the figure

part ("pull" operation) and thus will add and immediately detect the recently overflowed object.

In order to stop the attacker from locating the buffers by scanning the heap, the memory page containing them is surrounded by guard pages which will cause the process to be terminated if read or written. Lastly, note that any additional active threads that are not under the attacker's control will continue to operate. If some threads need dynamic memory, they will continue to allocate and deallocate memory from the heap thus filling-up HeapSentry-U's buffers and causing the allocator to inform the kernel of the new memory canaries, including the one that was overflowed by the attacker. Additionally, if a thread performs a *High-Risk* system call as part of its regular operations, HeapSentry-K will "pull" the unreported canaries from the user space and thus also detect the recently overflowed object.

Given the foregoing, we reason that the grouping of allocations and deallocations significantly lowers the overhead of HeapSentry without compromising any of its original security guarantees.

3.5 Implementation details

HeapSentry is comprised of two parts: a library working on top of existing memory allocators in the user space of a process and a kernel-level module. In our Linux prototype, the library was compiled as a position-independent dynamic library that was loaded into existing binaries using the `LD_PRELOAD` directive of the dynamic linker.

The kernel-part of HeapSentry is a Loadable Kernel Module which uses the KProbes library [21] to hijack the control flow of each system call thread at the assembly instruction just before the dispatch of each specific system call. At the kernel-space the process identifier of a protected process is used to locate the HeapSentry-K structures specific to that process. As described in previous sections, when a heap overflow is detected the `eax` register is overwritten by HeapSentry to contain the number of the `exit` system call instead of the one requested by the attacked program. Normally, KProbes is meant to be a framework for inspecting data in the Linux kernel in order to measure statistics or investigate crashes and does not support changing the values of registers. More precisely, KProbes saves the values of all registers (by pushing them on the kernel stack) before handing-off execution to a function in a kernel module and restores them when the module returns execution to KProbes. In order to overcome this, when HeapSentry needs to terminate a process, we trace the stack of caller functions until we locate the register values that KProbes saved. Once they are located, the value corresponding to the `eax` register is modified and execution is handed back to KProbes. When KProbes restores the saved values in their appropriate registers, it will restore `eax` with the overwritten value and thus the system call-handling thread will call `exit` instead of the originally requested system call.

4 Evaluation

4.1 Attack Coverage

In the previous sections we presented the workings of HeapSentry and provided descriptive arguments concerning the attacks that it covers. In this Section, we quantify the protection against heap overflows provided by our system, using RIPE [38], an open source testbed which quantifies the protection of any given system against buffer overflows. RIPE is a process that attacks itself in hundreds of different ways and reports the success or failure of any given attacking technique. We used an Ubuntu 9.10 Linux distribution where we configured RIPE to launch all attacks specific to the heap and in Table 2 we summarize the results. By disabling all default countermeasures of the operating system (ASLR, W-xor-X and ProPolice), RIPE performed successfully 112 attacks against the heap. By turning them back-on, RIPE's successful attempts decreased to 22. We repeated the above runs with HeapSentry enabled on the operating system. When HeapSentry is enabled, and all other countermeasures are disabled, RIPE was able to perform 20 successful attacks. When HeapSentry was cooperating

HeapSentry	ASLR, W-xor-X & ProPolice	#Successful attack forms
OFF	OFF	112
OFF	ON	22
ON	OFF	20
ON	ON	10

Table 2. Attack coverage of HeapSentry compared to existing protection mechanisms - lower is better

with the other countermeasures, the successful attack forms dropped to 10. The attacks detected and stopped by HeapSentry but not by the default countermeasures, targeted function-pointers present on adjacent heap blocks or overwrote a critical memory location through an indirect pointer overwrite. In order to circumvent the W-xor-X countermeasure, the attacks used the *return2libc* technique to execute malicious code. HeapSentry however, can detect overflows regardless of the attacker's way of executing malicious code and thus could detect and stop the *return2libc* attacks in time.

One can make several observations based on the aforementioned data. First of all, even in modern operating systems with many countermeasures against code injections, when HeapSentry is available on the system, the system is immune to 50% more heap-specific attacks than if it wasn't present. Second, while there is dramatic decrease of successful attacks when the default countermeasures are turned-on, a legacy system that has available none of them but only HeapSentry, is already more secure against heap overflows than modern operating systems with all of the default countermeasures turned-on. The 10 remaining attacks that evaded detection, are variations of the attack exploiting a buffer and a function pointer allocated together as part of the same struct and are discussed in Section 5.

The results of this experiment highlight HeapSentry's effectiveness and ability of detecting and stopping heap overflows in modern operating systems in both the presence and the absence of other countermeasures.

4.2 Security Evaluation of Risk groups

In Section 3.4 we presented our categorization of the Linux OS system calls according to the usefulness of each one from an attacker's perspective, which resulted in three groups of system calls, namely *High-Risk*, *Medium-Risk* and *No-Risk*. In order to check whether our categorization was correct, in this section we test it against existing shellcode. For this purpose, we downloaded the latest 100 shellcode samples from `shell-storm.org`, a website providing information and resources for security testers. Using `strace` [36], we analyzed the system calls requested by each shellcode and recorded the risk category of each one. The purpose of this experiment was the following: supposing that each of these shellcode samples was injected and executed as part of an ongoing attack

that begun with a heap overflow, would HeapSentry detect the overflow and stop the attack in time? Note that the detection would be identical in a case of a *return2libc* or *return-oriented* programming attack performing the same operations since they too, would eventually result in the same maliciously-invoked system calls.

From the 100 samples, we removed 5 that were performing non-critical operations (such as printing obscene messages in all terminals). All the remaining 95 shellcode samples were using at least one High-Risk system call as part of their malicious payload. The majority were utilizing process-launching system calls (e.g. `execve` and `fork`) while others attempted to change the permissions, read and, in some cases, edit critical Linux system files that could give them access to the victim machine (e.g. reading and transmitting the `/etc/shadow` file to the attacker or adding a new user account in the `/etc/passwd` file). In the current configuration of HeapSentry, the `chmod` and `open` system calls are High-Risk system calls and all attacks against system files need to perform either one or both of them. As explained in Section 3.4, when a High-Risk system call is requested, HeapSentry checks the health of the entire heap before allowing the call to proceed. Thus, for all samples, the overflow would be detected and the process killed before the completion of the attack.

4.3 Performance

# Heap Objects	HS-U	HS-K	Total
1,000	16	81	97
100,000	412	1,665	2,077
1,000,000	4,012	16,065	20,077

Table 3. Memory overhead (in KBytes) of HeapSentry depending on the number of live heap objects

Memory Overhead Both components of HeapSentry need to add and maintain information in order to accurately detect heap-based buffer overflows. In the user space, HeapSentry-U augments each allocation request with the size of an integer where it will store its new canary. Additionally, HeapSentry-U needs a total of 3 memory pages, one where it stores the unreported canaries and two that serve as guard pages for the first page. In the kernel space, HeapSentry-K requires a hash table and doubly-linked tail-based lists for handling hash collisions. In our current configuration, the hash-table structure requires 16K integers and then each added canary requires another 4 integers. In Table 3 we present the memory overhead (for 32-bit architectures) depending on the number of live allocated objects in a process' heap. Not shown in the table is the overhead of HeapSentry due to the grouping of deallocations, which however is negligible in comparison to the aforementioned memory requirements.

Overall, these results show that HeapSentry imposes only a modest memory overhead, even for allocation-intensive programs (less than 20 MBytes for a process with 1 million active heap objects).

Benchmark	**HS 1/32**	**HS 1/16**	**HS 1/8**
400.perlbench	1.60	1.70	1.88
401.bzip2	1.00	1.00	1.00
403.gcc	1.04	1.04	1.06
429.mcf	1.00	1.00	1.00
445.gobmk	1.00	1.00	1.00
456.hmmer	1.00	1.00	1.00
458.sjeng	1.00	1.00	1.00
462.libquantum	1.00	1.00	1.00
464.h264ref	1.00	1.00	1.00
471.omnetpp	1.24	1.24	1.24
473.astar	1.00	1.00	1.00
483.xalancbmk	1.20	1.21	1.21
Average	1.090	1.099	1.116

Table 4. Runtime performance of HeapSentry on the SPEC Int 2006 Benchmarks - results normalized with GLIBC default allocator

Run-time Overhead In order to quantify the overhead of our system in real-world scenarios, we evaluated it using the SPEC CPU2006 Integer benchmark suite using the reference workload. The experiments were conducted on a machine with an Intel Dual Core processor at 2.66GHz and 4GB of memory. Each experiment was repeated three times and the average run-time of each benchmark is shown in Table 4, normalized by the time of the standard memory allocator in Linux systems. To show how different parameters affect the performance of HeapSentry, we measured the overhead of our solution with three different configurations for the *Medium-Risk* system calls (Sec. 3.4). In the first experiment, each time a *Medium-Risk* system call was requested by the running program, HeapSentry checked the canaries for 1 out of 32 active heap objects. In the second experiment, 1 out of 16 active objects was checked and lastly 1 out of 8. The larger the percentage of checked objects per system call, the longer the process has to wait before regaining control of the CPU and thus the longer it will take to fully execute. Note that in all three settings, requests for *High-Risk* system calls will always cause a scan of the entire set of heap objects.

The results show that only 3 out of the 12 benchmarks experience significant slowdown due to HeapSentry. The benchmark that is affected the most, `perlbench`, is a highly allocation-intensive program that combines many millions of memory allocations with tens of thousands of *Medium-Risk* system calls. In the third experiment (HeapSentry 1/8), `perlbench` experiences a 88% overhead over the non-protected version. The other two benchmarks, `omnetpp` and `xalancbmk` are also allocation intensive but have less *Medium-Risk* system calls

than `perlbench`. The average for the HeapSentry 1/8 over all 12 benchmarks is 11.6% percent. In comparison, the average overhead of DieHarder is 20% [25]. Cruiser [41], due to the use of dedicated threads, reports better results, however in real systems with more concurrent protected applications than available number of CPUs, the dedicated threads will be regularly scheduled-out by the kernel. This scheduling-out, apart from degrading the reported performance of Cruiser, will also create windows of opportunity for an attack to go undetected. We discuss in detail the security of DieHarder and Cruiser in Section 6.

5 Limitations

While HeapSentry can detect and stop a wide range of heap overflows there is one case where an overflow would go undetected. Since our system is canary-based, an attacker who manages to overwrite a critical location on the heap without first overwriting the canary will be able to avoid detection. This can happen only when the overflowing buffer and the target are in the same heap object, i.e., they are both part of the memory block that was allocated through a single memory allocation call. An example would be a dynamically allocated `struct` that contains a character buffer and a function pointer where the former could overflow the latter. This problem is shared by all canary-based systems, by all security-conscious allocators and by most bounds-checkers since the overflow happens within the same object (in-bounds write).

The same problem would also manifest in a program that does not rely on standard memory allocators, but rather first requests a large amount of memory from the operating system and then implements its own custom memory allocator on top of that space. In this case however, a program that would be willing to protect itself could use HeapSentry as an API where it would request the placement and maintenance of canaries in specific memory locations.

6 Related Work

Due to the plethora of research in code injection countermeasures, in this section we mainly discuss the work that is most relevant to HeapSentry. A broader survey of related work can be found in [40].

6.1 System call monitors

System call monitors have received a lot of focus by the research community due to several attractive characteristics, such as the fact that they cannot be circumvented by user space applications and the attacker's dependence on system calls. Bernaschi et al. [7] propose a system call monitor that checks the validity of system calls and system call arguments based on an access-control database. The downside of this approach is that the rules of the database must be manually encoded by the administrators of a system for all system calls and

applications that wish to be protected. Kc et al. [20] propose a similar monitor without the need of manual rule encoding. At the kernel-level they inspect the return address of the requested system calls to stop the injection of new code in the stack or heap of a process. Additionally they perform checks to ensure that a system call that originates from the `.text` section of a process was legitimately called by the process and not by an attacker, through analysis of the call-paths leading to all system calls and validation of them at run-time. Unfortunately, these techniques cannot stop non-control-data attacks, since the call-paths leading to the exploitable system call are the same. Other problems include, an attacker using return-oriented programming to de-randomize their stack layout and then mimic legitimate system calls, and possible impedance of Just-in-Time compilation techniques which create new call-paths at run-time [17].

Linn et al.'s work [22] suffers from similar problems since they cannot account for system calls and arguments that are not detectable through the inspection of a binary. Provos proposes a system call monitor, SysTrace [29], that can make decisions using data from earlier training sessions and/or interactively asking the user to allow or deny a system call. While this could be a viable security approach, we believe that non-technical users will not be able to use it or would just end-up allowing all requested system calls.

In comparison with the aforementioned system call monitors, HeapSentry does not require training or user interaction and it is not vulnerable to mimicry attacks. Additionally our system stops non-control-data attacks and does not use static analysis of a binary, allowing programs that use Just-in-Time compilation techniques to work without modification. On the other hand, since HeapSentry is a heap-specific solution, our system would need to be combined with other approaches in order to stop attacks that occur on a different data segment (e.g., the stack).

6.2 Canary-based approaches

StackGuard [13] introduced the use of random values as a way of identifying buffer overflows on the stack. The stack has a very specific caller-callee protocol (implemented through the function prologue and epilogue) which allows the checking of the canary values right before the execution flow is given back to the caller. ProPolice [18] later re-implemented StackGuard and added a series of new features that increased the overall security of the stack, e.g. re-organizing the local variables and placing character buffers right next to the canary. ProPolice is widely used in modern operating systems but it does not add any protection mechanisms on the heap of the running program. Robertson et al. [31] were the first ones to adapt the idea of canaries to protect the heap. In their approach, a global process-wide canary was placed at the beginning of each allocated object and was checked at the time the object was freed. Unfortunately this meant that an attacker could still perform a successful heap-based buffer overflow as long as a sensitive value in the overflowed object was used before the object was deallocated. Van Acker et al. [37] wrap all variables in canary-protected structures, but require access to source code and incur a significant overhead.

Recently Zeng et al. [41] presented Cruiser, a concurrent canary-based heap buffer overflow monitoring system. A major difference between our system and Cruiser is that Cruiser attempts to protect user space applications from within the user space thus becoming part of the program's attack surface. In comparison, the original canary values and detection functions of HeapSentry are situated within the kernel out of the attacker's reach. Cruiser's *modus operandi* is as follows: In Cruiser each heap block is prepended and appended with canaries that are then checked by a separate user-level thread, "cruising" over the address space of the process. Unlike HeapSentry, each canary is not random but is the result of a XOR operation between process-wide keys, the address and the size of the protected object. While this enables Cruiser to recompute canaries without the need of storing their original values, it also opens up the system to attacks. An attacker that achieves control of the execution flow, can read the neighboring canaries and, given that the size and address of them are known, can compute the XOR key needed to recreate the canary of the object that he overflowed. Additionally, depending on the load of the system, the number of available cores on a CPU and the number of canaries that need to be checked by the dedicated user-level thread, an attacker could successfully request one or more malicious system calls (e.g. `execve('/bin/sh')`) before the thread detects the overflowed canary. Contrastingly, HeapSentry *synchronously* stops all High-Risk system calls in the kernel and does not allow them to proceed before the health of all canaries is verified.

6.3 Security-conscious allocators

HeapShield [5] by Berger, is a memory allocator that instead of organizing objects in free-chunk lists, organizes them in pages, where each page holds objects of a specific size. HeapShield then intercepts all exploitable libc function calls, such as `strcpy`, and checks whether the size of the destination object is large enough for the requested operation.

The concepts behind HeapShield were later generalized and incorporated into DieHard [6], an allocator providing probabilistic memory safety for unsafe languages. DieHard approximates an "infinite heap" by randomly distributing objects on the heap and requiring the heap to be M times larger than needed. While DieHard helps applications to run correctly in the presence of heap errors and completely eliminates certain classes of bugs, such as double-frees, an attacker can still perform heap-based buffer overflows by adapting to the changes of the heap layout. For instance, as in HeapShield, objects of the same size are placed on the same memory pages. In this scenario, an attacker can still overflow freely from one heap object to the other, as long as they are part of the same size category. Additionally, since DieHard rounds-up objects to the nearest power of two, objects that may have been allocated "far-away" from each other by best-fit memory allocators, may now be allocated in the same page, thus enabling an attacker with a limited write-range to successfully overflow from the one to the other. In contrast, HeapSentry protects each object with its own unique canary which allows our system to detect a heap overflow even if the application

wrote just a single byte past its boundary. Although DieHard was extended in DieHarder [25] with additional security features, such as a "destroy-on-free" and "address space sizing", the aforementioned problems still remain.

Archipelago [23] is a similar approach to DieHard where the abundance of virtual memory pages of 64-bit systems is used to place objects far apart in the virtual address space, without consuming the underlying physical memory. Archipelago imposes a significant slowdown on allocation-intensive applications and it cannot be straightforwardly applied to 32-bit systems since memory-intensive applications would quickly exhaust the virtual memory alloted to the process. Younan et al. [39] modify the *dlmalloc* memory allocator to isolate heap metadata from data by placing the former in a contiguous space protected by guard pages. Even though this technique stops attacks against the metadata of the allocator, it cannot protect data in neighboring chunks from overflows nor can it detect that an overflow has occurred. Lastly, note that security-conscious allocators can hide a bug in the programs that utilize them, which may later resurface if the vulnerable programs are used with a different allocator. Contrastingly, HeapSentry works as a defense layer on top of existing allocators and thus does not change the semantics of allocations but protects the running applications regardless of their memory allocators.

7 Conclusion

In this paper we presented HeapSentry, a system designed to detect and stop heap overflows through the cooperation of the memory allocation library and the kernel of an operating system. We described how it is possible to further involve a kernel in the protection of applications and how this increases the security and resiliency of the protecting system against sophisticated attackers. Finally, we showed that HeapSentry scores better than existing countermeasures of modern operating systems and we demonstrated that HeapSentry stops all attacks involving common malicious code for a modest overhead in real-world applications.

Acknowledgments: This research was performed with the financial support of the Prevention against Crime Programme of the European Union (B-CCENTRE), the Research Fund KU Leuven and the EU FP7 project NESSoS.

References

1. Adobe. Security bulletins and advisories. `http://www.adobe.com/support/security/`.
2. P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, QC, Aug. 2009.
3. Aleph1. Smashing the stack for fun and profit. *Phrack*, 49, 1996.
4. C. Anley, J. Heasman, F. F. Linder, and G. Richarte. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley Publishing, 2 edition, 2007.

5. E. D. Berger. Heapshield: Library-based heap overflow protection for free. In *UMass CS TR 06-28*, 2006.
6. E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of 27th Conference on Programming Language Design and Implementation*, June 2006.
7. M. Bernaschi, E. Gabrielli, and L. V. Mancini. Operating system enhancements to prevent the misuse of system calls. In *Proceedings of the 7th Conference on Computer and communications security*, 2000.
8. S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Washington, D.C., Aug. 2003.
9. S. Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, Paris, France, July 2008.
10. S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of CCS 2010*. ACM Press, 2010.
11. S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, Aug. 2005.
12. M. Conover. w00w00 on heap overflows. `http://www.w00w00.org/files/articles/heaptut.txt`.
13. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.
14. S. Designer. lpr LIBC RETURN exploit. `http://insecure.org/sploits/linux.libc.return.lpr.sploit.html`.
15. D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceeding of the 28th international conference on Software engineering*, Shanghai, China, 2006.
16. M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '09, pages 88–106, Berlin, Heidelberg, 2009. Springer-Verlag.
17. A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *ACM conference on Programming language design and implementation*, 2009.
18. IBM. Gcc extension for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/`.
19. R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging*, pages 13–26, Linköping, Sweden, 1997.
20. G. S. Kc and A. D. Keromytis. e-NeXSh: Achieving an effectively non-executable stack and heap via system-call policing. In *Annual Computer Security Applictions Conference*, 2005.
21. J. Keniston, P. S. Panchamukhi, and M. Hiramatsu. Kernel probes (kprobes).

22. C. Lin, M. Rajagopalan, S. Baker, C. Collberg, S. Debray, and J. Hartman. Protecting against unexpected system calls. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, Maryland, Aug. 2005. USENIX Association.
23. V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII. ACM, 2008.
24. Microsoft. Security advisories. `http://www.microsoft.com/technet/security/advisory/`.
25. G. Novark and E. D. Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 573–584, New York, NY, USA, 2010. ACM.
26. National Vulnerability Database. `http://nvd.nist.gov`.
27. PaX. Documentation for the PaX project. `http://pax.grsecurity.net/`.
28. M. Payer. I control your code. In *Proceedings of the 27th Chaos Communication Congress (27c3)*, 2010.
29. N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, Washington, D.C., Aug. 2003.
30. U. Rivner. Anatomy of the rsa attack. `http://blogs.rsa.com/rivner/anatomy-of-an-attack/`.
31. W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time detection of heap-based overflows. In *Proceedings of the 17th Large Installation Systems Administrators Conference*, pages 51–60, San Diego, CA, Oct. 2003.
32. G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *25th Annual Computer Security Applications Conference*, 2009.
33. H. Shacham. 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*, 2007.
34. Solar Designer. Non-executable user stack. `http://www.openwall.com/linux/`.
35. E. H. Spafford. The internet worm program: An analysis. *Computer Communication Review*, 19, 1988.
36. strace(1): trace system calls/signals. `http://linux.die.net/man/1/strace`.
37. S. Van Acker, N. Nikiforakis, P. Philippaerts, Y. Younan, and F. Piessens. ValueGuard: Protection of Native Applications against Data-Only Buffer Overflows. In *Proceedings of the 6th International Conference on Information Systems Security*, Lecture Notes in Computer Science, pages 156–170. 2011.
38. J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. Ripe: Runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011.
39. Y. Younan, W. Joosen, and F. Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. In *Proceedings of the International Conference on Information and Communication Security*, Raleigh, NC, Dec. 2006.
40. Y. Younan, W. Joosen, and F. Piessens. Runtime countermeasures for code injection attacks against C and C++ programs. *ACM Computing Surveys*, 44(3):17:1–17:28, 2012.
41. Q. Zeng, D. Wu, and P. Liu. Cruiser: concurrent heap buffer overflow monitoring using lock-free data structures. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2011.