HeapTherapy: An Efficient End-to-end Solution against Heap Buffer Overflows

Qiang Zeng*, Mingyi Zhao*, Peng Liu
The Pennsylvania State University
University Park, PA, 16802, USA
Email: {quz105, muz127, pliu}@psu.edu

***Abstract*—For decades buffer overflows have been one of the most prevalent and dangerous software vulnerabilities. Although many techniques have been proposed to address the problem, they mostly introduce a very high overhead while others assume the availability of a *separate* system to pinpoint attacks or provide detailed traces for defense generation, which is very slow in itself and requires considerable extra resources. We propose the first efficient solution against heap buffer overflows that integrates attack detection, exploit diagnosis, and defense generation in a single system, named *HeapTherapy*. It conducts on-the-fly lightweight trace collection on the production system (instead of a *separate replaying* system), and initiates automated diagnosis upon exploit detection to generate defenses in realtime. It can handle both over-write and over-read attacks, such as the recently explosed Heartbleed attack. The system has no false positives, and keeps effective under polymorphic exploits. It is compatible with mainstream hardware and operating systems, and does not rely on specific allocation algorithms. We evaluated HeapTherapy on a variety of services (database, web, and ftp) and benchmarks (SPEC CPU2006); it incurs a very low average overhead in terms of both speed (6.2%) and memory (7.7%).**

I. Introduction

Programs written in C and C++ contain a large number of buffer overflow bugs, which involve writes or reads going beyond buffer boundaries.[1] In addition to erroneous execution, buffer overflow bugs can lead to various security threats, including data corruption, control-flow hijack, and information leakage. The recently published Heartbleed vulnerability, which has affected hundreds of thousands of servers, was due to a heap buffer over-read bug that leads to information leakage [17], [28].

Although there are many tools dedicated to finding buffer overflow bugs through static analysis [47], it is very unlikely to eliminate all the bugs through static analysis. The reality is that in 2014 one third of newly exposed software vulnerabilities published by CERT were related to buffer overflows [45]. Therefore, runtime measures that protect program execution against overflow attacks are important. Such measures can be roughly divided into three categories: (1) Full execution monitoring; (2) Approaches that learn from history to improve themselves; and (3) Measures that greatly increase the difficulty of exploiting buffer overflows. Examples of the third category include StackGuard [15], Data Execution Prevention [2], Address Space Layout Randomization [49], [8], and concurrent heap scanning [52], [44].

The first category contains a variety of approaches, which range from bounds checking [21], [38], [6]; to shadow memory based checking [25], [39], [5], [12]; to control/data flow monitoring [23], [4], [11]; to N-version/N-variant systems [16], [7], [30]. Even with many optimization methods they still lead to a very high overhead, or require significant extra computing resources. For instance, AddressSanitizer [39] uses a much more efficient shadow mapping and a more compact shadow encoding than Valgrind (Memcheck) [25] and TaintTrace [12], but still incurs 73% slowdown and over 3X memory overhead. As another example, N-variant execution [16] requires doubling resource consumption and system maintenance, which is costly.

Instead of performing expensive full execution monitoring, approaches in the second category generate tailored defenses against learned vulnerabilities. Given a zero-day vulnerability, they trade the prevention of the first buffer overflow(s) for subsequent low-cost protection. An example of the second category is patching. However, generating patches is a lengthy procedure. According to Symantec the average time for generating a critical patch for enterprise applications is 28 days [43]. A few approaches that generate defenses quickly after exploit detection have been proposed. One popular approach is to generate input filters to filter out suspicious input [22], [27], [50]. However, the false negative rate rises when dealing with input obfuscation, and false positives are a common issue for identifying innocuous requests as attacks. A type of methods, such as Vulnerability-Specific Execution-based Filtering (VSEF) [26] and Vigilante: [14], are based on the observation that, given a sample exploit, only a small portion of instructions are relevant to the exploit. A defense that monitors the relevant part of the program execution is generated to block further exploits. It is more efficient than full execution monitoring and performs better when handling input obfuscation.

However, VSEF relies on a separate system to provide sample exploits for analysis and defense generation; how to pinpoint malicious inputs efficiently is a challenging problem in itself, and such a system might be unavailable due to resource constraints. Second, a defense in VSEF takes effect by instrumenting instructions accessing a specific overflow target. While instructions that access a target near a vulnerable buffer on the stack, such as a return address, can be easily identified, it is very unlikely to determine the instructions that access adjacent regions of a vulnerable heap buffer, for a heap buffer can be allocated almost anywhere on the heap. A proper solution to heap buffer overflows is missing in that work. Vigilante has the same issues.

*These two authors have contributed equally.

[1] The term “overflow” in this paper refers to both over-writes and over-reads.

As stack-based buffer overflows are better addressed nowadays, heap buffer overflows gain growing attention of attackers [35], [52]. By noticing that full program execution monitoring usually incurs a high overhead, we target an efficient heap buffer overflow countermeasure falling into the second category, i.e., learning from history to protect against attacks. It should meet the following goals simultaneously: (G1) *All-in-one solution*: trace collection and defense generation should be built directly into the production system, so that it does not need a separate system and can respond to any detection of attacks instantly. (G2) *High efficiency*: the overall overhead should be low. (G3) *High accuracy*: the generated defense should have few to no false positives and negatives even with polymorphic attacks. (G4) *Compatible with mainstream hardware and runtime environment*: it should not require special hardware and can work with existing runtime environment; ideally, it does not depend on custom heap allocation algorithms. To our knowledge there is no such solution satisfying all the desirable requirements.

In this paper, we present HeapTherapy, a highly efficient end-to-end solution against heap buffer overflows that meets all the goals above. Unlike approaches applying costly bounds checking or data/control flow tracking, HeapTherapy employs inexpensive techniques to identify vulnerable heap buffers swiftly and enhance them locally. HeapTherapy contains in-memory trace collection, online exploit detection and realtime defense generation as part of the defense system. HeapTherapy identifies vulnerable heap buffers based on the intrinsic characteristics of an exploit, as opposed to filtering out malicious inputs based on signatures, so it is effective under polymorphic attacks. Finally, it can be easily deployed and does not rely on specific allocation algorithms.

To detect over-read attacks HeapTherapy places inaccessible guard pages randomly throughout the heap space, so that, when repetitive attacks such as Heartbleed are launched to collect jigsaw pieces on the heap, it is highly probable that a guard page is touched and hence the attack is detected before significant information is leaked. The widely deployed Address Space Layout Randomization (ASLR) is used to facilitate detection of over-write attacks, as a control flow hijack attack exploiting buffer over-writes causes the program to crash with a high probability due to the difficulty of guessing randomized addresses. This is used in many other defense techniques as well [34], [50].

Instead of relying on a separate system to pinpoint and replay offending requests, HeapTherapy collects lightweight in-memory traces during program execution to assist defense generation. Therefore, it avoids the cost and overhead due to a separate system.

Given a heap buffer overflow vulnerability, the vulnerable heap buffers must share some characteristics, and the characteristic used in HeapTherapy is their common allocation-time calling context. Thus, a defense generated by HeapTherapy contains the calling context when a vulnerable heap buffer was allocated. We employ a recent advance in calling context representation and retrieval — the calling context encoding technique [9], [42], [51], which continuously tracks the current calling context with a very low overhead representing it concisely as an integer, named a *calling context ID*. Whenever a heap buffer is allocated, our *malloc* wrapper compares the current calling context ID against the one contained in any defense. If they match, the new buffer is regarded vulnerable and a guard page is attached after it. Subsequently, without tracking access instructions or the information flow, out-of-bounds buffer access due to continuous read or writes are prevented by system protection automatically. While the guard page is an expensive enhancement [32], HeapTherapy applies it only to vulnerable heap buffers with a low overall overhead.

We have implemented HeapTherapy, and evaluated it on a variety of services (database, web, and ftp) and benchmarks (SPEC CPU2006). The throughput overheads incurred by HeapTherapy on service programs are all less than 8% with zero false positives. A thorough evaluation on SPEC shows that the speed overhead averages 6.2% and the memory overhead 7.7% when dealing with 10 synthesized vulnerabilities simultaneously.

We made the following contributions:

- We propose an end-to-end solution that integrates defense generation and overflow prevention in a single system. The defense is generated automatically on the user side, so the user system can be enhanced instantly, and the user does not need to maintain a separate system for defense generation.
- Compared to existing work, HeapTherapy incurs a very low speed and memory overhead.
- The defense generated by HeapTherapy does not have false positives, and keeps effective under polymorphic attacks.
- It is compatible with existing hardware, systems, and libraries. It does not require a custom heap allocation algorithm. Thus, its deployment is convenient.[2]
- While none of the techniques employed in HeapTherapy, such as canaries, guard pages, probabilistic detection, and calling context encoding, is new or complex, we creatively combine them and deliver a practical low-cost solution. To our knowledge it is the first one that meets all the goals G1-G4 simultaneously.

The remainder of the paper is organized as follows. Section II summarizes the related work. Section III gives an overview of HeapTherapy's design. Section IV describes the design and implementation of HeapTherapy. In Section V, we present the result of evaluation. Limitations of HeapTherapy are discussed in Section VI and potential applications in Section VII. We conclude this paper in Section VIII.

II. Related Work

Due to extensive research in buffer overflows, we do not intend to make an exhaustive list of work on the problem. Instead, we examine how techniques applied in HeapTherapy are used in other work.

Canaries: StackGuard [15] uses canaries to detect stack-based buffer overflows. It has been widely deployed in modern

[2] While our current implementation uses recompilation, it is feasible to implement through binary instrumentation; thus, recompilation is not an intrinsic limitation of the solution.

compilers, and greatly increased the difficulty of stack-based buffer overflow attacks. Robertson et al. proposed to detect heap buffer corruption by checking canaries placed between heap buffers [37]; however, the checks are only performed at buffer allocations or deallocations, which leaves a large exploitable time window. Cruiser [52] and Kruiser [44] dramatically reduce the time window, but they do not prevent data corruption itself. In our work, canaries are only used to detect buffer over-writes. Once an over-write is detected, HeapTherapy generates a defense preventing data corruption from repetitive attacks.

Guard pages: Electric Fence puts one inaccessible page immediately after or before a buffer [32]. DYnamic Buffer Overflow Containment (DYBOC) surrounds every buffer with two inaccessible pages [40]. The full enhancement for all heap buffers incurs prohibitively high overhead, while HeapTherapy applies guard pages only to over-read detection probabilistically and vulnerable buffers deterministically, incurring a very low overhead.

Context sensitive defense: The value of calling context beyond debugging was recognized early. For example, region-based heap allocation tags heap objects with allocation calling context information [53]. Calling context was recently used to generating context sensitive defenses [50], [26], [31], [46], [20]. However, they commonly use costly call stack walking. HeapTherapy is the first work that employs the calling context encoding technique to generate context sensitive defense. It largely reduces the overhead compared to using other calling context retrieval techniques such as stack walking [9], [42], [51]. Through the calling context encoding technique HeapTherapy is able to represent the characteristics of buffers being exploited with one integer and to identify vulnerable buffers through integer comparison.

Learning from attacks: The principle has been widely applied to defense generation systems [22], [27], [26], [50], [31], [14]. They usually use or assume a separate system to pinpoint the attacking request, but the separate system can introduce significant costs or deployment difficulties. HeapTherapy collect lightweight traces in memory to assist defense generation without the need of finding out the malicious request. It is an end-to-end solution built into the system being protected.

HeapTherapy is built on a series of different techniques. Canaries incur a low overhead but provides no overflow prevention; it is used in data corruption detection. Guard pages are expensive but can prevent overflows; it is applied to over-read detection probabilistically and buffers identified as vulnerable deterministically. The glue is the calling context encoding, which characterizes and identifies vulnerable buffers. Those techniques work together to deliver a new efficient solution against heap buffer overflows.

III. HeapTherapy Overview

Figure 1 shows the architecture of HeapTherapy, which contains three main components: a compiler pass for adding the calling context encoding functionality, a shared library for interposing the memory allocator, and a diagnosis engine for generating temporary patches upon detection. The memory allocator interposition library enforces the installed patches by enhancing the heap buffer being allocated.

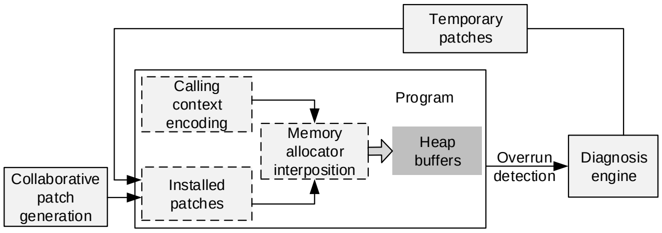


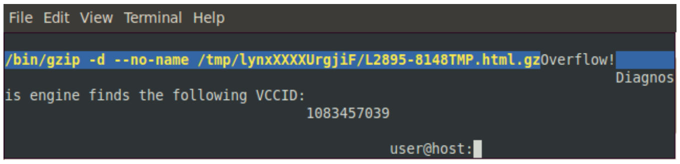
Fig. 1: The architecture of HeapTherapy.

As an example, we show how HeapTherapy can be used to harden the text-based browser `Lynx` against a heap buffer over-write attack. When parsing an HTML link whose hostname includes a % character in the last two bytes, the URL decoding code of `Lynx 2.8.8dev.1` will write attacker controlled data after a heap buffer's boundary, leading to a crash or arbitrary code execution (CVE-2010-2810). In the following steps, we use an exploit provided here [24].

A user applies HeapTherapy to hardening `Lynx` as follows:

Compilation: The user compiles `Lynx` with a LLVM compiler, to which we added a calling context encoding instrumentation pass (`PCC.so`), and links it with the `heaptherapy` shared library for memory allocator interposition by providing the flags:

```
CFLAGS= -Xclang -load -Xclang PCC.so
LDFLAGS= -ldl -L. -lheaptherapy
```

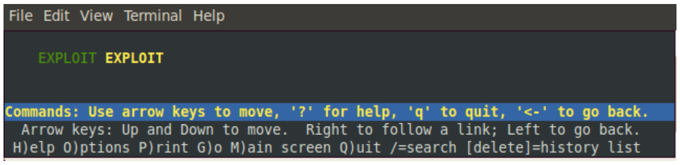


```
/bin/gzip -d --no-name /tmp/lynxXXXXUrgjiF/L2895-8148TMP.html.gzOverflow!
                                                                  Diagnos
is engine finds the following VCCID:
                                 1083457039

                                         user@host:
```

Fig. 2: A crash due to the malformed HTML link, and the detection and diagnosis result of HeapTherapy.

Online detection and diagnosis: The user then starts `Lynx`. Once the malformed link is visited, an over-write will occur. HeapTherapy detects this over-write, terminates the program and generates a core dump file. The core dump file is then analyzed by the diagnosis engine to locate the overwritten buffer and to generate a temporary patch containing the vulnerable buffer's allocation calling context encoding. Figure 2 shows the result of this step.



```
   EXPLOIT EXPLOIT


Commands: Use arrow keys to move, '?' for help, 'q' to quit, '<-' to go back.
  Arrow keys: Up and Down to move.  Right to follow a link; Left to go back.
 H)elp O)ptions P)rint G)o M)ain screen Q)uit /=search [delete]=history list
```

Fig. 3: Lynx displays the malformed links without crashes.

Defense: The temporary patch is then stored in a configuration file, which will be loaded once the user starts `Lynx` again. This time when `Lynx` parses a link exploiting the same

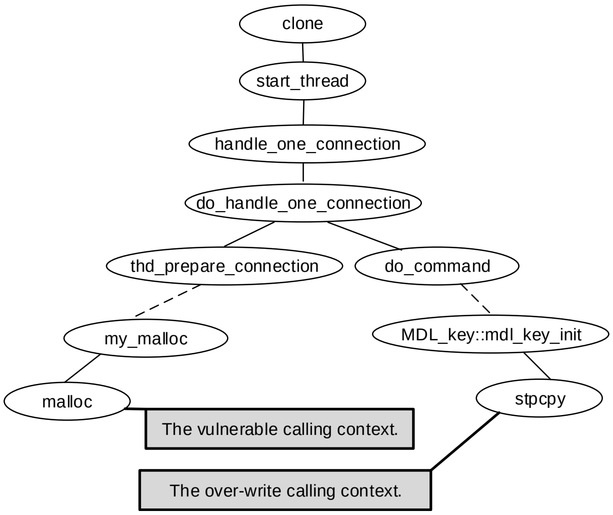


Fig. 4: The calling context tree of `MySQL5.5.19` related to CVE-2012-561. Dashed lines indicate omitted functions.

vulnerability, HeapTherapy prevents the over-write and allows `Lynx` to display the page normally without a crash, as shown in Figure 3. We mutate the attack and find the patch still effective.

Collaborative patch generation: In addition to locally generated patches, the patches can also come from other machines. The shared and collaborative patch generation makes an early response to large scale attacks exploiting zero-day vulnerabilities possible.

IV. Design and Implementation

In this section, we present the design and implementation of HeapTherapy. We first introduce the idea of applying calling context encoding to identifying vulnerable heap buffers instantly, and then explain overflow detection, attack diagnosis, and defense generation.

A. *Calling-context Sensitive Defenses*

HeapTherapy uses calling context as a characterization of vulnerabilities and the guidance for applying defenses. A *calling context* is a sequence of unreturned method invocations that lead to a runtime operation. It is also loosely referred to as *call stack* and *back trace*.

Given an attack that exploits a bug to overflow a buffer, our observation is that the allocation calling context of the buffer is the same when the attack recurs. Figure 4 illustrates such an example. It contains part of the calling context tree of `MySQL 5.5.19` with a heap buffer overflow vulnerability (CVE-2012-5612). The left branch shows the calling context when a vulnerable buffer is allocated, while the right one the over-write calling context,. We mutated the script exploiting the vulnerability and the calling contexts were reproduced. The observation is confirmed by our experiments on a wide variety of programs (Section V-A), and it is consistent with existing work that employs calling context for security purposes [50], [26], [31]. We do encounter cases where a single vulnerability corresponds to multiple vulnerable allocation calling contexts (the `Nginx` example in Section V-A); however, the number is very small, and HeapTherapy handles such cases easily by generating one defense for each vulnerable calling context.

```
foo() {
    int tmp = V; // Added: backup encoding
    ...
    V = 3 * tmp + cs_1; // Added: update it
cs_1: bar1();
    V = 3 * tmp + cs_2; // Added: update it
cs_2: bar2();
    ...
    V = tmp; // Added: recover the encoding
}
```

Fig. 5: An example of PCC encoding. Four lines are added to maintain the encoding.

HeapTherapy requires frequent retrieval and comparison of calling contexts. It is notable that if these operations incur high overhead, e.g., through stack walking, the overall performance degradation will be significant. Therefore, we employ calling context encoding techniques to speed up. A few encoding techniques, which represent a calling context using one or very few integers, have been proposed to track calling contexts continuously with a very low overhead [9], [42], [51]. We use the approach called *probabilistic calling context* (PCC) encoding [9], for it does not need static analysis and encodes each calling context into only one word. It uses a very simple hash scheme to update the calling context encoding value right before each call site: $V \leftarrow 3 \times V + cs$, where V is a thread-local integer variable storing the current calling context encoding value and cs is a hash value of the call site, which is calculated at compilation time based on the file name and line number.

Figure 5 shows an example of PCC encoding. To implement this, we write a compiler pass in LLVM that instruments the program code. The original work of PCC, which works with Java programs, shows a high efficiency. It is confirmed by our implementation for C and C++ programs, incurring only 1.9% average slowdown on SPEC CPU2006 benchmarks.

Due to the hash nature of PCC encoding, collisions may occur such that the encoding values of two different calling contexts are the same. Thus a patch might lead to unnecessary protection applied to safe buffers, resulting in some overhead. However, it has been shown in theory and practice that PCC can encode millions of contexts in a program with very few hash collisions [9].

In HeapTherapy, the encoding value of a calling context is called a *calling-context identifier*, or CCID; and hence the encoding value of a vulnerable calling context is called a vulnerable CCID (VCCID). Since each calling context is encoded into a single integer, the comparison operation of a pair of calling contexts is transformed to an integer comparison. By storing all the VCCIDs into a hash table, a buffer being allocated can be identified as vulnerable or not in $O(1)$ time.

The calling-context sensitive defense has two major benefits. First, *the VCCID can be used to precisely guide security enhancement to vulnerable objects during program execution.* This avoids a global enhancement and minimizes the perfor-

Program	Unique allocation CCIDs
vsftpd	176
Nginx	1361
MySQL	8180

TABLE I: Unique allocation calling contexts of three service programs in one execution. The inputs used to obtain this result are described in Section V-B. Note that the total number of unique allocation calling contexts in these programs should be much larger, and can be approximated using a test suite with high code coverage.

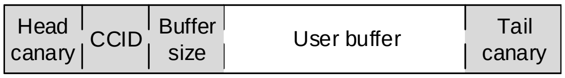


Fig. 6: Buffer structure I: a buffer structure for over-write detection.

mance overhead. To further understand this benefit, we create a profiler to collect unique allocation calling contexts of three service programs during normal execution, and display the result in Table I. Given a vulnerability, typically only one of them is vulnerable, and the calling-context sensitive defense will apply enhancements to buffers with that vulnerable calling context only.

The second benefit is that *the calling context information captures certain semantic invariants of a vulnerability.* Given a vulnerability, no matter how a piece of attack code is obfuscated, the corresponding calling context information can be used to defend against polymorphic attacks.

Next, we explain how HeapTherapy detects and diagnoses buffer over-write and over-read attacks.

B. Buffer Over-write Detection and Diagnosis

1) Buffer Structure

To detect buffer over-write, HeapTherapy interposes the allocation functions of the underlying memory allocator to surround each buffer that has not been associated with an overflow vulnerability with a head canary and a tail canary, and fills the allocation CCID and buffer size after the head canary. The buffer size indicates the number of bytes in a user buffer. A buffer size is always a multiple of the word size, so we borrow the last two bits of the *buffer size* field to indicate the buffer type; other types of buffers are introduced later. Figure 6 shows the layout of such a buffer.

2) Detection and Diagnosis

Before a buffer is freed, HeapTherapy checks the tail canary of the buffer and terminates the program if the canary is corrupted. An attack may exploit an over-write vulnerability to hijack the control flow, which upon success can evade the pre-deallocation checking. However, due to ASLR such an attack will trigger a segmentation fault signal highly probably, which is also considered as a successful detection. In both cases, a core dump file is generated at the time of detection. Then our diagnosis engine scans the core dump for a buffer with an intact head canary but a corrupted tail canary. This buffer is the origin of this over-write attack, and its CCID value will be identified as VCCID. In rare cases, multiple over-write vulnerabilities exist and are exploited simultaneously. In such cases, the same detection and diagnosis procedure is applied to each of them. A patch is then generated based on the VCCID to defeat attacks exploiting the same over-write vulnerability, which is discussed later.

3) Discussion

Although canaries have proven effective in practice, this approach has several limitations. First, there have been attacks revealing canaries, for example, the format string attack and attacks based on repetitive probings that guess the canary value byte by byte. The format string bugs have been largely reduced recently. Plus, each canary in our system is an `XOR` of the canary address and a value randomly assigned at the program start; thus, given a revealed canary, it is still difficult to guess the canary of another buffer. A single probing attack has a very low probability to succeed, while HeapTherapy generates a defense once a single probing is detected.

Second, HeapTherapy checks canaries only when a buffer is deallocated. An advanced attack may have hijacked the control flow before the canary checking is conducted. However, due to the wide deployment of ASLR, the attacker usually needs a large number of tries before a successful control flow hijack. Other attacks that increase the chance of bypassing ASLR exist, for example, heap spraying attacks. With the improvement of ASLR itself as well as other defenses, such as Data Execution Prevention and Nozzle [35], it is increasingly difficult to achieve control hijacking with a single over-write attempt. Again, HeapTherapy reacts upon a single failed attempt by generating a patch to defeat further attacks. In this sense, HeapTherapy complements ASLR for enhancing heap security.

C. Buffer Over-read Detection and Diagnosis

Since a buffer over-read does not corrupt canaries, checking canaries cannot be used to detect it. Other techniques, such bounds checking, incur a very high overhead. We propose to place guard pages probabilistically between heap buffers to detect heap over-read attacks.

1) Buffer Structure

A *guard page* is a memory page set as inaccessible using, for example, `mprotect` in Linux. A naive over-read detection approach, as in Electric Fence [32], is to append a guard page to every buffer, then any over-read will touch a guard page and trigger a segmentation fault. However, this would incur a prohibitively high performance overhead. HeapTherapy attaches a guard after a buffer with a monitoring probability P_m, which is a small value such as 0.01. A higher P_m offers a higher chance to detect any single buffer over-reads, but also incurs a higher overhead. The user can determine this probability according to its own preference of the trade-off between security and performance.

Figure 7 shows the structure of a monitored buffer. The user buffer is placed with its end aligned with the page boundary followed by a guard page, which contains a magic word and the allocation CCID for diagnosis and patch generation purposes.

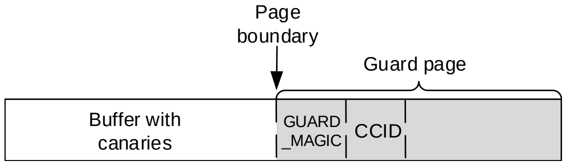


Fig. 7: Buffer structure II: a buffer structure for over-read detection.

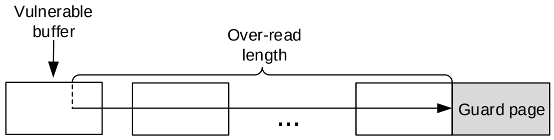


Fig. 8: A buffer over-read that spans multiple buffers and touches a guard page.

2) Detection and Diagnosis

Once an over-read touches a guard page, a segmentation fault is triggered and the signal handler installed by HeapTherapy is invoked. The magic word helps the handler determine whether the signal was due to an access to a guard page or not. The handler then terminates the program and produces a core dump for diagnosis. In Linux, we can determine the segmentation fault was due to a read or write based on the information saved in the `context` variable passed to the signal handler. Thus, this works as another approach to detecting over-write attacks.

Given a single over-read attack, the actual detection probability P_d may be larger than P_m, because an over-read might span several adjacent buffers. We evaluate the actual detection probability P_d in Section V-A1. Large volumes of attacks is a common exploitation of an over-read bug for information theft. Due to the random distribution of guard pages, n repetitive over-read attacks is detected at a probability $1-(1-P_d)^n$; that is, the detection probability increases quickly when n grows.

It is challenging to locate the vulnerable buffer directly by scanning the core dump. As shown in Figure 8, the over-read might span several buffers, so the buffer right before the touched guard page is actually not vulnerable. HeapTherapy uses a *two-stage identification method* to find the vulnerable buffer. In the first stage, the diagnosis engine searches back from the starting address of the guard page to the ending address of a previous guard page or inaccessible area. Since an overflow due to continuous read cannot pass a guard page or inaccessible area, the vulnerable buffer must lie between these two addresses, and all buffers between these two addresses are suspect buffers.

The CCID of each suspect buffer is used to generate a temporary patch. These temporary patches will guide HeapTherapy to append a guard page to every buffer with a suspect CCID in the next run, so that the same attack will touch the guard page appended to a vulnerable buffer. In the second stage, the diagnosis engine retrieves the CCID from the guard page and removes temporary patches due to other suspect CCIDs.

3) Discussion

The detection and diagnosis has some disadvantages. First, if the attacker finishes the exploitation with one or very few attack without being detected, the approach fails. However, in practice the attacker usually launches a large number of attacks to get enough chunks until target information is obtained. For example, in Heartbleed exploitation millions of attacks are launched to steal critical information [13].

Second, extra performance overhead can be incurred due to protection applied to suspect buffers. The period of the stage can be very short, as long as repetitive attacks are launched in a short time. Moreover, our evaluation in Section V-A shows that when $P_m = 0.01$ the average number of suspect CCIDs is 28.74, which is small compared with the total number of allocation CCIDs (Table I), and this number reduces to 3.7 when P_m increases to 0.04.

4) Buffer Release

When a `free` is invoked, HeapTherapy first turns the appended guard page, if it exists, to be accessible and resets the magic word and canaries, and then deallocates the buffer using the underlying memory allocator's original `free` function. To locate the guard page, the *buffer size* field at the head of the buffer is used.

It is worth mentioning that our implementation does not depend on specific allocation algorithms. *The memory allocator of HeapTherapy is implemented as a wrapper of the underlying allocator*, which hooks the buffer allocation and deallocation requests. The additional buffer size field in the buffer structure is critical for our implementation to keep independent from the underlying allocation algorithm.

D. Defenses

Given a VCCID, the diagnosis engine generates a temporary patch as defined below:

Definition 4.1: **Temporary Patch.** A temporary patch is a tuple of integers $\langle VCCID, T, L, G\rangle$, where $T \in \{OVERREAD, OVERWRITE\}$ indicates the bug type, $L \geq 0$ is the number of bytes used as padding, and $G \in \{YES, NO\}$ indicates whether a guard page is needed.

The use of padding and the case of avoiding guard pages safely are explained later. Temporary patches for a program are stored in a configuration file. When the program starts, HeapTherapy loads each patch into a hash table with its VCCID used as the key. During program execution, HeapTherapy interposes all memory allocation functions including `malloc`, `calloc`, `realloc`, and `memalign`.

We use `malloc` as an example to demonstrate how HeapTherapy handles an allocation request. As illustrated in Figure 9, it searches the current CCID in the hash table. It it does not match any installed patch, a structure I or II buffer is allocated to detect over-write and over-read attacks. Otherwise, a structure III buffer is allocated to shield vulnerable buffers from attacks.

Figure 10 shows the structure of such a shielded buffer, which is constructed according to some patch $\langle VCCID, T, L, G\rangle$. If $G = YES$ and $L = 0$, a guard page is appended to the user buffer directly, so that whenever an over-read or over-write occurs, the guard page is touched, which

```
void* malloc(size_t size) {
  int t = V; // Read the current CCID
  Patch *p = hashtable.search(t);
  if(p == NULL) {
    if(rand() > Pm) {
      // For over-write detection
      allocate a structure I buffer;
    } else
     // For over-read detection
     allocate a structure II buffer;
    }
  } else {
    // Shield the buffer according to p
    allocate a structure III buffer;
  }
  return buffer address;
}
```

Fig. 9: The pseudo code of the `malloc` function.

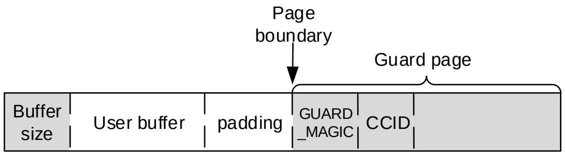


Fig. 10: Buffer structure III: a shielded buffer.

triggers a segmentation fault signal automatically to prevent data corruption, control hijacking and information leak. This protection is similar to that provided by those memory-safe languages, such as Java and C#, which throw an exception and terminate program execution whenever a buffer overflow occurs. It is also the best protection that can be provided by most countermeasures against buffer overflows.

We consider some other enhancement which potentially provides better protection. A desirable protection is that, under attacks, the program execution continues safely without being exploited. While protecting the continuity of program execution is not the focus of this work, as a preliminary step towards this goal, we explore the application of *padding* to buffer allocation in order to mitigate program termination.

Padding is a straightforward and commonly used idea, but it will be very expensive if it is applied to every buffer allocation. A unique advantage of HeapTherapy is that it identifies vulnerable buffers, which are usually a small portion of the whole set of buffers; therefore, expensive enhancement, such as guard pages and padding, can be applied to vulnerable buffers only without incurring a high overall overhead.

Currently HeapTherapy infers the size of padding based on some simple heuristics. Given an over-write attack, it finds out the length of overflow based on corrupted canaries. Given an over-read attack, it combines the information collected in the two stages to infer the length. In both cases, it is possible that the padding is not large enough to contain the overflow, and the guard page will still be touched. In that case, HeapTherapy will increase the padding size based on predefined policies. In our current implementation, HeapTherapy doubles the padding size. The adaptive padding growth works well in cases when the overflow length is limited due to the program interface or logic. Web servers, for example, typically set the limit on length for URLs up to 4096 characters [3]. For a DNS serve, the full domain name may not exceed the length of 253 characters [1]. Due to the 16-bit `size` field in a Heartbleed request, the maximum length of read in a Heartbleed response is 64KB. If in a patch $T = OVERREAD$, to avoid information leakage the padding is zeroed when a buffer is allocated. Finally, if a thorough analysis shows that some reasonable size of padding is large enough to contain all overflows exploiting a given vulnerability, the guard page is not needed in this case.

In the cases where the overflow length is unlimited or very large, in order to avoid exhausting memory, HeapTherapy enforces an upper bound for padding defined by the user. Some more advanced protection can be applied conveniently when the padding is not large enough and the guard page is touched. For example, *failure-oblivious computing* omits the overrun operations and continues [36]; the *reactive immune system* returns an error code for the current function invocation that leads to an overflow [41]. Both techniques incur a high overhead mainly due to expensive methods identifying such attack operations as overflows, while HeapTherapy provides an efficient and automatic way to capture overflows. Therefore, it will be an interesting research topic by combing these ideas with HeapTherapy to support program execution continuity.

E. Additional Features

1) Instant patch generation

Conventional patch generation is a long process. The user needs to provide the input that reproduces the problem to the software company, and then waits until the software gets back with the patch, which usually takes more than one month [43]. HeapTherapy can be used as an offline tool to quickly generate temporary patches once an input reproducing the overflow attack is available. The user can simply set P_m to be 1 and then run the program with the input. In this situation, HeapTherapy accurately locates the vulnerable buffer, which is right before the touched guard page, and generates the temporary patch.

2) Collaborative patching

A temporary patch generated at one site can be shared with other machines running the same vulnerable program. Therefore, the effort of detection and patch generation can be aggregated across machines to handle large scale zero-day attacks. Once a machine detects and generates a patch, the patch can be distributed to protect other machines. So that a large scale attack can be defeated before it plagues the Internet.

3) Patching without restart

In our current implementation the program is restarted after a patch is generated. For complicated programs, the time to restart may be long. To speed up the service recovery upon patching, the checkpointing and recovery technique can be used to resume the service from a clean state [33]. This is particularly useful for request-handling services, where most heap buffers allocations and deallocations are associated with per request. Once the patch is installed, when the attacker sends a new malicious request, shielded buffers will be allocated to prevent attacks.

Program	Vulnerability	Reference
Heartbleed	over-read	CVE-2014-0160
MySQL 5.5.19	over-write	CVE-2012-5612
Lynx 2.8.8dev.1	over-write	CVE-2010-2810
libtiff 4.02	over-write	CVE-2013-4243
SAMATE Dataset	12 heap buffer overflow cases	N/A

TABLE II: Vulnerabilities for effectiveness evaluation.

V. Evaluation

We first evaluate the effectiveness of HeapTherapy against buffer overflow attacks, and then measure the efficiency of HeapTherapy on service programs and the SPEC CPU2006 Integer benchmark suite.

A. *Effectiveness*

We evaluate the effectiveness of HeapTherapy using 4 real-world vulnerabilities and 12 test cases provided by NIST listed in Table II. The Lynx case was covered in Section III.

1) Heartbleed

Heartbleed attack. The recently exposed Heartbleed vulnerability in OpenSSL threatens millions of Internet services [17]. By sending an ill-formed heartbeat request, the attacker can over-read a buffer on the heap and steals up to 64KB data from the memory. While a Heartbleed attack is widely classified as an over-read attack, our investigation shows that the attack can actually exploit two heap-based vulnerabilities: an uninitialized read bug and an over-read bug. Specifically, the victim buffer has 34KB, while the attacker can manipulate the length l of the read over this buffer. If $l \leq$ 34KB, it is just an uninitialized read attack that leaks old data inside the buffer. Otherwise, the attack is a mix of uninitialized read and over-read. In this case study, we focus on over-read, and avoid the uninitialized read simply by zero-filling the buffer. A more systematic study of this issue can be found in [48].

Experiment setting. Nginx is the third most widely used web server. We use Nginx 1.3.9 and OpenSSL 1.0.1f to create a vulnerable HTTPS service. The program is compiled through our PCC encoding pass and linked with HeapTherapy's shared library. OpenSSL optionally uses a freelist to manage heap buffers, so that when a buffer of certain lengths is freed, it will be stored in the freelist. In order that HeapTheray interposes heap memory management, we use the flag OPENSSL_NO_BUF_FREELIST provided by OpenSSL to disable the use of the freelist. We obtain a Heartbleed attack script from [19] and set the l as 64KB, which enables maximum amount of data leakage.

Offline patch generation. We first evaluate how HeapTherapy can be used to generate a patch offline instantly, when the attack input is available. In this case P_m is set as 1 and the patch generation is run on an experimental system rather than a production system. The malicious request immediately crashes the worker process of Nginx, and a core dump is generated. Then the diagnosis engine produces a temporary patch containing the VCCID based on the core dump. Next we set the initial padding size as 4KB, and double the size whenever it is not large enough to contain the overflow attack, that is, the guard page is touched and the process crashes. It takes 1 round to obtain the VCCID and another 4 rounds to determine that the 32KB padding is large enough to prevent crashes due to the Heartbleed request. We repeat the experiment 100 times and reproduce the same result every time. Once the patch generated, it can be installed to protect the production system from Heartbleed attacks efficiently. The system is then immune from information leakage when waiting for the *official patch* that fixes the bug to be generated and installed.

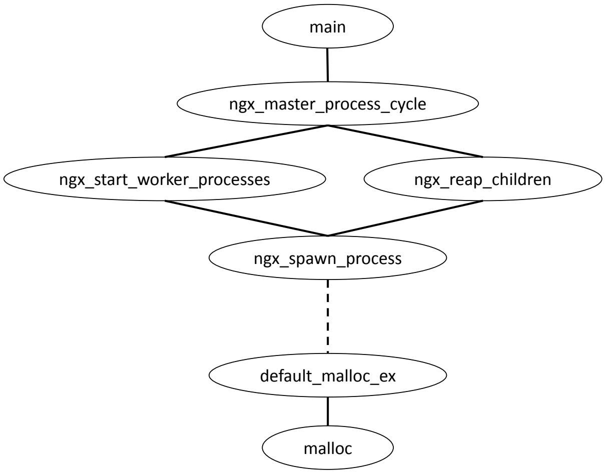


Fig. 11: Two vulnerable calling contexts identified by the diagnosis engine. Dashed lines indicate omitted functions.

The result contains the following two temporary patches:

$$< 0xE2FF92B2, OVERREAD, 32KB, YES >$$
$$< 0x6E3D3954, OVERREAD, 32KB, YES >$$

These two temporary patches indicate that there are two vulnerable calling contexts in the service. The first VCCID (0xE2FF92B2) was found in the core dump due to the first attack; while the following attacks are all related to the second VCCID (0x6E3D3954). Figure 11 shows the details of the two calling contexts. Our further investigations shows that a Nginx service, when it is started, has a master process and a worker process. The initial worker process is forked by the master process using the function ngx_start_worker_processes. Once the master process detects that a worker process has crashed, it will reap the worker process and fork a new one using ngx_reap_children. That is why all the vulnerable buffers, except for in the first attack, is related to the second VCCID. HeapTherapy handles the case of multiple VCCIDs smoothly by generating a patch for each unique VCCID.

Online protection. We next evaluate how HeapTherapy detects and defeats zero-day attacks. We set the P_m to be a small value ranging from 0.01 to 0.15. For each value of P_m, we record the following measurements: (1) the detection delay

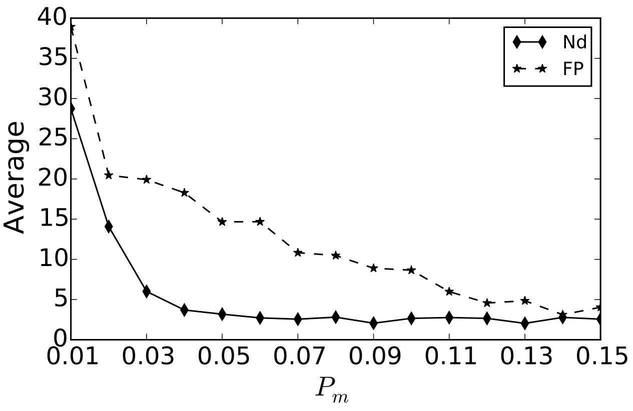


Fig. 12: Heartbleed detection and diagnosis result.

N_d, which is the number of successful over-reads before the first guard page is touched, and (2) the average number of temporary patches, FP, generated in the first stage due to the suspect buffers, as discussed in Section IV-C.

We conduct 100 experiments for each value of P_m and obtain the average values of N_d and FP, as shown in Figure 12. In all experiments, HeapTherapy successfully detects the attack. When $P_m = 0.01$, N_d averages 28.74. As P_m increases to 0.04, N_d quickly drops to 3.7, and it does not change much when P_m further increases. This shows that for this particular scenario by randomly monitoring a small set (4%) of buffers, HeapTherapy can quickly detect a zero-day Heartbleed attack. Since attackers usually need to send many Heartbleed requests before achieving their goals, e.g. $100k \sim 2.5M$ attacks for obtaining the SSL private key [13], the short detection delay allows HeapTherapy to detect and respond to the attacks before important information is leaked. In addition, N_d in all cases is much smaller than the expected value $1/P_m$, because an Heartbleed attack may access multiple buffers.

We next analyze the number of temporary patches generated at the first stage of detection and diagnosis. When $P_m = 0.01$, FP is 39; and it decreases to around 5 when P_m increases to 0.12. Compared with the total number of unique allocation calling contexts shown in Table I and Table III below, FP is very small. In addition, all of these temporary patches except one will be removed after the second stage of detection and diagnosis. We will further evaluate the overhead due to temporary patches in Section V-B and Section V-C.

Defense. After installing the two patches, the Heartbleed attack can only read zeros in the padding space, which is 32KB long and large enough to contain the over-read. We sends 10,000 Heartbleed requests to the patched `Nginx` service, the result shows that HeapTherapy successfully prevented information leakage in all attempts *without crashing the service*.

We then tried another three different Heartbleed attack scripts collected from the Internet, and launched all the attacks against `proftpd` and `MySQL` in addition to `Nginx`. In all the cases HeapTherapy detects the attack and prevents further attacks after generating and installing the patches.

2) MySQL

`MySQL 5.5.19` contains a heap buffer over-write vulnerability that allows a remote attacker to launch denial of service attacks using crafted database commands to over-write a heap buffer and corrupt the heap meta data. The heap corruption leads to a segmentation fault and crashes the `MySQL` service when the connection is closed. We applied HeapTherapy to the service and ran the attack script [18] 20 times against it. In the first run, HeapTherapy detects the attack when the segmentation fault is triggered. The diagnosis engine retrieves the VCCID of the vulnerable buffer from the core dump file, and generates a temporary patch, which is then installed to the service. In all the following runs, HeapTherapy successfully prevents data corruption from occurring.

3) Libtiff

`Libtiff` is a popular library for processing TIFF images. A heap buffer overflow vulnerability was found in the `gif2tiff` tool in `libtiff 3.4-4.03`. By manipulating height and width of a GIF image, a remote attacker can exploit this vulnerability to overwrite a heap buffer. We first reproduce the exploitation using an attacking GIF image input [10]. Upon detection, the diagnosis engine automatically finds the VCCID in the core dump. When we use `gif2tiff` to open the crafted image again, the tool is able to avoid crash and reports "`illegal GIF block type`".

4) NIST SAMATE reference dataset

The SAMATE dataset [29] maintained by NIST contains 12 programs with heap buffer overflow vulnerabilities caused by contiguous writes through, for example, assignments, `memcpy`, `strcpy`, and `snprintf`.

In all these cases, HeapTherapy retrieves the vulnerable calling contexts accurately and uses padding to prevent both data corruption and program crashes.

B. Efficiency on service programs

We evaluate the overhead due to HeapTherapy using three real world service programs, `Nginx`, `proftpd` and `MySQL`. We simulate 10 VCCIDs using the method described in Section V-C, and set P_m as 0.05. To measure the service throughput, we use `ApacheBench` to send web requests to `Nginx`, and use the official test script for `MySQL`; For `proftpd`, we write a script that generates concurrent clients to upload files, and measures the throughput.

The result shows that HeapTherapy has a low overhead for these service programs. The overhead on `Nginx`, `proftpd` and `MySQL` is 7.6%, 4.7%, and 6.0%, respectively.

C. Efficiency on SPEC CPU2006

1) Methodology

Benchmarks and platform. We then measure the efficiency of HeapTherapy on SPEC CPU2006 Integer benchmark suite with respect to speed and memory. All results presented are normalized by the execution time of the original benchmark programs without HeapTherapy. We also measure the memory overhead in terms of the average Resident Set Size (RSS) for all benchmark programs. We write a script to read the `VmRSS` value of `/proc/[pid]/status` 50 times per second and then calculate the average.

Benchmark	Buffer allocation count	Unique allocation-time CCID count
400.perlbench	**360,728,131**	**13,909**
401.bzip2	168	10
403.gcc	**28,458,470**	**913,747**
429.mcf	5	5
445.gobmk	658,034	5,404
456.hmmer	2,474,268	191
458.sjeng	5	5
462.libquantum	179	10
464.h264ref	177,779	258
471.omnetpp	**267,064,936**	**162,332,040**
473.astar	4,799,955	184
483.xalancbmk	**135,155,557**	**131,848,405**

TABLE III: Buffer allocation and CCID profiling. Benchmark programs in bold text are allocation intensive.

The experiments are performed on a Dell Precision Workstation T5500 with 2.26GHz Intel Xeon E5507 processor and 16GB RAM. The operating system is Ubuntu 12.04 with Linux kernel 3.2.0.

Vulnerability simulation. To simulate the performance of HeapTherapy when handling multiple vulnerabilities, we create several sets of simulated VCCIDs for each benchmark program. Specifically, we first develop a profiler to count the number of buffer allocations, the number of unique allocation-time CCID values and obtain the number of buffer allocations associated with each unique CCID. Table III lists our profiling result. From the table we can see that these benchmark programs have a diverse profile of buffer allocation. Programs, such as `perlbench`, `gcc`, `omnetpp` and `xalancbmk`, have intensive memory allocations.

We then generate several sets of simulated VCCIDs for later experiments. To pick the VCCIDs fairly, for each benchmark program we sort the CCIDs according to their allocation counts in descending order. Next, for each benchmark program we generate 3 sets of VCCIDs containing 1, 5 and 10 elements, respectively. For the 1-VCCID set, the median CCID is selected. For the 5-VCCID set, CCIDs at 0.01%, 20%, 40%, 60% and 80% points are chosen. The 10-VCCID set is created in a similar fashion with an interval of 10%. Since benchmark programs `mcf` and `sjeng` have less than 10 unique CCIDs, we use all of their CCIDs for their 10-VCCID sets.

2) Overhead due to PCC Encoding

Our first evaluation was focused on the speed and memory overhead due to PCC encoding alone. All benchmark programs are instrumented with PCC encoding but not linked with HeapTherapy, so no detection overhead is incurred. The average speed and memory overhead is 1.9% and 0.2%, respectively. The low overhead is consistent with the result of the original PCC encoding on Java programs [9].

3) HeapTherapy Efficiency

Next, we evaluate the speed and memory overhead due to HeapTherapy using the 3 sets of VCCIDs. For each VCCID, a patch is created manually. Over-read detection is not turned on because these benchmark programs are not service programs and the concern of information leakage is rare. With regard to speed overhead, the result in Figure 13 shows that HeapTherapy caused 4.3% average overhead in the case of 0 patch. This shows the overhead when HeapTherapy works in the detection status with no patch installed. The average overhead increases slightly to 6.2% when the 10-VCCID set and 5-page padding are used, which demonstrates the high efficiency of HeapTherapy even when multiple vulnerabilities are handled simultaneously.

When no patch is applied, the average memory overhead incurred by HeapTherapy is 5.9%. The overhead in the case of 10 patches and 5 padding pages increases to 7.7%. For majority of the benchmark programs, the memory overhead changes little when the size of padding increases, mainly because a physical page is not mapped until the corresponding virtual memory region is accessed.

We also compare HeapTherapy with DieHarder, a memory allocator against heap-based attacks [30]. Figure 13 shows that DieHarder incurs a much higher speed overhead for allocation-intensive benchmark programs. The average speed performance penalty due to DieHarder is 20.3%, which is also much higher than HeapTherapy. For allocation-intensive programs, the average overhead due to DieHarder is 86.1% while HeapTherapy only incurs 15.8% overhead on average. In addition, DieHarder only provides probabilistic defense against known vulnerabilities, while HeapTherapy provides deterministic protection in such cases. Other comprehensive defense methods, such as AddressSanitizer [39], provides a wider range of protection than HeapTherapy. However, their high overheads (e.g., 73% speed overhead and 337% memory overhead on SPEC 2006 for AddressSanitizer) making them more suitable for offline testing.

In summary, we conclude that HeapTherapy incurs a low overhead in terms of both speed and physical memory.

VI. Limitations

First, it is possible that an overflow vulnerability can be exploited in different VCCIDs, and the attacker may invest to develop different attack input to overflow buffers in new allocation calling contexts. However, as we have shown in the `Nginx` case, whenever the attack overflows a buffer allocated in a new calling context, HeapTherapy simply treats it as a new vulnerability and starts another defense and diagnosis cycle. Moreover, based on our evaluation and observations in previous research [50], [26], [31] on the high reproducibility of calling contexts, this is not common.

The second limitation is that HeapTherapy can only deal with buffer overflow due to continual write or read operations, which are the main form of buffer overflows. It cannot handle overflows due to discrete read or write. Also, if an overflow occurs inside a data structure, HeapTherapy cannot detect it. This limitation is common in many existing countermeasures against buffer overflows.

Third, although, with the current recompilation based implementation, users of open-source software or software companies can recompile their programs with HeapTherapy to protect heap memory security, binary programs cannot be protected conveniently. However, recompilation is not an

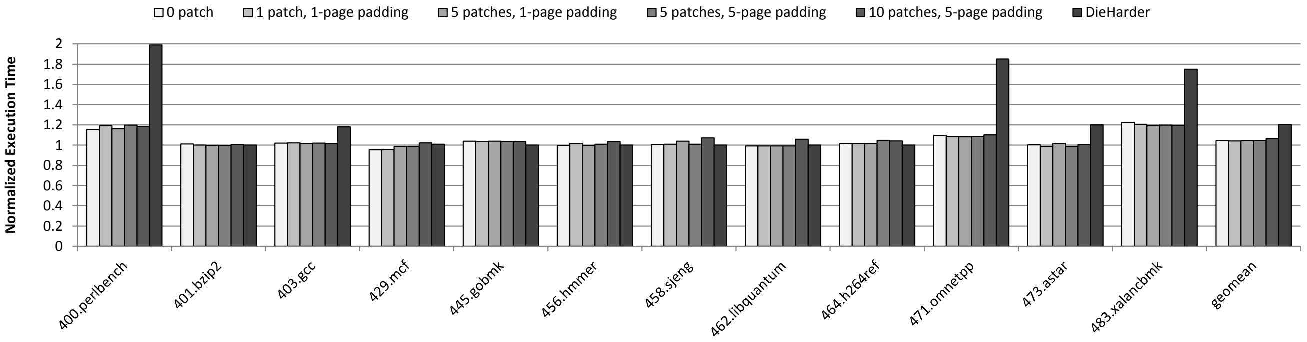


Fig. 13: Speed overhead due to HeapTherapy. Each bar represents one experimental setting. For example, "5 patches, 5-page padding" represents the setting where 5 simulated patches are applied based on the 5-VCCID set and each vulnerable buffer is padded with 5 pages. The guard page is enabled for all vulnerable buffers. The last bar of each group is based on the related work DieHarder.

inherent limitation of the technique covered in the paper. A binary-instrumentation based implementation is possible. We are building tools to build PCC encoding into programs through binary instrumentation, while the shared library of HeapTherapy can be loaded through LD_PRELOAD.

VII. Other Potential Applications

While the focus of this paper is the heap buffer overflow problem, HeapTherapy can generate patches to deal with many other memory errors, such as double frees, dangling pointers and uninitialized heap buffer read. We can extend the specification of the temporary patch to support more bug types, and add bug-specific predefined handling to the memory allocation wrapper. We discuss some examples in the section.

One type of dangling pointer bugs is due to a premature deallocation of some buffers. Assume the temporary patch that fixes a dangling pointer bug is generated. We can code a simple handling, which is invoked when a `free` call is hooked: the handling identifies whether the buffer's CCID matches the VCCID in the patch; if so, the buffer's deallocation is delayed. Such that the pointer variable previously containing a dangling pointer can be dereferenced safely.

To address an uninitialized heap buffer read bug, we write a simple handling that zero-fills the newly allocated buffer. `malloc` calls the handling only when the buffer's CCID matches the VCCID of a patch that treats an uninitialized read bug. So that the effort of zero-filling is tailed to buffers where an uninitialized read may occur.

The memory management wrappers search in the hash table assembled with patches to determines what actions are needed, and then enforce them on the buffers. Therefore, HeapTherapy is extensible and new handling functions and patches can be defined to deal with new types of heap bugs.

VIII. Conclusions

We propose HeapTherapy, an end-to-end solution that performs diagnosis and generates defenses against zero-day heap buffer overflow attacks in realtime. HeapTherapy creatively employs the calling context encoding to describe and identify vulnerable buffers precisely and efficiently. It does not have false positives, and remains effective under polymorphic attacks. Our evaluation shows that it incurs a low speed and memory overhead even when dealing with multiple vulnerabilities simultaneously. It does not need infrastructure for request recording and replaying, so HeapTherapy can be used to protect personal applications as well as enterprise services conveniently. The technique can be extended to deal with other heap bugs, such as immature deallocation and uninitialized read.

Acknowledgment

The authors would like to thank the anonymous reviewers for their invaluable comments. This work was supported by ARO W911NF-09-1-0525 (MURI), NSF CNS-1223710, NSF CNS-1422594, and ARO W911NF-13-0421 (MURI).

References

[1] Domain name resolution request length limit. http://tools.ietf.org/html/rfc1034.

[2] The pax project. https://pax.grsecurity.net/.

[3] Url request length limit. http://www.checkupdown.com/status/E414.html.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353. ACM, 2005.

[5] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *IEEE Symposium on Security and Privacy, 2008.*, pages 263–277. IEEE, 2008.

[6] 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*, pages 51–66, 2009.

[7] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 158–168, 2006.

[8] E. 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, 2003.

[9] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 97–112, 2007.

[10] Bugzilla. Bug 2451 - CVE-2013-4243 libtiff (gif2tiff): possible heap-based buffer overflow in readgifimage(). http://bugzilla.maptools.org/show_bug.cgi?id=2451.

[11] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 147–160. USENIX Association, 2006.

[12] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the 11th Symposium on Computers and Communications (ISCC)*, pages 749–754. IEEE, 2006.

[13] CloudFare. The results of the cloudflare challenge. https://blog.cloudflare.com/the-results-of-the-cloudflare-challenge/.

[14] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, pages 133–147, 2005.

[15] C. Cowan and C. Pu. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, January 1998.

[16] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: a secretless framework for security through diversity. In *Proceedings of the 15th USENIX Security Symposium*, volume 6, pages 105–120, 2006.

[17] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, et al. The matter of heartbleed. In *Internet Measurement Conference (IMC)*, pages 475–488. ACM, 2014.

[18] Exploit. MySQL (Linux) Heap Based Overrun PoC Zeroday. http://www.exploit-db.com/exploits/23076/.

[19] Exploit. Openssl heartbeat poc with starttls support. https://gist.github.com/takeshixx/10107280.

[20] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 62–75, 2003.

[21] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *the International Workshop on Automatic Debugging*, pages 13–26, 1997.

[22] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th Conference on USENIX Security Symposium*, 2004.

[23] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, volume 92, 2002.

[24] Launchpad.net. Heap overflow when parsing malformed URLs. https://bugs.launchpad.net/ubuntu/+source/lynx-cur/+bug/613254.

[25] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[26] J. Newsome, D. Brumley, and D. Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the 13th Symposium on Network and Distributed System Security*, 2005.

[27] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of Network and Distributed System Security Symposium*, 2005.

[28] NIST. CVE-2014-0160. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160.

[29] NIST. SAMATE Reference Dataset. http://samate.nist.gov/SRD.

[30] G. Novark and E. D. Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 573–584. ACM, 2010.

[31] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. In *Proceedings of the Programming Language Design and Implementation*, 2007.

[32] B. Perens. efence(3) - Linux man page. http://linux.die.net/man/3/efence.

[33] D. K. Pradhan and N. Vaidya. Roll-forward checkpointing scheme: A novel fault-tolerant architecture. *IEEE Transactions on Computers*, 43(10):1163–1174, 1994.

[34] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, pages 235–248, 2005.

[35] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th USENIX Security Symposium*, pages 169–186, 2009.

[36] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, pages 303–316, 2004.

[37] W. K. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time detection of heap-based overflows. In *The 17th Large Installation Systems Administration Conference*, volume 3, pages 51–60, 2003.

[38] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 12th Symposium on Network and Distributed System Security*, pages 159–169, 2004.

[39] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Address-sanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[40] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis. A dynamic mechanism for recovering from buffer overflow attacks. In *Proceedings of the 8th International Conference on Information Security*, pages 1–15, 2005.

[41] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *Proceedings of the USENIX annual technical conference*, pages 149–161, 2005.

[42] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 525–534, 2010.

[43] Symantec. Internet security threat report. http://www.symantec.com/security_response/publications/threatreport.jsp.

[44] D. Tian, Q. Zeng, D. Wu, P. L. 0005, and C. Hu. Kruiser: Semi-synchronized non-blocking concurrent kernel heap buffer overflow monitoring. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012.

[45] US-CERT. Vulnerability ranking. http://www.kb.cert.org/vuls/bymetric/.

[46] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, pages 156–168, 2001.

[47] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *In Proceedings of Network and Distributed System Security Symposium*, pages 3–17.

[48] J. Wang, M. Zhao, Z. Qiang, D. Wu, and P. Liu. Risk assessment of buffer "heartbleed" over-read vulnerabilities (practical experience report). In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2015.

[49] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of 22nd International Symposium on Reliable Distributed Systems*, pages 260–269, 2003.

[50] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 223–234, 2005.

[51] Q. Zeng, J. Rhee, H. Zhang, N. Arora, G. Jiang, and P. Liu. DeltaPath: Precise and Scalable Calling Context Encoding. In *Symposium on Code Generation and Optimization*, 2014.

[52] 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*, pages 367–377, 2011.

[53] B. Zorn and M. Seidl. Segregating heap objects by reference behavior and lifetime. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, 1998.