Risk Assessment of Buffer “Heartbleed” Over-read Vulnerabilities

Jun Wang, Mingyi Zhao, Qiang Zeng†, Dinghao Wu, Peng Liu
The Pennsylvania State University, University Park, PA 16802, USA
{jow5222, muz127, dwu, pliu}@ist.psu.edu, †quz105@cse.psu.edu

***Abstract*—Buffer over-read vulnerabilities (*e.g.,* Heartbleed) can lead to serious information leakage and monetary lost. Most of previous approaches focus on buffer overflow (*i.e.,* over-write), which are either infeasible (*e.g.,* canary) or impractical (*e.g.,* bounds checking) in dealing with over-read vulnerabilities. As an emerging type of vulnerability, people need in-depth understanding of buffer over-read: the vulnerability, the security risk and the defense methods.**

This paper presents a systematic methodology to evaluate the potential risks of *unknown* buffer over-read vulnerabilities. Specifically, we model the buffer over-read vulnerabilities and focus on the quantification of how much information can be potentially leaked. We perform risk assessment using the RUBiS benchmark which is an auction site prototype modeled after eBay.com. We evaluate the effectiveness and performance of a few mitigation techniques and conduct a quantitative risk measurement study. We find that even simple techniques can achieve significant reduction on information leakage against over-read with reasonable performance penalty. We summarize our experience learned from the study, hoping to facilitate further studies on the over-read vulnerability.

I. Introduction

The buffer over-read vulnerability [1] has gained much attention after the Heartbleed [2] bug was discovered, which threatens millions of Web services on the Internet [3]. A buffer over-read happens when a program overruns a buffer's boundary and reads the adjacent memory. It is similar to buffer overflow which is an over-write, but had been paid less attention than over-write as it usually does not lead to memory taint or control hijack. However, sensitive and security related information such as passwords and keys can be leaked through buffer over-read.

The root cause of buffer over-read is similar to that of buffer overflow. It is typically resulted from insufficient or a lack of bound checking for buffer read (write for overflow). Extensive research has been devoted to this issue related to buffer overflow and practical tools exist for partial mitigation. For example, StackGuard [4] injects a canary right after the return address to detect stack-based buffer overflow before a function returns. Address Space Layout Randomization (ASLR) [5], [6] makes the addresses of the target code such as standard libc less predictable and thus increases the difficulty of control transfer hijacking when a buffer is overflowed. These techniques have been implemented in many compilers and systems, but they are not applicable to the over-read issue. There is also a large volume of research on direct bound checking (*e.g.,* [7], [8], [9]) and safe type system retrofitting (*e.g.,* [10], [11]). Bounds checking and safe type system retrofitting can mitigate the over-read issue, but few is widely adopted due to either excessive runtime overhead, expensive cost of manual work, or insufficient mitigation in practice.

A number of programming techniques for writing solid and secure code have been around to preserve memory safety. Although some of them are initially intended to make the resulted software more reliable, they are helpful in mitigating the buffer over-read vulnerability. Zero out memory blocks or buffers have been advocated and implemented in many systems. In particular, Chow et al. [12] measured the performance overhead on zero out heap buffers and stack frames at deallocation time. They also experimented on zeroing out currently unused stack part periodically. Initializing and padding buffers with special characters are proposed to facilitate easier debugging and fault localization. Maguire [13] uses “0xA3” and “0xCC” for padding and initialization of buffers for Macintosh models and Microsoft applications, respectively, due to a number of reliability and debugging benefits.

These techniques are useful in mitigating buffer over-read vulnerabilities or information leak in general; there is, however, no quantitative study on their effectiveness with regard to information leak through buffer over-reads. Some of these techniques are known in the field, but were not reported with any quantitative measures before to the best of our knowledge. Moreover, one could view “Heartbleed” as a family of buffer over-read vulnerabilities and *there could be unknown zero-day vulnerabilities exploitable by unknown “Heartbleed” attacks*. It is thus too restricted and not objective to evaluate the entire risk of buffer over-read vulnerabilities solely based on the specific Heartbleed bug. As an emerging type of vulnerability, we need more research on the mitigation of the buffer over-read vulnerability as well as the quantitative risk assessment of the software systems deployed in the field.

In this paper, we intend to fill in the gap by providing a preliminary quantitative risk measurement on information leak associated with buffer over-read for some popular mitigation methods such as zeroing and padding buffers. We explore and implement both in-line and concurrent versions of these methods. We apply these techniques to the RUBiS benchmark [14], and report our experience on four metrics we developed for measuring information leak risks.

In summary, our main contributions are:

- We report the first experience on quantitative risk measurement of information leak through buffer over-read. The methodology can also be applied to other programs for a quick risk assessment.
- We provide a summary of the current mitigation techniques that are applicable to buffer over-read and measure their effectiveness.

- The preliminary quantitative risk measurement and experience we reported can facilitate further studies on the issue.

The remainder of the paper is organized as follows. In Section II, we introduce background, threat models, motivation, and examples. In Section III, we describe the details of our methods and present the metrics on information leak risk measurement. We validate our methods with a few case studies in Section IV and present the experimental results in Section V. We discuss some other issues and limitations in Section VI. We then present the related work in Section VII and conclusion in Section VIII.

II. Overview

A. Background

Although heap buffer overflow has been extensively researched for many years, heap buffer over-read remains understudied and under-reported [15]. Our focus is the risk assessment of heap buffer over-read vulnerabilities. Conceptually, a buffer over-read attack involves a source buffer, a destination buffer, and the vulnerable operation(s) that are possibly resulted from a bug in the program. Therefore, the victim of over-read is the source buffer rather than the destination buffer, whereas the victim is typically the destination buffer in a buffer overflow attack.

We classify heap buffer over-read into three categories.

(1) **`memcpy`-based**. This is a major type of buffer over-read vulnerabilities and possibly the most damage-causing (due to Heartbleed) one. This happens when the `size` argument of a `memcpy` function is accidentally or maliciously enlarged so that more information than necessary or allowed is copied to the destination. As this is the main focus of this paper, further details will be introduced shortly in the next section. Real world examples of this type of vulnerability include CVE-2014-0160 (*i.e.,* Heartbleed) and CVE-2009-2523.

(2) **`strcpy`-based**. This is another common vulnerability in which unintended extra information is copied out (*e.g.,* CVE-2009-2523). Different from `memcpy`-based over-read, `strcpy`-based over-read is mostly due to the improper null termination of the source string. Although there have been a handful of studies on `strcpy`-based over-read in terms of safe function alternatives [16], vulnerability detection [17], and writing secure program in general [18], a real world vulnerability leading to large-scale information leak does not exist yet and thus the corresponding thread model is still unclear. Therefore, in this report, we exclude this category from consideration.

(3) **Byte-level over-read and others**. In addition to the category (1) and (2) that are "chunk-level" over-read, byte-level over-read vulnerabilities are also witnessed by the real world (*e.g.,* CVE-2004-0112, CVE-2004-0184). Instead of reading out a chunk of memory as a whole piece, byte-level over-reads usually access one element (of an array or a table) at a time or loop over a range of elements. Insufficient or even lack of bounds checking is always the root cause. Using an index that exceeds the lower limit and the upper limit will result in under-read and over-read, respectively. However, only given the program binary, it is highly non-trivial to identity the set of instructions with potential risks of being affected by such over-read. Even with the availability of source code, it is missing from existing literature on how to identify the potential culprits. More importantly, the amount of information leakage caused by byte-level over-read is usually minimal compared to category (1) and (2), so we decide to also omit this category in our risk assessment.

Therefore, this paper only focuses on a particular type of heap buffer over-read: the `memcpy`-based buffer over-read. Although the scope is limited to some extent, we argue that this is the most attainable way to do risk assessment for heap buffer over-read vulnerabilities at the time of this writing, which we think is still in the early stage of buffer over-read research.

B. Threat Model

We now explain the details of `memcpy`-based buffer over-read vulnerabilities. We will briefly introduce the Heartbleed vulnerability and use it as the illustrative example. For more details of the Heartbleed vulnerability, please refer to [2].

```
2580  buffer = OPENSSL_malloc(1 + 2 + payload +
          padding);
2581  bp = buffer;
2582
2583  /*Enter response type, length and copy payload*/
2584  *bp++ = TLS1_HB_RESPONSE;
2585  s2n(payload, bp);
2586  memcpy(bp, pl, payload);
2587  bp += payload;
2588  /* Random padding */
2589  RAND_pseudo_bytes(bp, padding);
2590
2591  r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT,
          buffer, 3 + payload + padding);
```

Fig. 1: A code snippet related to the Heartbleed vulnerability (from t1_lib.c in openssl-1.0.1f).

OpenSSL implements the TLS Heartbeat Extension [19], which allows the client to send a heartbeat request to the server, who will then reply the same payload back to keep the connection alive. Fig. 1 shows a code snippet of `openssl-1.0.1f` for constructing heartbeat response message. In line 2586, `memcpy` function copies data of `payload` size from the source buffer `pl`, which contains the heartbeat request message, to the destination buffer `bp`. However, since both the `payload` variable and the `pl` buffer are controlled by the attacker, and there is no check to make sure that the actual length of the `pl` buffer is equal to `payload`, the attacker can provide a very short heartbeat request message while setting the `payload` variable to a large value up to 2^{16}. Thus, the `memcpy` function will read beyond the boundary and copy sensitive data to the `bp` buffer, which is eventually sent to the attacker (line 2591).

Fig. 2 further illustrates the information leak caused by the `memcpy`-based over-read. There are actually two types of information leak. The first type is to steal information previously used and left in the heap memory. For example, if the vulnerable buffer `pl` has been used to handle a previous user's request, and the data in the buffer is not cleared or initialized (*e.g.,* the uninitialized area in Fig. 2), then `memcpy` will copy such data to the intermediate buffer which will be then sent to the attacker. Note that existing approaches on bounds checking [7], [8], [9] does not really handle this type

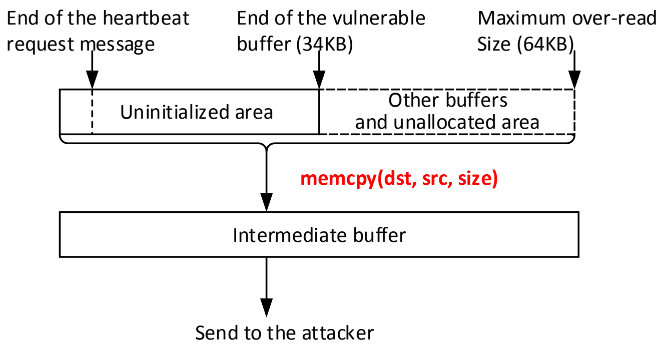

End of the heartbeat request message
End of the vulnerable buffer (34KB)
Maximum over-read Size (64KB)
Uninitialized area
Other buffers and unallocated area
memcpy(dst, src, size)
Intermediate buffer
Send to the attacker

Fig. 2: Illustration of heap buffer over-read in the Heartbleed vulnerability.

of information leak, because they only check memory access at the end of the buffer. The second type is to steal information that are stored out of the current buffer. More specifically, if the over-read reaches a buffer that is located after the vulnerable buffer and the buffer is currently filled with sensitive data, then those data will be leaked also. Since the attacker utilizes both types of leakage to steal sensitive information, we will not separate them in later discussions.

C. Motivation and Challenges

To evaluate the risks of heap buffer over-read vulnerabilities associated with `memcpy`, our initial attempt was to capture the instances of `memcpy` that are exploitable by the attacker. If we can achieve this, the risk assessment will be accurate. In practice, however, we found it is extremely hard to do so. The main challenge is that there lacks a reliable method that can accurately pinpoint those vulnerable `memcpy`s.

Although a few commercial tools, such as Coverity [20] and CodeSonar [21], can now detect the Heartbleed bug, none of them did it before the bug was disclosed. Otherwise the Heartbleed bug might not have stayed there for that long, namely around two years. In this sense, it is reasonable to suspect that there could be more complicated and obscure zero-day buffer over-read vulnerabilities associated with some other `memcpy`s that are unknown. Thus, only using the Heartbleed vulnerability to assess the entire risk of buffer over-read vulnerabilities of a program is neither objective nor representative. If an attacker exploits a different bug to do buffer over-read, what the attacker can get might be very different from what he can get using Heartbleed. Moreover, it is fundamentally difficult to verify a `memcpy` is never exploitable; and it is also hard to predict how soon an exploit will happen. Therefore, our intention is to come up with a generic methodology for buffer over-read vulnerability risk assessment before the next "Heartbleed" arrives. As the result, we choose a conservative principle for our risk assessment, that is, *every* `memcpy` *could be abused by the attacker*.

D. First Glance

To see how different it is to look at all `memcpy`s instead of only considering the Heartbleed bug in regard of evaluating the risk, we first try to get some early sense. Since Apache and Nginx are the two major targets of Hearbleed attacks [3], we perform a preliminary analysis on the two Web servers.

TABLE I: Statistics of OpenSSL, Apache, and Nginx

Program	# memcpy	LOC
openssl-1.0.1f	184	50K
httpd-2.2.14	380	283K
nginx-1.3.9	156	122K

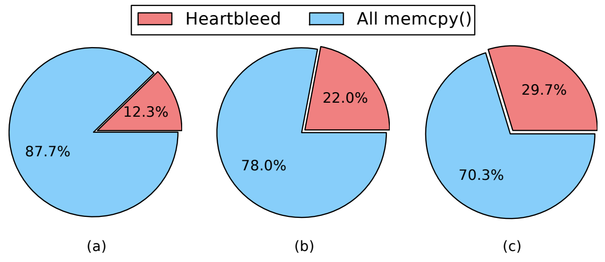


Fig. 3: A rough estimate on the ratio of data leakage of Heartbleed and all `memcpy` when Heartbleed attack happens (a) one, (b) two, and (c) three times during a one-minute user session

First, we count the number of `memcpy`s as well as lines of code in OpenSSL, Apache and Nginx, shown in Table I. In OpenSSL, we already know that 184 `memcpy`s out of 50K LOC lead to one Heartbleed vulnerability. Therefore, we probably can infer that there *might* be two heap buffer over-read vulnerabilities that are still unknown in Apache and another one hidden in Nginx.

Second, we further run the RUBiS benchmark (the detailed experiment setup will be introduced in Section V) using Apache with OpenSSL to get a rough ratio of data leakage between the Heartbleed bug and all the other `memcpy`s. Here, we run 16 user sessions for about one minute. We perform the Heartbleed attack on each user session by over-reading 32KB data for once, two times, three times, respectively. For all the other `memcpy`s, we set the size of data over-read to be the original (intended) memory copy size. Fig. 3 (a), (b), and (c) show the ratio of data leakage between Heartbleed and all `memcpy`s from the three experiments. We can see that the Heartbleed bug only contributes around 10% to 30% to all the potential data leakage.

As the result, given a technique targeting at heap buffer over-read vulnerabilities, it is insufficient to only evaluate the effectiveness against one particular known bug. Instead, one needs to consider all instances of `memcpy` to perform a comprehensive risk evaluation.

III. Risk Assessment

The risk assessment for the `memcpy`-based heap buffer over-read vulnerabilities boils down to two questions: (1) *How to collect the potential heap buffer information leak?* (2) *How to quantify the potential heap buffer information leak?* In this section, we describe our methodology to address these two questions.

A. How to collect the potential heap buffer information leak?

Since we consider every `memcpy` potentially vulnerable, we dynamically hook each occurrence of `memcpy` and simulate buffer over-read attacks by copying out additional N bytes (after the intended `size` bytes in the `src` buffer) and write to a log file. The N bytes are therefore regarded as information leak. As the size N is usually controlled by the attacker and hence hard to predict, we do not specify the exact value as part of our method. Nevertheless, there could be many heuristics and strategies on choosing the over-read size. For example, in Heartbleed attackers can read at most 64KB (*i.e.,* 2^{16}) because the `size` argument is converted from a short integer of two bytes. Some heuristics can also be employed to estimate the upper limit of N used by a reasonable attacker, for instance, the maximum size of `malloc` request. In addition, one can also use a stochastic process (*e.g.,* Gaussian process) to generate a sequence of variable over-read size. In sum, it is better for system administrators or security officers who adopt our method to determine the size N depending on the target platform and workload, such as the type of web server, the type of client requests, the characteristics of web pages being hosted (*e.g.,* static or dynamic, page size, *etc.*). In our experiment, we select fixed size of 1KB, 16KB, and 32KB to simulate the buffer over-read attack and collect data leak.

The implementation is straightforward. We simply hook every `memcpy` using the `LD_PRELOAD` trick, calculate the range of victim buffer, and copy the data out and dump to a log file. The benefit of using `LD_PRELOAD` is that our risk assessment can directly handle off-the-shelf binaries without the need of source code. It should be pointed out that in real world programs the source buffer of `memcpy` can also be on stack and data segment. To remove these `memcpy`s from our consideration, we check the virtual addresses of source buffers against the memory map `/proc/self/maps` and skip those `memcpy`s. After running the target program with certain test inputs or benchmarks, the log file will contain the (simulated) leaked data.

B. How to quantify the potential heap buffer information leak?

The quantification of information leak is the top challenge of our risk assessment. We develop four metrics to quantify the information leakage in different aspects. In reality, since people may or may not know what data is targeted by an attacker, we come up with two assumptions: a weak assumption and a strong assumption. In the weak assumption, we assume we are uninformed of what data is targeted by the attacker. We propose Metric 1 and 2 below to perform a macroscopic measure on the gross information leakage. The strong assumption means that we are aware of the target of the attacker (*e.g.,* private key, username-password pair). We propose Metric 3 and 4 below to conduct a fine-grained examination. Note that although we only demonstrate in Section V the application of these metrics using a particular benchmark, the metrics themselves are meant to be general. The four metrics are described as follows:

Metric 1: Volume of Information-carrying Bytes. Not every byte leaked to the attacker carries meaningful information. For example, if the victim buffer is filled with zeros, there is virtually no meaningful information being leaked. So we can use both the size and ratio of the information-carrying bytes to quantify how much gross information is leaked. How to differentiate meaningful bytes and meaningless bytes depends on the particular defense techniques. In our case studies (introduced shortly in Section IV), we pick a special ASCII control code `0x06` (ACK) to be used in the defense techniques to clean the heap memory. For example, if a 1MB log file contains 700KB non-`0x06` characters, the ratio of information carrying bytes would be 70%.

Metric 2: Information Entropy (Compression Ratio). In information theory, entropy is used to indicate the quantity of information. In the literature of quantitative information flow, various entropy measures have been explored, including Shannon entropy, guessing entropy, and min-entropy [22]. In our context, since there is not a clear way to do word segmentation or text segmentation against the log file, we use *compression ratio* to approximate the information entropy of the leaked data. Particularly, we leverage three different compression algorithms (`gzip`, `bzip2`, and `xz`). The higher the compression ratio is, the higher entropy the leaked data contains.

Metric 3: Sensitive Data Quantity. Apparently, if we know the exact sensitive data (*e.g.,* server private key) or we are able to identity sensitive data based on the patterns (*e.g.,* email address), we can perform targeted analysis to quantify the amount of information leakage. In our experiment on the RUBiS benchmark, for instance, the user name and password of a user with ID 1234 will be "user1234" and "password1234", respectively. As a result, we can search for such patterns from the log file and count the total number of valid username-password pairs.

Metric 4: Unique Sensitive Data Quantity. Since the data over-read by multiple `memcpys` could have overlaps, it is necessary to remove the redundancy and check the amount of unique sensitive data leaked. Different from Metric 3 which implies how big the chance is for an attacker to get something valuable, this metric reveals how much unique entities can be affected. The entities can be registered users, customers, and websites. In our experiment, for example, the number of leaked username-password pair indicates the number of users of the auction site that are affected.

IV. Case Studies

To validate our risk assessment methodology, we perform case studies on four different sets of approaches adopted from existing literature, which are all aimed at mitigating information leakage in buffer over-read attacks.

A. Padding at Allocation

Assuming the same attack with the same length of over-read, one intuition to reduce the amount of information leakage is to increase the memory allocation size, in other words to pad the memory object with additional memory, and initialize the memory (including both original and padding) with zero bytes. Actually, similar padding ideas have already been proposed in DieHard [23] and OpenBSD to tolerate memory errors such as buffer overflows, dangling pointers, and reads of uninitialized data.

Ideally, if the padding bytes are longer enough, larger than the upper limit of over-read size used by attacker, the attacker can get nothing except his own memory content. This will lead to zero information leak. In reality, however, the upper limit of over-read size could be very large (*e.g.,* 64KB in Heartbleed) and even not foreseeable in case of zero-day attacks. Too large padding size could cause dramatic increase in the memory consumption as well as performance overhead. We choose a simple heuristic, *i.e.,* making the padding size to be the same as the original allocation size. The merit of doing so is the deterministic space overhead, that is, the heap will be at most as twice large as the original heap. To achieve this, we simply hook every `malloc`, double the requested size, perform real `malloc`, and zero out the allocated memory. The following code snippet demonstrates this procedure.

```
void* malloc_hook(size_t size) {
   void *ret = malloc(size*2);
   if (ret) {
      memset(ret, 0, size*2);
   }
   return ret;
}
```

B. Erasing at Deallocation (Inline)

It has long been a secure programming guideline that sensitive data should be properly erased after the processing has done with it. This idea can also help to lower the chance and amount of information leakage. This idea has been explored in data lifetime research to defend against memory disclosure attacks [12]. We reimplement this technique for glibc heap memory management by hooking `free`, retrieving the chunk size, and then zeroing out the chunk. The code snippet shown below demonstrates this idea.

```
void free_hook(void *ptr) {
   if (ptr) {
      size_t size = malloc_usable_size(ptr);
      memset(ptr, 0, size);
   }
   free(ptr);
}
```

C. Erasing at Deallocation (Concurrent)

We also attempt to reduce the performance overhead of the inline heap erasing by exploring a concurrent technique proposed in Cruiser [24]. As shown in Fig. 4, the key idea is to migrate the heap erasing task to a concurrent eraser thread and leverage lock-free data structure to achieve non-blocking and efficient synchronization between user threads and the eraser thread. Since the design and implementation of this technique is not the focus of this paper, we only provide a brief overview of this technique as below.

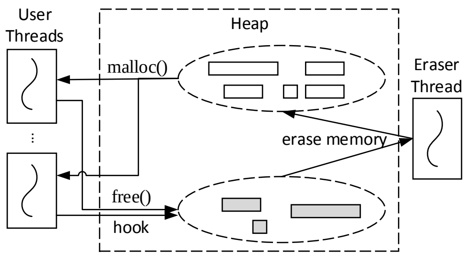


Fig. 4: Architecture of concurrent heap buffer erase

We hook every call to `free`, push the target pointer to a queue, and return immediately. Meanwhile, we create a dedicated eraser thread to read pointers out of the queue, perform memory erasing, and then call the glibc `free` to do real deallocation. We implement the queue data structure based on the single-producer single-consumer FIFO lock-free ring buffer proposed by Lamport [25]. This data structure (not shown in Fig. 4) enables a producer thread and a consumer thread to concurrently perform operations (*i.e.,* enqueue and dequeue) on the ring buffer. The following pseudocode highlights the workflow of this technique.

```
void free_hook(void *ptr) {
   if (ptr) {
      enqueue(ptr);
   }
}
void* eraser_thread() {
   while(!stop){
      void *ptr = dequeue();
      memset(ptr, 0, malloc_usable_size(ptr));
      free(ptr);
   }
   return NULL;
}
```

D. Concurrent Erasing plus Padding

Based on the concurrent heap erasing, we further combine it with the technique of padding at allocation. Theoretically, we simply need to integrate the two techniques together. In practice, we realize that all the newly allocated memory returned by `malloc` have already been erased by an earlier `free`. Therefore, it is unnecessary to zero out the memory again. So we only need to hook the `malloc` and do memory allocation with double request size.

V. Experimental Results

A. Experiment Setup

We base our experiment on the RUBiS [14] benchmark which is commonly used for evaluation by the systems research community, especially on emulating real world workload of websites that use dynamic content. The RUBiS benchmark is an auction site prototype modeled after eBay.com. It contains three types of user sessions, namely buyers, sellers, and visitors, as well as 26 types of interactions, such as ViewItem, PutBid, BuyNow, Sell, *etc.* The workload generator can simulate many clients that follow a markov model to browse and take actions on the auction site and there is also some "thinking time" between the actions.

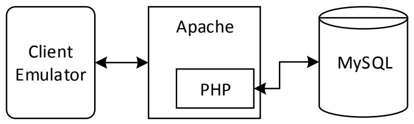
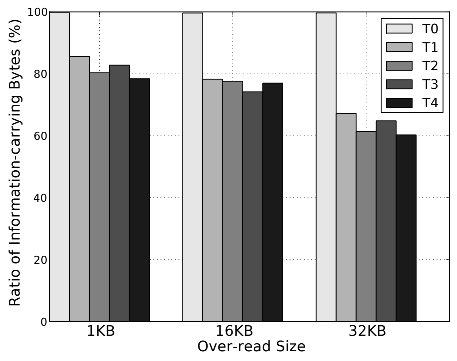
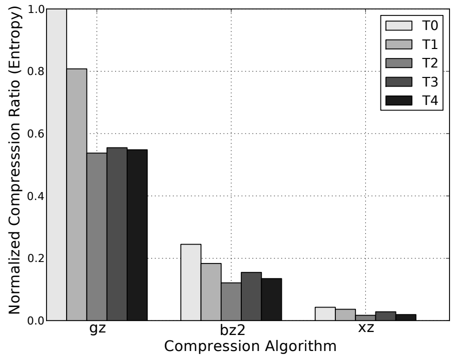


Fig. 5: Experiment setup of the RUBiS benchmark

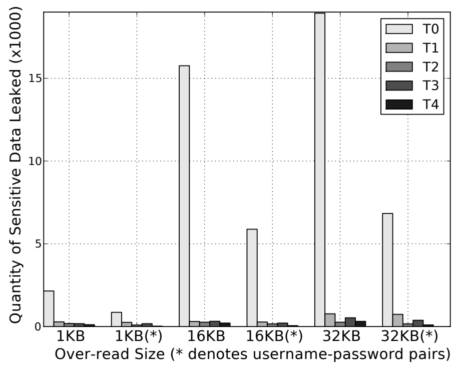
We select the PHP version of RUBiS implementation and set up a three-tier testbed, including a web server, a database, and a client emulator as shown in Fig. 5. We use Apache version 2.2.12 with PHP version 5.3.2 and MySQL version 5.1.73. The client emulator is compiled and run using Java 6. We collect the information leakage from the Apache web server. On the client side, we simulate 100 users concurrently browsing the website. Since the size of the log file grows so quickly, we pause the client emulator every one minute, perform the measurement, delete the log files and then resume the client emulator for another one minute. All the results reported below are averaged over 10 rounds of such iterations.



(a) Metric 1: Volume of Information-carrying Bytes



(b) Metric 2: Information Entropy



(c) Metric 3: Sensitive Data Quantity

Fig. 6: Experimental results of Metric 1, 2, and 3 on baseline and the four techniques: T0-baseline, T1-padding, T2-erasing (inline), T3-erasing (concurrent), T4-erasing (concurrent) + padding.

B. Risk Assessment

Metric 1. In the log files dumped from the baseline run (*i.e.,* without applying any technique), we found that the ASCII control code `0x06` rarely appears under our experiment setup. So we choose `0x06` for the four techniques to do memory cleaning. We then count the number of this particular value in the log files and compute the ratio over the entire file size. In order to get a more consistent comparison, buffer over-read attacks using size 1KB, 16KB, and 32KB are simulated in the same run. That is, we hook `memcpy` and do three `write`'s to three log files that correspond to the three over-read sizes.

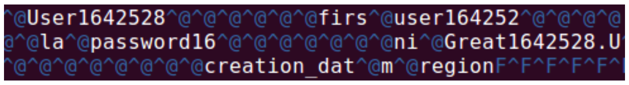
The results are shown in Fig. 6(a). The ratio of information-carrying bytes for the baseline is nearly 100%, while the others are mostly below 80%. This indicates that all of the four techniques can reduce the information-carrying bytes. The larger the over-read size is, the less percentage of information-carrying bytes the attacker can get. For the over-read size of 32KB, the average ratio of information-carrying bytes reduces to 63.4%. It is interesting to point out that attackers usually tend to try over-read with a longer length as they think they will gain more information. However, using a longer size will increase the possibility of failure (*e.g.,* causing a crash). Now our results seem to indicate that the belief of "longer over-read size will lead to more data leakage" does not always hold. So an attacker using a shorter size can still gain similar amount of information. From the attackers' perspective, however, this sounds to be a good news.

Comparing the four different techniques, the rough conclusion is that erasing plus padding is better than purely erasing, which is in turn better than purely padding. The inline erasing is oftentimes better than the concurrent erasing because in concurrent erasing, user threads calling `free` and the eraser thread erasing the memory are asynchronous events so that there is a small time window during which the data is still left on memory. However, this advantage comes with some performance penalty (see Section V-C).

Metric 2. As shown in Fig. 6(b), we use three compression algorithms to approximate the entropy of information contained in the leaked data. We first calculate the compression ratio for each compression algorithm on every test cases and use the compression ratio of `gz` on the baseline as the normalization factor to normalize the compression ratio of others. The over-read sizes of the test cases shown in Fig. 6(b) are all 16KB. The difference between the average compression ratios of the three algorithms is attributed to their inherent difference in compression and encoding schemes. Overall, the techniques using erasing are more effective in reducing the information entropy than the padding technique. The average entropy of T2, T3, and T4 is almost only half of the baseline's entropy, indicating that these techniques are indeed able to mitigate the information leakage to a great extent.

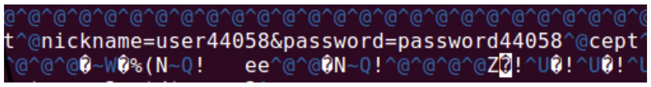
Metric 3. For the RUBiS benchmark, we define the user's information as sensitive and thereby targeted by attackers. Specifically, the sensitive information of a user include first name, last name, email address, nickname (*i.e.,* username), and password. As mentioned earlier, in the RUBiS benchmark every user has a unique user ID and these sensitive information all share the same pattern: [keyword][user ID]. For example, all of the usernames and passwords have the pattern of "user[user ID]" and "password[user ID]", respectively. Consequently, we analyze the log file and search for the sensitive information by pattern matching with regular expressions.

Interestingly, we found that not all the results that match the patterns are valid. The log file fragment shown in Fig. 7(a) gives an example. The strings "User1642528" and "Great1642528" are the valid last name and first name of the user whose user ID is "1642528". However, the strings "user164252" and "password16" are truncated by other data and thus invalid. We regard these invalid results as false positives and filter them out through correlating with the client emulator to check the validity of user IDs. It is worth mentioning that knowing the first part of a password greatly reduces the search space for a brute-force password attack. Since further quantifying the difficulty of password cracking is beyond the



```
^@User1642528^@^@^@^@^@firs^@user164252^@^@^@^@
@^@la^@password16^@^@^@^@^@^@ni^@Great1642528.U
^@^@^@^@^@^@^@creation_dat^@m^@regionF^F^F^F^F^
```

(a)



```
@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
t^@nickname=user44058&password=password44058^@cept
^@^@^@�~W�%(N~Q!   ee^@^@�N~Q!^@^@^@^@Z�!^U�!^U�!^U
```

(b)

Fig. 7: Leaked data showing (a) some sensitive information and (b) a valid username-password pair

TABLE II: Metric 4 – Quantity of Unique Sensitive Data

Over-read Size	Type I: all, II: username-password pair	T0	T1	T2	T3	T4
1KB	I	9	5	2	2	1
	II	2	1	1	1	0
16KB	I	9	5	2	3	2
	II	2	1	1	1	1
32KB	I	9	7	3	5	3
	II	2	2	1	1	1

focus of this paper, we simply regard these truncated password as invalid. In addition, to also avoid counting the sensitive information leaked by the same user (which is virtually not an information leak), we modify the client emulator to create different groups of users for every 1-minute interval. As such, we check every interval (except the first one) against the users in the previous intervals to determine the sensitive data leak.

Fig. 6(c) shows the quantity of sensitive information retrieved from the leaked data. Since the username-password pair is normally considered much more sensitive and can be used to directly compromise a user account, we additionally measure the quantity of valid username-password pairs. Fig. 7(b) shows a fragment containing a complete pair of username and password. For all the test results in Fig. 6(c), we can observe a clear difference on the quantity of sensitive data leaked with and without the mitigation techniques. The huge difference suggests that even a simple technique might be able to reduce the risk to a large extent.

Metric 4. Lastly, we apply Metric 4 to measure the quantity of unique sensitive data leaked. Table II lists the numbers of different test cases over the three sizes. As for the over-read size, we can see that the difference it makes is quite minimal, especially for the baseline which is totally the same across 1KB, 16KB, and 32KB. On the other hand, the four techniques are further proven to be effective. The number of unique username-password pairs for T4 is even reduced to zero under the 1KB over-read size.

C. Performance Comparison

We further compare the performance of the four techniques in terms of microbenchmark runtime overhead, web server throughput overhead and memory overhead. First, we write a microbenchmark which allocates a series of memory ranging from 10 bytes to 20KB ($\sim$200MB in total), does some computation, sleeps for a while (to simulate I/O), deallocates all the memories, and repeats this iteration for 50 times. Fig. 8 shows the runtime of the microbenchmark in different test cases. Compared to (inline) padding and inline erasing, concurrent erasing incurs much less runtime overhead, which is even lower than the baseline. The reason is that in the concurrent erasing, the main thread does not even need to do the real stuff in `free` (which is delegated by the eraser thread).

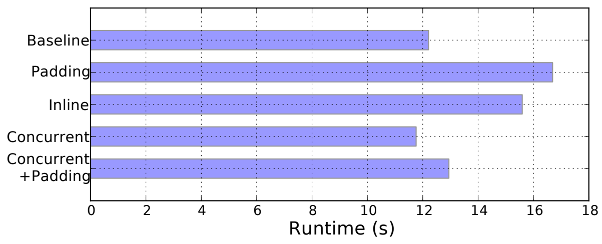


Fig. 8: Runtime comparison on microbenchmark

TABLE III: Throughput and memory overhead on Apache

Techniques	Throughput Overhead	Memory Overhead
Padding	0.4%	2.6%
Inline	0.2%	0%
Concurrent	0.1%	3.9%
Concurrent + Padding	0.1%	6.2%

Second, we use Apache web server and ApacheBench version 2.3 to measure the web server throughput overhead and RSS memory overhead. The concurrency of ApacheBench is 50 and the number of request is 10,000 for each run. We repeat the test 10 times and take the average for each measurement. The experiments are done on a DELL T310 server with 2.53GHz quad-core CPU and 4GB RAM. The result in Table III shows that the throughput overhead for all four techniques is negligible. As for memory overhead, padding and concurrent erasing will each increase roughly 3%, which is still very small.

VI. Discussion and Limitations

It should be pointed out that the results of risk measurement and performance of different mitigation techniques are affected by various factors, including the characteristics of programs, the workloads, how the sensitive data is defined, *etc.* Even in our own experiment, for example, the performance comparison for the microbenchmark and the web server are not in line with each other. Therefore, we encourage system administrators or security officers to do their own measurement if they are interested in adopting our method to do risk assessment.

In Metric 3 and 4, we only measure the leakage of sensitive data across intervals. In theory, it is also useful to measure the leakage within the same interval (*i.e.,* one user steals the data of a concurrent user). However, this requires hacking into the web server to associate each invocation of `memcpy` with the corresponding user ID and involves modifying and recompiling the program, which may not be a practical option in many cases. We plan to explore non-intrusive ways to achieve this in the future.

Another limitation is that given the limited disk space, our current approach suffers from the fast growing speed of the log files. One solution is to embed the analysis component into the data collection component so that the analysis component can consume the data on the fly and the log files are no longer needed. However, doing this integration might change the heap memory layout and thus affect the results. We leave this issue for future work.

VII. Related Work

It has been well accepted that sensitive data (*e.g.,* passwords, SSN, and bank accounts) scatters in memory and thus becomes a favorable target of attacks. The investigation by Chow et al. further reveals that many popular applications, such as Mozilla and Apache, take virtually no measures to limit the lifetime of sensitive data they handle [12]. While their work aims at tracking the lifetime of tainted sensitive data, we focus on quantifying the risks of information leakage on sensitive data.

There exist approaches that quantify information leaks in a few different aspects [22], [26], [27]. For example, [26] proposes to minimize and constrain information leaks in outbound web traffic by checking fixed data pattern against the HTTP protocol. Heusser and Malacaria [27] introduces a technique to decide if a program conforms to a quantitative policy via model checking. Baring a similar purpose but a different focus in mind, we present the first quantitative study on the information leaks of buffer over-read vulnerabilities.

Zeroing data upon deallocation is an important security practice [18], [28], which, however, is not well taken in practice. Chow et al. thus propose to zero data upon deallocation automatically [12]. We have included this strategy in our comparison. In addition to considering deallocated data, our method also evaluates the risk of information disclosure targeting allocated data. Harrison and Xu [29] have focused on the disclosure of cryptographic keys at both allocated and deallocated data, and proposed combined solutions to minimize the threat. However, their approaches require modification of library and kernel code. Zeng et al. [30] propose a new end-to-end defense against zero-day buffer over-read vulnerabilities and evaluate their solution with the Heartbleed attack. In our work we examine a set of simple non-intrusive approaches that can be applied to mitigate information leakage caused by buffer over-read.

VIII. Conclusion

The research on buffer over-read vulnerability is gaining more and more attentions from both academia and industry recently. This is mainly due to the vast damage that the Heartbleed bug has caused and, more importantly, it is much more difficult to detect and fully defeat compared to buffer overflow attacks. As the Heartbleed bug has remained undiscovered for about two years, it is entirely possible that zero-day "Heartbleed" over-read vulnerabilities still exist or will be introduced. To evaluate new counter-measures against buffer over-read attacks, people would need to perform risk assessment on all the potential information leakage caused by unknown "Heartbleed" vulnerabilities.

We present a set of feasible metrics to quantitatively evaluate the risks associated with `memcpy` operations in buffer over-read attacks. We develop practical methods to collect and measure information leakage in real world programs. We report our experiments on an auction site prototype modeled after eBay.com. Our experience reveals that even some simple non-intrusive techniques can achieve reasonable defense against buffer over-read in terms of information leakage as well as performance.

Acknowledgment

We would like to thank anonymous reviewers for their insightful feedback. This work was supported in part by ARO W911NF-09-1-0525 (MURI), NSF CNS-1223710, NSF CNS-1422594, and ARO W911NF-13-1-0421 (MURI). Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of any organization.

References

[1] "Cwe-126: Buffer over-read," http://cwe.mitre.org/data/definitions/126.html.

[2] "The heartbleed bug," http://heartbleed.com/.

[3] 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 *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM, 2014, pp. 475–488.

[4] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *SSYM'98*.

[5] Y. Yarom, "Method of relocating the stack in a computer system for preventing overrate by an exploit program," uS Patent 5,949,973.

[6] "PaX," http://pax.grsecurity.net/.

[7] R. W. M. Jones and P. H. J. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs," in *Proceedings of the Third International Workshop on Automated Debugging*, May 1997.

[8] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," ser. SSYM'09.

[9] D. Ye, Y. Su, Y. Sui, and J. Xue, "WPBound: Enforcing spatial memory safety efficiently at runtime with weakest preconditions," in *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering (ISSRE'14)*, Naples, Italy, 2014.

[10] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy code," ser. POPL '02.

[11] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer, "CCured in the real world," ser. PLDI '03.

[12] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding your garbage: Reducing data lifetime through secure deallocation," ser. SSYM'05.

[13] S. Maguire, *Writing Solid Code*, ser. Code Series. Microsoft Press, 1993.

[14] "Rubis," http://rubis.ow2.org/.

[15] "Cwe-125: Out-of-bounds read," http://cwe.mitre.org/data/definitions/125.html.

[16] T. C. Miller and T. De Raadt, "strlcpy and strlcat-consistent, safe, string copy and concatenation." in *USENIX ATC, FREENIX Track.*, 1999.

[17] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities." in *NDSS*, 2000, pp. 2000–02.

[18] J. Viega and G. McGraw, *Building secure software: how to avoid security problems the right way*. Pearson Education, 2001.

[19] "Transport layer security (tls) and datagram transport layer security (dtls) heartbeat extension," http://tools.ietf.org/html/rfc6520.

[20] "On detecting heartbleed with static analysis," http://security.coverity.com/blog/2014/Apr/on-detecting-heartbleed-with-static-analysis.html.

[21] "Finding heartbleed with codesonar," http://www.grammatech.com/blog/finding-heartbleed-with-codesonar.

[22] M. Backes, B. Kopf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *S&P Oakland'09*.

[23] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," ser. PLDI '06.

[24] Q. Zeng, D. Wu, and P. Liu, "Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures," ser. PLDI '11.

[25] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Softw. Eng.*, vol. 3, no. 2, pp. 125–143, Mar. 1977.

[26] K. Borders and A. Prakash, "Quantifying information leaks in outbound web traffic," in *S&P Oakland'09*, pp. 129–140.

[27] J. Heusser and P. Malacaria, "Quantifying information leaks in software," ser. ACSAC '10, pp. 261–269.

[28] "Protecting sensitive data in memory," http://www.ibm.com/developerworks/library/s-data.html?n-s-311.

[29] T. P. Parker and S. Xu, "A method for safekeeping cryptographic keys from memory disclosure attacks," ser. INTRUST'09.

[30] Q. Zeng, M. Zhao, and P. Liu, "Heaptherapy: An efficient end-to-end solution against heap buffer overflows," in *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE, 2015.