Detecting leaks of sensitive data due to stale reads

Will Snavely, William Klieber, Ryan Steele, David Svoboda, Andrew Kotov

Software Engineering Institute
Carnegie Mellon University
{wsnavely,weklieber,rsteele,svoboda,aakotov}@cert.org

***Abstract*—It is well-established that out-of-bounds buffer reads and writes pose a major threat to an application's security. Less appreciated is that even in-bound reads have the potential to lead to a violation of a security property of the application, especially confidentiality-based requirements. For example, an uninitialized or stale portion of a buffer might contain sensitive information and generally should not be read. This paper introduces a heuristic-driven dynamic analysis that aims to detect reads that may be accessing stale sensitive data. We apply this analysis to two real-world programs where in-bounds stale reads led to leakage of sensitive data: the Jetty web server (with the JetLeak vulnerability) and OpenSSL (with the Heartbleed vulnerability). Our approach was able to detect JetLeak, and with some modifications was also able to detect Heartbleed. We furthermore applied our analysis to the GNU Coreutils, and report results from that experiment. We suggest a number of directions for future work to refine and extend our approach.**

I. Introduction

Computer security research has investigated memory misuse to a considerable depth. A well-accepted tenet: if a program allocates an array, all accesses of the array should occur within its allocated bounds. Otherwise, a terrifying variety of security compromises become possible, e.g., a buffer overflow attack that overwrites a return address, subverting control flow. There are various strategies for managing the risk generated by out-of-bounds reads and writes. One strategy involves interposing an agent that performs bounds checks before every array access. In this paper, we aim to push this concept further and consider applying access controls to data *within* the bounds of the array. In brief, we are asking the question: under what circumstances is it be appropriate to restrict access to elements within the legal bounds of an array?

There is one immediate answer: uninitialized elements of an array should not be read. CWE-457 (*Use of Uninitialized Variable*) [14] identifies uninitialized reads as a general security risk, with a wide range of potential negative impacts. However, there are also cases where accessing an initialized element of an array may violate a security requirement of the application. Principally, we are concerned with violations of confidentiality requirements, whereby an actor receives information they are unauthorized to view (CWE-200, *Information Exposure* [14]).

Within this domain, we are narrowing our focus to *stale data*. To understand this, consider the following scenario. A simple web server services multiple clients. When a new client request arrives, it is stored in a global (or thread-local) buffer, processed, then discarded. The next request will then be stored into the same buffer, processed, discarded, and so on. In the interest of efficiency, the server implements the "discard" operation by simply setting a *size* attribute of the buffer to 0, without clearing out data from the request. Therefore, the next request will overwrite some amount of data from the previous request, as shown in Figure 1. If, subsequently, any data remains from the first request, we call this *stale data*—data which was once valid, but some time later becomes invalid. A confidentiality violation could occur if—due to some defect in the server software—an adversarial client could arrange for the server to send them stale data from the shared buffer. In this example, the valid portion of the array (i.e., the portion of the array containing non-uninitialized, non-stale data) is from index 0 to the most recently written location of the buffer.

There are several high profile software defects that have exhibited this characteristic. The OpenSSL vulnerability CVE-2014-0160 (aka Heartbleed) featured a shared buffer, the contents of which could be exfiltrated with a malformed request. Moreover, a Heartbleed exploit could read past the end of this buffer into adjacent memory. Thus, Heartbleed facilitated all of the memory abuses outlined in this paper so far: reading out of bounds, reading uninitialized data (e.g., uninitialized elements of the shared buffer), and reading stale data (e.g., elements of the shared buffer initialized by previous requests).

CVE-2015-2080 (aka JetLeak) was a defect discovered in the Jetty web server, written in Java. Again, a malformed request could be used to retrieve stale data from a shared buffer, due to a bug in an error handling procedure. The technical details of this vulnerability are available in a blogpost courtesy of Gotham Digital Science [5]. JetLeak demonstrates

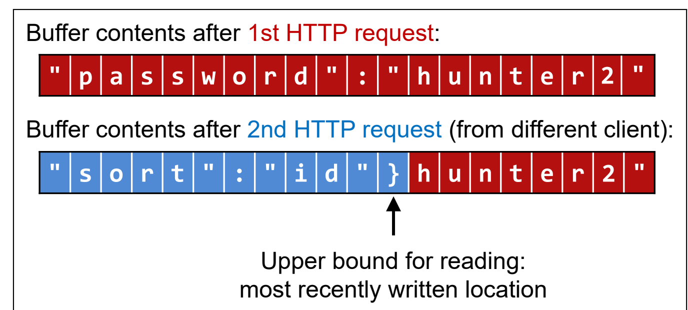

Fig. 1. Reused buffer with stale sensitive data

that the problem of stale reads is not unique to memory-unsafe languages like C or C++. Memory-safe languages like Java, which tend to be resistant to out-of-bounds reads and writes, are also susceptible to stale reads.

We posit that stale reads are, in some cases, detectable by automated means, just as other classes of memory misuse are (see section V). In this paper we consider a dynamic approach to detecting such reads.

II. Approach

This paper principally investigates one particular heuristic for detecting stale reads, which we will refer to as the *sequential write* heuristic.

Definition (Qualifying). A buffer B is a *qualifying* buffer under the sequential write heuristic iff every write to the buffer is either to index 0 or to the successor of the last written position (LWP) of B. ("Last" here means "most recent".)

Note that in addition to writing sequentially, the definition of *qualifying* allows restarting at the beginning (index 0).

Definition (LWP-Bounded). A buffer B is *LWP-bounded* iff the portion of B with valid data is from index 0 to the last written position of B.

The sequential write heuristic posits that qualifying arrays are LWP-bounded. This heuristic is designed with the hope of discovering shared buffers, specifically those where a new generation of data is written over the previous generation, starting with the beginning of the array. It is assumed that once a new generation commences, data from previous generations is now stale. For example, return to our web server scenario. Each request represents a new generation of data written into the shared buffer, overwriting part of the previous generation but possibly leaving some stale data behind.

We propose a runtime monitoring agent that seeks to detect stale reads, as determined by a staleness heuristic such as the sequential write heuristic. The abstract definition of this agent should be reasonably language-agnostic, since programs in many languages are susceptible to this vulnerability. In this paper we will focus on C and Java, motivated by the Heartbleed (C) and JetLeak (Java) vulnerabilities discussed above. At a high level, this agent intercepts primitive memory operations at runtime, in order to maintain state information needed to employ the staleness heuristic. Specifically, this agent tracks the following events at runtime: (1) allocations of buffers in memory; (2) writes to allocated buffers; (3) reads of allocated buffers; and (4) deallocations of buffers. The term "buffer" here refers to any array-like data structure, including simple C arrays as well as higher abstractions such as the Java `ByteBuffer` class. Similarly, the concepts of allocation, read/write, and deallocation are somewhat abstractly defined, and may have slightly different interpretations for different languages and buffer data structures.

The main activity of this agent is determining whether each read of an allocated buffer is stale; such reads are *flagged*. This determination is based on monitoring previous writes of the buffer and using a heuristic to make inferences about which segments of the buffer are likely to contain stale data. This reveals a significant difference between this approach and existing dynamic analyses that look for out-of-bounds accesses. The latter analyses can typically identify out-of-bounds activity with high confidence, for example by tracking the size of a buffer and looking for reads and writes past that size. Our proposed analysis, on the other hand, may suffer a high false-positive rate.

A. Use in developing secure software

There are two main ways that our technique can be useful in developing secure software. The first is to use it as an aid in testing, debugging, and fuzzing a program. The second is to use it as a dynamic enforcement mechanism in production use, to detect invalid reads immediately before they occur and handle the error in some manner (such as aborting the program, throwing an exception, or zeroing the stale data). Either way, we need to solve the problem of false positives (i.e., qualifying buffers in which there are reads of valid data beyond the last written position).

One way to address the problem of false positives is as follows. First we run the dynamic analysis on a representative set of 'good' traces (i.e., traces that do not exhibit confidentiality violations), recording the *allocation site*[1] of buffers with flagged reads. The set of 'good' traces can come from a test suite or otherwise exercising the program in a controlled manner. Any flagged allocation sites on the 'good' traces will be presumed to be false positives, and the instrumentation of the program would then be adjusted to exclude these allocation sites from our analysis.

III. Implementation

This section describes the implementation of the proposed dynamic analysis for Java and C programs.

A. Shared Algorithm

Both the Java and the C implementations have a similar core algorithm. We maintain state information for every active buffer B allocated by the running program. $State(B) \in \{Init, Qualified, Disqualified\}$, where $Init$ indicates that the buffer has been allocated, but has not yet been read or written; $Qualified$ indicates that B is a qualifying buffer and has at least one write; and $Disqualified$ indicates that B is not a qualifying buffer.

$LWP(B)$ is a scalar value representing the last written position of B. Note that the sequential write heuristic implies that $LWP(B)$ is the greatest byte-offset in B that holds non-stale data. In particular, every offset $i \leq LWP(B)$ holds non-stale data, while every offset j such that $LWP(B) < j < len(B)$ holds stale (or uninitialized) data.

We represent writes to B with $Write(B, off, size)$, where off is the offset (in bytes) from the beginning of the array and

[1] The *allocation site* of a memory region is the location in the program text (either source code or binary executable code, depending on context) at which the memory region was allocated. Our analysis works on the binary, but also maps it back to the source code if the necessary debugging information was included during compilation.

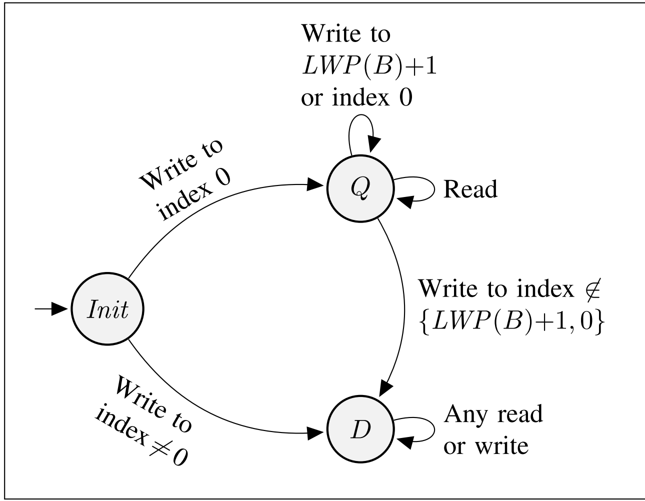

Fig. 2. The basic state machine driving the dynamic analysis, with initial, qualified (Q), and disqualified (D) states.

$size$ is the number of bytes written. Likewise, we represent reads with $Read(B, off, size)$.

Pseudo-code for the core algorithm is shown in Algorithm 1, and Figure 2 shows a state-machine representation for $State(B)$. When the allocation of a buffer B is detected, denoted "$Alloc(B)$", $State(B)$ is set to $Init$. Subsequent writes and reads of B are then monitored.

Upon $Write(B, off, size)$ where $State(B) = Init$, we check if $off = 0$. (Note that if $State(B) = Init$, then $LWP(B) = -1$, so $LWP(B) + 1$ simplifies to 0.) If so, B is marked as $Qualified$ and $LWP(B)$ is set to $(off + size - 1)$; else B is marked as $Disqualified$.

Upon $Write(B, off, size)$ where $State(B) \neq Init$, we check whether the new write is at position $LWP(B) + 1$ or at the beginning of the array ($off = 0$). If so, $LWP(B)$ is updated with the offset and size of the new write. Else B is marked as $Disqualified$.

Upon $Read(B, off, size)$, if $State(B)$ is $Qualified$, then we check whether the read occurs after $LWP(B)$. If so, then a warning is issued, indicating that a read of stale data may have occurred. We currently ignore reads that occur while the buffer is in the $Init$ or $Disqualified$ state, although the former likely indicate a bug in the program. Namely, if a buffer is in the $Init$ state, then we have observed no writes to the buffer, therefore a read of the buffer in this state is almost certainly accessing uninitialized data. Future work could add an option to the tool to flag these reads.

Finally, we detect the deallocation of B, or $Free(B)$, so that we can remove B from our list of active allocations (implementation not shown in Algorithm 1).

B. Java Implementation

The Java implementation of this dynamic analysis has two components: (1) a Java agent plugin, which performs some minor modification of Java class files to insert callbacks and (2) a runtime, which provides implementations of the inserted

Algorithm 1 Dynamic Analysis State Machine

1: **procedure** ALLOC(B)
2: $\quad State(B) \leftarrow Init$
3: $\quad LWP(B) \leftarrow -1$
4: **end procedure**

5: **procedure** WRITE($B, off, size$)
6: $\quad$ **if** $off = LWP(B) + 1$ **or** $off = 0$ **then**
7: $\quad\quad LWP(B) \leftarrow off + size - 1$
8: $\quad\quad$ **if** $State(B) = Init$ **then**
9: $\quad\quad\quad State(B) \leftarrow Qualified$
10: $\quad\quad$ **end if**
11: $\quad$ **else**
12: $\quad\quad State(B) \leftarrow Disqualified$
13: $\quad$ **end if**
14: **end procedure**

15: **procedure** READ($B, off, size$)
16: $\quad$ **if** $State(B) = Qualified$ **and** $off > LWP(B)$ **then**
17: $\quad\quad$ ISSUESTALEREADWARNING($B, off, size$)
18: $\quad$ **end if**
19: **end procedure**

callbacks. The bytecode rewriting in (1) is done with the ASM library [1]. The JetLeak vulnerability involved the class `java.nio.ByteBuffer`, therefore we targeted this class for rewriting. However, any array-like class could be rewritten in a similar way (`ArrayList`, `Vector`). Java arrays require slightly different treatment; instead of instrumenting higher-level method interfaces, one must instrument array-specific opcodes (e.g., `newarray`, `iaload`, `iastore`, and so on). However, this is well within the capability of ASM.

The `ByteBuffer` class implements an array-like data structure for storing binary data. Data is inserted with various `put` methods, and retrieved with various `get` methods. We instrumented the `get` methods with callbacks to detect reads, and the `put` methods with callbacks to detect writes. The two principal callbacks in our implementation are:

```
1) static void captureRead(ByteBuffer b,
                           int off, int size)
2) static void captureWrite(ByteBuffer b,
                            int off, int size)
```

These are static methods in a class called `Runtime`. Both accept a reference to the `ByteBuffer` being read/written as their first argument. The second argument specifies the offset in the `ByteBuffer` where the read/write begins, and the third argument specifies the size of the read/write. These callbacks implement the basic algorithm described in subsection III-A. Data for each buffer (e.g., its state, its last written position) is maintained in a dictionary data structure within the `Runtime` class, keyed off a unique identifier associated with each `ByteBuffer` instance.

Listing 1 illustrates the results of the bytecode rewriting on the `ByteBuffer` class, with our additions highlighted. A few details are glossed over here; the shown `get` and `put` methods are both abstract, so the actual instrumented methods were in

```
class ByteBuffer /*...*/ {
  public byte get(int index) {
    Runtime.captureRead(this, index, 1);
    // Implementation of get() starts here...
  }

  public void put(int index, byte b) {
    Runtime.captureWrite(this, index, 1);
    // Implementation of put() starts here...
  }
}
```

Listing 1. Inserting callbacks with bytecode rewriting

```
// A global allocation
static char global[1024];

void func(size_t size) {
  // A stack allocation
  char stack[1024];

  // A heap allocation (with dynamic size)
  char *heap = (char*)malloc(size);
  // Heap allocations require explicit free
  free(heap);
}
```

Listing 2. Buffer allocations in C

a subclass (e.g., HeapByteBuffer). Also, only two methods are shown in the listing, but our implementation targeted a larger collection of get and put variants.

The two components of this system are packaged into two separate JAR files. To apply the analysis to a given Java application, the JVM must be invoked with the -javaagent argument, and the agent plugin JAR file passed as an argument. The agent plugin runs before the application is launched. The first thing it does is add our runtime JAR to the bootstrap class loader. Next, the desired bytecode modifications are performed. Finally, the Java application is launched. No source code modifications to the original Java application are required.

C. C Implementation

The C implementation of the dynamic analysis has two principal components: (1) a set of compiler instrumentations, added with a slightly modified AddressSanitizer pass in the Clang compiler [11]; and (2) a runtime agent that intercepts allocations, reads, and writes, and implements the algorithm described in subsection III-A. The runtime is built on top of Intel Pin, a dynamic binary instrumentation tool [9]. The compiler instrumentation was built on top of Clang 5.0, and the dynamic instrumentation was performed with Pin 3.5.

The compiler instrumentation piece is required to detect different types of buffer allocations in C programs. In the Java implementation, we had the luxury of inserting code into class definitions. Thus, we did not have to explicitly track object allocations, since instances of the instrumented classes would automatically trigger our callbacks. In C, we have no such luxury, and therefore must have some mechanism for tracking buffer allocations.

We are concerned with three types of allocation: stack, heap, and global. Stack allocations are typically implemented by adjusting a stack pointer at runtime, for example subtracting the size of the desired buffer from the stack pointer (common in x86 programs). The stack allocation is reclaimed by simply moving the stack pointer back to its original position. Heap allocations, often called dynamic allocations, occur via an explicit interface (e.g., malloc, calloc), and must be explicitly freed by the programmer (e.g., by free). These allocations are "dynamic" in the sense that their size can be specified at runtime, opposed to stack allocations which generally (though not always) are sized statically at compile time. Finally, global allocations are statically sized and are stored in a dedicated region of memory such as the BSS or DATA region. Listing 2 gives examples of the different flavors of allocation.

```
static char global[1024];

void premain() {
  capture_global_alloc(global, 1024);
}

void func(size_t size) {
  char stack[1024];
  capture_stack_alloc(stack, 1024);

  // Heap allocations don't receive
  // additional instrumentation
  char *heap = (char*)malloc(size);
  free(heap);

  capture_stack_free(stack);
}

void postmain() {
  capture_global_free(global);
}
```

Listing 3. Inserting callbacks with compiler instrumentations

Listing 3 demonstrates the effect of the compiler instrumentations added by our tool on the code in Listing 2. We add callbacks to detect global and stack allocations and frees.

Note that dynamic allocations do not receive any special instrumentation during compilation; neither do reads or writes. Instead, we use Intel Pin to dynamically instrument the binary executable to handle those cases. If the program being instrumented uses any dynamic allocation functions other than the

standard `malloc` function, we rely on the user to provide a list of such functions (including wrappers[2] around `malloc`). We then use Intel Pin to detect calls to those functions. We also use Pin to detect reads and writes, at the x86 instruction level. This raises the question of: why use dynamic instrumentation over linking the application with a shared library, in the manner of AddressSanitizer?

The deciding factor ultimately was the ability of each approach to capture reads and writes to the allocated buffers. AddressSanitizer does instrument reads and writes (rather, load and store instructions in LLVM), however it is possible that some reads/writes will be missed, for example if they occur in a non-instrumented third-party library, or if they occur within inline assembly. To avoid this possibility, we decided to use Pin to capture reads and writes.

In summary, the Pin tool has the following responsibilities. It intercepts (1) calls to dynamic memory allocation functions, which are specified by the user, (2) calls to functions inserted by the compiler instrumentation, to track stack/global allocations, and (3) read/writes of memory, at the x86 instruction level. When an allocation is detected, a record holding the state of the allocation is inserted into a balanced binary search tree. The records in this tree are sorted in reverse order by the starting address of the allocation. A given read or write is associated with a buffer by finding the allocation in the tree with the starting address closest to the read/write address, without exceeding it, a $\mathcal{O}(\log n)$ operation, where n is the number of allocations currently tracked. Next, it is checked whether the read/write occurs within the bounds of the discovered allocation (the size of the buffer is recorded at allocation time). If so, then the algorithm described in subsection III-A is followed. Otherwise, the read/write is ignored. Out-of-bounds reads and writes are detected by AddressSanitizer. When an allocation is freed, it is removed from the tree.

Extensions: A few extensions were made to the C implementation to address issues faced during experiments. First, we noticed that some implementations of the `memcpy` function write data starting from the end of the destination array and ending at the start. As a workaround, we treat `memcpy` as a special case: we treat it as a single monolithic read followed by a single monolithic write rather than a series of individual reads and writes. There are other similar functions that could be treated similarly, e.g., `memmove`, `memset`, and so on, but our current implementation only addresses `memcpy`. Additionally, we relax the constraint that new generations of data must begin at offset 0, and instead allow new generations to begin in the range $[0, \epsilon]$, where ϵ is a user-specified value. See subsection IV-B for more information on the motivation behind these changes.

[2]Wrapper functions must be identified in order to provide useful information about allocation sites. If a program exclusively uses a wrapper function, say `xmalloc`, and this function is not identified as an allocation function, then the program would be considered to have a single allocation site (*viz.*, the underlying call to `malloc` inside the definition of `xmalloc`), rather than considering each call site of `xmalloc` to be its own allocation site. Identification of wrapper functions can likely be automated to some degree by using heuristics, but we have not yet attempted to do so.

IV. Results

A. JetLeak

We installed a version of the Jetty webserver that is vulnerable to JetLeak (9.2.8[3]), and ran the server with our agent attached. The server hosted a very basic website containing only static content.

We sent two types of this request to this server: "bad" requests and "good" requests. Bad requests were based on the JetLeak testing script provided by Gotham Digital Science, the firm that disclosed the vulnerability[4]. In brief, the bad requests have a malformed "Referer" field in their HTTP header. Good requests were well-formed HTTP POST requests. We arranged for the application to dump a stack trace and terminate when a stale read was detected by the agent. There are several basic experiments we ran: (1) sending only good requests, (2) sending only bad requests, and (3) sending good requests followed by a bad request.

Experiment (1) resulted in no stale reads. Both experiments (2) and (3) resulted in stale reads being detected. The error message and stack trace produced by these experiments is excerpted in listing 4. The error message contains the unique ID of the buffer that produced the warning, the offset of the read in the buffer, and the size of the read (always 1 byte). The stack trace reveals that the stale read occurs during HTTP header parsing. In particular, an illegal character is discovered, and while building the error message (in `toDebugString`), a potential stale read of a `HeapByteBuffer` via `get` is detected. These observations are consistent with the description of the JetLeak vulnerability from Gotham Digital Science [5]. We conclude that the agent was successful in detecting the stale read that led to the JetLeak vulnerability.

Testing on Patched Jetty: We installed a patched version of the Jetty webserver, and reran our experiments. Surprisingly, we discovered that the same stale reads were still detected. However, critically, the reads did not propagate back to a client connected over a socket. Instead, the stale buffer contents are dumped to a server-side log. Said another way, the sensitive data does not appear to cross a trust boundary. Therefore, in this case the detector has issued *false positives*. This is a place where source and sink information is needed to refine the detector. As it stands, the detector does not know if the value returned by a suspicious read ever crosses a trust boundary. We cover this topic more in section VI.

B. Heartbleed

We constructed a standalone test application using version 1.0.1a of the OpenSSL library, which is susceptible to Heartbleed. This application allows the user to send customized heartbeat messages to a local OpenSSL server. In particular, the user can control the length of the heartbeat payload

[3]https://github.com/eclipse/jetty.project/commit/26b759

[4]https://github.com/GDSSecurity/Jetleak-Testing-Script

```
SUSPICIOUS READ
Buffer: 1595576509, Offset: 99, Length:1
at java.lang.Thread.dumpStack
at org.sei.jetbleed.runtime.MyRuntime.captureRead
at org.sei.jetbleed.runtime.MyRuntime.on_get
at java.nio.HeapByteBuffer.get
at org.eclipse.jetty.util.BufferUtil.appendDebugString
at org.eclipse.jetty.util.BufferUtil.toDebugString
at org.eclipse.jetty.http.HttpParser$IllegalCharacter
   .<init>
at org.eclipse.jetty.http.HttpParser.next
at org.eclipse.jetty.http.HttpParser.parseHeaders
```

Listing 4. JetLeak detector stack trace

(l_{actual}), as well as the value of payload length field in the request (l_{user}). Heartbleed can be reproduced by setting l_{user} to a value larger than l_{actual}. To facilitate the scenario of stale reads (opposed to uninitialized reads), the application first sends a well-formed heartbeat request with a 500 byte payload. It then sends the user-configured request. Each payload consists of a different, repeated byte value, so that we can easily tell if the second request leaks information from the first.

We built OpenSSL using our customized Clang compiler, with the `fsanitize=address` flag specified, which enables AddressSanitizer (recall that our instrumentation piggy-backs on ASan). We then built our test application with the same compiler configuration (not strictly necessary, as only OpenSSL requires instrumentation). Finally, we ran the application with our dynamic instrumentation agent (Pin tool) attached, supplying various l_{actual} and l_{user} values. Similar to the Java application, the Pin tool issues a warning when a potential stale read is detected.

Confirming the presence of in-bounds stale reads: We first confirmed that Heartbleed exhibited the specific defect of interest—namely, reading stale data within the bounds of an array. It is possible that setting l_{user} greater than l_{actual} will always trigger an out-of-bounds read or write, in which case AddressSanitizer is sufficient to detect Heartbleed, and our analysis is unnecessary.

We fixed the value of l_{actual} to some small integer, and fed increasingly large values of l_{user}. We did not receive an out-of-bounds access error from AddressSanitizer until l_{user} exceeded 17725. This bound was observed when we varied the value of l_{actual} as well. These observations suggest that when $l_{actual} < l_{user} \leq 17725$, the server will leak stale or uninitialized bytes from the in-bound portion of the shared buffer, the behavior we are hoping to detect with our dynamic analysis. In summary, AddressSanitizer is not sufficient to detect all occurrences of the Heartbleed vulnerability, and our additional analysis is justified.

Detecting stale reads: Our tool initially did not mark the shared buffer in OpenSSL as *Qualified*, thus we did not observe a stale read alert for this array. Our investigation into this revealed two factors: (1) the initial writes into the buffer of interest were not at offset 0, but at offset 3; (2) the majority of the writes into the buffer occurred via a `memcpy` call, and this `memcpy` sometimes wrote data from back-to-front (descending offsets) instead of front-to-back (ascending offsets). Our tool only looks for ascending offsets between writes, therefore this `memcpy` pattern was problematic.

To resolve these issues, we made two modifications to the tool. First, we relaxed the constraint that a generation of data must begin at index 0, instead allowing new generations to begin within some small ϵ of 0. Next, we abstracted `memcpy` calls so that they appeared as a single write, to avoid issues with the direction of the copy.

After making these changes, the tool successfully marked the shared buffer as *Qualified*, and correctly identified a stale read from this buffer due to a tampered heartbeat request. Listing 5 summarizes a log file produced by the tool, focusing on the buffer of interest. After sending the first, valid heartbeat request (500 byte payload), we see the allocation of a buffer 17736 bytes long. There are two stores to this buffer, first a small store of 5 bytes at offset 3 (after which the LWP becomes $3+5-1=7$), then a large store of 519 bytes at offset 8. Following these stores, there is a load of 500 bytes at offset 11, where the server is reading the heartbeat payload so that it can echo it back to the client.

Upon sending the second, invalid heartbeat request (payload of 100 bytes, but falsely reported to be 200 bytes), we again see a 5 byte store at offset 3, followed by a 119 byte store at offset 8. Finally, we see a 200 byte load at offset 11, which reads past the end of the current heartbeat request and into the payload of the previous request. Due to our tool modifications, the buffer remains *Qualified* throughout, so this final load is marked as suspicious.

```
Send heartbeat request 1 (valid):
    Payload length = 500, Length field = 500
Allocation of size 17736 at address 0x629000005200
Stored 5 bytes at offset 3
Stored 519 bytes at offset 8
Loaded 500 bytes at offset 11

Send heartbeat request 2 (tampered):
    Payload length = 100, Length field = 200
Stored 5 bytes at offset 3
Stored 119 bytes at offset 8
Loaded 200 bytes at offset 11:  Suspicious Read
```

Listing 5. Heartbleed detector log summary

C. GNU core utilities (“Coreutils”)

We have seen above that our technique is able to detect leaks of sensitive information in Heartbleed and JetLeak. In this section, we will examine the technique’s performance on a codebase where we do not expect to find any reads of stale sensitive data. We ran the tool on the GNU core utilities version 8.21 using the test suite included in the coreutils repository. (This codebase consists of 80,000 lines of source code, plus another 486,000 lines from included libraries.)

Running the test suite produced a total of 34,000 traces (executions). In Figure 3, row 1 shows the number of allocation sites that are disqualified (D), qualified (Q), and flagged (F) on at least one trace. There is a relatively large number of flagged allocation sites, which we presume to be

false positives. Manual investigation revealed that (at the x86 machine-code level) there were many cases where an odd number of bytes were allocated, but a word-sized word-aligned read was used, with subsequent masking out of the unwanted trailing bytes. This is safe, but it still triggered a read-beyond-LWP warning. To avoid these false positives, we added an option *LoadOverlap* to our tool to ignore reads of beyond-LWP bytes up to a word boundary. This reduced the number of flagged allocation sites by 79% (row 2 in Figure 3).

A small subset of executions failed with *LoadOverlap* turned off, and a different small subset of executions failed with *LoadOverlap* turned on. As a result, there were 5 (resp. 17) alloc sites that appeared qualified (resp. disqualified) in either the base configuration or *LoadOverlap* but not both. This accounts for the slightly different number of qualified and disqualified allocation sites in rows 1 and 2.

We noticed that sometimes a beyond-LWP might occur in an array that is qualifying in one run of the program but which is disqualified on another run. (Arrays are identified between different runs based on their allocation site.) To account for this, we added an option *MergeDQ* which postprocessed the data from the *LoadOverlap* runs. With this option, an array is considered to be disqualified for all runs if it is disqualified on any run. Results are shown in row 3 in Figure 3.

Finally, some of the allocations are fairly small (less than 32 bytes), so we added another postprocessing option *MinSize*=N, under which allocated memory regions of size less than N bytes are ignored. Rows 4 and 5 in Figure 3 show the results for $N = 32$ and $N = 64$.

Some of the remaining false positives are due to idiosyncrasies of individual tools. In `diff`, the program constructed an array of pointers, with a sentinel null pointer marking the end of the list of pointers. Later, it rewrote the first element of the array (with the same value, so in effect a no-op), resetting the LWP to 0. It then read all the pointers in the array, triggering a false-positive flagged read.

Configuration	D	Q	F
1. Base configuation	1852	1409	278
2. LoadOverlap	1849	1412	59
3. LoadOverlap, MergeDQ	1849	1310	38
4. LoadOverlap, MergeDQ, MinSize=32	1515	291	26
5. LoadOverlap, MergeDQ, MinSize=64	1178	186	17

Fig. 3. Number of allocation sites (out of 3160) that are disqualified (D), qualified (Q), and flagged (F) on at least one trace

For Configuration 5, the 17 flagged reads consisted of 12 confirmed false-positives (5 `struct` field-access reads, 5 SHA-specific reads, 2 Clang Address Sanitizer reads) and 5 unconfirmed flagged reads (missing debug information to map the instruction pointer to source code locations).

V. Related Work

There is a significant amount of work devoted to spatial and temporal memory safety for languages like C and C++. A general thrust of this work is introducing runtime enforcement of memory safety without unreasonable impact on performance. A paper from Nagarakatte [10] provides a thorough overview of approaches in this domain, organizing them into three categories: "tripwire" approaches, which insert guard blocks around valid memory regions; object-based approaches, which associate metadata with each object allocated in memory (such that each pointer to an object shares the same metadata); and pointer-based approaches, which associate metadata with each individual pointer.

Our approach is most similar to the object-based method, in that we store metadata for each individual allocated object. Moreover, we augment this with additional checking for out-of-bounds accesses. In our C implementation, we rely on AddressSanitizer (ASan), a tripwire approach ([11]). In our Java implementation, we can of course rely on the Java Virtual Machine to detect out-of-bounds accesses (as an aside, the Java approach to bounds-checking is, generally and unsurprisingly, an object-base approach, the job of which is significantly simplified due to absence of pointer arithmetic in the language). Our approach differs from the body of memory safety work in that we are not principally interested in spatial or temporal memory safety. We are more interested in detecting irregularities in how the valid portion of the array is used. Moreover, at this time we are not focusing on performance, though achieving acceptable performance is a longer-term goal.

Some have applied static analysis to the problem of array content analysis. Most notably, there are various approaches based on abstract interpretation, for example the work of Halbwachs and Perón [6], Cousout et al. [2], and Gange et al. [4]. Our work describes a dynamic analysis, and therefore has little similarity with these techniques.

The problem that this paper addresses is related to the problem of reads of uninitialized memory. Memcheck [12] uses dynamic binary instrumentation to detect reads of uninitialized memory, incurring a typical slowdown of 20x. More recently, MemorySanitizer [13] instruments C/C++ code at compile time to dynamically detect reads of uninitialized memory. The authors report a slowdown of 2.5x and doubled memory usage.

In a sense, the type of vulnerability that we address is one of information flow: the program reads sensitive data from a source and sends it to an undesired sink. Therefore, techniques based on taint tracking (e.g., [16], [3]) might profitably be employed. For example, for OpenSSL and Jetty, information read from one TCP connection (identified by pair of remote IP and remote port) might be tagged with the connection ID to prevent the information from flowing to a different connection. Li et al. successfully use dynamic taint analysis to detect Heartbleed [8]. However, there are two complications with this approach. First, the sources and sinks must be identified. This entails a significant one-time cost for manual analysis of each library or module that exposes sources or sinks. Second, some inter-connection information flow might be intended; enabling communication between users is of course one of the main functions of the Internet.

VI. Conclusion and Future Work

Our current implementation uses a combination of compile-time instrumentation and dynamic binary instrumentation. This has the advantage of catching all memory accesses, even those in code that were not instrumented at compile-time. However, it has disadvantages: (1) it is unsuitable for deployment in production use, and (2) it does not play nice with white-box fuzzing tools. In the future we plan to perform all instrumentation as part of compilation. Then we would be able to use white-box fuzzing tools such as AFL [15] to attempt to find inputs that trigger flagged reads. There would still be some manual analysis necessary to determine whether the flagged read is a true positive or a false positive.

This manual effort can be minimized by employing a slight variation on existing fuzzing techniques. Given the original codebase C, we create a variation C' that zeroes out any data that the staleness heuristic posits to be stale. We then use fuzzing techniques to try to find an input vector that not only triggers a flagged read but also causes C' to produce different output than C. In that way, manual code inspection can be avoided if the judgment of true/false positive can be made simply on the basis of which of the two outputs (those of C and C') is correct for the input vector.

This paper only explores one heuristic for detecting stale reads, but other heuristics are possible. For example, our heuristic requires that writes be sequential, but we could develop a heuristic that supports non-sequential writes. (For example, this would be needed to handle a 2-dimensional array that stores n headers each of length ℓ.) Instead of maintaining a last written position, we could instead maintain a set of *intervals* within a buffer that contain initialized data. In this case, we still require a mechanism for determining when one generation of data ends and another begins, so that we can properly invalidate the initialized intervals. For this, we could use the approach in this paper of looking for writes to an offset close to 0, or develop a new approach.

Programs will often explicitly store the length of the valid portion of a buffer, e.g., in a *size* property or variable (distinct from and typically less than the total length of the buffer). Future work could attempt to detect cases of this and insert bounds-checks based on such *size* properties.

In certain cases, the valid portion of a buffer can be determined in a specialized fashion. For example, some buffers mark the end of the valid portion with a *sentinel value*, such as NULL-terminated strings in C. Future work could attempt to detect invalid reads in such sentinel-terminated buffers, by checking if each read occurs past the end of the sentinel value.

A shortcoming of our current approach is that we do not attempt to detect erasures of stale data. For example, a program might zero out a buffer in between generations of data. Our current dynamic analysis would view this zeroing activity as just another generation of data, and consider subsequent reads of the zeroed data as stale. A refinement to our analysis would detect this activity and not issue warnings on reads of the zeroed (or otherwise-sanitized) data.

Early in the course of this work we considered static-analysis-driven approaches to this problem, e.g., statically discovering in-bounds array accesses that are outside the valid portion of the array. Further developing such analyses is an interesting direction for future work.

Our ultimate goal with this work is to develop techniques for *repairing* code that permits stale reads of data. There are two directions we are considering to get closer to this goal: developing more efficient runtime detection mechanisms, ala Softbound, or developing techniques for directly repairing the source code (as proposed in [7]).

References

[1] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30(19), 2002.

[2] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *ACM SIGPLAN Notices*. ACM, 2011.

[3] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[4] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. A partial-order approach to array content analysis. *arXiv preprint arXiv:1408.1754*, 2014.

[5] Gotham Digital Science. JetLeak vulnerability: Remote leakage of shared buffers in Jetty web server . https://blog.gdssecurity.com/labs/2015/2/25/jetleak-vulnerability-remote-leakage-of-shared-buffers-in-je.html, 2015. [Online; accessed 19-February-2018].

[6] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *ACM SIGPLAN Notices*. ACM, 2008.

[7] W. Klieber and W. Snavely. Automated code repair based on inferred specifications. In *SecDev 2016*. IEEE, 2016.

[8] W. Li, Y. Yan, H. Tu, and J. Xu. A dynamic taint tracking based method to detect sensitive information leaking. In *Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2014.

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[10] S. Nagarakatte, M. M. Martin, and S. Zdancewic. Everything you want to know about pointer-based checking. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32, 2015.

[11] 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.

[12] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[13] E. Stepanov and K. Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2015.

[14] The MITRE Corporation. Common Weakness Enumeration. https://cwe.mitre.org, 2018. [Online; accessed 19-February-2018].

[15] M. Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/, 2007.

[16] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 45(1):142–154, 2011.

Acknowledgements

Copyright 2018 IEEE. All Rights Reserved.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. Carnegie Mellon® and CERT® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. DM18-0296