MemSherlock: An Automated Debugger for Unknown Memory Corruption Vulnerabilities*

Emre C. Sezer, Peng Ning,
Chongkyung Kil
Dept. of Computer Science
North Carolina State University, Raleigh, NC
ecsezer,pning,ckil@ncsu.edu

Jun Xu
Google, Inc.
Mountain View, CA
jxu3@ncsu.com

ABSTRACT

Software vulnerabilities have been the main contributing factor to the Internet security problems such as fast spreading worms. Among these software vulnerabilities, memory corruption vulnerabilities such as buffer overflow and format string bugs have been the most common ones exploited by network-based attacks. Many security countermeasures (e.g., patching, automatic signature generation for intrusion detection systems) require vulnerability information to function correctly. However, despite many years of research, automatically identifying unknown software vulnerabilities still remains an open problem.

In this paper, we present the development of a security debugging tool named *MemSherlock*, which can automatically identify unknown memory corruption vulnerabilities upon the detection of malicious payloads that exploit such vulnerabilities. MemSherlock provides critical information for unknown memory corruption vulnerabilities, including (1) the corruption point in the source code (i.e., the statement that allows the exploitation of memory corruption vulnerability), (2) the slice of source code that helps the malicious input to reach the corruption point, and (3) the description of how the malicious input exploits the unknown vulnerability. We evaluate MemSherlock with a set of 11 real-world applications that have buffer overflow, heap overflow, and format string vulnerabilities. The evaluation results indicate that MemSherlock is a useful tool to facilitate the automatic vulnerability analysis process.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: Security and Protection; D.4.6 [**Operating Systems**]: Security and Protection—*invasive software*

*The authors would like to thank Prachi Gauriar and Keith Irwin for their valuable discussions and James Newsome and Dawn Song for sharing TaintCheck. This work is supported by the U.S. Army Research Office under the Cyber-TA Research Grant No. W911NF-06-1-0316.

General Terms

Security, Experimentation

Keywords

Vulnerability analysis, Debugging, Memory corruption

1. INTRODUCTION

Software vulnerabilities have been the main contributing factor to the Internet security problems such as fast spreading worms. Among the software vulnerabilities, memory corruption vulnerabilities such as buffer overflow and format string have been most commonly exploited by network-based attacks.

There have been attempts to retrofit legacy code to prevent memory corruption and guarantee memory safety, as represented by CCured [17,18]. However, these approaches require porting, and are not automated fully. Furthermore, due to the conservative memory protection, the additional instrumentation imposes permanent non-negligible performance overhead. For example, CCured requires annotation of program with pointer qualifiers, and introduces in the worst case 87% performance overhead in its evaluation [17]. Thus, identifying and removing software vulnerabilities is still an attractive option to provide software security.

Many security countermeasures have been proposed to remove software vulnerabilities once they are identified. Patching has been adopted by almost all mainstream operating systems and applications, such as Microsoft Windows, Linux, Mac OS, and Microsoft Office, to remove newly discovered vulnerabilities. Moreover, Shield [27] was developed to provide temporary protection of vulnerable systems after the vulnerabilities are identified but before patches are properly applied. Recently, a filtering technique was developed to defend against (polymorphic) exploits of known vulnerabilities [9], and automatic generation of vulnerability-based signatures (for known vulnerabilities) was also investigated [1]. All these approaches require specific vulnerability information in order to function correctly.

There have been many years of research efforts to identify software vulnerabilities automatically. Static analysis techniques have been applied to find potential software vulnerabilities (e.g., [2,3,8,14]). However, most static analysis techniques tend to generate a large number of false positives without guaranteeing the detection of all vulnerabilities.

Dynamic approaches have also been investigated. In particular, several dynamic approaches have been proposed recently to detect exploits of (unknown) vulnerabilities (e.g., address space randomization [10, 21], TaintCheck [20], Minos [6]), analyze such exploits (e.g., DACODA [7], COVERS [15]), and sometimes recover from such attacks (e.g., DIRA [25], STEM [23], [29]). However, despite the detection of potentially unknown attacks, most of such approaches cannot give precise information of the exploited vulnerabilities. One exception is [29], which identifies the corruption points used by exploits of unknown memory corruption vulnerabilities through back tracing from the program crash point [29]. However, as indicated in [29], this method can handle special cases only, and does not guarantee the identification of the corruption point in general. Moreover, it does not give specific information about the exploit of unknown memory corruption vulnerabilities either. As a result, it may still take hours or days of manual effort to understand and patch the unknown vulnerabilities being exploited.

In this paper, we present the development of a security debugging tool named *MemSherlock*, which is aimed at automatically identifying unknown memory corruption vulnerabilities upon the detection of malicious payloads that exploit such vulnerabilities. MemSherlock provides three pieces of information for unknown memory corruption vulnerabilities: (1) the corruption point in the source code (i.e., the statement that allows the exploit of memory corruption vulnerability), (2) the slice of source code that helps the malicious input to reach the corruption point, and (3) the description of how the malicious input exploits the vulnerability.

Unlike previously proposed methods (e.g., [6,7,15,20,29]), MemSherlock detects memory corruption of not only control flow data (e.g., return addresses), but also non-control data (e.g., local variables). This feature is critical in detecting non-control-flow attacks, such as those identified in [4]. Moreover, MemSherlock automatically analyzes the vulnerability that leads to the memory corruption, and outputs the vulnerability information at the *programming language level*, with variable names and line numbers involved in the vulnerability in source code as well as the connection between them. Such information is presented in an intuitive way to the programmer to facilitate the understanding and patching of the vulnerability. Finally, MemSherlock keeps a mapping for the entire virtual memory, providing monitoring at multiple levels of granularity.

We evaluate the security debugging tool with a set of 11 real-world applications with known vulnerabilities, including stack overflow, heap overflow, and format string vulnerabilities. MemSherlock is able to identify all but one of the vulnerabilities with very few false positives. It is important to note that the false negative and false positives are due to the limitation of the proof-of-concept implementation, not the proposed method.

The contribution of this paper is three-fold. First, we develop a suite of source code rewriting, static analysis, and dynamic monitoring techniques to provide automated debugging of unknown memory corruption vulnerabilities. Second, we implement the proposed techniques as a security debugging tool, MemSherlock, which allows automated and efficient identification of unknown memory corruption vulnerabilities in real-world applications. Third, we perform substantial experimental evaluation of MemSherlock using a set of real-world applications, demonstrating the feasibility of this approach.

The rest of the paper is organized as follows. The next section gives an overview of the proposed approach. Section 3 discusses pre-debugging phase preparation for MemSherlock. Section 4 describes the debugging process aimed at identifying the memory corruption vulnerabilities. Section 5 presents the implementation of MemSherlock. Section 6 gives the experimental evaluation of MemSherlock using a set of real-world applications. Section 7 discusses related work. Section 8 concludes this paper and identifies several future research directions.

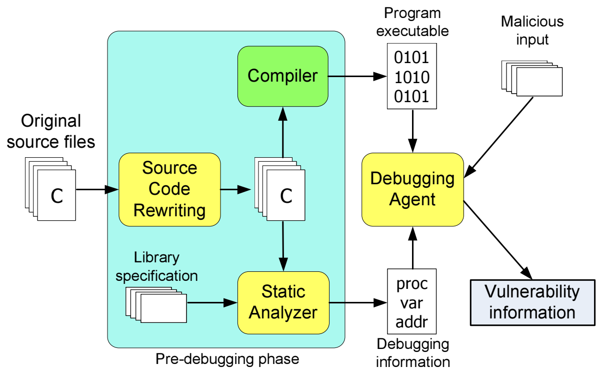

Figure 1: Overview of MemSherlock

2. OVERVIEW OF MEMSHERLOCK

The goal of MemSherlock is to assist programmers in understanding and patching unknown memory corruption vulnerabilities by automatically detecting and providing information about such vulnerabilities. We concentrate on memory corruption vulnerabilities in network service programs (e.g., httpd, ftpd) in this paper, since they are the primary targets of network-based attacks (e.g., worms).

To identify memory corruption, we take advantage of an observation made in [30]. That is, in most programs, a given variable typically is accessed by only a few instructions (or the corresponding statements in the source code). This observation can be further extended in the context of memory corruption attacks: in order for a memory corruption attack to succeed, an attacker needs to use an instruction (in the victim program) to modify a memory region onto which the instruction should not write. To exploit this observation, we keep track of memory operations during the debugging process, and verify whether an instruction writes to a memory location that it is not supposed to modify. Specifically, we determine the memory regions and associate with each of them a set of instructions that can modify it. For a given memory region m, the set of instructions that can modify m is called the *write set* of m, denoted $\text{WS}(m)$.

Figure 1 illustrates the procedure for using MemSherlock. MemSherlock requires a pre-debugging phase to collect the information needed for security debugging. In particular, it needs to collect the write set of each critical memory region. It is non-trivial to obtain such write sets and track the write operations during debugging, particularly due to the complications caused by pointers and complex program constructs. As illustrated in Figure 1, during the pre-debugging phase, MemSherlock first performs source code rewriting to handle pointers and complex program constructs, then uses static analysis of source code to collect information necessary for debugging (e.g., write set information), and finally invokes static analysis of binary code to associate the collected information with memory locations.

Once invoked for debugging, MemSherlock takes as input the instrumented version of the program, the auxiliary debugging information (e.g., the variables in the program along with their sizes and their write sets) generated during the pre-debugging phase, and malicious network payloads. During the debugging process, MemSherlock verifies the modifications to memory regions with the write set information, and identifies an illegal write when the updating instruction is not in the write set. Thus, MemSherlock can capture memory corruption at the time of the modification, pinpointing the exact instruction or statement in source code that is responsible for the corruption.

In addition to the above verification, MemSherlock also keeps track of the propagation of input data as well as the program instructions involved in the propagation. As a result, upon the detection of memory corruption, MemSherlock can identify precisely parts of the program involved in the propagation of the malicious input and determine how the malicious input lead to the memory corruption. By further integrating the auxiliary information collected during the pre-debugging phase, MemSherlock presents all the vulnerability information at source code level to facilitate the understanding and patching of the vulnerabilities.

One critical input to MemSherlock is malicious network payloads that exploit memory corruption vulnerabilities. We assume the method used in [16, 29] to capture such data. For example, we may run network service applications using address space randomization (e.g., PaX ASLR [21]), and log the messages to the service programs in a *message log.* (Note that the logged messages can be discarded upon the completion of a non-crash session.) When a memory corruption attack (e.g., a new worm) attempts to exploit an unknown vulnerability in such a service program, it typically causes the corresponding process to crash [10,21], which triggers the automated debugging of the vulnerability. We then run the instrumented version of the service program under MemSherlock, with the logged network messages replayed to replicate the error and obtain the vulnerability information.

Though based on the same observation as AccMon [30], MemSherlock differs from AccMon in several ways. AccMon relies on a training phase to collect the access instructions for the monitored objects, and offers no guarantee of collecting all access instructions. Indeed, missing instructions will result in false alarms during access monitoring. In contrast, MemSherlock uses static analysis combined with dynamic monitoring to get precise write set information, not suffering from the same problems. Moreover, AccMon requires hardware architectural supports, such as iWatcher [31] and Check Look-aside Buffer (CLB) [30], which are not available in current computer systems. AccMon uses Bloom filter to implement the CLB, and may introduce false positives in recognizing normal instructions, which imply false negatives in detecting memory related bugs. This gives a malicious attacker an opportunity to bypass detection. In contrast, MemSherlock assumes existing hardware and software supports in modern computer systems, and does not suffer from the same false negative problem.

3. GENERATING WRITE SETS

The primary objective of the pre-debugging phase is to generate the write sets of memory regions used by applications. In this phase, we need to determine all program variables and extract their write sets. Moreover, we need to provide information for the debugging agent so that during the debugging phase, it can link memory regions to program variables and their write sets. A particular challenge in this phase is handling pointers and certain dereferences (e.g., chained dereferences and `struct`).

3.1 Extracting Write Sets from Source Code

We perform source code analysis to determine all the program variables and extract their write sets. The write set of a variable `v` includes statements that assign `v` or library function calls where `v` is passed as a modifiable argument (e.g., `memcpy(v, src)`). To facilitate this process, we provide the static analyzer with not only the source code, but also a specification file for every shared library linked to the program. The specification file includes the names of library functions that modify their arguments and identifies the modified arguments. An entry in the write set is a pair consisting of a file name and a line number. We believe that using line numbers is a reasonable approximation to using instructions. As an immediate benefit, this method provides information directly at the source code level. The static analyzer also determines the size of the variables, and for local variables, the function they appear in. Such information will be used by MemSherlock during the debugging phase.

3.1.1 Handling Pointers

Pointers require some special attention, since given a pointer variable, the statements that modify the pointer variable and those that modify the pointer's referent object modify two different memory regions. To address this issue, we keep two separate write sets for every pointer variable `p`: One for the pointer variable itself (WS(`p`)), and the other for the referent object ref(`p`) (WS(ref(`p`))). Note that a pointer may point to different objects during the course of execution. During the debugging process, when the referent object `ref(p)` is determined, the debugging agent adds WS(ref(`p`)) to the referent object's write set.

Note that WS(ref(`p`)) represents the write set of `p`'s referent object possibly updated through pointer `p`. Thus, when `p` is updated, for example, to point to a different object, WS(ref(`p`)) should be removed from the write set of the referent object to which `p` previously pointed, since it is no longer possible to update this object through pointer `p`.

```
1 int i = 0;
2 int *p = &i;
3 *p = 1;
4 p = NULL;
```

(a) Code example

```
WS(i) ={1}
WS(p) ={2,4}
WS(ref(p)) ={3}
```

(b) Write sets after static analysis

Line	ref(p)	WS(i)
1	N/A	{1}
2	i	{1,3}
3	i	{1,3}
4	NULL	{1}

(c) ref(p) and WS(i) during monitoring

Figure 2: Example illustrating model of pointers during static analysis and security debugging

Figure 2 shows an example of the write sets of pointers and their referent objects. We can see that after static analysis, WS(`i`) contains the instruction on line 1 and WS(ref(`p`)) contains only line 3. Note that ref(`p`) remains unresolved during static analysis. During security debugging, however, `p`'s value is updated on lines 2 and 4. At these points, we can see that ref(`p`) is resolved to `i` and `NULL`, respectively. During the execution of lines 2 and 3, while `p` points to `i`, WS(`i`) changes to include the instructions in WS(ref(`p`)).

However, once p's referent object changes to NULL on line 4, WS(i) goes back to its original value.

Since we use the debugging agent to determine dynamically when a pointer variable is updated and find the corresponding memory region, we can avoid pointer alias analysis during static analysis. Indeed, general pointer alias analysis is known to be an undecidable problem [13, 22]. Our approach allows us to bypass it without sacrificing the analysis accuracy. We will discuss the details of pointer updating and tracking in Section 4, since it occurs during debugging.

3.1.2 Handling Chained Dereferences

Chained dereferences make it difficult for the debugging agent to track the memory writes and verify the write set constraint. Examples of chained dereferences include `**p`, `array[1][2]` and `*(p+q)`. We use source code rewriting to transform chained dereferences to simple ones so that the techniques discussed in Section 3.1 can be applied. We perform this transformation only if the expression potentially is updated. For example, `x = var.arr[5]->name` need not be transformed since the modified variable `x` is already in a simple form and `var.arr[5]->name` is not updated at all.

```
1 int z;
2 int *y = &z;
3 int **x = &y;
4 **x = 10;
```

```
1 int z;
2 int *y = &z;
3 int **x = &y;
4 int *temp = *x;
5 *temp = 10;
```

Figure 3: Example of chained dereference

Figure 3 shows an example of chained dereferencing on the left. The static analyzer models the chained dereference on line 4 as a simple dereference, and adds line 4 to WS(ref(x)). This is because we model variable updates as low-level write instructions, which do not have any access to type information. However, we cannot determine the number of dereferences that have occurred in calculating the final target address of a write instruction. Thus, at line 4, the agent is unable to determine the relationship between `x` and `z`. When it detects a write to `z`, it first checks if line 4 is in WS(`z`) and then WS(`y`). Both checks fail, since line 4 is only in WS(ref(`x`)) and `z` is not the referent of `x`.

To handle such chained dereferences, we use automatic source code rewriting. Any chained dereference can be translated into simple dereferences by introducing one or a few temporary variables. For a chained dereference of the form $*X$, we declare a temporary variable t whose type is that of X and assign the value of X to t. In the above example, we can replace line 4 with lines 4 and 5 on the right in Figure 3.

After the transformation, upon executing line 4, the debugging agent sets `temp`'s referent to `z` and adds `temp` to `z`'s list of references. When line 6 attempts to write to `z`, the debugging agent determines it as a legitimate write, because `temp` is one of `z`'s references and line 6 is in WS(ref(`temp`)).

3.1.3 Separating `struct` *fields*

Another complication with C is in dealing with `struct` constructs. Modeling a `struct` variable as a single memory region can introduce false negatives. An instruction that operates on one field could illegitimately modify another field without being detected. This may happen since the instruction is in the `struct` variable's write set and is therefore considered as a legitimate instruction. For example, in the code segment shown in the left part of Figure 4, `strcpy` overflows the `str` field and writes into the `num` field.

```
typedef struct {
    char str[4];
    int  num;
} entry;

1 int main() {
2   entry var;
3   strcpy(var.str,
           "Hello");
4 }
```

```
typedef struct {
    char str[4];
    int  num;
} entry;

1 int main() {
2   entry var;
3   char* temp;
4   temp = var.str;
5   strcpy(temp,
           "Hello");
6 }
```

Figure 4: Example of `struct` field dereference

In order to solve this problem, we need to treat each field in a `struct` as a separately monitored memory region. Once again, we turn to source code rewriting to generate individual memory regions for each field in a `struct`. A field reference of a `struct` is considered a dereferencing itself. We replace every field expression with a temporary variable of the same type. For the example shown in Figure 4, MemSherlock adds a temporary variable of `char *` type and assigns it `var.str` before line 4. Line 4 is then added to the WS of `temp`. When the statement in line 5 overflows the buffer and modifies the memory region of `num`, the debugger can detect the overflow and raise an error, since `temp` points to the memory region of `str` and not of `num`.

The current implementation of MemSherlock treats an array or a `union` as a single memory region. Therefore, arrays of `struct`s or `struct`s within `union`s cannot be handled in the same way. This prevents MemSherlock from capturing overflows from one field (or element) to another. In practice, we have not observed any false negatives due to this limitation. We will discuss more implementation details on `struct`s in section 5.2, and point out the possible false positives and false negatives that may arise in Section 6.

3.2 Mapping Variables to Memory Regions

The aforementioned static analysis at the source code level allows us to extract write sets of variables. To facilitate the debugging process, we have to provide additional information to the debugging agent so that it can associate the variables with memory regions and identify the write sets of those memory regions during debugging time.

We perform binary analysis to determine the location of memory regions corresponding to variables. This is trivial for global variables, since global variables are assigned static addresses after compilation. Local variables, however, have dynamic addresses depending on when the functions containing the local variables are called. To address this issue, we use the addresses of functions that contain the local variables and their frame pointer offset values to identify local variables. The debugging agent can use these values and the actual function calls to calculate the real memory addresses during debugging. The binary analysis also provides information about segment sizes and locations as well as the addresses of functions.

3.3 Output of Pre-debugging Phase

The information gathered during the pre-debugging phase is written to a text file and passed to the debugging agent. The pre-debugging phase need be done only once per program. The file contains a listing of variables along with their write sets in the form of file name and line number pairs. It contains additional information about variables to simplify the debugging process. Variables are distinguished as global or local. For local variables, we also output the variable's enclosing function. In addition, pointer variables and formal parameters that are pointers are flagged as such. Finally, line numbers in a pointer's write set are marked with a flag if the statement modifies the referent instead of the pointer. This enables the debugging agent to divide the write set of a pointer variable into two separate write sets for the pointer and its referent, respectively.

4. DEBUGGING VULNERABILITIES

During the debugging phase, MemSherlock monitors the program execution to detect memory corruption and infer vulnerability information. As discussed earlier, MemSherlock verifies the modifications to memory regions with the write set information, and identifies an illegal write when the updating instruction is not in the write set. To accomplish automated vulnerability analysis, the MemSherlock debugging agent needs to perform three primary tasks:

1. *State Maintenance:* Keep track of the memory regions along with their write sets as program executes. This is necessary, because the active memory regions and their write sets change as program executes.
2. *Memory Checking:* Track and verify memory update operations to detect memory corruption.
3. *Vulnerability Extraction:* Generate vulnerability information once a memory corruption is detected.

In the following, we first discuss a few key data structures, which will be used in the later discussion, and then explain the three primary tasks in detail.

4.1 Key Data Structures

The MemSherlock debugging agent uses several key data structures. For each monitored memory region `r`, the agent creates a `MemoryRegion` object `m`, which stores `r`'s address, size and write set. Additionally, the `MemoryRegion` object for each pointer `p` stores WS(ref(`p`)) and a pointer to its referent object's `MemoryRegion`. At any time during debugging, MemSherlock maintains all the active memory regions. For the sake of presentation, we collectively refer to these memory regions as `ActiveMemoryRegions`, though our implementation manages global variables, local variables, and heap-allocated variables separately for performance reasons. For each user-defined function in the executable, a `Procedure` object is created to store the function's name, its address in the code segment, and a list of `MemoryRegion`s corresponding to its local variables. For local variables, their addresses are stored as frame pointer offsets.

Certain memory regions, such as a function's return address or saved registers, should never be written by source-level instructions[1]. These memory regions, along with meta data adjacent to dynamically allocated memory regions and segments in the virtual memory that do not have write permissions (e.g., code segment, kernel space), are stored in `NonWritableRegions`.

Since functions may be called recursively, MemSherlock maintains a `ProcedureStack`, a stack of `Procedures` whose elements correspond to the user-level functions currently on the execution stack. Maintaining this stack is necessary to ensure that MemSherlock can monitor local variables correctly when there are several instances of the same function on the execution stack.

[1]MemSherlock begins monitoring these regions after the frame pointer is set. Therefore, they should not be updated until the function returns

These key data structures facilitate the MemSherlock debugging process. In particular, state maintenance actions update the data structures so that the current state of execution is reflected accurately, while memory checking actions ensure that only legitimate write instructions are executed.

4.2 State Maintenance

It is necessary to maintain the list of active memory regions and their current write sets at any time of program execution. MemSherlock updates its internal data structures at certain runtime events. For example, when a function call is made, the local variables of the function should be added to the list of monitored memory regions and their write sets should be generated accordingly.

We discuss the critical events and the corresponding state maintenance in detail below:

Pointer Value Updates and Pointer-Type Function Arguments: When an update to a pointer variable `p` with the address of a `MemoryRegion` m is detected, the MemSherlock debugging agent first determines the new referent object by searching through `ActiveMemoryRegions`. The referent pointer of m is set accordingly if m is found. If the new referent cannot be matched to a monitored memory region, this implies that there could be a potential dangling pointer or misuse of a pointer.

Function Calls and Returns: When a user-defined function is called, MemSherlock pushes a `Procedure` record associated with this function onto `ProcedureStack`. MemSherlock then calculates the real addresses of its local variables by adding their offsets to the current frame pointer. The function's return address, the saved frame pointer, and any padded regions between local variables are added to `NonWritableRegions`, enabling MemSherlock to capture illegal writes to these regions. (Note that the static analysis performed in the pre-debugging phase does not provide sufficient information about these memory regions.) This is especially useful in detecting stack buffer overruns. When the function returns, MemSherlock pops the corresponding `Procedure` record off the `ProcedureStack`, and removes its return address, the saved frame pointer, and the padded regions from `NonWritableRegions`.

MemSherlock uses the knowledge of frame pointers. Therefore, the above operations can be done once the frame pointer is set, rather than when a `call` instruction is executed. A side benefit is that all the memory writes that take place to initialize a function's activation record on stack do not cause alarms, since the memory regions are not added yet to `ActiveMemoryRegions`. When the frame pointer is initialized, MemSherlock looks for any pointer-type formal-in arguments. If the procedure has any, MemSherlock reads them and determines the referent object in the same way as when a pointer is updated. Again, this allows the static analysis performed in the pre-debugging phase to be fairly simple, since the MemSherlock debugging agent takes care of inter-procedural dependencies.

Heap Memory Management: When a heap memory region is allocated using the `malloc` family of functions, MemSherlock creates a new `MemoryRegion` object and adds it to `ActiveMemoryRegions`. In addition, any memory manager meta data adjacent to the block is added to `NonWritableRegions`. This not only ensures that the meta data is protected, but also facilitates the detection of heap buffer overruns.

When `free` is invoked on a memory region, MemSherlock first checks that the region is in `ActiveMemoryRegions`. If so, MemSherlock frees the region, and removes the corresponding `MemoryRegion` record from `ActiveMemoryRegions`. Otherwise, MemSherlock generates an error message, indicating that the program has tried to free a non-heap allocated region, which might indicate a double free error.

Dynamically Linked Libraries: MemSherlock keeps track of memory regions allocated for shared libraries. Each shared library has an executable region and a read-only region. In some cases, libraries also have `.bss` sections. Shared libraries can be loaded to arbitrary locations in the virtual memory and their location is determined at runtime. Another feature is lazy binding, which loads a library only when a function from that library is called. MemSherlock reads the memory region information from the process map in the `proc` file system, and can infer that a new library has been loaded while performing write checks. It then checks if the global offset table (GOT) has been modified, and reads the map file only if GOT has changed. MemSherlock groups library regions with respect to their permissions and allows a library to modify any writable regions of the library. We discuss the ramification of this simplification in section 6.3.

4.3 Memory Checking

When a memory write to an address `addr` occurs, MemSherlock searches through `ActiveMemoryRegions` to look for a `MemoryRegion` that *covers* `addr` (i.e., `addr` falls in this `MemoryRegion`). Moreover, MemSherlock also searches for pointer-type `MemoryRegion`s pointing to such a `MemoryRegion`. Once found, MemSherlock verifies that the write instruction's address is in the WS of this `MemoryRegion`, or in one of the memory regions whose pointers point to it. Note that this implies that the memory region m for a pointer-type variable `p` can be verified in two ways. If the destination address is in m then the membership is checked for WS(`p`). If the destination address is in the referent object's memory region then the membership is checked for WS(ref(`p`)).

If MemSherlock cannot find a `MemoryRegion` corresponding to the write destination address `addr`, it will perform the same search in `NonWritableRegions`. If a match is found, this means the write instruction is trying to corrupt a non-writable region, and MemSherlock emits an error message.

If the destination address does not match any of the entries in `ActiveMemoryRegions` or `NonWritableRegions`, there are several things that could be happening. Depending on the program counter (PC) and the destination address, this could be a `call` instruction pushing values onto the stack. Since the frame pointer of the callee is not set, we do not monitor its memory regions at this time. Second, it could be a library function writing onto its stack. MemSherlock keeps track of the lowest memory address of client function activation records for this purpose. Third, this could be a library function writing to dynamically allocated memory.

Because the static analyzer outputs write sets as file name and line number pairs, the PC must be translated into a file name and line number before performing a write check. The translation from instruction address to file name makes library functions a challenge. When a library function is called, the execution jumps to the shared library memory region where multiple function calls may occur. When this occurs, the PC is an instruction address in the shared library region for which no source code is available. In order to address this problem, the agent needs to find the call site of the library function in the user code. This can be done either through a stack walk or by keeping track of the last jump instruction. Once the original call site has been recovered, the memory write check can proceed as describe before.

4.4 Generating Vulnerability Information

Unlike most other memory level monitoring tools, MemSherlock detects memory corruption at the time of memory write. This enables MemSherlock to pinpoint the exact statement in the source code responsible for the corruption. In many cases, just knowing the point of corruption is sufficient to determine the vulnerability. For example, most programmers look for a buffer overflow when the problem statement is a `strcpy`. However, to provide more vulnerability information, MemSherlock incorporates the taint analysis from TaintCheck [20] to check if the value written to the destination address during the corruption is tainted. If so, MemSherlock performs additional analysis to report the source of the tainted data (e.g., network packet) and a dynamic slice of the source code that propagated the tainted data. The programmer can see how the tainted data is introduced and causes the vulnerability to be exploited.

As described earlier, MemSherlock keeps a close watch on memory regions and operations performed on them. In return, MemSherlock can determine the memory region being modified and the program variable to which the memory region corresponds. It also determines if the memory region was updated through the use of the variable or dereferencing of a pointer variable. This greatly simplifies the analysis of the vulnerability, since the programmer does not have to iterate through the call stack and pointer aliasing to determine the original memory region being modified.

When generating the dynamic program slice for the exploited vulnerability, MemSherlock uses the Taint data structure from TaintCheck. Every tainted memory region is associated with a Taint data structure. When the taint is propagated to a new memory region, a new Taint data structure is created. This data structure stores the instruction that propagated the taint, the tainted memory address, the current execution stack, and a reference to the Taint structure of the source memory regions. By using these data structures, particularly the execution stacks, MemSherlock can identify the part of the program that propagated malicious network input, the involved memory regions, and the dependency among the memory regions.

When generating the output of the analysis, we can highlight the statements in the source code (through the translation from instructions to file name and line number pairs), and associate these statements with the memory regions involved in the exploit. Figure 6 in Section 6.1 shows an example of the output, using one of our test cases.

In most cases, checking whether the value being written is tainted is sufficient. One exception is when a tainted value is used as a size argument during memory allocation. One of our test cases (Null HTTPD) has such an overflow vulnerability, where a user-provided value is used in calculating the size for a heap buffer, which is then overflowed. In this example, the influence of the tainted data is indirect. In order to deal with such indirect effects, we use source code rewriting and a functionality of Valgrind that allows it to receive client calls from the client program during debugging. We modify the source code such that every time a user variable is passed as a size argument to a `malloc` family of function call, a client call is made to the debugging agent to inform the memory location of the variable. When a buffer overflow occurs, we not only check if the value is tainted but also if the size used during allocation is tainted as well.

5. IMPLEMENTATION

MemSherlock is implemented as two pre-debugging tools and a security debugging agent. In this implementation, we try to reuse existing software as much as possible to reduce the development cycle. In the following, we present the implementation details.

5.1 MemSherlock Preprocessing Tools

In the current implementation, MemSherlock uses two programs to facilitate the pre-debugging phase: *SrcRewrite* performs source code rewriting, as discussed in Section 3.1.2, while *WriteSetGen* performs source code and binary static analysis to generate write sets and auxiliary information.

5.1.1 Source Code Rewriting via SrcRewrite

SrcRewrite uses C Intermediate Language (CIL) to rewrite source code files. The CIL executable *cilly* supports an OCaml scripting interface that allows users to define their own rewriting rules. *cilly* supports rewriting at different levels such as per function, per statement and even per l-value. We use statement based rewriting, which allows us to define new temporary variables and insert new statements or alter the existing one.

For every l-value and function argument that is a chained dereference or a reference to a field of a `struct`, we insert a temporary variable *temp* of compatible type and insert an assignment statement that sets *temp* to the expression. The l-value is then replaced with *temp* in the assignment statement.

For each library, *SrcRewrite* takes as input a specification file for the library which lists the functions that modify their arguments and the argument numbers that are modified. *SrcRewrite* rewrites all such arguments in the source code and converts them to simple dereferences if necessary.

Finally, in order to aid MemSherlock in security debugging phase, *SrcRewrite* inserts client calls into the source code whenever a `malloc` family of function call is passed a non-static size (i.e., variable or expression). Expressions are first transformed to l-values. This client call informs the debugger of the location of the variable being used as the size, which in turn is checked for taintedness.

5.1.2 Generating Write Sets via WriteSetGen

During the pre-debugging phase, *WriteSetGen* performs static analysis in two steps to produce the information for its debugging agent. In step 1, *WriteSetGen* uses Code Surfer [5], a commercial static analysis tool, to identify variables and their write sets. Code Surfer analyzes a program and creates its own internal data structures, including data and control-flow dependency graphs. It is equipped with a scripting interface that allows users to access these internal data structures. *WriteSetGen* uses a script to determine the variables and their legitimate write sets automatically. This script also contains specifications for standard library functions that potentially modify their arguments just as in *SrcRewrite*). When such a function is called with a monitored variable as a potentially modified argument, *WriteSetGen* adds the call site to the variable's legitimate write set.

The Code Surfer script outputs a text file listing every variable along with its WS in the form of file and line number pairs. Variables are distinguished as global or local. For local variables, it also outputs the variable's enclosing function. In addition, pointer variables, pointer type formal parameters and `struct` variables are flagged as such. Finally, line numbers in a pointer's WS are marked with a flag if the statement modifies the referent object instead of the pointer. This enables the MemSherlock debugging agent to distinguish between the write sets of the pointer and its referent object.

In step 2, *WriteSetGen* analyzes the program executable to determine the global variable addresses, function addresses, and offsets of local variables. To further facilitate the debugging process, we compile the code using debugging flags, and use the `dwarfdump` tool to determine the addresses for global variables and functions, and offsets for local variables. `dwarfdump`'s output effectively provides a mapping from program source variables to their corresponding runtime memory locations. *WriteSetGen* includes a pre-debugging script written in Ruby to parse `dwarfdump`'s output and combine the relevant data with Code Surfer's output to produce the final input file for the debugging agent. The script also splits `struct` variables into its fields, and outputs individual memory region information for each field as well as its WS.

5.2 MemSherlock Debugging Agent

The MemSherlock debugging agent is implemented as an extension to Valgrind [19], which is an open-source CPU emulator for x86 architectures that provides facilities to monitor all aspects of program execution, including memory writes, memory allocation events, function calls and system calls. The MemSherlock debugging agent is implemented as a Valgrind skin that logs all memory operations, monitors memory regions, and performs checks to ensure that only legitimate memory writes occur. When a program is executed under Valgrind, the binary code undergoes certain transformations one basic block at a time. In our implementation of the MemSherlock debugging agent, we instrument the basic blocks with calls to our own functions when certain events are observed, including function calls, returns, memory writes, and system calls. MemSherlock also incorporates the taint analysis from TaintCheck [20]. This information is used in extracting the dynamic slice of the program responsible for the vulnerability.

Another implementation detail worth mentioning is the way MemSherlock handles `struct` type pointer assignments. MemSherlock assumes `MemoryRegions` are non-overlapping. Moreover, to capture overflows from one `struct` field onto another, we represent these fields as individual memory regions. The challenge then, is to determine whether a pointer points to the field itself or the entire `struct` variable. Essentially, the two are differentiated by a flag associated with the pointer variable during pre-debugging static analysis. During memory write checking, when this pointer is analyzed, MemSherlock first checks the `MemoryRegion` of the variable itself, and then its referent object's `MemoryRegion`, just like

```
--20361--
--20361-- Error type: Heap Buffer Overflow
--20361-- Dest Addr: 3AB3E360
--20361-- IP: 0x804E5C7: ReadPOSTData (http.c:108)
--20361-- Dest address resolved to:
--20361-- Global variable "heap var"
          @ 3AB3E280 (size: 224)
--20361--
--20361-- Memory allocated by 0x804E531:
          ReadPOSTData (http.c:100)
--20361--
--20361-- TAINTED destination 3AB3E360
--20361-- Fully tainted from:
--20361--     0x804E5C7: ReadPOSTData (http.c:108)
--20361--
--20361-- TAINTED size used during allocation
--20361-- Tainted from:
--20361--     0x804E456: ReadPOSTData (http.c:100)
--20361--     0x804FBB5: read_header (http.c:153)
--20361--     0x805121B: sgets (server.c:211)
--20361--
```

Figure 5: A typical error message from the debugger

any other pointer variable. In the case when the pointer variable is flagged as a `struct` pointer, the debugger further checks the `MemoryRegion`s of the subsequent fields. Therefore, the `struct` flag determines whether this instruction is allowed to modify the field alone or the entire `struct`.

6. EXPERIMENTAL EVALUATION

We performed a series of experiments to evaluate MemSherlock. In our evaluation, we used 11 real-world applications with a variety of vulnerabilities, along with the attack programs that exploit these vulnerable applications. Table 1 gives the information about these test applications. The first three columns in Table 1 show the list of applications, their vulnerability type and a brief description. Six of the test cases have stack buffer overflow vulnerabilities, three have heap overflow vulnerabilities, and the other two have format string vulnerabilities. It is worth noting that other types of memory corruption attacks rely on these three vulnerabilities. For example, return-to-library attacks are a variation of stack overflows, whereas the malloc-free attack relies on overflowing a heap buffer and corrupting the meta data used by the memory manager.

Table 1 also summarizes the evaluation results, including whether the vulnerabilities are captured and the number of false positives. Moreover, Table 1 presents the false positives in three classes based on their reasons. It is worth pointing out that all the false positives were due to the limitation of our implementation rather than the proposed method. In the following, we describe the evaluation results in detail.

6.1 Automated Debugging

MemSherlock can provide crucial information about exploited vulnerabilities to aid programmers in debugging, signature generation, patching, etc. To demonstrate the depth of information MemSherlock can provide, we use the vulnerability output from Null HTTP as an example.

Figure 5 shows the error message displayed by MemSherlock when NullHTTP's heap is overflowed. The first paragraph displays the location of the error; both the instruction number and the source file location which states that line 108 was responsible for this memory corruption. The error message then provides the destination memory address and the memory region to which it corresponds. In this particular example, since the destination address is the meta data of the heap memory region, the heap memory region that was allocated is shown rather than the meta data's.

Knowing the corruption point, a programmer easily can guess that the `recv` function call is responsible for the overflow. What is not apparent from this information alone is that the reason the buffer is overflowed is not due to an oversized packet alone. The size of the overflowed buffer is calculated from user data, and a negative value provided by the user can cause the buffer to be smaller than expected. The error message states that the buffer was allocated from line 100 in http.c and also performs taint analysis on both the array and the size value that was used during allocation.

MemSherlock produces enough information to detail this vulnerability. A more intuitive display of the vulnerability can be generated by extracting a dynamic slice of the program and presenting it as a graph. Figure 6 shows the fragments from the source code, highlighting the statements involved in the propagation of the tainted data. It includes the critical program steps from the time when the malicious input is introduced to the time of memory corruption.

As highlighted in Figure 6, the function `read_header` calls `sgets`, passing its local variable `line` as an argument. The `sgets` function taints the memory region belonging to `line` through the `recv` library function call. Note that the argument used while calling `recv` is `buffer`. This assignment is captured during the function call to `sgets` and the connection is clearly shown in Figure 6. Once `sgets` returns, the value in `line` is converted into a decimal number at line 153 in `read_header`. This statement propagates the taint into another heap memory region belonging to `conn[sid].dat->in_ContentLength`. The dotted line between the two memory regions show the taint propagation. Later, the tainted heap memory region is used as the size argument in `ReadPOSTData` at line 100, where the `calloc` function call at line 100 creates a new memory region. (Note that TaintCheck itself cannot capture the connection between the tainted size argument and the newly created memory region. MemSherlock uses a Valgrind client call inserted by *SrcRewrite* to capture it.) Finally, with the call to `recv` at line 108, `ReadPOSTData` taints the newly created memory region and also overflows it at the same time, for which the debugger issues the error message.

As illustrated in Figure 6, MemSherlock can simplify the security debugging process greatly by providing the information on how a memory corruption vulnerability is exploited, and thus significantly reduce the time and effort required in understanding and fixing unknown memory corruption vulnerabilities.

6.2 False Positive Analysis

Our evaluation shows that MemSherlock generates very few false positives. We observed a total of 25 false positives in our 11 test applications. Most of them were due to the same implementation limitations manifesting themselves in different locations within the program. We categorize these false positives into three groups, as discussed below.

Embedded Assembly: Code Surfer cannot perform source code analysis on embedded assembly code. This prevents *WriteSetGen* from including the statement in the WS of the variable on which it operates. This missing information causes the debugger to label the memory modification as illegal.

Application Name	Vuln. Type	Description	Captured	#FP (total)	#FP due to asm	#FP due to clib	#FP due to `struct`
GHTTP	S	A small HTTP server	Yes	7	4	2	1
Icecast	S	A mp3 broadcast server	Yes	0	0	0	0
Sumus	S	A game server for 'mus'	Yes	0	0	0	0
Monit	S	Multi-purpose anomaly detector	Yes	0	0	0	0
Newspost	S	Automatic news posting	Yes	2	0	1	1
Prozilla	S	A download accelerator for Linux	No	0	0	0	0
NullHTTP	H	Null HTTP, HTTP server	Yes	0	0	0	0
Xtelnet	H	A telnet server	Yes	4	0	4	0
Wsmp3	H	Web server with mp3 broadcasting	Yes	0	0	0	0
OpenVMPS	F	Open source VLan management policy server	Yes	2	0	2	0
Power	F	UPS monitoring utility	Yes	10	6	4	0

Table 1: List of test applications. Type abbreviations: (S)tack overflow, (H)eap overflow and (F)ormat string.

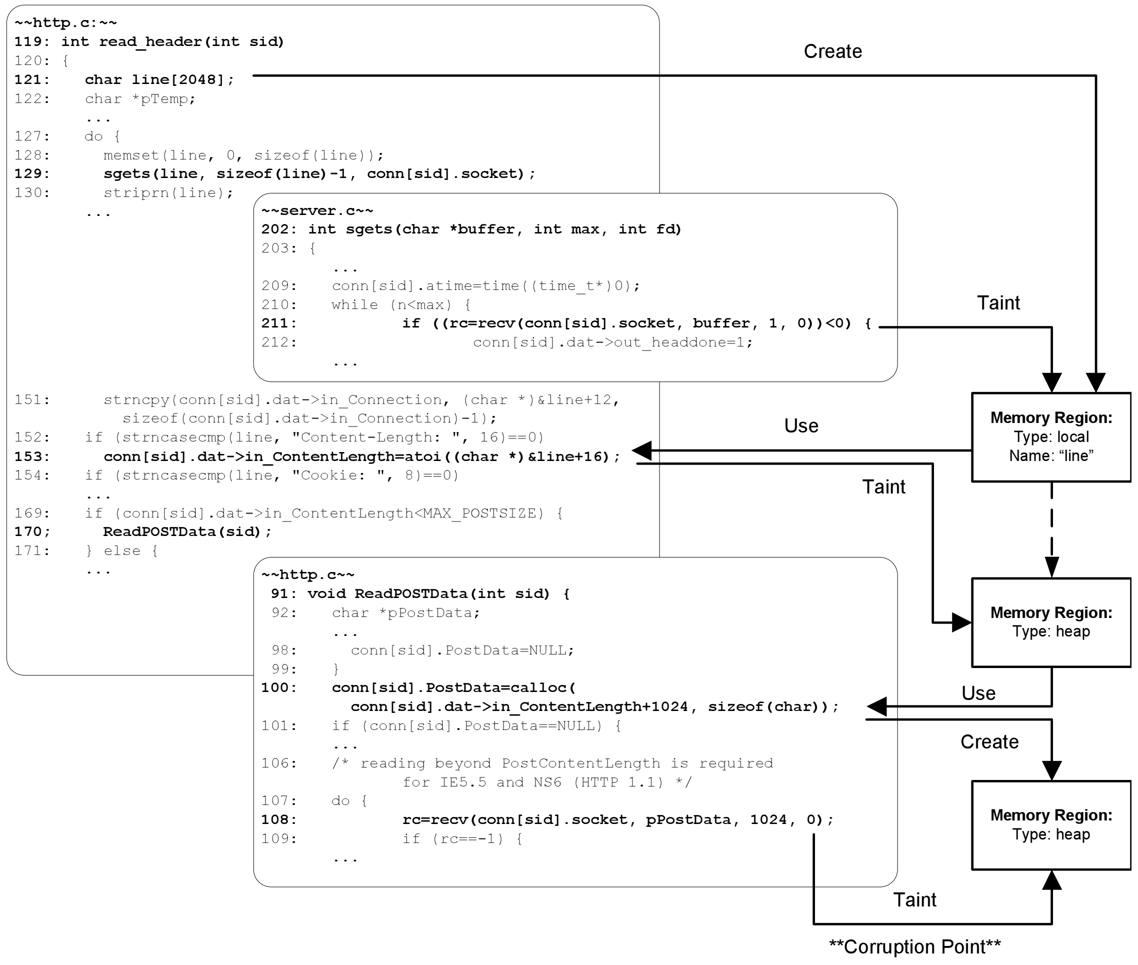
```
~~http.c:~~
119: int read_header(int sid)
120: {
121:    char line[2048];
122:    char *pTemp;
        ...
127:    do {
128:      memset(line, 0, sizeof(line));
129:      sgets(line, sizeof(line)-1, conn[sid].socket);
130:      striprn(line);
        ...
```
```
~~server.c~~
202: int sgets(char *buffer, int max, int fd)
203: {
        ...
209:    conn[sid].atime=time((time_t*)0);
210:    while (n<max) {
211:            if ((rc=recv(conn[sid].socket, buffer, 1, 0))<0) {
212:                    conn[sid].dat->out_headdone=1;
        ...
```
```
151:      strncpy(conn[sid].dat->in_Connection, (char *)&line+12,
            sizeof(conn[sid].dat->in_Connection)-1);
152:    if (strncasecmp(line, "Content-Length: ", 16)==0)
153:      conn[sid].dat->in_ContentLength=atoi((char *)&line+16);
154:    if (strncasecmp(line, "Cookie: ", 8)==0)
        ...
169:    if (conn[sid].dat->in_ContentLength<MAX_POSTSIZE) {
170;      ReadPOSTData(sid);
171:    } else {
        ...
```
```
~~http.c~~
 91: void ReadPOSTData(int sid) {
 92:    char *pPostData;
        ...
 98:      conn[sid].PostData=NULL;
 99:    }
100:    conn[sid].PostData=calloc(
          conn[sid].dat->in_ContentLength+1024, sizeof(char));
101:    if (conn[sid].PostData==NULL) {
        ...
106:    /* reading beyond PostContentLength is required
                for IE5.5 and NS6 (HTTP 1.1) */
107:    do {
108:            rc=recv(conn[sid].socket, pPostData, 1024, 0);
109:            if (rc==-1) {
        ...
```

Figure 6: A graphical representation of the vulnerability in Null HTTP

Incomplete Library Specification: Our testing allowed us to observe certain properties of the C library which require a more expressive specification than the one our current implementation uses. `strtok(char *str, char *delim)` is a C library function that tokenizes a given string. It traverses the string until a delimiter character is reached and returns the pointer to the beginning of the token. The original string can be parsed further by calling `strtok` with a NULL argument. These subsequent calls are not included in the WS of the original string, since the string does not appear as an argument. This results in the memory write being interpreted as illegal during debugging.

Certain library functions modify global variables as side effects. For example, the function that parses the program argument `getopt` returns the argument through the global variable `optarg`. The statements issuing the function calls are not in `optarg`'s WS, causing false positives during debugging. Another example is `errno`, which is used to return the error number throughout the C library.

Finally, some library functions return pointers to global variables that are hidden from the client program through the use of *attribute-hidden*. Examples include `getdatetime`, `gmtime` in `time.h` and `gethostbyaddr`, `gethostbyname` in `socket.h`. When the returned value is assigned to a program variable, MemSherlock fails to find a corresponding `MemoryRegion` and raises a false alarm.

The false positives due to library functions can be prevented by the use of a more expressive library specification.

`struct` Pointers: MemSherlock relies on type information as little as possible. Unfortunately, `struct` pointers is one of the few instances where MemSherlock requires type information supplied through flagging the variable. As mentioned earlier, this allows the debugger to determine if the pointer is pointing to a field or the entire `struct`. Two of our test applications use `void *` type pointers to refer to `struct` variables using explicit type casting. When the `struct` variable is modified through the use of this pointer, MemSherlock raises an error upon the modification of the second and later fields. These false positives can be prevented by checking for such type casting during static analysis and communicating this information to the debugger either through the input file or client calls at debugging time.

6.3 False Negative Analysis

The MemSherlock debugging agent is implemented as a Valgrind skin. Unfortunately, Valgrind is unable to trace into kernel instructions. As a result, our current implementation cannot detect memory region modifications done by kernel instructions. This case is different from system calls which are handled successfully. While experimenting with Prozzilla, we noticed that `vfprintf` makes a call to `mempcpy`, which is identical to `memcpy` except that it returns the destination pointer instead of the number of bytes copied. When copying large chunks of memory that span multiple virtual memory pages, `mempcpy` uses kernel functions to modify the page table. The current implementation of MemSherlock debugging agent is not able to see such memory writes, and failed to detect the memory corruption.

It is easy to prevent such false negatives by writing wrappers for library functions that modify memory without using client program instructions. Newer versions of Valgrind support wrapper functions which can intercept calls to library functions. Such a wrapper function simply would check if the write is legitimate before the actual call to the library is made and perform any pointer related assignments once the function returns.

Although we have not encountered any other false negatives during our experiments, we are aware that the current implementation can cause several types of false negatives. The most obvious one is the use of `struct`s within arrays or unions. Since we currently handle such memory regions as single blocks, we cannot distinguish writes to individual fields. In the case of arrays, it is possible to subdivide the memory region and monitor each element. The problem is that arrays can be quite large, and increasing the number of memory regions to monitor can degrade performance to an unacceptable level. We observe that this problem can be solved by using compressed data structures; however, due to time limitations, we did not include this functionality in our proof of concept implementation.

Another type of false negatives can occur when dealing with chained dereferencing expressions such as $var[i].field[j] = some\_exp$. `SrcRewrite` would convert the l-value into a temporary variable $temp = var[i].field$. Our current implementation fails to detect an illegal write when the index i is out of bounds. Bounds checking on these indices can remove the possibility of false negatives. Inserting bounds checking for arrays can be done during source code rewriting, since the bounds are known. For heap buffers, the bounds checking can be deferred to the debugger via client calls.

Ideally, we would have liked to perform fine-grained monitoring on library regions as well. However, in our current implementation, we treat dynamically linked libraries as grey boxes, in which they are only defined by their specifications. Even though this could potentially cause a false negative, an attack that exploits this shortcoming would have to modify library data alone to succeed.

7. RELATED WORK

MemSherlock is related closely to intrusion detection systems that perform memory level monitoring [20,25,30]. Minos [6] and TaintCheck [20] can detect the improper use of tainted data by tracking the propagation of untrusted data. MemSherlock also incorporates taint tracking; however, it relies on completely different detection mechanisms, and thus can provide vulnerability information that Minos and TaintCheck cannot offer.

Brumley *et al.* [1] recently investigated automatic generation of vulnerability-based signatures. A precondition of their approach is the specification of vulnerability point and vulnerability condition. Moreover, Newsome *et. al.* proposed self-hardening programs [9], which can remove vulnerabilities from the program to make it immune to exploits attacking the vulnerability. MemSherlock can complement these approaches by providing vulnerability information.

A few address space randomization techniques, such as PaX ASLR [21], TRR [28] and ASLP [10], have been proposed to detect memory corruption attacks. Such approaches can be used to trigger the MemSherlock debugging phase.

Network based IDSs such as [11,12,24] can automatically generate signatures for unknown attacks. These systems do not provide vulnerability information, rather extract common syntax from the network packets. However, IPSs relying on such syntactic signatures have been shown to be vulnerable to attacks [26].

8. CONCLUSION AND FUTURE WORK

In this paper, we presented the development of MemSherlock, a security debugging tool that can identify unknown memory corruption vulnerabilities automatically upon the detection of malicious payloads that exploit such vulnerabilities. MemSherlock provides critical information for unknown memory corruption vulnerabilities, including (1) the corruption point in the source code, (2) the slice of source code that helps the malicious input to reach the corruption point, and (3) the description of how the malicious input exploits the unknown vulnerability. We evaluated MemSherlock with a set of 11 real-world applications that have buffer overflow, heap overflow, and format string vulnerabilities. Our results demonstrated that MemSherlock is a useful tool to facilitate the vulnerability analysis process.

Our future work is two-fold. First, we will improve the implementation of MemSherlock to address its implementation oriented limitations, such as the inability to deal with assembly code and the coarse-grained monitoring of memory regions allocated by shared libraries. Second, we will improve the automated analysis and the presentation of the analysis results so that the analysis results are more intuitive and easier to use.

9. REFERENCES

[1] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2006.

[2] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of c code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, February 2004.

[3] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS'02)*, November 2002.

[4] S. Chen, J. Xu, and E. C. Sezer. Non-control-data attacks are realistic threats. In *Proceedings of 14th USENIX Security Symposium*, 2005.

[5] CodeSurfer. http://www.grammatech.com/products/codesurfer/.

[6] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 221–232, December 2004.

[7] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 235–248, 2005.

[8] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizingsensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, May 2004.

[9] D. S. James Newsome, David Brumley. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS '06)*, Feb 2006.

[10] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22st Annual Computer Security Applications Conference (ACSAC '06)*, pages 339–348, December 2006.

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

[12] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.

[13] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[14] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[15] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 213–222, 2005.

[16] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of 12th ACM Conference on Computer and Communication Security (CCS '05)*, pages 213–222, 2005.

[17] G. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transaction on Programming Languages and Systems*, 27(3):477–526, May 2005.

[18] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 128–139, 2002.

[19] N. Nethercote. Dynamic binary analysis and instrumentation, 2004. valgrind.org/docs/phd2004.pdf.

[20] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of The 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, February 2005.

[21] PaX Team. `http://pax.grsecurity.net/docs/aslr.txt`.

[22] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, September 1994.

[23] S. Sidiroglou, M. Locasto, S. Boyd, and A. Keromytis. Building a reactive immune system for software services. In *Proceedings of USENIX Annual Technical Conference*, pages 149 – 161, April 2005.

[24] S. Singh, C.Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, December 2004.

[25] A. Smirnov and T. Chiueh. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *Proceedings of The 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, February 2005.

[26] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 21–30, New York, NY, USA, 2004. ACM Press.

[27] H. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of ACM SIGCOMM*, August 2004.

[28] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of 22nd Symposium on Reliable Distributed Systems - SRDS 2003*, pages 260–269, 2003. IEEE Computer Society, Oct.

[29] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookhold. Automatic diagnosis and response to memory corruption vulnerabilities. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 223–234, 2005.

[30] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In *MICRO 37: Proceedings of the 37th annual International Symposium on Microarchitecture*, pages 269–280, Washington, DC, USA, 2004. IEEE Computer Society.

[31] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient architectural support for software debugging. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*, 2004.