

Automated Security Debugging Using Program Structural Constraints

Chongkyung Kil*, Emre Can Sezer*, Peng Ning*, and Xiaolan Zhang[†]

Department of Computer Science*
North Carolina State University
{ckil, esezer, pning}@ncsu.edu

IBM T. J. Watson Research Center[†]
cxzhang@us.ibm.com

Abstract

Understanding security bugs in a vulnerable program is a non-trivial task, even if the target program is known to be vulnerable. Though there exist debugging tools that facilitate the vulnerability analysis and debugging process, human developers still need to manually trace the program execution most of the times. This makes security debugging a difficult and tiresome task even for experienced programmers. In this paper, we present the development of a novel security debugging tool called CBones (SeeBones, where bones is an analogy of program structures). CBones is intended to fully automate the analysis of a class of security vulnerabilities in C programs, the exploits of which would compromise the integrity of program structures satisfied by all legitimate binaries compiled from C source code. In other words, CBones automatically discovers how unknown vulnerabilities in C programs are exploited based on program structural constraints. Unlike the previous approaches, CBones can automatically identify exploit points of unknown security bugs without requiring a training phase, source code access (analysis or instrumentation), or additional hardware supports. To validate the effectiveness of this approach, we evaluate CBones with 12 real-world applications that contain a wide range of vulnerabilities. Our results show that CBones can discover all security bugs with no false alarms, pinpoint the corrupting instructions, and provide information to facilitate the understanding of how an attack exploits a security bug.

1 Introduction

Understanding security bugs in a vulnerable program is a non-trivial task, even if the target program is known to be vulnerable. Though there are existing debugging tools (e.g., gdb, Purify [21]) to facilitate the vulnerability analysis and debugging process, human developers still need to manually trace the program execution most of times. This makes security debugging a difficult and tiresome task even for experienced programmers.

A critical step in the security debugging process is the analysis that recognizes the vulnerable point in a program and identifies the cause of the bug. A major challenge in this analysis process is to identify how unknown vulnerabilities can be exploited, since these vulnerabilities are often short-lived during the exploit process and thus are difficult to trace. Static analysis techniques (e.g., [24–26]) have been proposed to discover vulnerabilities in a program via source code analysis (e.g., model checking). However, such methods are mainly focused on known vulnerabilities and often require human interventions to write assertions or specifications. In addition, some require access to source code, which is not always available (e.g., commodity software). Runtime checking methods (e.g., [10, 21–23]) have also been studied to detect security bugs dynamically by checking legitimate memory accesses, inserting canary values, or using additional hardware support. These approaches, however, still mostly focus on existing vulnerabilities and suffer from non-negligible false alarms and performance overhead.

A few approaches have been investigated recently to discover security bugs using statistical program invariants [11–13]. These approaches detect bugs by identifying deviations from program invariants statistically derived during normal program executions. DAIKON [12] and DIDUCE [13] automatically extract likely program invariants among variables through multiple normal program executions in the training phase, and use the violations of such invariants to detect security bugs at analysis time. AccMon [11] introduced Program Counter (PC)-based invariant detection (i.e., a memory object is typically accessed only by a small set of instructions), and detects security bugs when memory objects are accessed by instructions not observed during normal program executions in the training phase.

These approaches extended the capability of analyzing unknown security bugs. However, they still suffer from several limitations. In particular, all these approaches require a training phase during normal program executions to derive the program invariants, which offers no guarantee in generating all program invariants. Moreover, each of

these approaches have additional limitations. For example, DAIKON [12] requires access to the program source code, while AccMon [11] requires hardware supports not available on modern computer systems.

In this paper, we identify another type of program invariants called *program structural constraints* (or simply *structural constraints*), which are complementary to the above program invariants. Unlike the program invariants used by the previous approaches (e.g., the variable invariants used by DAIKON [12] and DIDUCE [13]), such program structural constraints are satisfied by all binaries produced by certain software development tools. Moreover, these program structural constraints can be *statically* and *entirely* extracted from program binaries. Thus, no training phase is required, and it is guaranteed that we can extract all instances of program structural constraints.

Based on program structural constraints, we develop an automated security debugging tool named *CBones* (*See-Bones*, where *bones* is an analogy of program structures). CBones automatically extracts program structural constraints from a program binary, and verifies these constraints during program execution to detect and isolate security bugs. Compared with the previous approaches [11–13], CBones provides several unique benefits:

- **Complete automation.** CBones extracts program structural constraints (invariants) via static analysis of the compiled program executable. This has two implications. First, CBones does not require any training phase, which differentiates CBones from most runtime monitoring tools (e.g., AccMon [11], DIDUCE [13], DAIKON [12]). Second, CBones does not require manual specification as an input. This differentiates CBones from most static analysis and model checking tools (e.g., [24–26]).
- **No access to source code or additional hardware is required.** CBones dynamically instruments the program binary using Valgrind [20] during the program executions. Thus, it does not need to access the source code (in comparison with DAIKON [12]), nor does it need additional hardware support (in comparison with AccMon [11]).
- **No false alarms.** Since the program structural constraints should be satisfied by all the binaries produced by the compiler of concern, violation of any of them during runtime indicates a bug. Thus, the proposed approach produces no false alarms. Moreover, as indicated in our experiments, CBones can detect some data attacks recently discovered in [1], which many other approaches fail to recognize.
- **CBones provides both coarse-grained and fine-grained debugging modes,** which can be combined to isolate security bugs efficiently. The coarse-grained mode allows a user to quickly zoom into a region that has a security bug, and the fine-grained mode enables the user

to pinpoint the bug.

To validate the practicality and effectiveness of the proposed approach, we evaluate CBones with 12 real-world applications that have different vulnerabilities. The result shows that CBones can discover all 12 security bugs with no false alarms. Moreover, CBones can pinpoint the corrupting instruction points precisely, which is critical in understanding how an attack exploits a security bug.

The rest of the paper is organized as follows. Section 2 describes the program structural constraints and the debugging techniques based on these constraints. Section 3 discusses the implementation of CBones. Section 4 presents the experimental evaluation of CBones. Section 5 discusses related work, and Section 6 concludes the paper.

2 Security Debugging Using Program Structural Constraints

A process’s virtual address space is divided and used according to the operating system and the compiler with which the program was compiled. Furthermore, each segment of memory is usually logically divided into smaller chunks that represent memory objects in the program or metadata used by the compiler to determine the program state. In this study, we use the structural properties that these memory objects or metadata always satisfy to derive program structural constraints for each program. These constraints can be verified at runtime to ensure that the program complies with the assumptions made by the operating system and/or the compiler. Violations of these constraints thus indicate the existence of security vulnerability.

CBones uses a combination of static analysis and dynamic monitoring to accomplish its goal. Static analysis is used to automatically determine structural constraints for a given program. These constraints are then passed to the dynamic monitoring component which verifies whether the constraints are satisfied during execution. Any violation of these constraints signals a misbehavior, which indicates an attack or a bug. For convenience, we refer to the static analysis portion as *Constraint Extractor*, and the dynamic monitoring part as the *Monitoring Agent*.

In this paper, we focus our attention on Linux operating systems running applications written in C and compiled with the GCC compiler [3]. Our set of constraints include those generated for the operating system and compiler, and some others generated for the standard C library. It is worth mentioning that similar structural constraints can be generated for different platforms, compilers, etc. Indeed, the proposed method is applicable to a broad range of operating systems and compilers despite our choice in this study.

Figure 1 shows the security debugging process used by CBones. The binary executable is first analyzed by the Constraint Extractor to determine the structural constraints. The set of constraints are then passed to the Monitoring Agent along with the program executable. The Monitoring Agent

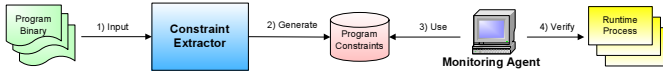


Figure 1. Security debugging process with C Bones

executes the program and checks for any constraint violations, possibly with previously captured malicious inputs exploiting one or more security vulnerabilities of the target program. If a structural constraint is violated, the execution is halted and an error message is generated. The error message states the violated constraint, outputs the program state, and indicates the instruction responsible for the violation.

In the following subsections, we first present our structural constraints, and then show how these structural constraints can be used in security debugging.

2.1 Program Structural Constraints

The Linux executable file format ELF (Executable and Linkable Format) [2] has a typical virtual memory layout as shown in Figure 2. Although some of the addresses can be user-specified, by default, the program code, data, stack and other memory segments are located as depicted in the figure and the ordering of these segments are fixed. For example, the stack segment is always at a higher address than the heap and the code segments, and the heap is always higher than the code and the data segments. We present our program structural constraints with respect to these segments, namely the stack, the heap and the data segments.

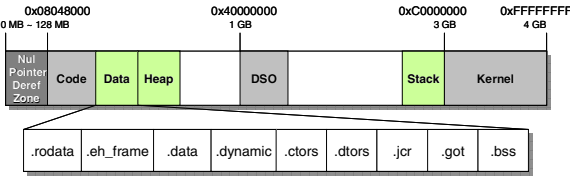


Figure 2. An example of an ELF program runtime process memory layout

2.1.1 Stack Constraints

The stack is perhaps the most commonly attacked segment of the virtual memory. For each procedure called, an activation record of an appropriate size is pushed onto the stack. Figure 3 illustrates a simple program and a simplified stack layout after function `main` calls `cal_Sum`, which in turn calls `get_Num`. The figure also shows the various logical subdivisions in the activation record. For x86 architectures, the special register `$ebp` points to the return address in the activation record of the last function called. *Arguments* hold the values of the arguments that the caller passes to the callee when making the function call. *Return Address* is where the execution will return once the function completes execution. *Previous Frame Pointer* is the saved `$ebp` register value. When the function returns, the `$ebp` value is

restored to point back to the previous activation record of the caller function. Sometimes additional registers can be pushed onto the stack to free up more registers for the current function. These registers are saved in the *Saved Registers* area and are restored upon function return. The rest of the activation record holds the local variables of the function.

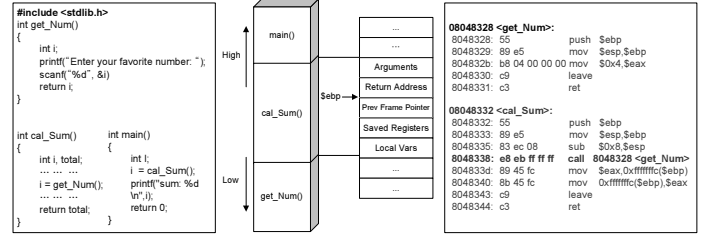


Figure 3. An example of stack structure and caller-callee relationship

The return address has been the most frequent target for attackers; however, a recent attack trend study [15] shows that other elements in the stack (e.g., frame pointer, saved registers) have also been exploited [1, 16, 17]. Such stack-based exploits require illegal modification of the stack structure. Therefore, security bugs in a program can be detected by monitoring the structural changes of the stack during program execution. Next, we present the program structural constraints for the stack region.

Caller-callee relationship constraint: When a function A (caller) calls another function B (callee), we say that A and B have a caller-callee relationship. A given function generally calls a small number of other functions. We can find out all possible caller-callee relationship between functions, and represent such relationships in a *call-graph*, in which all functions are represented as nodes, and a possible caller-callee relationship is represented as a directed edge from the caller to the callee. The constraint here is that, at runtime, every function call should represent an edge in this call-graph.

Return address constraint: Return address is a critical element in an activation record, since it changes a program's control flow. A valid return address should always point to an executable region such as the code section or the executable section of the dynamically loaded libraries.

Frame pointer constraint: The frame pointer (`$ebp` register) is a reference point for the stack frame and is used as a base address to access memory objects in the stack that are defined as offsets from this pointer. The invariant is that the value of the frame pointer should not be changed during a function's execution once it has been set in the function's prologue. The frame pointer is changed at a function's epilogue (`leave` instruction) according to the IA-32 instruction set manual [18].

Saved registers constraint: Saved registers generally

hold function pointers or values that refer to memory objects in a program’s runtime process. Although saved registers are not critical to the program’s control flow, they can be used as a bridge to a successful exploit. For example, one of the data attacks in [1] uses an address register($\$esi$) to execute an attacker-provided command. The saved registers constraint is that their values should not be changed during a function execution once they are set at the function’s prologue.

Saved frame pointer constraint: Upon a function call, the caller’s frame pointer is saved in the callee’s activation record. Since the frame pointer is held in the $\$ebp$ register, the constraint for saved registers applies directly. However, due to its special use, we found more constraints for the saved frame pointer. First, the saved frame pointer should point to a higher memory address than where it is stored. Second, since a frame pointer actually points to the saved frame pointer in the activation record, one should be able to walk the stack (following the saved registers as a linked list), and traverse every activation record. Finally, at the top of the stack, the saved frame pointer for function `main` (or the alternate entry function) should be `NULL`.

Frame size constraint: The content of the activation record is determined during compile time. Therefore, at runtime, the activation record size for a function should remain constant until the function returns.

Stack limit constraint: The maximum size of a stack can be increased or decreased depending on a program’s needs. The two register values, the stack pointer ($\$esp$) and the frame pointer ($\$ebp$) should be in-bounds during execution (i.e. point to the valid stack region). The default stack size limit in GCC is 8 MB, but it is a soft limit that can be changed during link time with the `-stack_size` linker flag or during runtime via the `setrlimit` system call. Therefore, the size of the stack and any changes made to it during runtime need to be determined.

Alignment constraint: The GCC aligns the stack as much as possible to improve a program’s performance. The default alignment is word aligned, but a user can choose other alignment options to adapt different computing environments such as 64 bit or 128 bit. Therefore, each stack frame should be aligned according to the alignment option.

2.1.2 Heap Constraints

A powerful feature of the C language is its dynamic memory management. Generally, dynamic memory is manipulated via the `malloc` family of library functions. There are various dynamic memory management schemes, Lea [5], Hoard [6], and OpenBsd [7] to name a few. In this paper, we follow Lea’s scheme, which is used in Linux systems, to derive our heap based program structural constraints. Lea’s scheme uses boundary tags to manage allocated and freed memory chunks. Each memory block is preceded by a small bookkeeping region that stores the information about the block in use, or in the case of available

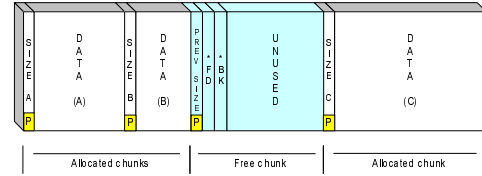


Figure 4. An example of heap structure

blocks, the next available block. Figure 4 shows an example heap structure. A boundary tag of allocated chunk (e.g., DATA (A)) includes information about its size and indicates whether the chunk is in-use with the last bit of the size field (`PINUSE_BIT`). If a chunk is free, the tag includes pointers to the previous and the next free chunks. We are able to identify six structural constraints for the heap segment.

Memory allocation/deallocation requests constraint: Since the structure of heap is changed dynamically by `malloc` related function calls, the first thing we need to check is whether such memory allocation/deallocation requests are made from valid locations. The idea is similar to the caller-callee relationship constraint in the stack, since we verify valid structural changes in a heap using caller-callee relationships for `malloc` family of function calls.

The heap constraints are checked for programs that use the C library to manage their heap. Some applications use their own memory management utilities, either by allocating a large chunk through `malloc` and managing the block themselves or using the `brk` system call to eliminate the library entirely. In this case, CBones does not check heap constraints. It is worth noting that such systems are not vulnerable to metadata manipulation like the C library is.

Boundary tag (metadata) constraint: Boundary tags, or hereafter referred to as metadata, are used and manipulated by the library that implements the memory management scheme. In our case, the metadata should only be modified through legitimate calls to `malloc` family of functions.

Allocated memory chunks link constraint: The metadata allows the memory manager to traverse the list of allocated or available memory blocks as a linked list. Therefore, a verification program should be able to traverse the same list using the metadata. A broken link between allocated memory chunks indicates the corruption of the heap structure.

Heap boundary constraint: Dynamic memory allocation must be performed within the given boundary of the heap memory range in a program’s runtime process. It is possible for an attacker to expand the heap size up to the kernel’s memory space to shutdown the target system or to escalate the privilege by making invalid heap allocation requests. For example, a real threat has been found in the previous Linux kernels [8] that exploits no boundary checking in the kernel’s `do_brk()` function so that an adversary can expand the heap area to the kernel space to take control

of the affected system.

Chunk boundary constraint: Heap memory chunks always begin on even word boundaries. When the program requests n bytes, Lea’s memory allocator provides more than $8+n$ bytes to keep the heap memory area aligned. Consequently, the returned pointer (beginning of the memory chunk) of `malloc` function call should be aligned with even word boundary.

Consecutive free chunks constraint: Upon deallocation of a memory block, before the memory block is added to the linked list of available memory, the memory manager looks for adjacent available memory blocks. If found, the two are merged together to form a larger available memory block. Therefore, after the memory manager returns from a `free` function call, the adjacent memory blocks should be in use. Having adjacent free chunks indicates a corruption of the heap structure and a heap corrupting attack. For example, double free attack puts fake free chunks with malicious forward (*FD) and backward (*BK) pointers to overwrite function pointers in the global offset table when the malicious chunk is freed.

2.1.3 Data Constraints

Although there are not many structural changes in the data section, we have found that some of the structural characteristics can help detect security bugs.

Shared library function pointers constraint: The global offset table (GOT) is a table of library function pointers. These function pointers initially point to the dynamic library loader and are updated via the loader when a library is loaded at runtime. Various attacks have tried to exploit these function pointers (e.g., [14]). The structural constraint is very simple: the library function pointers should point to library functions.

Constructor function pointers constraint: Constructor section (`.ctors`) consists of pointers to functions that are called before the program begins execution. These constructor functions mainly check or initialize the runtime environment. It has been shown that a format string attack can change these pointers to hijack the control flow [4]. Being function pointers, these entries should point to the program’s code section, not to stack or heap regions. Another observation is that these pointers are determined during compile time. Thus, once the program is loaded, they should remain constant.

Destructor function pointers constraint: Just like the constructors, the destructor function pointers point to the functions that are called once the program terminates. The structural constraints are the same as those of the constructor function pointers.

2.2 Security Debugging through Constraints Verification

As discussed earlier, CBones performs security debugging by verifying whether program structural constraints are

violated at run time. To do so successfully, we have to answer two important questions related to the verification of these constraints: “what to verify”, and “when to verify”. We have described the program structural constraints in the previous subsection, which answered the first question. In this subsection, we clarify the second question, i.e., when to verify.

We first introduce some terms to facilitate our explanation. Most structural constraints state that a memory region should be constant. Obviously, we need to clarify the time frame during which such a constraint is valid. We define the *lifetime* of a memory region to be the duration from the time when the memory region is set for the current context to the time when the memory region is discarded or released. Take for example an activation record in the stack. We described that the return address and the saved registers are structural invariants and should be constant throughout the execution of the function. The lifetime of the memory region holding a saved register does not start at function call time, but rather after the function prologue has pushed the register value onto the stack and ends when the function call returns. A metadata’s lifetime starts with the dynamic memory allocation and ends with the deallocation of the same memory region. For a data constraint, the lifetime of the `.ctors` and `.dtors` segments start with program execution and end at program termination.

Our second term describes the program state. A program is said to be in a *transient state w.r.t. a structural constraint* if the memory region related to the structural constraint is being modified. For example, consider a heap memory region allocated in a program. The program is in a transient state w.r.t. the memory region from the time when a `malloc` family of function is called to its return. In general, for a heap related structural constraint, this transient time frame is the period from the call to a `malloc` family of function to the function return. For a stack related structural constraint, this time frame includes the period from the time a function call occurs to the time the function prologue sets up the activation record.

Most of the constraints are based on memory segments that are dynamic. The stack changes with every function call/return, and the heap is modified with every memory allocation/deallocation. In theory, we can verify all the structural constraints continuously at every instruction. Indeed, any structural constraint that relies on a specific memory region can be checked at any given time, provided that the program is not in a transient state w.r.t. that constraint. However, such an approach will introduce substantial overhead, which is prohibitive in practice. On the other hand, the structural constraints related to a memory region must be checked at least before the memory region becomes inaccessible, so that potential violation of structural constraints will not be missed.

A simple solution is to perform *coarse-grained* con-

straint verification. That is, to verify the structural constraints before function returns and memory deallocations, since both the activation record for the returning function and the metadata of the dynamic memory region will become inaccessible after these actions. This allows us to capture violations of program structural constraints (as long as the exploit does not “fix” such violations). However, we will not be able to pinpoint the instruction that causes the constraint violation. This is certainly highly undesirable in security debugging.

We propose to use a two-stage approach in CBones to address the dilemma between unacceptable performance overhead and inaccuracy in identifying the constraint violations. In the first stage, CBones narrows the location of a constraint violation point down to a single function call, and then in the second stage, it determines the constraint violation point precisely.

Specifically, in the first stage, CBones is executed in the coarse-grained debugging mode, where the CBones monitoring agent verifies the structural constraints before function returns and memory deallocations. CBones then identifies the function call after which a constraint violation is detected. CBones is then re-executed with the same target program and input for the second time to start the second stage. In order to obtain more information and provide the instruction responsible for the corruption, in the second stage, CBones switches to a fine-grained debugging mode when it reaches the function call identified in the first stage. CBones then monitors all the memory writes during the function call. If a memory write instruction causes the constraints to be violated, CBones raises a flag and outputs the instruction attempting to corrupt the memory. As discussed earlier, the fine-grained debugging mode incurs high performance overheads; CBones works around this problem by only performing fine-grained monitoring during the function call identified in the first stage.

3 Implementation

As mentioned previously, utilizing a combination of static analysis and dynamic monitoring techniques, CBones consists of two components, the static analysis component called *Constraint Extractor*, and the dynamic monitoring component called *Monitoring Agent*. In the following, we describe the two components in detail.

3.1 Constraint Extractor

Written as a Ruby script [19], *Constraint Extractor* utilizes a number of programs and scripts to extract a program’s structural information and constraints from the target program binary. Since a C program is generally structured by multiple user-defined functions including *main*, *Constraint Extractor* first extracts each function’s information such as its name, address, activation record size and number of saved registers from the debugging information included in the program binary. In order to obtain the in-

formation from the binary code itself, *Constraint Extractor* takes as input the program executable compiled with the debugging flag (-g) and without any optimization. It uses *dwarfdump*, a publicly available C program that outputs the debugging information in DWARF format and we wrote a parser called *dwarf_parser* in Ruby to parse *dwarfdump* output.

To derive the caller-callee constraints from the target program, *Constraint Extractor* uses *objdump* to disassemble the program executable and extract all the *call* instructions, then parses the instruction number and the procedure name, and adds the next instruction number to the valid return address list of the procedure. Finally, the *Constraint Extractor* outputs the procedure information in a text file.

Note that *Constraint Extractor* takes as input an ELF binary that is compiled without any optimizations. The reason for a non-optimized binary requirement is that *Monitoring Agent* works based on a number of assumptions such as the *\$ebp* being used as a frame pointer and the specific protocols during function calls and returns. When optimized, binaries may invalidate these assumptions, causing false positives and/or negatives to occur. We leave the optimized binaries and seamless integration with commodity software as future work.

3.2 Monitoring Agent

The CBones Monitoring Agent is responsible for verifying the program structural constraints and reporting any violations. We implement the CBones Monitoring Agent as a Valgrind skin. Valgrind is an open-source CPU emulator which has been used for debugging and profiling Linux programs [20]. When a program is run under Valgrind, Valgrind executes the program one basic block at a time. It initially translates the instructions in a basic block to an intermediate assembly-like language called UCode. The UCode basic block is then passed on to the *SK_(instrument)* function defined in the skin, which allows programmers to instrument the binary code by calling their own functions or altering the basic block. The instrumented UCode block is translated back into machine language for execution.

The Monitoring Agent uses some internal data structures to store the procedure information provided by the *Constraint Extractor* and to keep program state during debugging. A *Procedure* data structure is created for every entry in the input file and a *CallStack* stores activation records during the runtime. The Monitoring Agent also keeps another data structure *ChunkList* to keep track of the dynamically allocated memory regions.

The Monitoring Agent uses a procedure called *cb_check_all_constraints* to verify all program structural constraints that are available. As discussed earlier, we cannot verify whether a structural constraint is violated when the program is in transient state w.r.t. this constraint. Since the only times when such cases may happen is when the target program makes a function call or

a call to a `malloc` family of functions, in coarse-grained debugging mode, the Monitoring Agent captures function calls to validate the structural constraints. Once the program leaves its *transient* state, the Monitoring Agent calls `cb_check_all_constraints` to verify all the remaining structural constraints. Moreover, the Monitoring Agent also marks a “safe point” if no constraint violation is detected. The last “safe point” before the constraint violation will be used in stage 2 as the indication to switch to the fine-grained debugging mode.

In the following we list the events of interest and the actions taken by the Monitoring Agent.

Function calls: The Monitoring Agent handles function calls in several stages. Initially, when a jump due to a call instruction is captured, the Monitoring Agent determines the caller and the callee and verifies the caller-callee relationship constraint. This constraint is only checked for client program functions and not the library functions. The second stage occurs when the callee sets its frame pointer. The Monitoring Agent creates a new activation record for the callee and adds it to the current thread’s `CallStack`. The alignment of the frame pointer is also checked. The third stage only applies to procedures that save registers in their activation record. Once all the registers are pushed onto the stack, a snapshot of the invariant region in the activation record is taken and stored in the activation record. Since no further changes to the invariant region is expected, the program is no longer in its transient state. The Monitoring Agent calls `cb_check_all_constraints` to verify the other structural constraints, and marks a “safe point” if there is no constraint violation.

Function returns: When a procedure is returning, the Monitoring Agent captures the jump due to a `ret` instruction and verifies that the return address is a valid return address for the returning procedure. A function epilogue contains at the very least a `leave` and a `ret` instruction. The `leave` instruction, which precedes the `ret`, restores all the saved registers. Therefore, when the Monitoring Agent captures a function return, the registers, including the frame pointer, are already restored to the caller’s values. Nevertheless, the activation record of the callee is still intact and can be examined. The Monitoring Agent verifies that the invariant region of the activation record is intact and removes the activation record of the returning function from the current thread’s `CallStack`. It then calls `cb_check_all_constraints`, and marks the return address as a “safe point” if there is no constraint violation.

malloc/free calls: The Monitoring Agent intercepts `malloc` family of function calls via wrapper functions. These function allow the Monitoring Agent to perform bookkeeping on dynamically allocated memory regions. For each newly allocated memory, the Monitoring Agent first calls `malloc` to allocate the memory, and then creates a new chunk and add it to `ChunkList`. Two additional

checks verify that the heap boundary constraint and the alignment constraint are satisfied. When a `free` call is intercepted, the Monitoring Agent first verifies that the metadata is intact. It then calls `free` and finally removes the chunk corresponding to the memory from `ChunkList`. During deallocation, the Monitoring Agent simply calls `cb_check_all_constraints` to verify that the metadata is intact. This is possible since the Monitoring Agent determines when to actually deallocate the memory, and hence the program is not in a transient state until it does.

Memory writes: When running in the fine-grained debugging mode in the second stage, the Monitoring Agent captures all memory writes by instrumenting the binary code. If the destination address belongs to any of the invariant regions in stack or heap, a flag is raised to mark the instruction attempting to violate the corresponding structural constraint. Capturing memory writes is not always trivial, since memory can be updated through system calls and kernel functions. The Monitoring Agent’s current implementation captures system calls and performs the necessary checks before the memory is updated. In one of the test cases, a large memory copy operation is performed by manipulating the page table through the use of kernel functions. Since Valgrind cannot trace into kernel space, such a memory modification would go unnoticed. This means that the current implementation of CBones would not be able to determine the instruction responsible for the corruption. It should be noted that this does not mean that the attack is unnoticed.

We briefly mentioned in section 2.2 that not all program structural constraints need be checked to verify our list of constraints. We deferred this discussion previously to present the events and actions related to the Monitoring Agent first. `cb_check_all_constraints` verifies that for each activation record in `CallStack`, the invariant region of the activation record is intact. This ensures that the return address and saved registers, including the saved frame pointer, have remained constant, satisfying a number of structural constraints. We assume that if every activation record is intact, and each activation record is created by a legitimate instruction conforming to the call-graph, then other structural constraints such as the linked list of saved frame pointers and frame sizes are also satisfied. A similar approach is taken with heap related structural constraints. Our assumption is that, as long as the metadata have not been modified by any means other than the library function calls that are designated for the task, the constraints are satisfied.

4 Experimental Evaluation

We performed a series of experiments to evaluate CBones, using a group of 12 real-world applications with known vulnerabilities and exploits. The objective of the experiments is to understand both the security debugging capability and the performance overhead in CBones.

In the following, we first illustrate how CBones is used

for security debugging through a running example, then present our security evaluation aimed at understanding the security debugging capabilities provided by CBones, and finally describe the performance evaluation.

4.1 Security Debugging Using CBones: A Running Example

We use one of our test cases, Sumus, to demonstrate how to use CBones for security debugging. Sumus [9] is a game server with a stack overflow vulnerability. Figure 5(a) shows the snippet of the source code responsible for this vulnerability. The boldface line copies the contents of `p1` into the local array `tmpCad`. `p1` points to the string after `GET` in the HTTP request. The programmer assume that this input string have a small size. An attacker may exploit this assumption by sending a message longer than the size of `tmpCad`. At a first glance, this looks like a trivial stack overflow; the overflow should first corrupt the local variables and then the return address. However, as the buffer is overflowed, the instruction first overwrites `faltan` and then `kk`, which is actually used to index `tmpCad`. With a carefully designed input, the overwrite skips to the formal arguments’ memory region, not overwriting the return address. This behavior of the attack makes it much more difficult and time-consuming to debug manually. Another important note is that this attack cannot be captured by systems looking for control-hijacking attacks alone, since the return address remains intact.

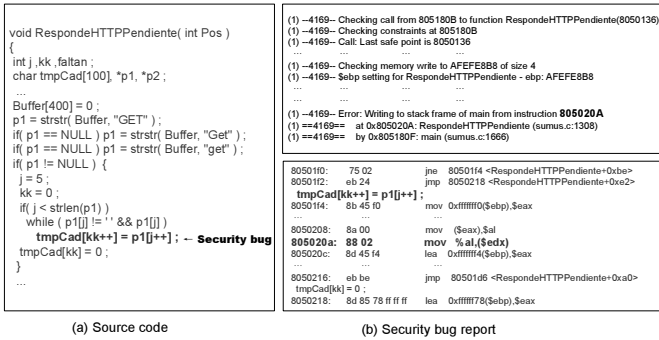


Figure 5. An example of automated bug diagnosis using CBones

For comparison purposes, we first ran Sumus under `gdb`. However, `gdb` was only able to capture the program’s crash after `ResponseHTTPPendiente` calls the `send` function (not shown in figure). Therefore, it is non-trivial to locate the corrupting instruction using `gdb`, if not entirely impossible.

We then ran Sumus under CBones. CBones, however, was able to detect the exploit and determine the corrupting instruction fully automatically. Although this attack does not corrupt the invariant region of `ResponseHTTPPendiente`, it does corrupt the invariant region of the caller function `main`. Therefore, CBones

was able to detect a violation of the caller-callee relationship constraint in the first stage. In the second stage, CBones started the fine-grained debugging mode from the last “safe point” (0x8050136), detected the (illegal) memory write into the return address in the stack frame of the `main` function, and raised a flag accordingly. Figure 5(b) shows the output of CBones. The error message clearly states that a memory write to an invariant region has occurred and displays the instruction number responsible for the memory write.

4.2 Security Evaluation

Our debugging example with CBones clearly reflects the effectiveness of using program structural constraints to discover security bugs in a vulnerable program. To further evaluate the effectiveness of our method, we used 12 real-world applications with various vulnerabilities. Seven applications have stack overflow vulnerabilities, three have heap overflow vulnerabilities, and two have format string vulnerabilities. The effectiveness is measured by how precisely CBones can locate a security bug in a program.

Table 1 summarizes the evaluation result. In all tested applications, CBones captured constraint violations at the time of structural corruption and pinpointed the corrupting point (instruction) in the program, and raised no false alarms. The last column of Table 1 shows the constraint that was violated in each test case. Most stack overflow exploits violated either the return address constraint or the saved registers constraint. All heap overflow cases violated boundary tag constraint, in which the exploits overwrote metadata next to an allocated chunk in the heap. Two format string cases violated the return address constraint. Although we do not have malicious attacks to evaluate CBones with other types of attacks, it is worth noting that such attacks depend on the three types of memory corruption techniques mentioned here. For example, return-to-libc attack is a variation of stack overflow and malloc-free attack relies on a heap buffer overflow.

4.2.1 Discussion: False Positives and False Negatives

Our proposed approach guarantees no false positives, since all program structural constraints should be satisfied by all program binaries. The current version of CBones, however, does not reach the full potential of the proposal approach yet, due to a limitation in the implementation of Constraint Extractor. The current CBones Constraint Extractor cannot handle function pointers properly yet. A jump due to a function pointer usually dereferences a register or an address in assembly code. Even though the Constraint Extractor detects these instructions, it does not include analysis procedures to derive all possible caller-callee relationships. As a result, the Monitoring Agent may raise a false alarm due to some missing caller-callee relationship. These false positives can be suppressed by marking `call` instructions that dereference values instead of using function addresses and

Table 1. Security evaluation result. S:Stack overflow, H:Heap overflow, F:Format string

Name	Description	Vul.Type	Bug Detected?	Constraint Violation
streamripper	a Winamp plugin for recording radio stream	S	Yes	Return address constraint
ghhttpd	an open source web server	S	Yes	Return address constraint
sumus	a game server for 'mus'	S	Yes	Return address constraint
prozilla	a web download accelerator	S	Yes	Return address constraint
monit	a multi-purpose server monitoring utility	S	Yes	Saved registers constraint
newspost	a Usenet binary auto poster	S	Yes	Saved registers constraint
icecast	a audio broadcast server	S	Yes	Saved registers constraint
WsMp3d	an web server with audio broadcasting	H	Yes	Boundary tag constraint
Xtlnet	an open source telnet daemon	H	Yes	Boundary tag constraint
NULLhttp	an open source web server	H	Yes	Boundary tag constraint
Power	an UPS monitoring daemon	F	Yes	Return address constraint
OpenVMPS	an open source VLAN management server	F	Yes	Return address constraint

suppressing the errors when they are generated at these instructions. We will extend the implementation of Constraint Extractor to handle such cases in our future work.

CBones is intended to automatically identify exploits of vulnerabilities that violate *program structural constraints*. Thus, it cannot detect exploits that are outside of this scope, for example, illegal modifications of the memory regions belonging to program variables. Such cases can be resolved using techniques and tools complementary to CBones, such as DAIKON [12] and DIDUCE [13].

4.3 Performance Overhead

We also performed experiments to understand how much CBones slows down the test applications, though such performance penalty is non-critical for debugging. For simplicity, we chose to use throughput as a performance metric. This performance metric certainly reflects the extra overhead due to CBones. However, it is not applicable to all the test applications. For example, the Power daemon checks the status of the UPS periodically and idles/sleeps for the rest of the time. As another example, Xtlnet is a telnet daemon, and its runtime or throughput depends on the client activity. Thus, in our experiments for performance evaluation, we focus on the subset of server/client applications in our test suite. All programs were compiled using GCC version 3.2.2 with debugging option (-g). Our testbed ran Red Hat Linux 9.0 with a 2.66 GHz Intel Pentium 4 CPU, 512 MB of RAM, and a 7200 RPM ATA IDE hard disk drive.

For comparison, we first ran the test programs without Valgrind, under Valgrind's Nullgrind skin, and finally under CBones. Nullgrind is Valgrind skin without instrumentation. Thus, this reflects the performance slowdown introduced by Valgrind. To better understand the performance impact of CBones, we run CBones in two modes. The first mode is the default, coarse-grained mode, in which the relevant program structural constraints are checked after function calls/returns and after malloc family function calls. The second mode is the fine-grained mode, in which CBones starts the memory monitoring from the last "safe point". Such a "safe point" is usually very close to the corrupting instruction. For each test program except for sumus, we measured the time it took to download two files with sizes 700KB and 12MB, respectively, and take the average

of the slowdown factor over 15 iterations. As an exception, sumus uses multi-threads and only allows a small size file (up to 200KB) for data transmission. In this case, we used 200KB files.

Figure 6 shows the evaluation result. The Y-axis in the figure shows the slowdown factor of using Nullgrind and CBones compared to normal execution of each test program. For example, in ghhttpd, Nullgrind incurs 1.88 times slowdown and CBones in the coarse-grained mode incurs 2.46 times slowdown compared to the normal execution. The overall average slowdown factors for CBones in the coarse-grained and the fine-grained modes are about 5.23 and 12.57, respectively, compared to normal program execution. It is worth noting that a good portion of the overhead came from our implementation choice (Valgrind), which incurs 2.44 times slowdown on average by itself (Nullgrind). CBones only incurs 68% additional overhead on average compared to Nullgrind. NULLhttp shows an exceptional slowdown compared to other programs. It shows about 5 times slowdown under Nullgrind, 15.21 times slowdown with CBones in the coarse-grained mode, and 53.97 times slowdown with CBones in the fine-grained mode. This significantly increases the performance overhead. Our further investigation indicates that this significant performance overhead is due to the large number of function calls and returns during the test process (e.g., 708,355 function calls in NULLhttp v.s. 1,224 function calls in ghhttp when downloading a 700KB file). In contrast, the sumus test case shows very little overhead (1%), since sumus uses multi-threads and only allows a small size file (200K) for data transmission.

When the monitoring agent observed a constraint violation, we changed the constraint checking granularity to fine-mode using the last safe point reported from the monitoring agent. In such case, the average performance overhead can be increased up to 14.76 times compared to normal execution and 2.99 times compared to Nullgrind.

5 Related Work

A number of approaches have been proposed to provide automated debugging capability. iWatcher [10] uses expected access (e.g., read-only, read/write) to user-specified

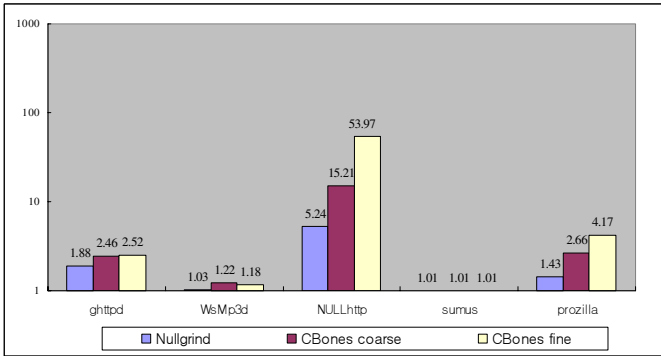


Figure 6. Slowdown factors of Nullgrind and CBones

memory regions to check whether there is any access violation and triggers further investigation if there is any. Users can turn monitoring on or off at run time to reduce the overhead. However, iWatcher requires additional hardware support, such as modified L1/L2 cache, victim watchflag table, and range watch table, and thus cannot be used directly on common computer systems. AccMon [11] uses the program counter based invariants (i.e., each memory object is generally referenced only by a few instructions) to monitor program execution. AccMon incurs low overhead (0.24-2.88 times) using additional hardware support, including iWatcher [10] and Check Look-aside Buffer. AccMon monitors global objects, heap objects, and return addresses in the stack. To reduce false alarms, AccMon provides a confidence level to help users decide whether an alarm is likely to be a bug. Similar to iWatcher, AccMon cannot be used on modern computer systems due to the additional hardware requirement. DIDUCE [13] instruments Java byte code and monitors variables in a program to generate dynamic invariants. DIDUCE reports detected anomalies by checking the history of variable usage (comparing a variable's previous and current values). To reduce false alarms, it provides options for users to tighten or relax the properties. DAIKON [12] introduced variable-based invariants debugging. It instruments a program's source code and traces the program executions to find the likely invariants. It then generates pre/post conditions at certain program points such as functions or variables to check violations.

6 Conclusion

In this paper, we identified a new class of program invariants called program structural constraints, and developed a novel security debugging tool called CBones to automatically extract program structural constraints from program binaries and analyze security bugs in vulnerable programs. Compared with the previous approaches, CBones provides several benefits: 1) full automation (neither training nor manual specification is required, 2) no need for source code analysis or instrumentation, 3) no requirement of additional hardware support, and 4) no false alarms. Our experimental evaluation with 12 real-world applications demonstrates

that CBones can identify all security bugs automatically, without generating any false alarm or false negatives.

We understand that the program structural constraints also offer a basis for detecting (unknown) exploits, in addition to security debugging. In our future work, we will investigate how to use program structural constraints to efficiently detect attacks that cause the violation of these constraints.

References

- [1] S. Chen, J. Xu, E.C. Sezer, P. Gauriar and R.K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security Symposium*, Baltimore, MD, August 2005.
- [2] Tool Interface Standard (TIS) Committee. Executable and Linking Format (ELF) Specification, 1995.
- [3] Programming Languages Supported by GCC. GNU Project, <http://gcc.gnu.org>.
- [4] Format string exploit, http://www.acm.uiuc.edu/sigmil/talks/general_exploitation/format_strings/.
- [5] D. Lea, A Memory Allocator, <http://gee.cs.oswego.edu/dl/html/malloc.html>.
- [6] E.D. Berger, K.S. McKinley, R.D. Blumofe, and P.R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In the *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*. Cambridge, MA, November 2000.
- [7] The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition.
- [8] iSec Security Research. Linux Kernel do_brk() Vulnerability, 2003.
- [9] Common Vulnerabilities and Exposures (CVE) 2005-1110, assigned on April 16 2005. available from <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1110>.
- [10] P. Zhou, F. Qin, W. Liu, Y. Zhou and J. Torrellas. iWatcher: Efficient Architecture Support for Software Debugging. In *Proceedings of the 31st annual International Symposium on Computer Architecture (ISCA'04)*, June, 2004.
- [11] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, J. Torrellas. AccMon: Automatically Detecting Memory-related Bugs via Program Counter-based Invariants. In *37th International Symposium on Microarchitecture (MICRO)*, Nov 2004.
- [12] M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.
- [13] S. Hangal and M. S. Lam. DIDUCE: Tracking down software errors using dynamic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, May 2002.
- [14] S. Cesare. Shared library call redirection using ELF PLT infection. Apr 2007. available from <http://vx.netlux.org/lib/vsc06.html>.
- [15] NIST national vulnerability database, <http://nvd.nist.gov/>.
- [16] The Frame Pointer Overwrite. klog, available from <http://doc.bughunter.net/buffer-overflow/frame-pointer.html>.
- [17] E. Chien and P. Szor. Blended Attacks Exploits, Vulnerabilities and Buffer-Overflow Techniques in Computer Viruses, Virus Bulletin Conference, 2002.
- [18] IA-32 Intel Architecture Software Developer's Manual, <ftp://download.intel.com/design/Pentium4/manuals/>.
- [19] Ruby programming language. <http://www.ruby-lang.org/en/>.
- [20] N. Nethercote. Dynamic binary analysis and instrumentation, 2004.
- [21] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter Technical Conference*, 1992.
- [22] W. Robertson, C. Kruegel, D. Mutz, F. Valeur. Run-time Detection of Heap-based Overflows. In *Proceedings of the 17th USENIX Large Installation Systems Administration Conference (LISA)*, October 2003, San Diego, CA USA.
- [23] C. Cowan, C. Pu, and D. Maier. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.
- [24] J.-D. Choi et al. Efficient and precise datarace detection for multithreaded objectoriented programs. In *PLDI*, June 2002.
- [25] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI*, June 2002.
- [26] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI*, December 2002.