2019 2nd International Conference on Mechanical, Electronic and Engineering Technology (MEET 2019)

Active checking buffer overflow vulnerability in binaries with symbolic execution

Bo Wu[a], Yufeng Ma[b], Qian Zhang[c], Fengchen Qian[d]

School of Information and Communication on National University of Defense Technology, Xi'an, China

[a]cherry_wb@163.com, [b]mauf@163.com, [c]zhangqian_nudt@163.com, [d]fchen_qian@163.com

Keywords: symbolic execution; software vulnerabilities; software exploit; reverse engineering

Abstract: Buffer overflows in C and C++ programs are among the most common and serious classes of software vulnerabilities for two decades. To mitigate and to eradicate this security threat, many detection approaches based on static and dynamic analysis techniques have been proposed. This paper aims to provide an alternative and multipath solution to existing symbolic execution approaches by catching the red-zones in memory, rather than the strict memory object bounds, which are absolute vulnerable to buffer overflows. From the observations of buffer overflow attacks that are commonly overwrite a special position to exploit the vulnerabilities, in this paper, we propose a multipath dynamic buffer overflow detecting approach (symbolic-execution-based), relying on catching the red-zones in the stack and heap memory. We examine every memory store operation both in concrete addresses and symbolic pointers, whose writing size should never overlap or cross a red-zone position. We implement a practical dynamic checking tool called MACBin based on S2E platform. We apply it to test several real world software, the results show that MACBin is useful and effective at detecting buffer overflow vulnerabilities.

1. Introduction

Buffer overflow is a common memory corruption error, which allows an attacker to cause the program to write more data in a buffer than it was intended to hold. This may cause danger when overwriting a security-critical data structure, almost always a code pointer or a data pointer. It is well documented that buffer overflows can be maliciously exploited, such as elevate privilege, code injection, remote execution and so on. Classic buffer overflows have been a constant threat for years, such as WannaCry[1]. The programs that vulnerable to buffer overflow cover a wide range of applications that include an OS kernel, device driver, database engine, embedded system, networking and web applications.

To date, the academic and industry research communities have proposed and developed a lot of different approaches to mitigate or to eliminate buffer overflows and their exploitation. One category of preventing techniques attempts to alleviate the damage by halting exploits via operating system support, for which some people consider that such errors are inevitable but defensible. These mitigating techniques include canary-based stack protection [2], non-executable stack solution [3], Data Execution Protection (DEP) [4], and address space randomization and artificial heterogeneity [5].

Another category is more positive, which is committed to eliminate the errors with compiler-enforced protections, or find the errors with runtime detecting. These techniques range from static analysis based on abstract interpretation in compile time, to bounds checkers that perform runtime checks dynamically for out-of-bounds accesses.

Static analysis tools attempt to inspect the source code that all memory accesses are checked in bound[6]. They can test or analyze almost all the execution paths. However, without actually running the program execution with real inputs, it has difficulties inferring precise runtime effect. Sound static tools tend to generate too many false alarms due to its overly aggressive abstraction and unsound tools can miss errors in the code. Although some work tries to reduce the false positives, it can't fundamentally eradicate them.

Compared to static analysis, dynamic buffer overflow detection is a more attractive approach which has a key benefit: the input itself demonstrates the presence of a bug and there are no false alarms[7]. Dynamic detecting tools mainly focus on memory operations on stack and heap by instrument checking functions in the program. These tools can detect memory errors such as stack smash, heap overrun, and memory leak and so on. One of the most weakness of this approach is that requires an input to reveal the error, which can just analysis a single input relevant execution path at a time.

An appealing alternative approach to analyze multipath in program is symbolic execution. Instead of running a program on concrete input, symbolic execution runs the program on symbolic input initially allowed to have any value, and automatically generates a set of input values that will exercise as many program paths as possible. By leveraging recent advances in program analysis and automated constraint solving, symbolic execution technique achieves fairly higher statement coverage than other testing strategies. By actually running code, it sees the precise runtime effects of code and eliminates false positives. Unfortunately, existing symbolic execution systems are not designed to detect buffer overflow in binaries, they are mainly designed for bound checking in source code, or passively testing bugs by program crashes.

Detecting buffer overrun with symbolic execution in binaries have not been adopted widely compared with that in source code. That because they either (1) lack of high-level program language semantic information, (2) can't recognize each memory object bound precisely on execution, or (3) can't give an aggressive check unless a real memory corruption happening.

In this paper, we present a symbolic-execution-based dynamic detecting approach for the buffer overflow detection in binary programs. In order to overcome the deficiencies of binary programs described above, we use red-zones to measure the bound for each meaningful memory block. The red-zone can be a return address, a heap header or a function pointer, which shouldn't be overwritten by the program instructions or extern data. To explore multipath and monitor the execution, we use a state-of-the-art symbolic execution binary analysis platform to implement our tool. We check each memory writing operation both in concrete addresses and symbolic pointers, whose writing size should never overlap or cross a red-zone position.

Our insight consists of two steps: Firstly, we maintain a run-time data structure to record all red-zones that we can get in execution. We mainly consider three forms of un-writable addresses as the red-zones: stack frame's bound, heap chunk afterward bytes and tainted function pointers. Secondly, we checking the address's legality at both writing and execution time. To check the legality of writing operation, we intercept each memory access and examine whether the sized storing operation is touching a red-zone. Due to the checking address can either be a concrete value or a symbolic expression, we consider them separately. For the concrete address, we use the actual size to evaluate it. For the symbolic address, we first get the range of the symbolic expression under current execution state constraints, then we check whether the range will cross a red-zone. For checking the illegal execution, we intercept symbolic execution calling instruction to see whether it is a symbolic pointer, which means that a function pointer may be modified by the extern data.

Based on our key insight, we have implemented a tool on top of S2E[8] platform, a selective symbolic execution system. We develop MACBin, a tool for Memory Active Checking on Binary programs, which is based on S2E memory-check plugins. Compared with other source code based bound checking tools, MACBin takes two key trade-off but benefit characteristics: First, instead of preventing pointers from overwriting from one memory object into another, it prevents that of red-zone positions. With this handling, we do not need to reason the memory objects in detail, which is difficult for binary programs. Secondly, it delays checking the influence of modifying an un-red-zone location in latter use. For this, we can avoid to detect a slight influence or benign overflow. Similar to most previous dynamic approach, MACBin can record and replay the program path leading to each error it detects. We show that our tool's detecting ability in several real-world applications in both of concrete and symbolic checking modes.

The remainder of this paper is organized as follows. We first introduce and explain red-zones definition(§II). We take an overview and its implementation for our MACBin (§III). And we present its evaluation for effectiveness(§IV). We finally conclude our work (§V).

2. Red-Zones Definition

Typically, a buffer is used when moving data between processes or functions, however, without a proper termination condition, it would overwrite several meaningful memory positions behind the buffer, where we call them as red-zones. These red-zones always have a special significance for the system, which are not supposed to be modified by the program instructions.

Considering the memory layout of a program, the process virtual address space should be composed of five primary segments: 1.the text segment, holding the executable code. 2.the initialized data segment, containing those data that are initialized to specific non-zero values at process start-up. 3.the bss segment, containing data initialized as zero at process start-up. 4.the heap segment, containing uninitialized data and the process heap. 5.the stack, containing function's local data and context switching data. Due to the receiving buffer of extern data are mainly located in the data areas, we discuss the red-zones in these data areas.

The initial or un-initial data area provides space for non-transient variables such as global or static variables. These addressed are always determined after the operate system load the executable file from the disk. Buffer overflow occurs here may cause a data flow or control flow change in future execution. For example, the overlap of a global function pointer in data area may make the pointer reference to jump a contrived address. Therefore, the global function pointer can be seen as the red-zone that should not be modified by extern data.

2.1 Red-Zones in Stack

The stack is the most frequently used area which consists of a continuous series of stack frames or active records. Each stack frame provides a context environment for function executing, which contains parameters, local variables, protected registers and return address. Stack frame are dynamically changed as the program instructions execution, and the stack frame's top can be created by the function call instruction, where will firstly push the return address. Once the return address is pushed onto the stack, it shouldn't be modified by the program, so this position can be marked as the red-zone. Other data in the stack such as function pointers, protected registers or variables that change the control flow can be also seen as the red-zone, however, they can only possible be identified when they are in latter using. Figure 1 shows the red-zones with depth of 4 layers function call.

2.2 Red-Zones in Heap

The heap is a large pool of memory within the process space which is used to provide memory to an application when requested. The memory management system must track outstanding allocations to ensure that they do not overlap and that no memory is ever "lost" as a memory leak. Typically, the heap is often managed to cycle reuse by chunking. To organize the heap area and allocate and de-allocate chunks, the system often provides a specific algorithm which is interlinked with the kernel, such that an application can allocate and free the memory block by run-time library functions (such as malloc, realloc, calloc and free) or application program interface functions. During the allocating, the heap management routine will store its management information at the first few bytes of the chunk, which can be identified by system as an used one. And when de-allocated, the used chunk's header would be changed to a freed chunk header. Figure 2 shows the difference between the used chunk and the freed chunk(in windows2003 sp2). The used chunk header contains 8 bytes, and that of freed chunk(freelist) is 16 bytes. The used chunk and freed chunk may be alternately distributed in memory. Figure 3 shows the dynamic change of allocating and freeing manipulation. We can see that it must be another chunk's header behind each in-using heap chunk. Writing data past the end of a chunk boundary would overwrite next chunk’s management fields, hence, the bytes after each available chunk can be marked as the red-zones.

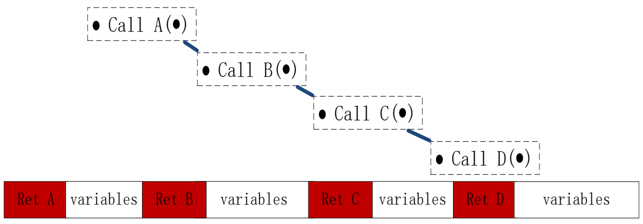

Fig. 1. Red-zones in stack.

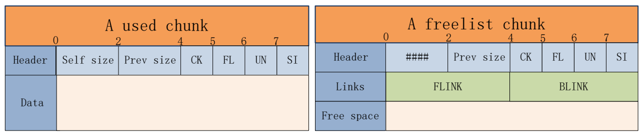

Fig. 2. Heap header in used chunk and freed chunk.

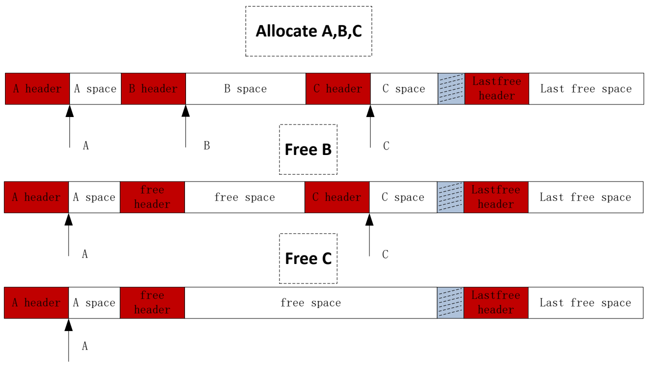

Fig. 3. Heap dynamic management in memory.

From the program language pointer of view, process memory is divided into blocks by declared and typed variables, which can be seen as a series of sized objects. Specially, buffer overflow is an anomaly where a program, while writing data to an object, overruns the object's boundary and overwrites adjacent object. As noted, it always assumes that a buffer overflow will cause a security breach because that it's out of programmer's intention. However, that is not necessarily true. If we consider the possible influence of a buffer overflow, it has varying degrees of damage. For example, if the buffer overflow just overwrite a limited bytes of next variable, and the modified variable may cause a harmless influence in latter use, moreover, it may also overwrites some insignificant bytes(such as padded bytes allocated by compiler for alignment), we can think it as a weak or benign overflow.

In single path analysis, modifying memory often means uncertain, because the influence of modified bytes can't be predictable. Fortunately, symbolic execution can trace all of the influence by its two central implement mechanisms: symbolic collecting and path forking. For the symbolic collecting, it means that all the input tainted memory are substituted by symbolic expressions. If the buffer overflow modifies a byte, this byte will get a symbolic expression, and how this byte takes effect can be examined in latter using. For the path fork, it accumulates the constrain which records the symbolic condition on each branch and forked path. In this way, the influence of modified byte can be checked in different paths, and this overcomes the deficiency of traditional single path dynamic detecting approaches.

Most existing symbolic execution systems use a strict bound check to detect the buffer overflow, which aim to correct the fault of the programmer based on source code. However, in most situations, the compiler will do some transformations during compiling the source code to assemble, such as rearranging the memory for align or optimizing. These manipulations could import or eliminate the errors. In others words, if we only consider buffer overflow in source code, we can't predict the change introduced by the compiler, which may cause an imprecise result.

In this paper, we consider detecting the buffer overflow by evaluating their actual and potential damages, which contains two aspects in symbolic execution consideration: Firstly, we select the red-zone positions from the real memory layout, which should never be modified by the program or external data. Secondly, we consider remainder memory locations as ordinary variables, replace them with symbolic expression if they are modified by the external data, and delay check them in using time. If they are used in improper way such as illegal address reference, unbound function pointer or else, we will detect it and emit a mistake.

3. Design and implementation

3.1 MACBin Overview

The goal of MACBin is to detect buffer overwrite red-zones such as a stack bound, a heap header, a heap freed area or a function pointer in a binary program. As discussed, MACBin is built on S2E for achieving multipath analysis and monitoring available memory. Figure 4 shows the workflow of MACBin. We explain this workflow as well as how the components interact, the numbered labels in the figure correspond to the steps as list below.

(1) Given a binary program, S2E intercepts QEMU engine to get intermediate representation of the basic block and invokes Dynamic Block Translate(DBT) to generate the host machine code or LLVM Bitcode for concrete execution or symbolic execution respectively.

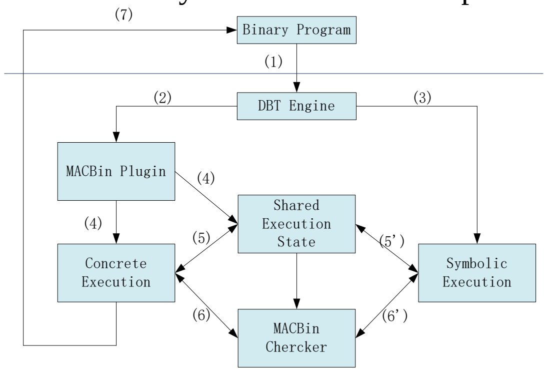

Fig. 4. MACBin workflow.

(2) If it is running in concrete mode, MACBin invokes the stack monitor plugin and heap monitor plugin. The stack monitor plugin intercepts each call instruction to record and update the stack frame information. The heap monitor plugin instruments the heap management functions to maintain and update the available heap information.

(3) If it is running in symbolic mode, it invoke the KLEE [11] to take a symbolic execution.

(4) The recorded stack and heap information are kept in their PluginState which can be shared in checking.

(5) The Concrete engine runs the instruction in native mode or the symbolic engine interprets the translated code in symbolic mode. Both of them are based on the shared execution state.

(6)When detecting a signal of store operation or a call/jump instruction in execution, MACBin invokes stack overflow checker, heap overflow checker and call/jump checker respectively. Both stack and heap checker work on concrete and symbolic mode. The call/jump checker just needs to work on symbolic mode, because it will throw out a system exception automatically if call/jump an illegal address in real execution. For the write operation in symbolic mode, it first gets the constraints and symbolic address pair from the shared execution state, then get the range of the symbolic address under current path constraints, and invokes the stack checker or heap checker to determine whether it would produce a error.

(7) Fetching and executing next basic block of the binary program.

When a bug is caught, the checker will report the related vulnerability information such as the type of the bug, the detecting mode, corresponding test cases, and instruction location in the modules. Due to MACBin checks in concrete and symbolic mode separately, it takes a different treatment: for the concrete checking mode, it will emit an bug information and terminate the execution state directly when detect a vulnerability. But for the symbolic mode, it will continue to execute. In this case, it may report the same vulnerability repeatedly. We can use the context information to distinguish the redundancy bugs.

3.2 Implementation

At the implementation level, MACBin leverages the plugin interface of S2E platform and extends the S2E execution state to store the monitored memory and symbolic address information. S2E implements shared execution state data structure for the CPU and the physical memory. This structure encapsulates an array of concrete bytes and symbolic expressions. Due to S2E concretizes symbolic addresses when they are written to the (always concrete) program counter, MACBin stores the current constraints and symbolic address pair to shared execution state before actual executing.

We have implemented MACBin as a part of the S2E platform. S2E automatically enumerates various execution paths and has a modular architecture, in which plugins communicate via events in a publish/subscribe fashion. MACBin has several components to support the vulnerabilities detecting, which are loaded by register a class of S2E events or handle a special function. S2E events are generated either by the S2E platform or by other plugins. MACBin's plugin invokes regEventX (callbackPtr), the event callback is then invoked every time EventX occurs, and it is passed parameters specific to the event [8].

Thanks to the S2E, MACBin can explore and check the program in multipath. It consists of two steps: it first monitors and identifies the memory allocated manipulation such as stack instructions or heap functions in execution, and maintains a structure to record the red-zone address, then checks whether the range of writing operation is crossing the red-zone. For the second step, because some write addresses could be symbolic pointers, and S2E can't deal with symbolic pointer directly. It will concretize the symbolic pointer. MACBin intercepts this symbolic address before concretizing it, and checks both the concrete and symbolic address at the storing time.

On the whole, we developed and improved plugins such as stackmonitor, heapmonitor, call/jump symbolic pointer checker, memorychecker, stackchecker, heapchecker and bugreporter. And our bugreporter plugin connects the OnVulnDetect signal emitted by the checkers, which contains the execution state, checking address located memory frame, bug type, and checking mode parameters. To avoid report the same bug repeatedly, we distinguishes the bugs by comparing the call stack of current instruction in the concrete checking mode. However, when working in symbolic checking mode, the stack checker and heap checker are checking at storing time, the call/jump symbolic pointer checker are checking at executing time. They may report the same vulnerability redundantly. For this case, except outputting the current PC address, the path constraints information will also be outputed.

4. Evaluation

We evaluate MACBin effectiveness against buffer overflow in symbolic and concrete checking mode respectively. We assume that the program can reach the vulnerable paths by concrete, symbolic or concolic execution. In particular, we focus on the detecting ability of our tool for the explored paths, the impact and effect in detecting various real-world binary software, and the precise and performance of our checkers.We study Microsoft Excel to evaluate MACBin's ability and performance in detecting real-world software buffer overflow vulnerabilities. Microsoft Excel is a popular and large spreadsheet application. It has a stack buffer overflow vulnerability in the Excel 2007 SP2 un-patch version[9]. This vulnerability occurs because the bytes that located in some offsets of the input file is involved in a pointer arithmetic, whose result is used as the start address of a writing block without strict restraining. The attacker can control these bytes to overwrite a return address for exploit. It could allow remote code execution if a user opens a specially crafted Excel file. Successful exploitation of this vulnerability may allow execution of arbitrary code on a target system, which could make the attacker gain the same user rights as the logged-on user.

Because both the input and path space in Excel were too huge to explore if we took all of the input as symbolic value, we tested it in concolic execution for saving the time and resource. To reach the vulnerable path with the sample file, we searched the POC file to find out the vulnerability related bytes. We verified it by setting them as some good-value which worked well, and bad-value which crashed.

We took the setup of our concolic execution experiment as following steps: Firstly, we obtained a vulnerable sample .xlb file that was generated by the POC exploit. Secondly, we found out the offset of the vulnerability related bytes in the sample file, and modified with some other value that not triggered the vulnerability. Thirdly, we wrote a configure tool to mark several bytes that contained the vulnerable related offset as symbolic variables, and generated a concolic result file. We used a RamDisk[10] to allocate a memory area as a RamDisk, and copied the concolic result file into the RamDisk, such that S2E can recognize it directly in memory. Finally, we used Excel to open the crafted file in S2E, and S2E would launch MACBin plugins to examine the bugs with path exploring.

The result is that while we marked 5 bytes as symbolic variables for the sample file, we detected the vulnerability successfully and quickly.

5. Conclusions

In this paper, we presented MACBin, an active buffer overflow multipath detector for binaries. MACBin is based on an automatic symbolic execution platform that can explore the input data related paths through a binary program at the granularity of bytes. The conventional wisdom is that buffer overflow should not overwrite any intended memory object, which needs to either obtain the detailed memory variables size information or analysis the program semantic of object reference. This approach is more suitable for checking the source code. Our approach considers the potential damage of the buffer overflow, which detects the vulnerability directly or delays the influence of overwrite memory to latter execution or examine. Due to our process needs not to analyze the program semantic to obtain the memory information in detail; it's more suitable and efficient for binary programs.

Our evaluation shows that MACBin works well in practice. It needs no source code to analysis the program semantic, scales to millions of lines of instructions, and has successfully found buffer overflow errors in each explored path. We believe that the buffer overflow vulnerabilities arise due to the fact that security checks before writing a memory object are not exact or premise, and it is often difficult to check if the correct set of checks are being performed on every program path, especially in complex programs. Without needing to understand the application logic and program semantic, our approach can provide a simple but effective checking policy.

However most of buffer overflows are depending on the length of the copied bytes. It can only cover the vulnerable path if the length is long enough. And the input is often fixed in symbolic execution for the convenience of solving the constraints, and a long input will introduce serious path explosion problem. We will expand the study of this work to solve this problem in future.

Acknowledgment

This work was financially supported by the National Natural Science Foundation of China (61701534).

References

[1] Kim H E, Yoo D, Kang J S, et al. Dynamic ransomware protection using deterministic random bit generator[C]// IEEE Conference on Application. 2018.

[2] Cowan C, Pu C, Hintony H, et al. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks[C]// Conference on Usenix Security Symposium. 1998.

[3] Kc G S, Keromytis A D. e-NeXSh: Achieving an Effectively Non-Executable Stack and Heap

via System-Call Policing[C]// Computer Security Applications Conference. 2005.

[4] Rohlf C, Yan I. The Security Challenges of Client-Side Just-in-Time Engines[J]. IEEE Security & Privacy, 2012, 10(2):84-86.

[5] Koo H, Chen Y, Lu L, et al. Compiler-assisted Code Randomization[C]//2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018: 461-477.

[6] Sadowski C, Aftandilian E, Eagle A, et al. Lessons from building static analysis tools at Google[J]. Communications of the ACM, 2018, 61(4): 58-66.

[7] Hu Z, Zhang Y, Wang H, et al. MIRAGE: Randomizing large chunk allocation via dynamic binary instrumentation[C]//Dependable and Secure Computing, 2017 IEEE Conference on. IEEE, 2017: 98-106.

[8] Chipounov V, Kuznetsov V, Candea G. The S2E platform: Design, implementation, and applications[J]. ACM Transactions on Computer Systems (TOCS), 2012, 30(1): 2.

[9] CVE-2011-0104 http://cve.mitre.org/cgi-bin/cvename.cgi?name =CVE-2011-0104.

[10] RAMDisk http://memory.dataram.com/products-and-services/soft-ware/ramdisk.

[11] Corin R, Manzano F A. Taint analysis of security code in the KLEE symbolic execution engine[C]//International Conference on Information and Communications Security. Springer, Berlin, Heidelberg, 2012: 264-275.