UAF Bug Checking Model Based on Dataflow Analysis

Yongji Ouyang, Qingxian Wang, Qiang Wei, Jie Liu
Zhengzhou Information Science and Technology Institute
Zhengzhou, Henan, 450002, China
E-mail:oyyj07@gmail.com,wangqingxian2008@vip.sina.com,funnywei@163.com, coco368@gmail.com

Abstract—**Although there are some tools supporting to detect the program's use-after-free vulnerability, their performance may be degraded because of the restriction they have. In order to detect the program's use-after-free vulnerability with a relatively higher efficiency, in this paper, we propose an automated approach for checking the use-after-free vulnerability in the application. First, we deliberate and choose the method of analyzing the dataflow statically. Then we track all definition and usage for the variables in the application. Finally, the techniques of the equivalent variable and alias analysis are introduced. We have implemented the approach of this thesis in a tool called UAFChecker which can detect use-after-free vulnerability automatically and have conducted experiments with several real-life case studies, experimental results show that the tool can detect the defects of actual application with low false positives and negatives probability.**

Keywords-use-after-free vulnerability; dataflow analysis; equivalent variable; alias analysis

I. INTRODUCTION

In recent years, the number of use-after-free (UAF) vulnerability becomes increasing, resulting in a serial of serious harm such as CVE-2010-0135， CVE-2010-0806 and CVE-2010-1297. The influence of this vulnerability occurs after the pointer to memory in the program is released. Because the pointer is not properly handled, when the released memory is quoted again, it may produce unforeseen circumstances. It is the simplest case that data corrupt, while it is serious that any procedure can run [1].

This vulnerability results from inherent flaws in programming language[2]. The C programming language, for instance, does not explicitly prohibit such a situation like UAF due to the reason that memory operation functions, such as malloc in the C, are actually a package of the interface of operating system memory management. When it is released, the memory pages being used by the process aren't usually released, and it is marked with "redistributed". This mechanism is the reason of UAF security problems. UAF is more common on the browser. When some objects are processed in asynchronous way, due to the flexibility of javascript [3], programmers cannot take all the circumstances into account, and some problems occur inevitably. Many vulnerabilities like this have occurred in 2010. Take CVE-2010-0249 vulnerability on the IE browser, for example, An attacker crafted web page code, and after CTreeNode was released, modified the original memory content by a particular value. Then combined with heap spray technology, the attacker filled the corresponding memory area, in order that CElement::Doc function calls shellcode code ,and the attacker can achieve the rights of the browser process to execute arbitrary code remotely [3,4].

II. RELATED WORK

Finding and detecting the program faults of memory operation is very important for improving the robustness and stability of procedure. At present the method of memory detection widely used can generally be divided into two kinds. Those are the static analysis and dynamic analysis methods [5]. Static analysis methods often be proposed based on the static disassembly process, and dynamic analysis method is to trace the program execution and search for memory state to carry out the dynamic program analysis.

Researchers produce some tools based on the both methods. But the tools have their certain applicability. Nicholas Nethercote develops a instrument named memcheck by using valgrind [6]，which can check all of the memory read / write operations and the interception of calls like *malloc*, *new*, *free*, *delete*. It is good for detecting memory management problems, but can not deal with aliasing problems. In addition, the tool only supports Linux platform. Similarly, Dr. Memory based on DynamoRIO made by Derek Bruening has the same problems [7], and it lacks support for C++ Language. As to say the open-source tools cppcheck [8], it can detect possible memory errors, and provides a simple analysis such as Mismatching allocation and deallocation. But cppcheck is for source-level static analysis method. There is no detection for binary program; so the scope of utilization is limited. As for the funsniff, a plugin of Immunity Debugger, it leaves away from restrictions from the source. But it relies on the debugger, it is limited by the debugger and can only automatically detect the UAF defect in the dynamic implementation in an inefficient way .In addition, it was also not function alias analysis [9].

Considering the situation above, UAF vulnerability detection method is introduced based on use-define and define-use chains in this thesis, and the main idea is: Use the static single assignment form to analyze the data flow for definition and further use. Then find the all equivalence pointer variables that point to the same memory address and depict UAF vulnerability model. Finally, statically analysis the instructions interprocedural and intraprocedual to find the existence of such vulnerability.

978-1-4244-6943-7/10/$26.00 ©2010 IEEE

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 08,2020 at 14:03:50 UTC from IEEE Xplore. Restrictions apply.

III. DEFINE-USE AND USE-DEFINE CHAIN MODEL

A Use-Definition chain (UD chain) is a data structure that consists of a use, U, of a variable, and all the definitions, D, of that variable that can reach that use without any other intervening definitions. A definition can have many forms, but is generally taken to mean the assignment of some value to a variable.

A counterpart of a UD chain is a Definition-Use chain (DU chain), which consists of a definition, D, of a variable and all the uses, U, reachable from that definition without any other intervening definitions.

A. The Relevant Definition of Model

To describe the variable, some agreement and set must be proclaimed first in the statements of program.

- The collection of statement can be expressed by using $s(i)$. In the signature, the variable i is confined as a integer in the interval $[1, n]$. n is to count the number of statements at basic blocks.
- The variable is marked in italics. For example the statement of variables u / v can be differentiate by use their block letters U / V or using $s(0)$.
- Suppose each variable in the program have a defined context.

By the constraints above, variable which is located to the left of an assignment could be defined, such as v. If the statement if $s(j)$, the $s(j)$ can be a definition of v ,and each variable at least one such definition. If a variable v is used, for example in $s(j)$, then the variable v is located to the right of $s(j)$. There is also a statement $s(j)$, in addition $i < j$, then there must be a variable v and a usage in $[i, j]$, such as $s(j)$.

B. Define-use Chain and Use-define Chain Models

Since the order of program statements is executed, assuming that a statement $s(j)$ is a definition statement. When $i < j$ and $k \geq j$, if the definition has a usage somewhere as $s(k)$, then it is effective to the j. In the statement i, the effective definition collection of i is defined as $A(i)$, the effective definition can be numbered in $|A(i)|$. And for $k < i$, a definition in the statement $s(i)$ has also been defined in the statement $s(k)$, then the previous definition would have become liable to the same definition of i.

A variable in the process for the implementation can be obtained with some certain steps [10], assuming variables to be dealt is v, the algorithm that look for its DU chain is shown in Fig.1:

Input: IDA disassembly code
Output: define-use chain

Algorithm
Begain
while(isEnd):
    Search(var the first_v);
    if(isDefine) label(writeAccessCode);
    if(isUse) label(writeAccessCode);
    record[v,label(writeAccessCode),label(writeAcces
sCode)];
return RecordsList
End

Figure 1. The define-use chain algorithm.

By the above algorithm, combined with read access and write access, a variable that can be completed.

And the method of building a UD chain is shown in Fig. 2:

Input: IDA disassembly code
Output: use-define chain

Algorithm
Begain
Set definitions in statement s(0);
For i in [1,n]:
    Find(usedLiveDefinitions) in statement s(i);
Linker(definitions,uses);
Set the statement s(i) as definition statement;
Remove previous definitions;
End

Figure 2. The use-define chain algorithm.

With this algorithm, two things are accomplished:

1) A directed acyclic graph (DAG) is created on the variable uses and definitions. The DAG specifies a data dependency among assignment statements, as well as a partial order (therefore parallelism among statements)

2) When statement $s(i)$ is reached, there is a list of live variable assignments. If only one assignment is live, for example, constant propagation might be used.

IV. USE-AFTER-FREE VULNERABILITY DETECTION MODEL

In this section we continue the discussion about handling the UD and DU chains which points to the memory variable pointer waiting to be released. Then it finds all such kinds of equivalent variables. In addition, in each path it individually records variables equivalent to each other. After that, it analyzes program instructions and models the UAF vulnerability.

A. Model Framwork

Based on the researches of UD and DU chains, this thesis designs and implements a testing framework which supports the executable binary pointer vulnerability by inaccurate pointer main module framework is shown in Fig.3:

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 08,2020 at 14:03:50 UTC from IEEE Xplore. Restrictions apply.

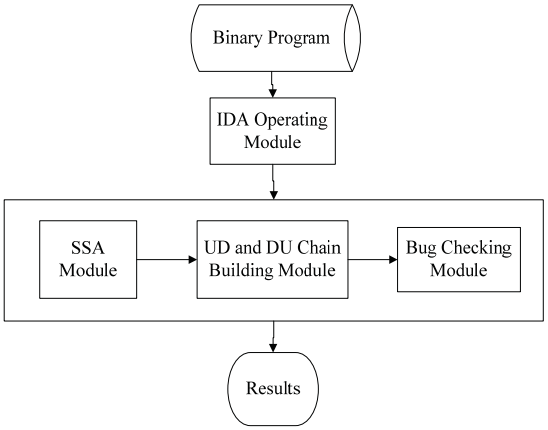

Figure 3. The framwork of the model.

In the system above, first test make binary data as the input and analyzed by IDA. Then it use SSA intermediate representation to treat the result from IDA. The next step is to set up UD chain and DU chain for variables from processed code in UD and DU chains establishment module. Then put the results in the leak detection module and at last, save the outputs in the shape of instruction and address of the problem.

B. Search for Equivalent Variable

UAF vulnerability is caused by improper operation in release memory. So the first thing should be handled is the pointer points to memory. Examine all the variables point to somewhere of a memory to find if they are released rightly. If not, you can believe that the use of these variables will have a bug. To get this row of equivalent variables，in the thesis the use of variables and the value which are calculated are made sure by execution path of the point from reverse reference department. Record all the equivalent variables and recursively process one of those variables with an equivalent one. The specific algorithm shown in Fig. 4

Input：IDA disassembly code
Output：Equivalent set of variables

Algorithm：

```
Begain
initialize Equivalent set of variables Set[equivVarList]
while：allDone
 if dst in Set[needDefList]
     src = insDataFlow.getSrcFor(dst);
     Set[needDefList].append(src);
         if src not in Set[clearedVars]
             if src not in Set[equivVarList]
                 Set[equivVarList].append(src);
     Set[needDefList].remove(dst);
     Set[clearedVars].append(dst);
return Set[equivVarList]
End
```

Figure 4. The algorithm of searching equivalent variable.

C. Analysis on Intraprocedure

For a function, because it uses some of the variables which is defined in its own function and some are not defined in the function, such as global variables. In this thesis, the examination includes function called by free has such problems; therefore, the analysis on intraprocedure is essential.

In this algorithm, at the beginning it should be started to use the API calls to find a function or function alias. To find the UAF error, it needs to be found the free function call or its alias function call. Such vulnerabilities can be triggered if the function is recorded as a binary array (f,n), where f is the function in question and n is going to release or be closed .The n can also be other these kinds of operations pointer variable. In addition, its alias can be recorded as (g,m), in which g is invoking function if x plays as the independent variable m. So in (f,n), f is invoking function if x plays as the independent variable n. By alias analysis, combined with the previous method, you can find the error intraprocedure.

First, use the equivalent variables algorithm which has been introduced previously to establish the set of equivalent variables about the variable n in the function f, which can be assumed as E. Consider the g which should be analyzed later, if there is independent variable established inside the collection been described in paragraphs above, then it can be defined that (g,m) is the alias to (f,n). The m is the serial number of variables, the error between the process can be carried out by the two-tuple (g,m) to find. In this process, the main difficulty is the tracking function of variables. This thesis talks about the methods uses the IDA's handling mechanisms, such as the form identification as arg_X, automated renaming of memory address, etc.

D. UAF Vulnerability Modeling

The reasons of UAF bug is mainly caused by the program, which uses the memory pointer has been released to calculate the new memory address, thus creating unpredictable results. Therefore, for the detection of such bug, the main danger is checking operation described above.

When those tests are in practice, it has been given an instruction inst, memory addresses or register x and the set of released pointer S. If you want to join x to the set of S, then if and only if the condition can be satisfied:

$$x \in inst_{dsts} \wedge (y \in inst_{srcs[x]} \mid y \in S) \neq \varnothing \qquad (1)$$

Among these, the $dsts$ is a set of the destination operand of one instruction. And $srcs[x]$ is the set of source operand which influent the destination operand.

If you want to remove x from set S, then if and only if：

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 08,2020 at 14:03:50 UTC from IEEE Xplore. Restrictions apply.

$$x \in inst_{dsts} \wedge (y \in inst_{srcs[x]} \mid y \in S) = \varnothing \qquad (2)$$

Through the description above, this approach this thesis chooses is: in each instruction we traverse through all of the base register used to calculate the pointer to determine whether they has been in the collection of the released pointer. Specific algorithm shown in Fig. 5:

Input：IDA disassembly code
Output：suspected Set of place

Algorithm：
```
Begain
CheckUseFreeBug(addr,src,dst)
for instruction in [src,dst]
        if isInstance(instruction, Taint_Regs)
          for useReg in instruction.regs
             if useReg in Set[freeVars]
                isBug(addr,useReg)
End
```

Figure 5. The algorithm of bug checking.

In the algorithm to find UAF bugs, Taint_Regs representing at least one operation for register to build memory address structure. Src and dst object represents the source operand and destination operand in a program instruction. FreeVars is the set of equivalent variables waiting to be released. The useReg is the register in use. The addr is the address in which program does such a dangerous operation.

V. EXPERIMENTS

This thesis implements a simple IDA's Python language verification tools UAFChecker. The tool runs on Windows systems. Test hardware environment: Intel Pentium (R) 4 processor 3.00GHz, Memory 2G. The tests mainly aim to the open source software or development packages which have been use-after-free vulnerability history such as Webkit, Clam Antivirus and libxml2. Then test the software to see if can find loopholes in the existing procedures. The results have proved the correctness of the algorithm. Table 1 is the specific experimental data.

TABLE I. THE DATA OF TESTING CASE

Application	Comparative Aspects			
	free calls	*alias*	*bugs*	*false positive*
Webkit	9035	0	5	1
libxml2	1621	8	4	1
Clam Antivirus	1412	9	3	0

After using this tool to test webkit, it totally reported of five errors. Then we manually analyze these errors and found four UAF defects. Fig. 6 Shows a defective parts. It is a part of core function in webkit. This function in webkit can be used as part of memory management system. When a render target releases, it will call this function. We can see from the code above, ptr +0 position will be filled with the address which has just been released. When you call the object function released, this function address is actually pointing to the location of a previously released attribute object. If you can control the value of this attribute, then the eip is controllable, thus to achieve the purpose of execution of arbitrary code. In fact, in most cases, this property can be controlled through custom properties, EIP is controlled [11].

```
.text:10003590 this          = dword ptr  4
.text:10003590
.text:10003590 size = eax
.text:10003590 ptr = edx
.text:10003590          add    size, 3
.text:10003593          shr    size, 2
.text:10003596          shl    size, 2
.text:10003599          cmp    ize, 190h
.text:1000359E          jnb    short locret_100035B1
.text:100035A0          mov  ecx, [esp+this]
.text:100035A4          shr    size, 2
.text:100035A7          lea    size, [ecx+size*4+1Ch]
.text:100035AB          mov   ecx, [size]
.text:100035AD          mov  [size], ptr
.text:100035AF          mov  [ptr], ecx
.text:100035B1
.text:100035B1 locret_100035B1:   ; CODE XREF:
WebCore::RenderArena::free(uint,void *)+E↑ j
.text:100035B1                retn   4
.text:100035B1 ?free@RenderArena@WebCore@
@QAEXIPAX@Z endp
```

Figure 6. Example of a use-after-free vulnerability.

In experiments of Clam Antivirus and libxml2, it has been found that some errors occur by free alias function call. such as cl_engine_free(). Using the algorithm of function alias, UAF errors can be correctly found. In Clam Antivirus, there was three such errors which were found by alias analysis. The places are: cli_ac_free() in contrib/clamdtop/clamdtop.c; free in cli_mp_hex2ui() of libclamav/mpool.c; phishing_done() in libclamav/readdb.c. There are two UAF libxml2 errors which are due to direct calls to free. The other two are caused by free alias. The specific reasons for those errors are concerned in CVE-2009-2414，CVE-2009-2416. Compare the results in Table 1. The method described in this article proved possible UAF defects. There are also a small number of false positive results.

In addition, we studied similar detection tool cppcheck, the plug-in funsniff in Immunity debugger and Dr. Memor . Table 2 is comparison in specific situation.

TABLE II. COMPARISON WITH OTHER TOOLS

Application	Comparative Aspects			
	Alias Support	*False Positive and Negative*	*Checking Type*	*Platform*
UAFChecker	yes	12%	static	windows, linux
cppcheck	no	13%	static	windows, linux
funsniff	no	28%	dynamic	windows

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 08,2020 at 14:03:50 UTC from IEEE Xplore. Restrictions apply.

Application	Comparative Aspects			
	Alias Support	*False Positive and Negative*	*Checking Type*	*Platform*
memcheck	no	—	dynamic	linux
Dr. Memory	no	15%	dynamic	windows, linux

From the data in the table above can be seen, the false positive rate of UAFChecker is about 12%, and static analysis tools cppcheck performance closer, a little better than Dr.Memory based on dynamic instrument, much better than funsniff detection capabilities. From Table 2, the UAFChecker which based on IDA has the good cross-platform. Overall, the proposed method and the model have good practicability.

But for the reason that the algorithm goes without path-sensitive method, and there can be some unexpected conditions, so some path cannot reach. So it will produce false positive test sometime [12, 13].

VI. CONCLUSION

This thesis implements a use-after-free vulnerability auto-detection tool, it can find UAF vulnerability and record the address and process context of the errors automatically. Some existing vulnerabilities has be verified, and achieved good results.

However, because of the limitations of the method mentioned in the following areas need further improvement in the future: the first, as constraint solving [14, 15] and other special operations have not been used for procedure path execution, false positives and false negatives are caused by the path does not reach. Therefore, in order to solve this problem, further method need to introduce constraint solver to reduce the path does not reach. The second is, since this thesis is proposed to improve the standard UD and DU algorithm, there is no In-depth analysis of the dataflow with complex path [16]. And the false negatives are hardly to determine. At last, the alias tracking and analysis in the binary is relatively more difficult than the source code. This thesis used the IDA analysis of treatment results briefly, and it can only deal with relatively easy targets. More in-depth aliasing analysis is needed.

ACKNOWLEDGMENT

This work was partially supported by the national high technology research and development program ("863"program) of china (grant no. 2008aa01z420).

REFERENCES

[1] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," Lecture Notes in Computer Science. Compiler Construction. Heidelberg: Springer Berlin, 2004, pp. 5–23.

[2] M. Costa, M. Castro, L Zhou, L Zhang, and M. Peinado, "Bouncer: securing software by blocking bad input," 21st ACM SIGOPS symposium on Operating systems principles, Stevenson, Washington, USA, 2007. New York, NY, USA: ACM Press, 2007, pp. 117-130.

[3] J. McDonald, C. Valasek. "Practical Windows XPSP3/2003 Heap Exploitation," Black Hat USA. Jul, 2009.

[4] HD Moore. "Metasploit3: Exploit Intelligence and Automation," Microsoft Blue Hat 3, 2006.

[5] Dawson Engler and Madanlal Musuvathi, "Static Analysis versus Software Model Checking for Bug Finding," VMCAI 2004,LNCS 2937,2004, pp. 191-210.

[6] Nicholas Nethercote. "Dynamic Binary Analysis and Instrumentation. PhD Dissertation," University of Cambridge, November 2004.

[7] derek bruening. "Memory Debugger for Windows and Linux." http://dynamorio.org/drmemory.html, 2010.

[8] http://sourceforge.net/apps/mediawiki/cppcheck/index.php?title=Main_Page.

[9] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," Proceedings of the 12th Annual Network and Distributed System Security Symposium, San Diego, California, 2005. The Internet Society, 2005.

[10] Kian Salem,Übung zu Software Engineering. http://www.wi.unimuenster.de/pi/lehre/ws0910/se/uebungen/Uebung6-Vorstellung.pdf, 2010.

[11] Alexander Sotirov. "Heap Feng Shui in JavaScript," Black Hat Europe. 2007.

[12] P. Boonstoppel, C. Cadar, and D. Engler, "RWset: attacking path explosion in constraint-based test generation," 14th International Conference, TACAS, Budapest, Hungary, 2008. New York: Springer-Verlag, 2008, pp. 351-366.

[13] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," Microsoft: MSR-TR-2008-123, 2008.

[14] L. de Moura and N. Bjørner. "Z3: An efficient SMT solver," In Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008, pp. 337−340.

[15] Robert Brummayer and Armin Biere. "Boolector:An effcient SMT Solver for Bit-vectors and Arrays" 2009, pp. 1174–177

[16] R. Chugh, J. W. Voung, R. Jhala, and S. Lerner. "Dataflow analysis for concurrent programs using datarace detection," In Conf. on Programming Language Design and Implementation(PLDI), 2008, pp. 316–326.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 08,2020 at 14:03:50 UTC from IEEE Xplore. Restrictions apply.