

Detecting Dangling Pointers in SVF

Siddhant Singh, Balaji Ravikumar, Isaac Chua
Georgia Institute of Technology

Abstract

Understanding the flow of information is crucial in software engineering. Static value flow analysis is a technique for collecting information about the set of values at different points in a program. A dangling pointer is a pointer that is pointing to a freed or deleted memory location. In this paper, we modify and perform static value flow analysis using SVF to detect dangling pointers which can lead to undefined behavior in software. By identifying these bugs in codebases, it is possible to discover new vulnerabilities in existing software and patch them, generally improving software security. This paper describes the implementation of a pointer usage identification tool to identify various memory usage errors that can be security defects. We evaluate the tool’s precision and soundness using known and potentially vulnerable codebases. This implementation is available at <https://github.com/aditmohan96/SVF-DPD/>

Keywords: SVF; dangling pointer; security, analysis

1 Introduction

In the software development life cycle, because of the complexity of modern software, it is often cheaper to discover and fix software bugs earlier rather than later in the process. Besides the considerable cost savings, detecting bugs before a software product is released will also result in higher quality software.

In software, static code analysis is the method of examining the source code and performing analysis on a program without executing said program. The benefits of performing static analysis are such as speed and depth of analysis. Reviewing code manually can take a significantly longer amount of time when compared to running automated static analysis tools. Static analysis can also provide software engineers with an in-depth analysis of the code that covers every possible execution path. In this paper, we focus on detecting dangling pointers by making the necessary changes to the SVF project.

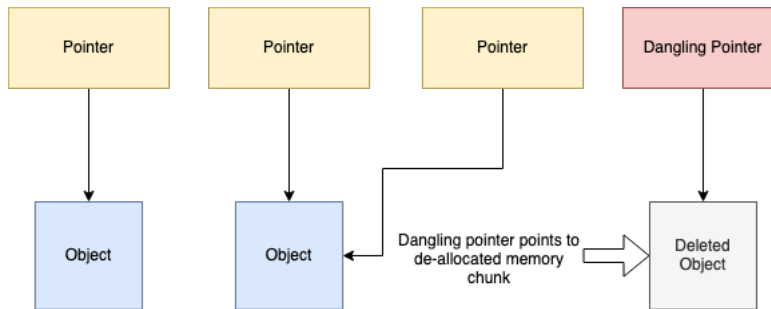


Figure 1: Diagram illustrating the difference between normal pointers and a dangling pointer

As shown in the above scenario, because the dangling pointer points to a memory chunk that is invalid, the use of this dangling pointer leads to unexpected or undefined behavior. Generally, such bugs result in applications or threads crashing. A more malign scenario can occur when the dangling pointer is used to point to malicious code. Hence, the detection and elimination of dangling pointers is crucial for preventing the exploitation of such pointers.

1.1 Motivating Examples

A notable example of a use-after-free (UAF) vulnerability is CVE-2012-4969. This particular vulnerability was a zero-day Internet Explorer bug that was exploited in September 2012.

In the scenario depicted in Figure 2, the use-after-free vulnerability appeared in the `CMshtmlEd::Exec()` function while the object that was freed was being pointed to by the this pointer. JavaScript DOM method

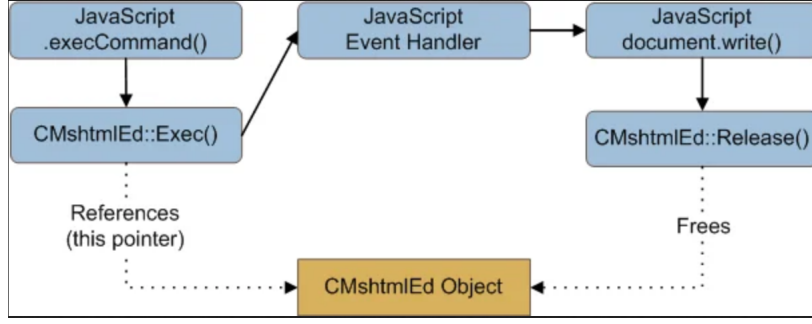


Figure 2: CVE-2012-4969 (IE CMshtmlEd UAF)

`execCommand()` is used to reach the `CMshtmlEd::Exec()` function. In order to free the `CMshtmlEd` object, the attacker invokes `execCommand()` to trigger a JavaScript event (attacker-controlled). When this attacker-controlled JavaScript event handler calls `document.write()`, the `CMshtmlEd` object is freed. The result of this is that the `this` pointer becomes a dangling pointer and is dereferenced causing a use-after-free scenario.

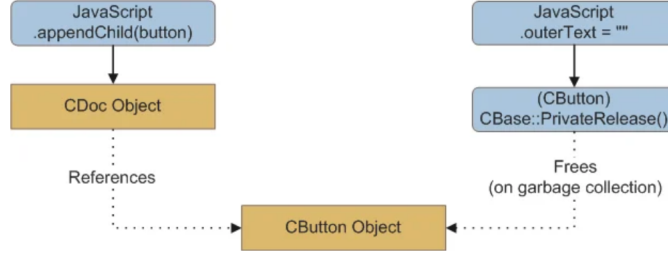


Figure 3: CVE-2012-4792 (IE CButton UAF)

Another illustrative example of a use-after-free zero-day vulnerability can be seen in CVE-2012-4792. In Figure 3, the pointer pointing to `CButton` Object was being stored in the `CDoc` object. The bug here is that the `CButton` object reference count was not being updated. Moreover, when `CButton` object was deleted through the invocation of `.outerText=""`, the `CDoc` object pointer was not being updated. `CButton` pointer becomes a dangling pointer and use-after-free condition arose when dangling `CButton` pointer points to dereferenced `CDoc` object.

2 Related Work

In exploring the existing work and tools available in the space of static analysis, we took the SVF framework and CodeQL into consideration. CodeQL is a semantic code analysis engine used for discovering vulnerabilities in a codebase. CodeQL converts a codebase into a CodeQL database which then allows users to write queries to find security vulnerabilities within the codebase. As such, we looked to see if there exists a CodeQL query that detects dangling pointers or use-after-free scenarios. Indeed, a `UseAfterFree` CodeQL query can be found in the default CodeQL query suite.

SVF is a sparse, selective, and on-demand interprocedural program dependence analysis framework which works on both sequential and multithreaded programs. The SVF framework has been widely used by researchers involved in security and program analysis. We found that the SABER leak detector tool within the SVF framework is particularly relevant to our work. Details on how we extend SABER to detect dangling pointers can be found in the methodology section below.

3 Methodology

In order to leverage the Source Sink analysis capabilities of SVF, specifically the SABER tool built on the SVF framework, we made a class inheriting from the `SourceSinkDDA` class. This allowed us to build our own analysis similar to the Leak Analysis within SABER by utilizing the path reachability solver for finding Source to Sink paths within the Value Flow Graph (VFG). To utilize this capability we transform dangling pointer usage into a Source-Sink reachability problem in the following ways:

3.1 Compilation

Our approach works by leveraging the basic optimization free compilation performed by clang to emit LLVM bytecode. As such, by not applying the mem2reg optimization pass upon compiled LLVM bytecode, all values are stored to and loaded from memory instead of being stored exclusively (or mostly) in LLVM SSA registers. As such, the memory allocated on the heap is immediately stored to memory and all dereferences or uses of this memory location by any pointer leads to a load from the same memory location. This allows us to treat all load statements referencing the memory location of interest as a use of the memory object. Hence, all load statements that occur after a call to a memory deallocation function upon the memory object can be considered a potential Use-After-Free or dangling pointer dereference. We recognize that this is not entirely accurate in terms of overwriting a dangling pointer value is still considered a Use-After-Free in this paradigm but this assumption still serves as a useful over-approximation allowing efficient solving. This stays in line with the allowed amount of false-positives by over-approximating the program state while not admitting false-negatives. This is done by the compilation command below, which is the standard command for usage of the SVF framework, where the `-g` flag is used to gather debug information that allows our tool to accurately print line numbers and file locations for identified bugs.

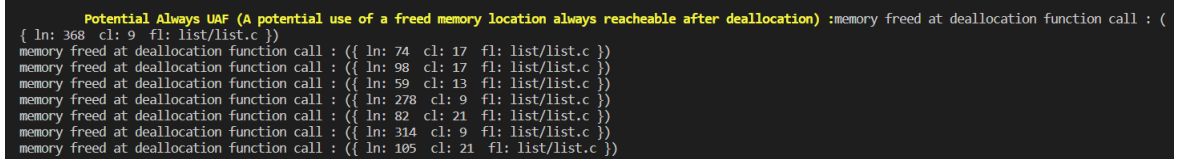
```
clang -S -c -g -Xclang -disable-OO-optnone -fno-discard-value-names  
-emit-llvm <Input>.c -o <Output>.ll
```

3.2 Identifying Sources

To identify sources, we begin by scanning through all the function call-sites identified by SourceSinkDDA in initializing the analysis. By examining all call-sites and the functions being called at them (callees), we identify all calls to memory deallocation functions (such as free, cfree, xfree, obstack free e.t.c). This can be done by comparing function names to a set of known deallocation functions, which is already implemented as part of SABER with its SABERCheckerAPI. By identifying these functions, we use the Program Assignment Graph (PAG) to get a mapping from callsites to the arguments list and retrieve the argument to the deallocation function.

After getting the argument value(s) for the deallocation function, we traverse outward from the node (VFG Node) in a Breadth First Search (BFS). This BFS traversal over the VFG allows us to find the stores of that value to memory, which serves as our Source for analysis. By virtue of the unoptimized compilation described earlier, these stores serve as the store to the memory location(s) that is subsequently used throughout the program for all load statements in order to access this memory object.

While initializing these sources, we also keep a mapping of all the callsites that we scanned over to the source identified from them in order to later retrieve this information for reporting. This allows us to use store statements as Sources while retaining the information on where the stored value was actually deallocated. This leads to reporting of this source associated information as seen in figure 4.



```
Potential Always UAF (A potential use of a freed memory location always reachable after deallocation): memory freed at deallocation function call : ( { ln: 368 cl: 9 fl: list/list.c } )  
memory freed at deallocation function call : ( { ln: 74 cl: 17 fl: list/list.c } )  
memory freed at deallocation function call : ( { ln: 98 cl: 17 fl: list/list.c } )  
memory freed at deallocation function call : ( { ln: 59 cl: 13 fl: list/list.c } )  
memory freed at deallocation function call : ( { ln: 278 cl: 9 fl: list/list.c } )  
memory freed at deallocation function call : ( { ln: 82 cl: 21 fl: list/list.c } )  
memory freed at deallocation function call : ( { ln: 314 cl: 9 fl: list/list.c } )  
memory freed at deallocation function call : ( { ln: 105 cl: 21 fl: list/list.c } )
```

Figure 4: A set of callsites being reported in a detected UAF as distinct function calls relating to the same Source in the program slice

3.3 Identifying Sinks

To identify sinks, we once again scan through all the function call-sites, identify calls to memory deallocation functions and retrieve the argument value(s) for these functions as before. However, at this point we use the Inter-Procedural Control Flow Graph (ICFG) generated by SVF by getting the ICFG Node corresponding to the specific argument being examined (which yields the ICFG node corresponding to the function call). Then, we traverse the ICFG outward in a BFS manner from this node in order to traverse over all the ICFG nodes that may be visited from this node in some execution of the program. This allows us to identify all the ICFG nodes reachable via loops, conditionals or sequential flow from the deallocation function callsite.

We subsequently traverse over all incoming and outgoing edges from the argument's VFG Node in order to locate all load statement VFG Nodes associated with this value. Specifically, we use the information gathered by the ICFG traversal to only identify all load statements corresponding to this value that relate to ICFG nodes reachable after the deallocation callsite as those are the only potential uses of the memory object after deallocation. These load statements that occur after the deallocation within the ICFG are then considered sinks on the VFG for our analysis. This leads to sink related reporting as seen in figure 5.

```

deallocated memory potentially used at : (uaf_ctf_1/uaf_ctf_1.c:39:22)
(uaf_ctf_1/uaf_ctf_1.c:12:61)
(uaf_ctf_1/uaf_ctf_1.c:12:56)
(uaf_ctf_1/uaf_ctf_1.c:12:12)
(uaf_ctf_1/uaf_ctf_1.c:36:32)

```

Figure 5: A set of pointer uses being reported in a detected UAF as distinct lines of code

3.4 Analysis and Reporting

After setting the Source and Sink nodes in the VFG as described above, we utilize the existing Source Sink solving via Program Slices (forward and backward slices utilizing Z3 for conditional edge constraints) to perform Source Sink analysis. By doing so, we are able to get distinct Program slices corresponding to each Source identified. By examining these slices for whether a Sink is reachable from the Source along all or some paths of the Program, we are able to infer potential Use-After-Frees that occur conditionally or during all program execution paths. This information is available via the graph dumps of SVF as seen in figure 6 showing a VFG with annotated source and sink nodes.

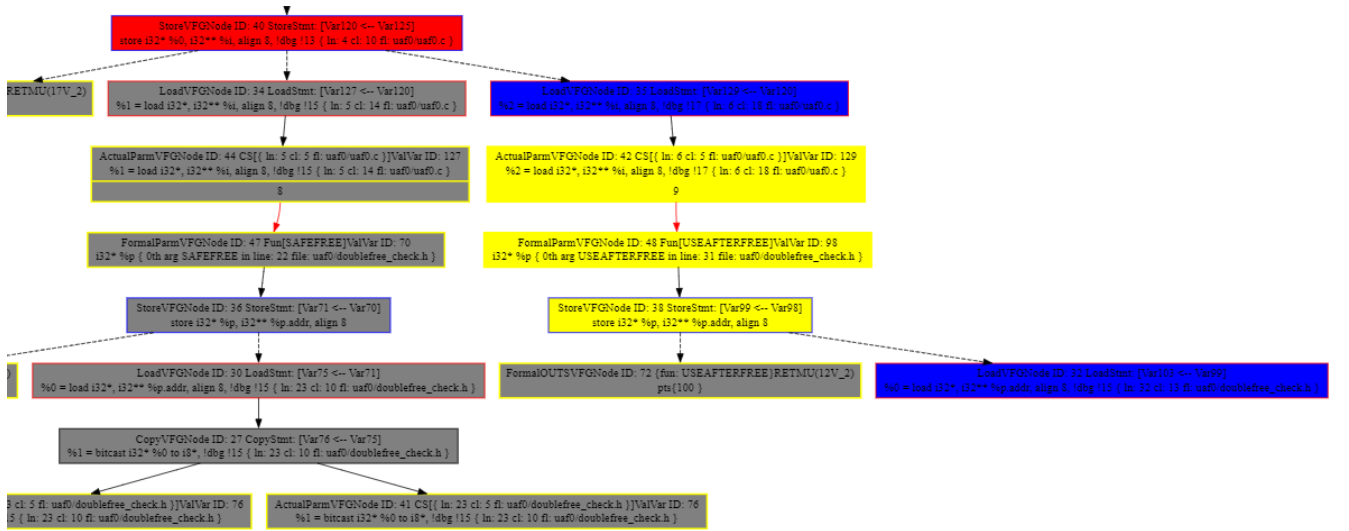


Figure 6: A program slice annotated onto a program VFG with red filled Source and blue filled Sink nodes

This information alongside the mapping of Sources to associated callsites allows us to report potential Use-After-Frees with source code locations via embedded debug information in the LLVM bitcode. This yields reports with Source code line and column information for the deallocation function calls and the potential uses of that memory object after the deallocation. This leads to the overall reporting scheme as seen in figure 7 and figure 8 based on all path or conditional Use-After-Frees.

```

Potential Always UAF (A potential use of a freed memory location always reachable after deallocation) :memory freed at deallocation function call : (
{ ln: 23 cl: 5 fl: uaf0/doublefree_check.h })
deallocated memory potentially used at : (uaf0/doublefree_check.h:32:13)
(uaf0/uaf0.c:6:18)

```

Figure 7: A potential always UAF being reported

4 Evaluation

For the evaluation of our methodology, we looked for examples of code that have a confirmed "dangling pointer". The authors of SVF had provided a micro-benchmark test suite for pointer analysis. This was found in <https://github.com/SVF-tools>. This benchmark contains tests for double free bugs. These double free tests were modified such that each instance of a double free bug was replaced with a pointer de-reference to test for use-after-free as summarised in

```

Potential Conditional UAF (A potential use of a freed memory location conditionally reachable after deallocation) :memory freed at deallocation funct
ion call : ({ ln: 7  cl: 3  fl: test2.c })
deallocated memory potentially used at : (test2.c:10:23)

conditional free path:
--> ({ ln: 9  cl: 6  fl: test2.c })|True)

```

Figure 8: A potential conditional UAF being reported alongside conditional path information

Testcase	Comment	Actual	Reported	False Positives
uaf0	basic use after free	1	1	0
uaf00	multiple free sites, always free	1	1	0
uaf2	use after second free	1	1	0
uaf3	uaf basic interprocedural	1	1	0
uaf34	basic uaf with potential false positives	1	7	6
uaf35	conditional uaf	1	5	4
uaf36	basic uaf depends on global constant	1	1	0
uaf37	basic uaf depends on global	1	5	4
uaf38	uaf inside switch	1	1	0
uaf4	use after free on different pointer to same address	1	1	0
uaf40	uaf with goto statements	1	1	0
uaf41	uaf with bad sink	1	3	2
uaf42	uaf after pointer aliasing (after free)	1	2	1
uaf43	uaf after pointer aliasing (before malloc)	1	2	1
uaf44	uaf inside for loop	1	1	0
uaf45	uaf with free inside funtion call	1	2	1
uaf46	uaf inside function call	1	2	1
uaf5	uaf on bad malloc source (Interprocedural)	1	2	1

Table 1: Evaluation on UAF Test Cases (Modified Double Free) from SVF Test-Suite

Table 1. There are no instances of a Use after free not being detected, however, there still remains some scope to improve the false positive reporting.

4.1 Example Capture the flag Test

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main()
6  {
7      char *name = 0;
8      char *pass = 0;
9      while(1)
10     {
11         if(name) printf("name address: %x\nname: %s\n",name,name);
12         if(pass) printf("pass address: %x\npass: %s\n",pass,pass);
13         printf("1: Username\n");
14         printf("2: Password\n");
15         printf("3: Reset\n");
16         printf("4: Login\n");
17         printf("5: Exit\n");
18         printf("Selection? ");
19         int num = 0;
20         scanf("%d", &num);
21         switch(num)
22         {
23             case 1:
24                 name = malloc(20*sizeof(char));

```

```

25         printf("Insert Username: ");
26         scanf("%254s", name);
27         if(strcmp(name, "root") == 0)
28         {
29             printf("root not allowed.\n");
30             strcpy(name, "");
31         }
32         break;
33     case 2:
34         pass = malloc(20*sizeof(char));
35         printf("Insert Password: ");
36         scanf("%254s", pass);
37         break;
38     case 3:
39         free(pass);
40         free(name);
41         break;
42     case 4:
43         if(strcmp(name, "root") == 0)
44         {
45             printf("You just used after free!\n");
46             system("/bin/sh");
47             exit(0);
48         }
49         break;
50     case 5:
51         exit(0);
52     }
53 }
54 }
55 }
56 }

```

In the above example, there are deallocations to pointers *name* and *pass* at lines 39 and 40. In this test case, we see three locations where we see a UAF.

- case 4 (line.no. 43) can be potentially get called after a pass of case3.
- In line.no. 11 and 12 name and pass can be used as arguments of printf() after a pass of case 3.

All of these potential UAFs were successfully reported by the Tool.

5 Conclusion

In this paper, we successfully extended SABER to include a dangling pointer detector, using the graph reachability source sink analysis. Through this implementation, apart from the issue of false positives, the tool correctly reports all instances of Use After Free in a given program. As the tool is used for a wider variety of code bases we believe, the existing methodology can be tweaked to be more precise by pruning false positives.

References

Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In Proceedings of the 25th International Conference on Compiler Construction (CC 2016). Association for Computing Machinery, New York, NY, USA, 265–266. <https://doi.org/10.1145/2892208.2892235>