Identifying Exploitable Memory Objects for Out-of-Bound Write Vulnerabilities

Runhao Li[1], Bin Zhang[1], and Chaojing Tang[1]

[1] *National University of Defense Technology, Changsha, 410072, China*

Email: lirunhao@nudt.edu.cn.

Exploit an out-of-bounds write vulnerability in general-purpose applications has become a current research focus. Given the large scale of code in programs, selecting appropriate memory objects for exploitation is challenging. In the letter, we propose a corrupted data propagation-guided fuzzing method. By tracking the propagation process of corrupted data among memory objects, we propose a multi-level fuzzing schedule to search the execution paths. Experimental results show that our proposed method, EMOFuzz, can effectively identify exploitable objects under various overflow lengths, significantly enhancing the efficiency of exploitability analysis.

Introduction: Software vulnerabilities are a crucial area of study in cybersecurity, with out-of-bound (OOB) write being a significant component [1,2]. In the process of researching vulnerabilities, the focus and challenge for researchers are how to exploit bugs to compromise the target program and gain higher privileges.

The process of exploiting a vulnerability often relies on the combination of exploit primitives [3]. Common primitives include arbitrary address reading (AAR), arbitrary address writing (AAW), and control flow hijacking (CFH). The exploit process usually involves combining these primitives to achieve a specific exploitation goal. The key to constructing an exploit primitive lies in rewriting critical objects of the program after triggering the vulnerability. For example, in the statement *memcpy(des, src)*, if the corrupted data of a vulnerability controls the *des* pointer, it becomes possible to write data to any address, achieving an AAW primitive. Therefore, researching how to influence critical memory objects in the program through flawed data, to reach an exploitable state (AAW, AAR, and CFH), is crucial for vulnerability exploitation [15].

Current methods for identifying exploitable objects of OOB write bugs primarily focus on the Linux kernel [4, 5]. These approaches concentrate on certain special objects, such as structures containing function pointers or objects with variable lengths [6]. However, in the case of general-purpose applications, which often have a large amount of code, complex logic, and significant variations in development styles, no effective analysis method is proposed to identify exploitable objects for memory vulnerabilities. Therefore, developing a method for analyzing exploitable objects for general-purpose programs, to assist in assessing the exploitability of OOB vulnerability, becomes a hot topic and a challenge in research.

In general-purpose programs within complex execution logic, after a vulnerability is triggered, it is often difficult to directly influence a memory object to reach an exploitable state, which makes the identification of exploitable objects challenging. In most cases, after rewriting a memory object via triggering bug, the flawed data propagates and spreads among memory objects, changing the program's state. For instance, as shown in Figure 1, after the vulnerability is triggered, object A is firstly affected, and then object A influences object B, which in turn affects the pointer *des*. When the program executes the statement *memcpy(des, B)*, AAW occurs. In this process, the program reaches an exploitable state by rewriting memory object A, making it an exploitable object. Considering the complexity of corrupted data propagation due to the program's complex logic, tracking the propagation of the flawed data is key to identifying the exploitable object.

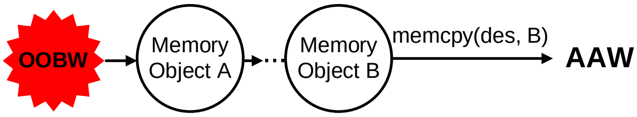

Fig 1 *An example of achieving AAW by triggering vulnerability.*

This letter adopts a data flow perspective to explore exploitable objects of OOB write bugs in general-purpose applications, thereby providing support for exploitability analysis. Initially, the letter utilizes static value flow tool to instrument the target program, constructing a weighted value flow graph. Subsequently, we integrate dynamic taint analysis, and develop a fine-grained model for propagation process of corrupted data. Following this, a fuzzing method is designed, using the extent of flawed data spread as a guidance to efficiently explore the exploitable memory objects. The contributions of this paper include two parts:

- Constructing a hybrid static-dynamic model for fine-grained analysis of corrupted data propagation, tracking the spread of corrupted data among memory objects, and evaluating the propagation impact.
- Proposing a corrupted data propagation-oriented fuzzing methodology, which can identify exploitable objects by efficiently explore the program execution paths.

System Design: The overall framework of our proposed method EMOFuzz is shown in Figure 2. Initially, we carry out instrumentation within static analysis tool and generate a Weighted Variable Flow Graph [7]. Based on that, a taint data propagation model is constructed to analyze the spreading process of corrupted data. Following this, we propose a flowed data-oriented fuzzing method. Through a multi-level fuzzing schedule, it searches for potential exploitable objects in the target program.

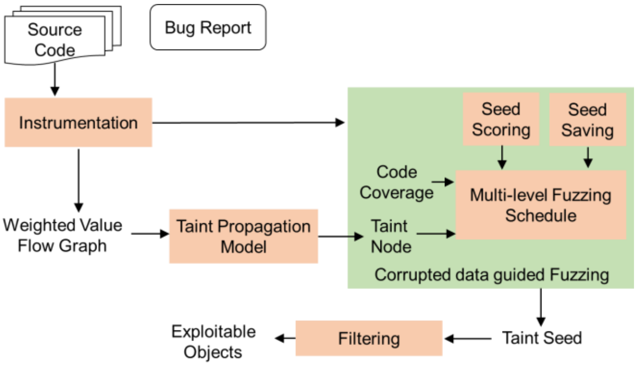

Fig 2 *System overview of proposed method.*

It is worth mentioning that one important assumption for our method is that the memory objects can be affected by the bugs through specific methods, such as heap layout manipulation [8,9]. In this way, after triggering a vulnerability, it becomes possible to drive the program into an exploitable state by affecting these critical objects.

Corrupted Data Propagation Model: To figure out the propagation process of corrupted data among memory objects and evaluate the effects, we construct a model combined static and dynamic method, which also provides support for fuzzing.

Firstly, this letter leverages the Static Value Flow (SVF) [7] framework for instrumenting the target program in LLVM level. SVF is a common data flow analysis framework that effectively supports cross-module and cross-procedure data flow analysis in large-scale programs. Given source code, we leverage SVF to construct a Value Flow Graph (VFG), which shows the data flow relationships among variables. For example, Figure 3 is part of VFG, where each node represents a variable or expression of the program, and each edge denotes the data flow relationship. In Figure 3, the variable %3 influences variable %10 through the *getelementptr*, and variable %10 further affects the variable %11.

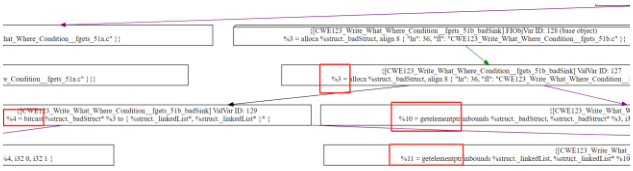

Fig 3 *Part of value flow graph of applications.*

In the VFG, the node possesses varying potential for spread. To assess the effects of propagation more accurately, we have advanced from the basic VFG to construct a Weighted Value Flow Graph (WVFG). Propagation potential refers to the extent to which a tainted variable node might affect other nodes. For example, in Figure 3, if variable %3 is tainted, variables %4 and %10 might also be tainted. Similarly, if variable %10 is tainted, variable %11 might be tainted as well. Therefore, the propagation potential of node %3 is 2, while that of node %10 is 1. Nodes with higher propagation potential are more significant for the spread of tainted data, as they are likely to facilitate wider dissemination once tainted. Based on the basic VFG, we quantify the propagation potential of each node according to the number of outgoing edges, thus resulting in the WVFG.

In OOB write bugs, the flawed data area varies as different overflow length, resulting in different exploitable memory objects. Therefore, to accurately assess the exploitability of a bug, we explored the exploitable objects under the influence of corrupted data within varying lengths. To finely track the propagation process of the corrupted data, we employed dynamic taint analysis, using the Dataflow Sanitizer [10] for online taint analysis. Additionally, to detect whether the flawed data has propagated to an exploitable position, we need to monitor important nodes during program execution. We employ Dataflow Sanitizer for dynamic runtime detection and conduct inspections at various critical points during the program's execution. The specific detection rules are outlined in Table 1.

Table 1. Detection rules of LLVM instructions for exploitable program state.

Instructions	Rules	Exploitable State
load %a, %b	Address of %b is tainted	AAW
store %a, %b	Address of %b is tainted	AAR
call func	Address of func ptr is tainted	CFH

Corrupted Data-oriented Fuzzing: Based on the analysis of propagation process, we developed a fuzzing method to efficiently explore the exploitable objects for OOB write bugs in general-purpose applications. Considering the large code scale and complex logical constraints of programs, methods such as symbol execution are inefficient and may suffer from constraint explosion [11]. Therefore, we adopted a search-based approach to explore the execution paths in programs.

The more extensively corrupted data propagate, the more likely the program reaches an exploitable state. Therefore, we use the extent of flawed data propagation to guide the fuzzer. Based on the corrupted data propagation model, we define the extent of corrupted data propagation. For a saved seed, if it can taint m more nodes, and the propagation potential of each node is p_1 to p_m, then the propagation extent of the seed (PES) is described in (1).

$$PES = \frac{m}{2} + \sum_{i=1}^{m} p_i \qquad (1)$$

To ensure thorough exploration of the program execution space [12,14], we also maintained code coverage as metric. Hence, we designed a multi-level fuzzing schedule by integrating both code coverage and PES. In terms of seed saving, we preserved seeds that discover new tainted nodes or trigger a new path. As for seed mutation strategy, we use random mutation algorithm and select seeds within a greedy algorithm. Regarding seed scoring, we evaluate each seed within a new scoring strategy, which is further illustrated in Algorithm 1.

Algorithm 1. Calculate the score of seed during fuzzing.

```
Algorithm 1: Seed Score Calculating
Input  : Current seed $q\_cur$;
         Number of new tainted node $n$;
         Propagation potential of each new tainted node $p_i$;
         Old score by original fuzzer $S_o$;
         Tainted score for seed $S_t$; Code coverage score for seed $S_c$;
Output: Score of cur test case $S_q$;
Initialize the list $S_q \leftarrow 100$, $S_t \leftarrow 0$, $S_c \leftarrow 0$.
if has_new_tainted_node(q_cur) then
    for $i \leftarrow 0$ to $n$ do
        $S_t = S_t + p_i * 100$
    end
    $S_c = S_c + n * 50$;
else
    $S_q = S_q * 0.5$;
end
if has_new_code_coverage(q_cur) then
    $S_c = S_c * 2$;
else
    $S_q = S_q * 0.5$;
end
$S_q = S_c + S_t$;
$S_o$ = get_score_by_original_method(q_cur);
$S_q = S_q + S_o$;
return $S_q$
```

Experiments: To validate the effectiveness of our tool EMOFuzz, we designed a series of experiments to verify the results of identification exploitable objects. We selected 5 OOB write bugs from several general-purpose programs. Under various overflow lengths, we conduct 3 fuzzing tests, within each fuzzing last for 24 hours. All the experiments were set up on Ubuntu 18.04 LTS, with an Intel(R) Xeon(R) Gold 6254 CPU @ 3.10GHz and 1TB of RAM. The experimental results are shown in Table 2.

Table 2 illustrates that EMOFuzz can effectively identifies exploitable objects for out-of-bounds write bugs, it can find out exploitable objects for all bugs within 24 hours fuzzing campaign. Generally, the more bytes the data are tainted, the more potential exploitable objects EMOFuzz can identify. Specially, as for CVE-2021-3156, which is heap overflow occurs in *sudo* program, within publicly available exploits. EMOFuzz identified a new exploitable object, *sudo_hook_entry* for exploit, which could lead to an AAW for *sudo*. As for gpac-issue-1317, we failed to find any exploitable objects initially until we increased the tainted length to 128 bytes.

Table 2. Experimental results of identifying exploitable objects by fuzzer.

Bug ID	**Program**	**Overflow Len**[a]	**# of Taint Node**[b]	**# of Exp. Obj.**[c]
CVE-2021-3156	sudo	2	5	0
		4	5	1
		8	5	1
		16	20	1
		32	71	1
CVE-2019-20162	gpac	2	298	1
		4	352	3
		8	288	3
		16	337	4
		32	1052	4
issue-1317	gpac	2	226	0
		4	282	0
		8	256	0
		16	376	0
		32	884	0
		128	2547	3
CVE-2022-26967	gpac	2	798	3
		4	677	3
		8	617	3
		16	594	4
		32	1057	4
CVE-2020-6851	openjpeg	2	16	0
		4	34	1
		8	149	1
		16	897	2
		32	1549	2

[a] It means the overflow length of the vulnerability.

[b] It means the number of corrupted nodes of WVFG.

[c] It means the number of identified potential exploitable objects.

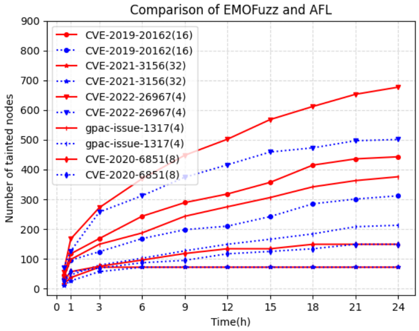

Fig 4 *Comparison of number of tainted nodes by EMOFuzz and AFL. The red line represents fuzzing process of EMOFuzz while blue line represents that of AFL. (16) means the tainted length is 16 bytes.*

To compare EMOFuzz with other state-of-the-art, we chose the widely used coverage-guided fuzzing tool AFL [13]. We selected five sets from Table 1 and compare the ability of in propagating tainted data. Specific results are drawn in Figure 4. As shown in Figure 4, in the 24-hour experiment, EMOFuzz demonstrated a stronger capability to propagate flawed data than AFL, within an average improvement of 31%. Therefore, it is more likely to identify exploitable memory objects.

Conclusion: In conclusion, this letter introduces EMOFuzz, an innovative corrupted data propagation-guided fuzzing method, specifically designed to address OOB write bugs in general-purpose applications. EMOFuzz is effective in identifying vital memory objects, thereby significantly enhancing the assessment of bug exploitability, and aiding in the development of effective exploits.

Received: xx January 2021 *Accepted:* xx March 2021
doi: 10.1049/ell2.10001

References

1. Brumley D, Poosankam P, Song D, et al. Automatic patch-based exploit generation is possible: Techniques and

implications[C]//2008 IEEE Symposium on Security and Privacy (sp 2008). IEEE, 2008: 143-157.

2. Avgerinos T, Cha S K, Rebert A, et al. Automatic exploit generation[J]. Communications of the ACM, 2014, 57(2): 74-84.
3. Bratus S, Locasto M E, Patterson M L. Exploit programming: From buffer overflows to “weird machines” and theory of computation[J]. 2011.
4. Chen W, Zou X, Li G, et al. {KOOBE}: towards facilitating exploit generation of kernel {Out-Of-Bounds} write vulnerabilities[C]//29th USENIX Security Symposium (USENIX Security 20). 2020: 1093-1110.
5. Wu W, Chen Y, Xing X, et al. {KEPLER}: Facilitating control-flow hijacking primitive evaluation for Linux kernel vulnerabilities[C]//28th USENIX Security Symposium (USENIX Security 19). 2019: 1187-1204.
6. Chen Y, Xing X. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel[C]//Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019: 1707-1722.
7. Source Code Analysis with Static Value-Flow. https://svf-tools.github.io/SVF/ 2023.
8. Heelan S, Melham T, Kroening D. Gollum: Modular and greybox exploit generation for heap overflows in interpreters[C]//Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019: 1689-1706.
9. Heelan S, Melham T, Kroening D. Automatic heap layout manipulation for exploitation[C]//27th USENIX Security Symposium (USENIX Security 18). 2018: 763-779.
10. DATAFLOWSANITIZER. https://clang.llvm.org/docs/DataFlowSanitizer 2023
11. Baldoni R, Coppa E, D’elia D C, et al. A survey of symbolic execution techniques[J]. ACM Computing Surveys (CSUR), 2018, 51(3): 1-39.
12. Aschermann C, Schumilo S, Blazytko T, et al. REDQUEEN: Fuzzing with Input-to-State Correspondence[C]//NDSS. 2019, 19: 1-15.
13. American fuzzy lop https://lcamtuf.coredump.cx/afl/ 2023.
14. Masahiro Yamada and Jani Nikula. 2019. kcov:code coverage for fuzzing. https://github.com/torvalds/linux/blob/master/Documentation/dev-tools/kcov.rst
15. Bao T, Wang R, Shoshitaishvili Y, et al. Your exploit is mine: Automatic shellcode transplant for remote exploits[C]//2017 IEEE Symposium on Security and Privacy (SP). IEEE, 2017: 824-839.