Understanding and Detecting Concurrency Attacks

Paper #82

The University of Hong Kong

@cs.hku.hk

Abstract

Just like sequential bugs lead to attacks, concurrency bugs also lead to concurrency attacks.

1. Introduction

Multi-threaded program is hard to be correct. Concurrency bugs are common in modern multi-threaded programs including atomic violation, ..., and especially data race[17, 18, 32, 33]. Extant work well explores interleaving that causes concurrency bugs, and efficiently detects explicit concurrency bugs that direct to severe consequences such as execution order violation, wrong output and program crash [17, 20, 26, 29, 32, 33].

Recent studies[24, 31] show rise of concerns about *concurrency attacks*. By triggering concurrency bugs, hackers may leverage the corrupted memory to conduct attacks including privilege escalations[8, 10], hijacking code execution[9], bypassing security checks[3–5], and breaking database integrity[24]. These vulnerabilities are often hidden in concurrency bugs and implicit regarding program behaviors. For example, a privilege escalation attack may cause no outrageous effect, but possess a permanent security hole hiding in the system. Also, despite the inputs that induce concurrency bugs, concurrency attacks may need other crafted inputs for exploitation of the vulnerabilities.

Unfortunately, although great progress has been made, there does not exist a general model and practical tools for understanding and detecting concurrency attacks. Our study over known concurrency bugs[1, 2] shows that even concurrency bugs have been successfully detected and reported, professionals may still lack knowledge about how severe consequences these bugs may cause. For instance, *apache-25520*[1] has been reported over years and well studied by researchers[18]. We are the first to exploit a new heap overflow attack leveraging on this bug and break the HTML integrity. Knowledge and detection of concurrency attacks is of crucial importance.

We studied 26 attacks and find two major challenges for detecting concurrency attacks. First, concurrency attacks may be implicit and hidden in common concurrency bugs. Existing concurrent bug detectors may not analyze whether a concurrency bug is vulnerable and exploitable. ConMem[33] first propose to consider concurrency bugs that may cause severe consequences (e.g.program crash), but our observation shows that explicit error (e.g.program crash) is not necessary for concurrency attacks. ConSeq[32] care about severe consequences and do intra-procedure analysis to help diagnose concurrency bugs. However, propagation path ... . Worse still, crash bugs may even lead to more severe vulnerabilities. In *CVE-2017-7533* conducted by our team, although the data race primarily causes kernel crash, we crafted the input and successfully conduct a privilege escalation attack without crashing the kernel. Current concurrency bug detecting tools are not designed to analyze this kind of latent vulnerabilities, and hence may direct wrong level of warnings towards the bugs.

Second, extant work ignores indicating extra inputs to conduct concurrency attacks. A concurrency bug may become much more vulnerable when attackers 1.craft inputs that trigger the bug; 2.employee another input running on victim threads to construct their attacks. In CVE-2017-7533, we do not only leverage two crafted inputs running on two threads to trigger a data race and construct kernel heap overflow, but also require another inputs running on *victim thread* to lay the target structure on the same heap. By corrupting the target structure, we finally achieve arbitrary code execution and get a root shell. Automatically indicating the inputs that construct concurrency attacks would be of vital helpful for developers to better understand the latent vulnerabilities.

To address the two challenges, we present a general model(§3.1) for understanding concurrency attacks. The model breaks down concurrency attacks into three stages: bug happening, bug-to-attack propagation, and attack happening. In this model, a concurrency bugs is triggered by two

Leveraging this model, we designed a two-phase framework(§3.2), XXX, for detecting concurrency attacks.

The first phase is *concurrency analyzer* to analyze bug-to-attack propagations. Our study found that most vulnerable races are already included in the race detectors reports, and concurrency attacks sites are often explicit in program code. Therefore, we can perform static analysis on only the data and control flow propagations between the bug reports and the potential attack sites, then we can collect relevant call stacks and branch statements as the potentially vulnerable input hints.

The second phase is *concurrency fuzzer?* to ... .

We implemented XXX on Linux, supporting both user space and kernel space attack detection.

We evaluated OWL on 6 diverse, widely used programs, including Apache, Chrome, Libsafe, Linux kernel, MySQL, and SSDB. OWLs benign schedule hints and runtime verifiers reduced 94.3% of the race reports, and it did not miss the evaluated concurrency attacks. With the greatly reduced reports, OWLs vulnerable input hints helped us identify subtle vulnerable inputs, leading to the detection of 7 known concurrency attacks as well as 3 previous unknown, severe ones in SSDB and Apache. The analysis performance of OWL was reasonable for in-house testing.

This paper makes two major contributions:

1. **A general model for understanding concurrency attacks.** This model explains most concurrency attacks in wild and providing two major direction for detecting concurrency attacks.
2. **A practical concurrency attack detection tool and its implementation, XXX.** XXX can easily employ existing concurrent bug detectors and vulnerability analyzer to improve the accuracy and ??? of detection.

The rest of this paper is structured as follows. §2 introduces the background of concurrency attacks. §3 gives an overview on the concurrency attack model and architecture of XXX. §**??**...

2. Background

2.1 Concurrency Bug

2.2 Concurrency Attack

Extant studies [24, 31] concurrency attacks into .. concurrency bugs and made three major findings on concurrency attacks. First, concurrency attacks make much more severe threats: 35 of the bugs can corrupt critical memory and cause four types of severe security consequences, including privilege escalations [7, 10], malicious code injections [8], and bypassing security authentications [4, 3, 5]. Second, concurrency bugs and attacks can often be easily triggered via subtle program inputs. For instance, attackers can use inputs to control the physical timings of disk IO and program loops and trigger concurrency bugs with a small number of re-executions. Third, compared to traditional TOCTOU attacks, which stem from corrupted file accesses, handling concurrency attacks is much more difficult because they stem from corrupted, miscellaneous memory accesses.

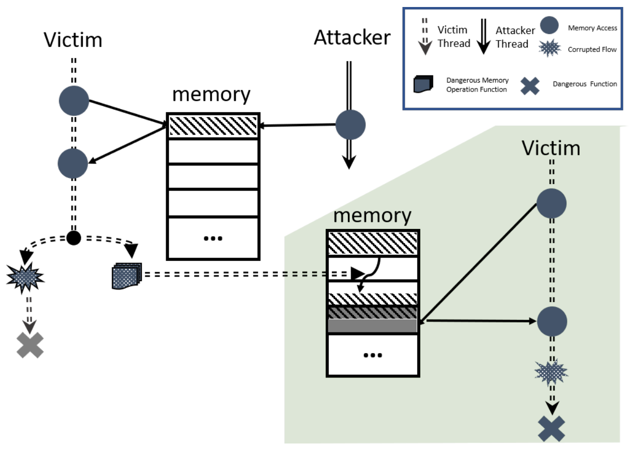

Figure 1: ***Concurrency Attack Model***

2.3 Bug-to-Attack Propagation

3. Overview

We introduce a model that explains most concurrency attacks we studied. Leveraging the model, we design a framework XXX to detect concurrency attacks. This section gives an overview of the model and architecture of XXX.

3.1 Concurrency Attack Model

3.2 XXX's Architecture

4. XXX's Framework

5. Implementation

6. Discussions

7. Evaluation

We evaluated XXX on 6 widely used C/C++ programs, including three server applications (`Apache` [12] web server, `MySQL` [11] database server, and SSDB [23] key-value store server), one library (`Libsafe` [16]), the 4.11.9 Linux kernel, and one web browser (`Chrome`). We used the programs' common performance benchmarks as workloads. Our evaluation was done on XXX.

We focused our evaluation on four key questions:

1. How many false reports from concurrency error detection tools can XXX reduce (§**??**)?
2. How many potential victim threads can XXX indicate(§3)?
3. Can XXX detect known concurrency attacks in the real-world (§**??**)?
4. Can XXX detect previously unknown concurrency attacks in the real-world (§**??**)?

8. Related Work

¡¡¡¡¡¡¡ HEAD ======= **TOCTOU attacks.** Time-Of-Check-to-Time-Of-Use attacks [13, 25, 27, 28] target mainly the file interface, and leverage atomicity violation on time-

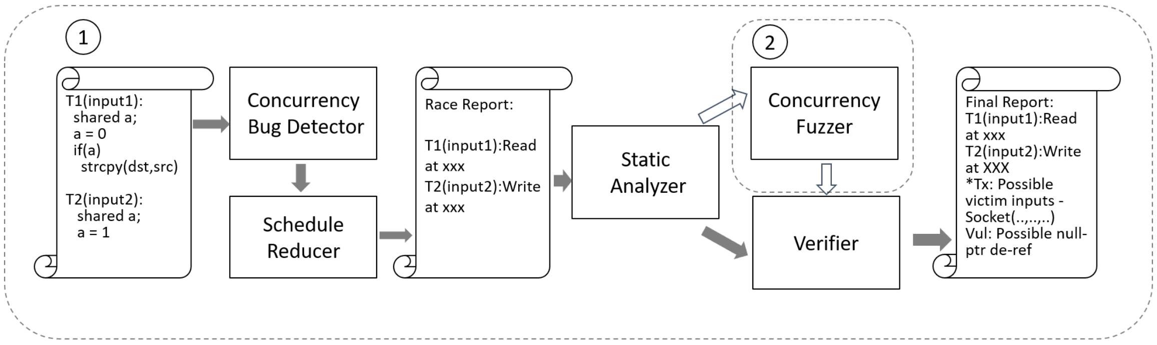

Figure 2: ***Architecture***

Name	LoC	# r.r.	# XXX's r.	# atks	# atks found
`Apache`	290K	715	10	3	3
`Chrome`	3.4M	1715	115	1	1
`Libsafe`	3.4K	3	3	1	1
`Linux`	2.8M	24641	34	2	2
`MySQL`	1.5M	1123	16	2	2
`SSDB`	67K	12	2	1	1
Total	-	-	-	-	-

Table 1: XXX ***race report reduction and concurrency attack detection results.*** Description

Name	Type	# Fed Inputs	# Victim Inputs	# atks
`Apache`	apr_palloc	-	-	-
`Linux`	kmalloc	180	6	1
Total	-	-	-	-

Table 2: XXX*'s Concurrency Fuzzer.* Description

of-check (access()) and time-of-use (open()) of a file to gain illegal file access.

A prior concurrency attack study [30] elaborates that concurrency attacks are much broader and more difficult to track than TOCTOU attacks for two main reasons. First, TOCTOU mainly causes illegal file access, while concurrency attacks can cause a much broader range of security vulnerabilities, ranging from gaining root privileges [8] , injecting malicious code [7], to corrupting critical memory [1]. Second, concurrency attacks stem from miscellaneous memory accesses, and TOCTOU stem from file accesses, thus handling concurrency attacks is much more difficult than TOCTOU.

Sequential security techniques. Defense techniques for sequential programs are well studied, including taint tracking [15, 19–21], anomaly detection [14, 22], address space randomization [70], and static analysis [38, 30, 76, 17, 19]. However, with the presence of multithreading, most existing sequential defense tools can be largely weakened or even completely bypassed [85]. For instance, concurrency bugs in global memory may corrupt metadata tags in metadata tracking techniques. Anomaly detection lacks a concurrency model to reason about concurrency bugs and attacks.

Concurrency reliability tools. Various prior systems work on concurrency bug detection [87, 64, 29, 51, 53, 89, 88, 44, 80], diagnosis [67, 59, 57, 16, 43], and correction [42, 78, 82, 41]. They focused on concurrency bugs themselves, while OWL focuses on the security consequences of concurrency bugs. Therefore, these systems are complementary to OWL. Conseq [88] detects harmful concurrency bugs by analyzing its failure consequence. Its key observation is that concurrency bugs and the bugs failure sites are usually within a short control and data flow propagation distance (e.g., within the same function). Concurrency attacks targeted in OWL usually exploit corrupted memory that resides in different functions, thus Conseq is inadequate for concurrency attacks. Conseqs proactive harmful schedule exploration technique will be useful for OWL to trigger more vulnerable schedules. Static vulnerability detection tools. There are already a variety of static vulnerability detection approaches [49, 84, 31, 15, 71, 90]. These approaches fall into two categories based on whether they target general or specific programs. The first category [49, 84] targets general programs and their approaches have been shown to find severe vulnerabilities in large code. However, these pure static analyses may not be adequate to cope with concurrency attacks. Benjamin et al. [49] leverages pointer analysis to detect data flows from unchecked inputs to sensitive sites. This approach ignores control flow and thus it is not suitable to track concurrency attacks like the Libsafe one in 4.3. Yamaguchi et al. [84] did not incorporate inter-procedural analysis and thus is not suitable to track concurrency attacks either. Moreover, these general approaches are not designed to reason about concurrent behaviors (e.g., [84] can not detect data races). OWL belongs to the first category because it targets general programs. Unlike the prior approaches in this category, OWL incorporates concurrency bug detectors to reason about concurrent behaviors, and OWLs consequence analyzer integrates critical dynamic information (i.e., call stacks) into static analysis to enable comprehensive dataflow, control- flow, and inter-procedural analysis features. The second category [31, 15, 71, 90] lets static analysis focus on specific behaviors (e.g., APIs) in specific programs to achieve scalability and accuracy. These approaches check web application logic [31], Android applications [15], cross checking security APIs [71], and verifying the Linux Security Module [90]. OWLs analysis is complementary to these approaches; OWL can be further integrated with these approaches to track concurrency attacks.

Symbolic execution. Symbolic execution is an advanced program analysis technique that can systematically explore a programs execution paths to find bugs. Researchers have built scalable and effective symbolic execution systems to detect software bugs [34, 68, 33, 35, 19, 86, 20, 23, 21, 63], block malicious inputs [24], preserve privacy in error reports [22], and detect programming rule violations [25]. Specifically, UCKLEE [63] has been shown to effectively detect hundreds of security vulnerabilities in widely used programs. Symbolic execution is orthogonal to OWL; it can augment OWLs input hints by automatically generating concrete vulnerable inputs. ¿¿¿¿¿¿¿ 44909b0199b9a4232c8650d2160eac4991a7dc75

9. Conclusion

References

[1] Apache bug 25520. `https://bz.apache.org/bugzilla/show_bug.cgi?id=25520`.

[2] Apache bug 46215. `https://bz.apache.org/bugzilla/show_bug.cgi?id=46215`.

[3] CVE-2008-0034. `http://www.cvedetails.com/cve/CVE-2008-0034/`.

[4] CVE-2010-0923. `http://www.cvedetails.com/cve/CVE-2010-0923`.

[5] CVE-2010-1754. `http://www.cvedetails.com/cve/CVE-2010-1754/`.

[6] CVE-2017-7533. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7533`.

[7] Freebsd CVE-2009-3527. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3527`.

[8] Linux kernel bug on uselib(). `http://osvdb.org/show/osvdb/12791`.

[9] MSIE javaprxy.dll COM object exploit. `http://www.exploit-db.com/exploits/1079/`.

[10] Mysql bug 14747. `https://bugs.mysql.com/bug.php?id=14747`.

[11] MySQL Database. `http://www.mysql.com/`, 2014.

[12] Apache web server. `http://www.apache.org`, 2012.

[13] M. Bishop, M. Dilger, et al. Checking for race conditions in file accesses. *Computing systems*, 2(2):131–152, 1996.

[14] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the 2nd Symposium on Principles and Practice of Parallel Programming (PPOPP '90)*, pages 1–10, Mar. 1990.

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 1–6, 2010.

[16] Libsafe. `http://directory.fsf.org/wiki/Libsafe`.

[17] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 103–116, 2007.

[18] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 329–339, Mar. 2008.

[19] A. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 129–142, 1997.

[20] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*, pages 89–100, June 2007.

[21] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, 2006.

[22] D. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 285–297, 1989.

[23] `ssdb.io/`.

[24] P. B. Todd Warszawski. Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*, pages 5–20. ACM, 2017.

[25] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva. Portably solving file tocttou races with hardness amplification. In *FAST*, volume 8, pages 1–18, 2008.

[26] Threadsanitizer. `https://code.google.com/p/data-race-test/wiki/ThreadSanitizer`, 2015.

[27] E. Tsyrklevich and B. Yee. *Dynamic detection and prevention of race conditions in file accesses*. PhD thesis, University of California, San Diego, 2003.

[28] J. Wei and C. Pu. Tocttou vulnerabilities in unix-style file systems: An anatomical study. In *FAST*, volume 5, pages 12–12, 2005.

[29] Z. Wu, K. Lu, X. Wang, and X. Zhou. Collaborative technique for concurrency bug detection. *International Journal of Parallel Programming*, 43(2):260–285, 2015.

[30] J. Yang. Concurrency attacks and defenses. Technical report, The Trustees of Columbia University in the City of New York DUNS 049179401 New York United States, 2016.

[31] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. In *the Fourth USENIX Workshop on Hot Topics in Parallelism (HOTPAR '12)*, June 2012.

[32] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: detecting concurrency bugs through

sequential errors. In *Sixteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 251–264, Mar. 2011.

[33] W. Zhang, C. Sun, and S. Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 179–192, Mar. 2010.