2020 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)

Study of Security Flaws in the Linux Kernel by Fuzzing

P.A. Teplyuk
Dept. of Computer Science, Engineering and Security
Altai State Technical University
Barnaul, Russia
teplyukpavel@yandex.ru

A.G. Yakunin
Dept. of Computer Science, Engineering and Security
Altai State Technical University
Barnaul, Russia
yakunin@agtu.secna.ru

E.V. Sharlaev
Dept. of Computer Science, Engineering and Security
Altai State Technical University
Barnaul, Russia
sharlaev@mail.ru

***Abstract* — An exceptional feature of the development of modern operating systems based on the Linux kernel is their leading use in cloud technologies, mobile devices and the Internet of things, which is accompanied by the emergence of more and more security threats at the kernel level. In order to improve the security of existing and future Linux distributions, it is necessary to analyze the existing approaches and tools for automated vulnerability detection and to conduct experimental security testing of some current versions of the kernel. The research is based on fuzzing - a software testing technique, which consists in the automated detection of implementation errors by sending deliberately incorrect data to the input of the fuzzer and analyzing the program's response at its output. Using the Syzkaller software tool, which implements a code coverage approach, vulnerabilities of the Linux kernel level were identified in stable versions used in modern distributions. The direction of this research is relevant and requires further development in order to detect zero-day vulnerabilities in new versions of the kernel, which is an important and necessary link in increasing the security of the Linux operating system family.**

Keywords — kernel, Linux, security threats, vulnerabilities, fuzzing, Syzkaller, use-after-free, stack overflow

I. Introduction

The main trend in the development of operating systems (OS) based on the Linux kernel is widespread use in cloud technologies, Internet of Things devices, and mobile devices. Meanwhile, active development by the open source community from around the world and the release of new versions of the kernel is accompanied by the emergence of vulnerabilities, both known and zero-day.

However, the search for security flaws is complicated by a large volume of source code, which, as technology rapidly develops, becomes more every year [1]. Therefore, it is necessary to perform various code checks to improve reliability and security. One of the solutions is the use of dynamic code analysis, which allows you to expand the scope of the program scan, allowing you to find more vulnerabilities, incl. "Zero day".

Fuzzing is a method of dynamic software testing, which implies automatic or semi-automatic transfer of incorrect, unexpected or random data to the application as input.

Despite the widespread popularity of Linux kernel-based distributions, there has been little systematic research into fuzzing it. Many kernel fuzzers have open source code, but in most cases the lack of at least a brief documentation of the operation algorithm makes them difficult to study.

However, some research is being done. For example, in [2], the automated detection of kernel flaws in commercial versions of OS based on the Linux kernel using fuzzing based on code coverage was considered. The following versions have been tested: 3.10, 4.9, 4.14-rt and 4.19. For each version, the authors found 6, 7, 18 and 10 reproducible vulnerabilities, respectively.

Recently, attempts have been made to apply the so-called "hybrid" fuzzing to Linux kernel testing. This approach combines symbolic performance and traditional fuzzing. In [3], a hybrid fuzzer based on Syzkaller [4] and the S2E symbolic execution platform is proposed.

II. Problem Statement

The goal of the work is to analyze the available approaches and fuzzing tools and to experimentally test the security of some current versions of the Linux kernel. First of all, it is important to detect such critical vulnerabilities as use-after-free (UAF), buffer overflow and others, which can potentially lead to code execution and elevation of privileges in the system [5-7].

The following critical vulnerabilities are distinguished:

978-1-7281-6951-4/20/$31.00 © 2020 IEEE

Authorized licensed use limited to: National Sun Yat Sen Univ.. Downloaded on December 15,2020 at 19:52:40 UTC from IEEE Xplore. Restrictions apply.

A. Stack overflow

Occurs as a result of writing the stack buffer after the end of the allocated memory area, which allows you to overwrite the stored instruction pointer in order to execute arbitrary code. A stack depth error occurs when the stack size exceeds its maximum size, as a result of which it allows manipulating the stack and thread parameters of other processes. This also includes attacks related to manipulation of local variables in the stack [8, 15]. Stack attacks are most dangerous for the main system when the process has a shared memory area for the stack with the OS kernel.

There are the following ways to counter this vulnerability:

- stack canary - a mechanism that detects buffer overflows before executing malicious code (it is possible to use the gcc -fstack-protector and -fstack-protector-strong program at the compilation stage);
- protection of kernel memory pages (option GRKERNSEC_KSTACKOVERFLOW);
- alloca checking (e.g. PAX_MEMORY_STACKLEAK);
- enabling the kernel stack randomization mechanism (KSLR);
- using shadow stacks - the procedure for storing the return address.

B. Integer overflow

Integer overflows occur when a multiplication occurs that exceeds the size provided by the data type. Usually this leads to writing to too small buffers, or to getting array indices outside the bounds. The most common exploitation is through heap overflows, since small buffers tend to be allocated on the heap. In addition, reference counts can overflow, which will lead to memory use after free.

To prevent exploitation of this vulnerability, there are the following approaches:

- using the PAX_REFCOUNT option - if an overflow is detected in the reference counter, the SIGKILL signal will be sent to the process;
- the PAX_SIZE_OVERFLOW option prevents various integer overflows of the size of the function arguments.

C. Heap overflow

Heap overflows typically occur due to integer overflows or bad bounds checks. Exploits overwrite contiguous heap memory or manipulate heap metadata values. Ways to counter this vulnerability:

- checking the correctness of the size of the variable in comparison with the size copy_to_user / copy_from_user (PAX_USERCOPY);
- guard pages;
- checking metadata (functions and glibc's heap protections).

D. Format string injection

When an input string is passed for output to a format string, an attacker can manipulate the result. In this case, the priority is to use the% n specifier, which writes to memory. All other specifier formats lead to information leakage. The ways to counteract this vulnerability are presented below:

- exclude the use of% n;
- detection of non-constant format string at compile time (gcc's -Wformat-security);
- detection of non-constant format string at runtime (glibc's -D_FORITY_SOURCE = 2).

E. Kernel pointer leak

When a kernel memory address (for example, from the stack, heap, etc.) leaks into user space, attackers can find out potentially sensitive information about data composition, kernel layout, stack layout, architecture layout, etc. constellations can be used to perform other types of attacks where these values are necessary for successful exploitation. If the locations are not determined correctly, an attacker can disrupt the entire system, making a kernel pointer leak critical for successful exploitation [9-11]. The ways to counter this vulnerability are presented below:

- removing visibility for kernel symbols (module GRKERNSEC_HIDESYM);
- detecting and blocking the use of% p or similar entries in seq_file or other custom buffers (modules GRKERNSEC_HIDESYM and PAX_USERCOPY).

F. Uninitialized variables

When variables (on the stack or on the heap) are used without explicit initialization, then the behavior is undefined. In fact, "uninitialized object" means that the variable has the previous value. When an attacker can control the previous values, it can lead to exploitation of vulnerabilities or leaks, both using conventional methods and by manipulating critical data that will be used during processing. The ways to counter this vulnerability are presented below:

- clearing the kernel stack before the system call (module PAX_MEMORY_STACKLEAK);
- forced initialization of all uninitialized structures (PAX_MEMORY_STRUCTLEAK module).

G. Use-after-free

The disadvantage arises when the allocated memory is cleared from one piece of code, but at the same time from another piece of code is still used by address. If an attacker can control a new memory allocation that fills the freed area, then it will not be difficult for an attacker to manipulate the contents so that the system executes arbitrary code. The ways to counteract this vulnerability are presented below:

- clearing all freed memory (module PAX_MEMORY_SANITIZE);

978-1-7281-6951-4/20/$31.00 © 2020 IEEE

Authorized licensed use limited to: National Sun Yat Sen Univ.. Downloaded on December 15,2020 at 19:52:40 UTC from IEEE Xplore. Restrictions apply.

- sharing the memory used by the kernel and user space can stop attacks (module PAX_USERCOPY);
- randomization of memory allocation;
- protection against overflow of the reference counter (module PAX_REFCOUNT, HARDENED_ATOMIC).

III. Research Methods

The research method, as noted earlier, is fuzzing.

Fuzzing is a technology for automated testing of software in order to detect potential vulnerabilities, which covers a large number of edge cases by generating incorrect input data. In this case, the input data can be files processed by the application, information transmitted over network protocols, functions of the application interface, etc.

There are 2 fuzzing approaches, depending on the data generation method:

1. Based on mutation data. New data is obtained through minor changes to existing ones.

2. Based on data generation. The data is prepared in advance, based on protocols or in accordance with predefined rules.

The following fuzzing types, one way or another, belong to one of the above classes:

1. Using pre-prepared text data. It is used to test the implementation of protocols. Together with the formal description of the protocol, the developer can prepare a series of textual data that must be correctly processed by the program implementing the protocol.

2. Using random data. It is the least efficient possible approach. A large amount of random data is transmitted to the program under investigation. When it crashes, it is very difficult to determine the cause of the failure.

3. Manual change of protocol data. The researcher knows the protocol, and he tries to achieve anomalous behavior of the investigated software by introducing erroneous data. The lack of automation is the reason for the inefficiency of this method.

4. Complete enumeration of data mutations generated according to the protocol. The approach reduces the volume of tests by leveraging knowledge of the protocol. However, all sorts of mutations are introduced into the data, due to which the data volume is large.

5. Deliberate modification of the data prepared according to the protocol. To conduct this type of testing, it should be further investigated and determined which parts of the data should remain constant and which should change. This is the most intelligent approach, but it is complicated by the increased time spent by the researcher [12].

With the development of core fuzzing, 4 main categories of this software have formed: randomization-based, typed, hook-based, and feedback fuzzers.

1. Based on randomization. The emergence of core fuzzers dates back to 1991, when the tsys fuzzer was released. The idea behind it is simple: it calls a series of randomly selected UNIX System V system calls with randomly generated arguments.

2. Typed fuzzers. In 1997, Koopman and others identified a set of all sorts of test data combinations for system calls that were generated based on the types of their arguments. In 2010, in the Trinity fuzzer, this idea was developed - some randomness was added to generate test cases.

3. Based on hooks (interceptions). Fuzzers in this category try to intercept API function calls during kernel execution.

4. Fuzzers with feedback. Some fuzzers like syzkaller, kernel-fuzzing, TriforceLinuxSyscallFuzzer use code coverage when generating system calls. In particular, they choose one of the randomly generated system calls that maximizes the coverage of the kernel code. The return code from the function [13-15] can be used as a feedback.

IV. Results and Discussion

The fuzzer syzkaller will be used for testing. The kernel versions under study will be 4.13, 4.19 and 5.2. The core configurations in both tests will be identical, and the time spent on fuzzing will be the same, so that in both cases it will have approximately equal coverage of the core code by tests. We will use KVM as a virtualization system. A server with an 8-core Intel Xeon E3-1245 V2 processor with a clock frequency of 3.4 GHz and 16 gigabytes of RAM will be used as a hardware platform for the test bench. System resource utilization is shown in Figures 3-4. The time allotted for fuzzing is limited to 150 hours. The configuration file for syzkaller is presented in Figure 1:

```
{
    "target": "linux/amd64",
    "http": "127.0.0.1:56741",
    "workdir": "/home/user/syzkalls/workdir",
    "kernel_obj": "/linux/",
    "image": "/home/user/wheezy.img",
    "sshkey": "/home/user/wheezy.img.key",
    "syzkaller":"/home/user/syzkalls/src/github.com/google/syzkaller",
    "disable_syscalls": ["keyctl", "add_key", "re-quest_key"],
    "suppressions": ["some known bug"],
    "procs": 1,
    "type": "qemu",
    "vm": { "count": 2,
            "cpu": 1,
            "mem": 1024,
            "kernel":"/home/user/bzImage"
    }
}
```

Fig. 1. Syzkaller configuration.

Qemu command to start fuzzing process is shown in Figure 2.

978-1-7281-6951-4/20/$31.00 © 2020 IEEE

Authorized licensed use limited to: National Sun Yat Sen Univ.. Downloaded on December 15,2020 at 19:52:40 UTC from IEEE Xplore. Restrictions apply.

```
qemu-system-x86_64 -m 1024 -smp 1
-net nic,model=e1000 -net
us-er,host=10.0.2.10,hostfwd=tcp::11747-:22
-display none -serial stdio -no-reboot
-enable-kvm -cpu host,migratable=off
-hda /home/user/wheezy.img -snapshot
-kernel /home/user/bzImage_4.19
-append earlyprintk=serial oops=panic
nmi_watchdog=panic pan-ic_on_warn=1 panic=1
ftrace_dump_on_oops=orig_cpu rodata=n
vsyscall=native net.ifnames=0 biosdevname=0
root=/dev/sda con-sole=ttyS0 kvm-intel.nested=1
kvm-intel.unrestricted_guest=1
kvm-intel.vmm_exclusive=1 kvm-intel.fasteoi=1
kvm-intel.ept=1 kvm-intel.flexpriority=1
kvm-intel.vpid=1 kvm-intel.emulate_invalid_guest_state=1
kvm-intel.eptad=1 kvm-intel.enable_shadow_vmcs=1
kvm-intel.pml=1 kvm-intel.enable_apicv=1
```

Fig. 2. Qemu command.

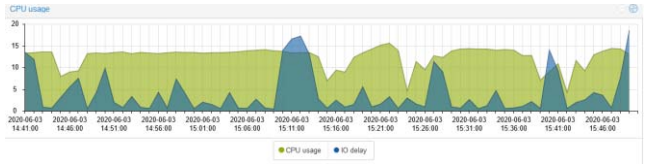

Fig. 3. CPU usage.

Fig. 4. RAM usage.

The fuzzing results are shown in Figure 3.

syzkaller

Stats:

revision	5d7b90f1
config	
uptime	169h32m10s
fuzzing	118h51m0s
corpus	14928
triage queue	0
cover	115748
signal	272028
syscalls	2358
crash types	21 (0/hour)
crashes	495 (2/hour)
exec candidate	0 (0/hour)
exec fuzz	75399 (444/hour)
exec gen	1141 (6/hour)
exec hints	0 (0/hour)
exec minimize	910212 (89/min)
exec seeds	0 (0/hour)
exec smash	592194 (58/min)
exec total	1828728 (179/min)
exec triage	249782 (24/min)
executor restarts	548 (3/hour)
new inputs	19963 (117/hour)
rotated inputs	0 (0/hour)
suppressed	9 (0/hour)
vm restarts	545 (3/hour)

Fig. 5. Fuzzing results.

During the fuzzing process, various types of crashes were discovered. The most common of them are shown in Table 1 (the investigated kernel version is 4.13).

TABLE I. Core flaws found with the most occurrences

№	Problem	Count
1	WARNING in loop_clr_fd	78
2	possible deadlock in rtnl_lock	32
3	general protection fault in sidtab_search_core	14
4	KASAN: use-after-free Read in ucma_close	14

As can be seen from Table 1, within the framework of the conducted testing using the KASAN sanitizer, a kernel security flaw was found - a use-after-free vulnerability.

Use-after-free (UAF) is a vulnerability associated with incorrect use of the dynamic operation of the program: when a memory cell is freed, the pointer to it is not reset, which allows attackers to exploit it for their own purposes.

Potential consequences of UAF exploitation:

- data destruction;
- emergency system termination;
- execution of arbitrary code.

V. Conclusion

The paper analyzes the main security flaws of the kernel of operating systems of the Linux family, such as Stack overflow, Use-after-free, Heap overflow, etc. As part of the study, an automated search for these vulnerabilities was carried out using fuzzing - an automated software testing technology aimed at detection of potential vulnerabilities and covering a large number of edge cases by generating incorrect input data.

The results of the work allow us to continue researching the security of the Linux kernel in order to detect other known vulnerabilities of various types, as well as zero-day vulnerabilities. As a tested version of the kernel, it is planned to use the 5th branch as the most relevant, but also the least studied at the moment.

References

[1] Security of the Unix OS kernel, URL: https://www.opennet.ru/base/sec/kernelsecure.txt.html.

[2] S. Heyuan, W. Runzhe, F. Ying, “Industry Practice of Coverage-Guided Enterprise Linux Kernel Fuzzing” in Proc. of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019. pp. 986–995.

[3] M.V. Mishechkin, “Review of various fuzzing tools as tools for dynamic analysis of software” in Young Scientist, 2017. No 52. pp. 28-31. (https://moluch.ru/archive/186/47575) (in Russian)

[4] Syzkaller - kernel fuzzer, URL: https://github.com/google/syzkaller

[5] V.V. Baklanov, “Protective mechanisms of the Linux operating system: textbook”, Ekaterinburg, UrFU, 2011, 370 p. (in Russian)

[6] G. Fisher, S. Smolski and C. Rodriguez, “The Linux Kernel Primer Upper Saddle River”, N.J., Prentice Hall PTR, 2005, 648 p.

[7] M. Kerrisk, “The Linux Programming Interface: A Linux and UNIX System Programming Handbook”, San Francisco, No Starch Press, 2010, 1556 p

[8] W. Brian, “How Linux Works: What Every Superuser Should Know”, San Francisco, No Starch Press, 2014, 392 p.

[9] R. Rosen, “Linux Kernel networking: Implementation and Theory”, N.Y., Apress, 2014, 636 p

978-1-7281-6951-4/20/$31.00 © 2020 IEEE

Authorized licensed use limited to: National Sun Yat Sen Univ.. Downloaded on December 15,2020 at 19:52:40 UTC from IEEE Xplore. Restrictions apply.

[10] G. Kroah-Hartman, “Linux Kernel in a Nutshell”, USA, O’Reilly Media, 2007, 202 p.

[11] R. Bharadwaj, “Mastering Linux Kernel Development”, Packt, 2017, 346 p.

[12] A.I. Avetisyan, “Modern methods of static and dynamic analysis of programs for automating the processes of improving the quality of software: dis. Cand. those. Sciences: 05.13.11: protected 06.08.12. (Moscow) 2012, 271 p. (in Russian)

[13] N.I. Fursova, “Methods for monitoring objects of the operating system running in a virtual machine”: dis. Cand. those. Sciences: 05.13.11: protected 21.12.17. Velikiy Novgorod, 2017, 120p. (in Russian)

[14] I.M. Slyusarenko, “Methodology for detecting and evaluating anomalies of information systems based on the analysis of system calls”: dis. ... Cand. those. Sciences:05.13.19: protected 01.12.05, St.Petersburg, 2005, 177 p. (in Russian)

[15] R. Love, “Linux kernel development”, Boston, Addison-Wesley Professional, 2010, 468 p.

978-1-7281-6951-4/20/$31.00 © 2020 IEEE

Authorized licensed use limited to: National Sun Yat Sen Univ.. Downloaded on December 15,2020 at 19:52:40 UTC from IEEE Xplore. Restrictions apply.