Approaches to determining the attack surface for fuzzing the Linux kernel

Pavel Teplyuk[1*] and *Aleksey* Yakunin[1]

[1]Polzunov Altai State Technical University, Lenina Ave., 46, Barnaul, 656038, Russia

Abstract. The purpose of the study was to analyze possible methods for determining the attack surface in relation to solving the problem of fuzzing testing the kernel of operating systems of the Linux family and to select the most suitable one. To evaluate and compare various models and practical approaches to attack surface analysis, as well as assess the possibility of combining them, theoretical research methods such as analysis, comparison, and deduction were used. An assessment and comparison of existing models and approaches to analyzing the attack surface of the Linux operating system kernel was carried out. A solution is proposed for the practical determination of the attack surface for effective testing of the kernel using the fuzzing method, which combines the studied approaches. The results of the study can be used to practically construct an attack surface, which will allow us to more accurately determine the goals of fuzz testing of the Linux kernel.

1 Introduction

Currently, along with the rapid development of computer technology, the task of ensuring the security of operating systems, network protocols and software is becoming increasingly urgent. Operating systems based on the Linux kernel, as the basis of system software, are used in information systems, including those that are objects of critical information infrastructure.

One of the methods for identifying undeclared capabilities and vulnerabilities in software in the secure development cycle is dynamic code analysis, including the use of fuzzing testing [1,2]. Fuzzing (fuzz testing) is a method of automated software testing in which incorrect, unexpected or random data is transferred to the program to identify operational defects and vulnerabilities [3].

The preparatory stage of fuzzing testing is to determine the attack surface of the software under study, that is, identifying all entry points into the program that could potentially be used by an attacker for an attack.

The study [4] provides three common kernel attack surface models: GenSec, IsolSec, and StaticSec. Each of them has its own application features. These models will be discussed in more detail below. Existing approaches to analyzing and determining the attack surface in the kernel will also be reviewed:

* Corresponding author: my@teplyukpavel.ru

- based on analysis of CVE databases;
- based on complexity metrics;
- using virtual machines introspection;
- by taint analysis (tracking labeled data).

The work analyzes models and approaches to determining the attack surface of the Linux kernel, defines comparison criteria, and selects the most effective approaches for solving the problem of fuzzing testing. In addition, further directions of research on fuzzing testing of the Linux kernel are outlined.

2 Methods

To construct the attack surface of the Linux kernel, it is necessary to define the following security models:

- a general model that includes a full-fledged core;
- a private model that covers local attacks on the kernel from unprivileged user space.

An important task in the kernel compilation process is to configure it correctly, since this directly affects the attack surface. The task of ensuring the security of the operating system kernel is the task of ensuring the confidentiality, integrity and availability of processes and data. An attacker can compromise data security by hijacking and executing arbitrary code in kernel space. An example of an attack aimed at violating data confidentiality is memory leaks. When considering an availability violation, it could be a denial-of-service (DoS) attack caused by a kernel failure. When specifying models, it is also necessary to assume that the hardware and firmware are reliable. The following is a discussion of publicly available attack surface models for the Linux kernel.

2.1 Attack surface models

2.1.1 GenSec model

The GenSec model covers all possible subsystems of the Linux kernel that may be susceptible to defects and failures (Fig. 1). The model assumes that the attacker has an account on the system, can interact with the hardware, and has access to privileged processes. In this case, failures in the kernel can only occur as a result of access from a privileged process. Kernel crashes can occur in both the main kernel and loadable modules. The GenSec model is quite extensive, so next we will consider more specific attack surface models.

2.1.2 IsolSec model

The IsolSec model (Fig. 2) describes a common pattern in systems where an attacker has gained local access by compromising an unprivileged isolated process on the system and then attempts to escalate its privileges.

Since the attacker has local access, he can use system calls to manipulate kernel space. At the same time, the IsolSec model assumes that the attacker does not have access to the hardware, which reduces the attack surface. However, it can implement arbitrary code execution through loadable kernel modules (LKMs). In the Linux kernel, operations with various pseudo-file systems, for example, securityfs, debugfs, are transferred to specific code paths, mainly kernel modules. This is often used to fine-tune interfaces for privileged user space applications. However, privilege checks are performed at the virtual file system level using POSIX access control lists.

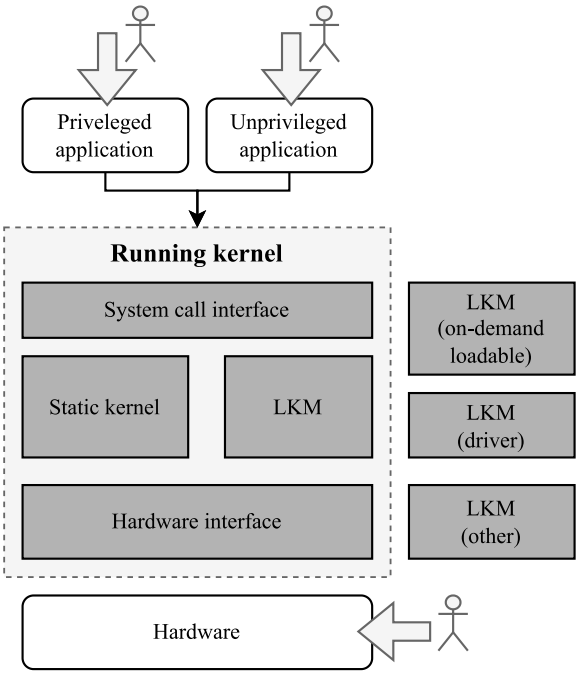

Fig. 1. GenSec model

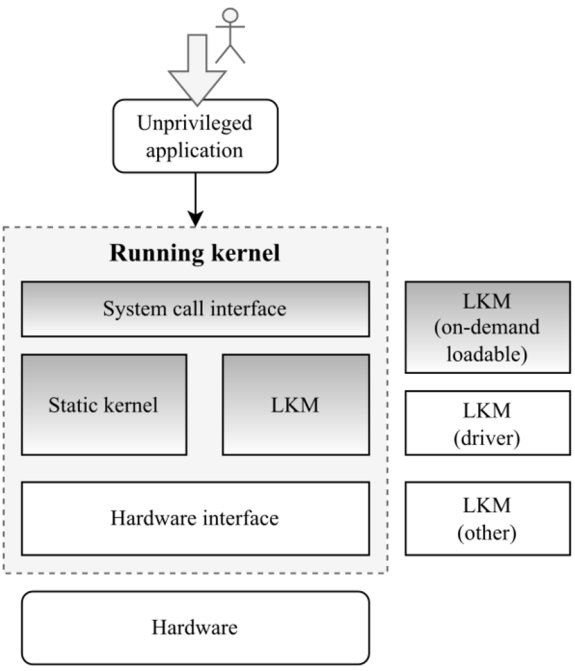

Fig. 2. IsolSec model

Because access to pseudo-file systems must be from an unprivileged isolated process, it is necessary to include the functionality of these file systems in the attack surface.

We will also include kernel modules that either are not loaded at runtime or can be loaded at runtime by a privileged user without any checks on the legitimacy of the module.

2.1.3 StaticSec model

The isolation model of StaticSec (Fig. 3) is similar to IsolSec, but its difference is that in StaticSec an attacker cannot modify the LKM.

Although loading modules by an attacker is possible due to the default settings in many Linux distributions, disabling this behaviour is quite simple (by enabling the modules_disabled control parameter).

The StaticSec model assumes that only LKMs that were initially loaded for work are available to the attacker.

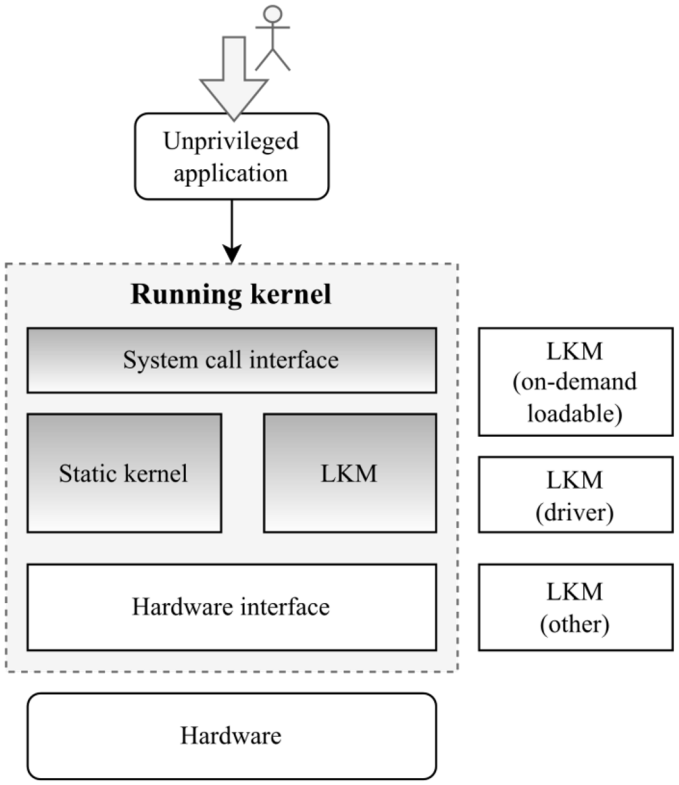

Fig. 3. StaticSec model

2.2 Attack surface approaches

2.2.1 Attack surface analysis based on CVE databases

One of the most well-known problems with fuzzing the Linux kernel is its extremely high complexity. As of November 2023, the kernel repository in the main code branch contains more than 1.2 million commits, and the code base is more than 30 million lines of code.

Work [5] proposes an approach to determining the attack surface for fuzzing using automated analysis of open databases of well-known CVE (Common Vulnerabilities and Exposures) vulnerabilities. The objective of the study is to identify vulnerable kernel

subsystems and, accordingly, exclude the most protected ones from the attack surface in order to increase the efficiency of fuzzing testing.

Determining the attack surface consists of the following steps:
- collection of existing Linux kernel CVEs;
- search for commits that propose a CVE fix and identify files containing the corresponding vulnerabilities;
- classification of kernel subsystems and comparison of found vulnerable files with these subsystems;
- generalization of the obtained data: identification of the most common types of vulnerabilities and the corresponding kernel subsystems.

The same work [5] provides a practical implementation of the chosen approach. To collect the Linux kernel CVE database from open sources on the Internet, the authors developed a web scanner that uses an HTML page parsing mechanism. The sources chosen were the National Vulnerability Database (NVD) and CVEDetails.

For subsequent analysis, the tool collects the following CVE parameters:
- ID (vulnerability identifier);
- type (type of vulnerability);
- CVSS score (vulnerability assessment according to the Common Vulnerability Scoring System standard).

In addition, links to commits in the Linux kernel repository that contain fixes for vulnerable code sections are additionally collected.

The classification of kernel subsystems is based on the interactive Linux Kernel Map. The mapping of vulnerable files to subsystems occurs by searching for keywords that identify the subsystem in the content, or in the names of the files or directories in which they are located.

A total of 2,162 Linux kernel vulnerabilities were discovered between 1999 and 2018, with more than 50% being Denial of Service (DoS) vulnerabilities. The largest number of vulnerabilities are found in such kernel subsystems as synchronization (synchronization mechanisms), logical memory (management of logical memory addresses) and threads (management of execution threads). According to the approach, these subsystems need to be included in the attack surface first.

The results of this study show that the collection and analysis of data on known Linux kernel vulnerabilities makes it possible to identify the most “problematic” subsystems that could potentially have defects and failures during kernel execution. These subsystems must be examined first as part of fuzzing testing.

2.2.2 Attack surface analysis based on complexity metrics

Work [6] proposes an approach to attack surface analysis based on complexity metrics.

The essence of the approach is to measure the cyclomatic complexity of Linux kernel code in different configurations in order to determine how specific kernel components affect the attack surface.

Cyclomatic code complexity is a quantitative metric for calculating the number of logical branches in a code. Whenever control flow is split, the metric counter is incremented by 1. The minimum complexity of each function is 1.

In accordance with the GenSec and IsolSec models described above, the attack surface includes a monolithic kernel and modules that allow you to expand the functionality of the system.

To measure cyclomatic complexity, it is necessary to select modules that are common to different Linux-based OS configurations, as well as hardware-dependent sections of the

kernel source code. The following modules and subsystems were tested in the study: SELinux, AMDGPU, KVM, ext4, xfs, btrfs and namespaces.

To collect complexity metrics, it is suggested to use static code analysis tools, for example, SonarQube. The sequence of operations for collecting complexity metrics is presented in Fig. 4.

Obtaining cyclomatic code complexity metrics allows to evaluate how different subsystems of the Linux kernel affect the overall attack surface. For example, using the AMDGPU module (graphics driver for AMD video cards) increases the complexity indicator by 16.09%.

However, the resulting metrics do not always correlate with the attack surface. According to the study [6], a significant part of the cyclomatic complexity is represented in such sections of the source code as “arch”, “driver” and “fs”. However, a deployed system will typically only use code that is relevant to the current hardware, so most of the source code will not be involved in kernel execution and thus will not be included in the attack surface.

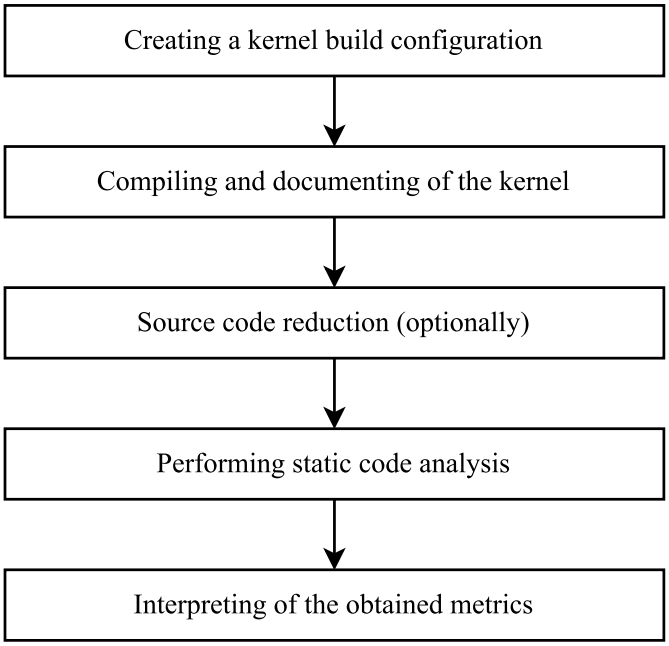

Fig. 4. Sequence of operations for collecting complexity metrics

2.2.3 Determining the attack surface using virtual machine introspection

Using virtual machines to run the Linux kernel during fuzzing has the advantages of ease of management and scalability. Therefore, kernel fuzzers, for example, Syzkaller, provide exactly this launch method.

However, when running a kernel in a virtual machine, there is a semantic gap problem between the data needed to analyze the attack surface and the actual representation of the data in the system. Introspecting virtual machines (VMI) helps bridge the semantic gap. [7].

Virtual machine introspection is a way to examine the current state of a virtual machine at the system level. In other words, introspection is the extraction of data from the OS that is used by the system for its operation while this data is hidden from the user.
As part of the task of determining the attack surface, two approaches to introspection are proposed:

1. An approach based on the integration into the guest system of special analysis modules that change the operation of the operating system. However, embedding a module does not allow you to deterministically reproduce the operation of the OS.

2. An approach based on a comparison of executable instructions and the source code of the operating system kernel. This allows you to obtain information about the location of data structures and their contents.

The Natch tool, developed by ISP RAS to determine the attack surface, implements the following methods of VMI:

- analysis of kernel data structures;
- interception of rarely changing events.

Such events in the operating system can be the execution of system calls. The system call interface rarely changes: usually old functions are no longer used or existing capabilities are expanded. The data that system calls use as arguments or return as a result of execution is not enough to obtain information about processes and modules.

Therefore, an additional solution is to obtain more detailed information about the operating system from the kernel data structures. At the same time, differences in kernel build version or configuration affect the layout of memory structures and addresses. In this regard, it is necessary to apply an introspection profile that contains offsets and addresses of structures and global variables of a particular system.

2.2.4 Determining the attack surface using taint analysis

Dynamic taint analysis is an attack surface approach that monitors data received from external sources during program execution [8].

Such an analysis involves identifying the processes, functions and modules involved in processing data from external sources [9]. This approach requires solving the following problems:

1. Define tracking data. For example, a mark may be placed on sensitive data or on data from unverified sources.

2. Determine the method for advancing the mark when transmitting data during program execution.

3. Determine cases that require an error warning.

DECAF [10] and Panda tools use dynamic binary translation to track tags. DECAF performs instrumentation at the internal TCG representation level. Panda uses a translation of the internal TCG representation into LLVM bitcode, which is tracked to promote tagging.

The Natch tool uses a special technique to reduce the strength of the mark as the number of transformations on the marked data increases. This allows you to more accurately build an attack surface, since the functions that receive weakly transformed data are primarily attacked.

3 Results and discussion

The general attack surface models of the Linux kernel analyzed in this paper describe the varying degrees of privileges in the operating system available to an attacker.

Modern methods of kernel fuzzing testing [11-12] make it possible to generate random input data both from user processes and from hardware interfaces using its emulation, for example, a USB interface [13] and from embedded devices [14]. Thus, fuzzing allows you to imitate the actions of an attacker who has access to hardware. It follows that the GenSec model is best suited for constructing an attack surface in relation to fuzzing testing of the Linux kernel. This model, covering a significant part of the kernel subsystems and hardware interfaces, allows us to maximize the number of fuzzing targets, which potentially allows the detection of previously unknown vulnerabilities in the kernel code.

However, the GenSec model, like the other models discussed, describes the attack surface in general terms and does not take into account many details. Therefore, an important task is to explore various practical approaches to defining the attack surface that can refine it. Due to the fact that the large volume of kernel code, which grows with each release, significantly affects the effectiveness of phasing, it is necessary to reduce the attack surface by identifying the most potentially vulnerable subsystems first.

The approach based on the analysis of the CVE database allows us to identify such subsystems based on the statistics of found vulnerabilities of the Linux kernel in different years. The advantage of the approach is its simple practical implementation: there is no need to deploy virtual machines and configure the kernel. However, the advantage is also determined by the disadvantage of this approach: it is not the current, currently running kernel that is examined, but the general shortcomings of kernels of various versions, many of which are no longer relevant today.

The complexity metrics approach, which is mostly static code analysis, allows you to numerically evaluate how different kernel subsystems affect the attack surface. Unlike the previous approach, this involves examining the current version of the kernel. The approach is not without its drawbacks: complexity metrics do not always correlate with the actual attack surface of the deployed system.

Approaches based on virtual machine introspection and taint analysis make it possible to describe the attack surface quite accurately, since it already involves monitoring and dynamic analysis of the deployed kernel.

The following criteria were defined for the selection of approaches:
- complexity of practical implementation;
- the need to deploy the kernel;
- relevance of kernel versions;
- the need to use proprietary software;
- accuracy of determining the attack surface.

Table 1 presents a comparison of the approaches to attack surface analysis discussed in the article.

From the analysis of the information given in this table, it follows that to determine the attack surface in relation to solving the problem of kernel fuzzing testing, it would be advisable to use a combination of the following approaches:
- based on measuring complexity metrics;
- based on virtual machine introspection;
- based on taint analysis.

The approach based on analysis of CVE databases is not very effective in terms of the relevance of kernel code versions and therefore will not be used in the research.

Table 1. Comparison of approaches to determining the attack surface of the Linux kernel.

	Difficulty of practical implementation	**The need to deploy the kernel**	**Relevance of the kernel**	**Requirement to use proprietary software**	**Attack surface accuracy**
CVE databases analysis	Low	No	Not always	No	Medium
Measuring complexity metrics	Medium	No	Relevant version	Yes (SonarQube)	Medium
Using virtual machine introspection	High	In a virtual machine	Relevant version	Yes (Natch)	High
Dynamic taint analysis	High	In a virtual machine	Relevant version	Yes (Natch)	High

4 Conclusion

Determining the attack surface is an important task in preparation for the kernel fuzzing process. Existing approaches to its construction make it possible to analyze kernel subsystems in different ways. In further research, it is planned to apply in practice a combination of the approaches studied in the article in order to more accurately describe the attack surface in relation to solving the problem of fuzzing testing of the Linux kernel. One of the areas of work is the search for defects and vulnerabilities using the fuzzing method in current kernel branches based on the constructed attack surface.

References

1. Gary McGraw/ Software Security: Building Security In (Addison-Wesley Professional, 2006)
2. The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities (Addison-Wesley Professional, 2006)
3. C. Beaman, M. Redbourne, J.D. Mummery, S. Hakak, *Fuzzing vulnerability discovery techniques: Survey, challenges and future directions*, Computers & Security, **120** (2022). DOI: 10.1016/j.cose.2022.102813.
4. A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schroder-Preikschat, D. Lohmann, and R. Kapitza, *Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring*, NDSS Symposium (2013). URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/03_2_0.pdf.

5. L. Peng, *Attack surface analysis and code coverage improvement for fuzzing*. Master's thesis, Nanyang Technological University, Singapore (2019). DOI: 10.32657/10356/105642.
6. S. Bavendiek. *Attack surface analysis of the Linux kernel based on complexity metric*, Master's Thesis in the study course "Applied Informatics / Software Engineering" (2021), DOI: 10.13140/RG.2.2.29943.70561.
7. P.M. Dovgalyuk, M.A. Klimushenkova, N.I. Fursova, V.M. Stepanov., I.A. Vasiliev, A.A. Ivanov, A.V. Ivanov, M.G. Bakulin, D.I. Egorov, *Natch: using virtual machine introspection and taint analysis for detection attack surface of the software*, Proc. ISP RAS, **34**, pp. 89-110 (2022). DOI: 10.15514/ISPRAS-2022-34(5)-6.
8. E.J. Schwartz E.J., Avgerinos T., Brumley D, *All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)*, Proc. of the IEEE Symposium on Security and Privacy, pp. 317-331 (2010). DOI: 10.1109/SP.2010.26.
9. Z. Jia, C. Yang, X. Zhao, X. Li, J. Ma, *Design and implementation of an efficient container tag dynamic taint analysis*, Computers & Security, **135** (2023). DOI: 10.1016/j.cose.2023.103528.
10. A. Davanian A., Z. Qi, Y. Qu., H. Yin, *DECAF++: Elastic Whole-System Dynamic Taint Analysis*. Proc. of the 22nd Intern. Symp. on RAID, pp. 31-45 (2019). URL: https://www.usenix.org/conference/raid2019/presentation/davanian.
11. C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, W. Liu, *A systematic review of fuzzing techniques*, Computers & Security, **75,** pp. 118-137 (2018). DOI: 10.1016/j.cose.2018.02.002.
12. C. Beaman, M. Redbourne, J. D. Mummery, S. Hakak, *Fuzzing vulnerability discovery techniques: Survey, challenges and future directions*, Computers & Security, **120**, art.102813 (2022). DOI: 10.1016/j.cose.2022.102813
13. N. Nissim, R. Yahalom, Y. Elovici, *USB-based attacks*, Computers & Security, **70**, pp. 675-688 (2017). DOI: 10.1016/j.cose.2017.08.002.
14. X. Li, L. Zhao, Q.Wei, Z. Wu, W. Shi, Y. Wang, *SHFuzz: Service handler-aware fuzzing for detecting multi-type vulnerabilities in embedded devices*, Computers & Security, **138**, art.103618, (2024). DOI: 10.1016/j.cose.2023.103618