**Exploit development for Linux kernel**

**ring 0 and userland vulnerabilities**

**Abstract**

Exploit development for Linux kernel ring 0 and userland vulnerabilities involves identifying and leveraging security weaknesses in the kernel space and userland to gain unauthorized access or control over a system. This process requires a deep understanding of the Linux kernel, its architecture, and the interplay between kernel space and userland. By crafting and executing tailored exploits, security researchers and ethical hackers can assess and improve system security, ensuring robustness against potential attacks. This abstract highlights the significance of exploit development in the context of Linux kernel and userland vulnerabilities, emphasizing the importance of responsible and authorized exploitation for security enhancement. This guide delves into exploiting kernel Ring 0 and userland vulnerabilities, with a focus on the C and C++ APIs. Kernel Ring 0 vulnerabilities are defects in the kernel space that, when exploited, can give an attacker complete control over the machine. In contrast, userland vulnerabilities occur in the user space and, if exploited, can result in privilege escalation or unauthorized access to sensitive data.

With the right authorization the live patch vulnerability can be, used by an attacker can escalate their privileges and possibly change system configurations, run arbitrary code, or obtain private data. An attacker can monitor the victim's actions on Chrome or Chromium-based browsers and obtain sensitive information by activating debugging and forwarding the debugging port using beug vulnerability. An attacker can obtain elevated privileges using kcalloc vulnerability through the use of CPU properties, giving them the ability to run arbitrary code, access restricted resources, and alter system configuration. An attacker may use trace syscall vulnerability to track down system calls and intercept data being sent between processes, which could lead to the disclosure of private data like plaintext login passwords. An attacker can remotely manipulate the system by injecting and executing malicious code by taking advantage of stacktrace vulnerability.

**Keywords:-** Ring 0, exploit development, userland vulnerabilities, assembly, C or C++, APIs, Metasploit

**1.Introduction**

An advanced field in cybersecurity called exploit development for kernel Ring 0 and userland vulnerabilities is dedicated to finding, evaluating, and fixing security flaws. To improve system security and safeguard sensitive data, ethical hackers and security researchers carry out this activity with the required authorization and consent. An in-depth grasp of the different facets of kernel Ring 0 and userland vulnerability exploit development will be provided by this comprehensive introduction.

Ring 0 Kernel Exploitation

The operating system kernel is located in Ring 0, the most privileged level of a computer's design, and this is where kernel exploitation takes place. Because it controls memory, hardware, and system resources, the kernel is a desirable target for hackers. Kernel exploitation requires an understanding of data structures, system call mechanisms, and memory management unique to the kernel. Typical methods for exploiting kernels include the following:

1. Stack Overflows:- Putting arbitrary code into execution by overwriting the return address via a buffer overflow in the kernel stack. Understanding stack canaries, non-executable stacks, and other protections is frequently required for this strategy.  
2**.** Write-What-Where (W^X):- Taking advantage of a flaw that lets an attacker alter the control flow and write data to any location in memory. Bypassing memory protection systems like Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) is a common part of this technique.

3. Null Pointer Dereference:-Using a vulnerability in null pointer dereference to run arbitrary code or deliver a denial of service attack. Utilizing race situations or memory leaks and comprehending the underlying architecture are frequently necessary for this strategy.  
4.Utilizing uninitialized stack:- variables is the fourth strategy. to run arbitrary programs and modify the control flow. This method frequently entails taking advantage of uninitialized memory and comprehending stack structures.  
5. Integer Overflow: - Taking advantage of arithmetic flaws in integers to corrupt memory or produce unexpected results. It's common for this strategy to demand knowledge of signed and unsigned integer operations as well as integer arithmetic.  
6. Use-After-Free (UAF):- Taking advantage of memory management flaws in which memory is allotted, used, deallocated, and then used once more, perhaps resulting in the denial of service or the execution of arbitrary code.

Userland Explotation (Ring 3)

Since user-mode programs and services run at a lower level of privilege (Ring 3) than the kernel, userland exploitation focuses on vulnerabilities in these programs and services. These flaws frequently result from coding mistakes, unsafe APIs, or inadequate input validations, giving attackers the chance to increase access, interfere with services, or run any random code. Typical methods for userland exploitation include the following:

1. Buffer Overflows: Overwriting the return address on the stack and executing arbitrary code by taking advantage of a buffer overflow vulnerability in a userland program.

2**.** Format String Vulnerabilities: By taking advantage of a format string vulnerability, one can write any data to the heap or stack, which frequently results in the execution of any code.

3**.** Integer Errors: Memory corruption or unexpected behavior caused by taking advantage of integer arithmetic flaws in userland applications.

4. API Misuse:- Making use of unsecure APIs to run arbitrary code or trigger unexpected behavior. It is necessary to become proficient in a variety of tools and methodologies in order to create exploits for userland vulnerabilities and kernel Ring 0. Learn how to use userland debuggers, disassemblers, and kernel-specific debugging tools like GDB, LLDB, and WinDbg, Ghidra and IDA Pro. Furthermore, knowing the fundamentals of shellcoding, fuzzing, and reversing can greatly improve your ability to construct exploits. In the hands of certified security experts, exploit development for kernel Ring 0 and userland vulnerabilities is a potent tool. You can help make systems more secure, safeguard sensitive data, and assist enterprises in staying ahead of potential dangers by becoming proficient in these ideas and practices.

Fuzzing is a very powerful exploit development approach that sends unexpected or incorrect inputs to a target program in order to find software flaws. Ethical hackers and security researchers use fuzzing, with the right authorization and permission, to find possible security flaws in software, especially memory corruption bugs. The best approaches, tools, and fuzzing techniques for exploit creation will be covered in this article.

Fuzzing Strategies

1. Mutation-based Fuzzing: This method entails changing individual bytes, adding or deleting data, or jumbling the input structure in order to modify already-existing input data. Among the mutation-based fuzzing tools are Peach Fuzzer, Honggfuzz, and AFL.

2. Generation-based Fuzzing**:** Using a preset model or language, this method generates new input data. Because generation-based fuzzing offers greater control over input generation, it's a good option for testing certain protocols. or structures. Generation-based fuzzing is used by programs like Syzkaller, zzuf, and American Fuzzy Lop (AFL), which use grammar-based fuzzing.  
3. Cadence-Aiding: This method combines intelligent input selection, feedback-driven analysis, and generation- and mutation-based fuzzing. Intelligent fuzzing tools, including BooFuzz, Sulley, and Radamsa, employ a variety of techniques to increase test coverage and more effectively find vulnerabilities.

Fuzzing Instruments

1. American Fuzzy Lop (AFL):- is a popular fuzzing program that combines coverage-guided and mutation-based fuzzing. To fuzz a wide range of applications, AFL supports many instrumentation techniques including as QEMU, LLVM, and native instrumentation.  
2. honggfuzz:- A coverage-guided fuzzing tool compatible with QEMU, LLVM, and native instrumentation, among other instrumentation techniques. Honggfuzz supports a variety of platforms and has sophisticated feedback methods including edge coverage.  
3. Fuzzer for Peaches: - A thorough fuzzing framework that facilitates both generation- and mutation-based fuzzing. Peach Fuzzer is extremely configurable and appropriate for testing complex protocols and formats because it lets users create input forms using XML or Python.

4**.** Radamsa:- A clever fuzzing tool that uses data mutation and statistical techniques to produce a variety of engaging input data. Radamsa is easy to combine with other fuzzing tools since it offers a straightforward command-line interface and supports a variety of output formats.

5**.** Sulley: **-** A very adaptable and configurable fuzzing framework with a modular design that supports both generation- and mutation-based fuzzing. Advanced capabilities like session management, feedback-driven fuzzing, and data modeling are included in Sulley.

Optimal Techniques for Exploit Development Fuzzing

1**.** Know the Target:- Prior to beginning the fuzzing process, have a thorough understanding of of the intended functionality, supported input types, and target application. You can use this knowledge to develop test cases and fuzzing procedures that are more successful.

2**.** Select the Appropriate Fuzzing Method:- Choose a fuzzing method based on input formats and the intended application. For example, generation-based fuzzing works better with text-based inputs, but mutation-based fuzzing is appropriate for systems handling binary data.

3**.** Build a High-Quality Seed Corpus: - The success of fuzzing depends on the seed corpus, or first input data. Take the effort to gather and select high-quality seed data that addresses a range of input possibilities and edge cases.  
4. Instrument the Target:- To gather coverage data that can direct the fuzzing process and uncover more vulnerabilities, instrument the target application. Numerous fuzzy Tools that enable different instrumentation approaches include honggfuzz and AFL.  
5. Monitor and Analyze Results:- To find potential vulnerabilities, fuzzing should be regularly monitored. Results should be analyzed. Utilize programs such as Debugging Symbols, AddressSanitizer, and Valgrind to assist identify problems and confirm any vulnerabilities.  
6. Automate and Integrate:- To guarantee that new vulnerabilities are found and fixed right away, automate the fuzzing process and incorporate it into your continuous integration (CI) or continuous delivery (CD) pipeline. In summary, fuzzing is an effective method for creating exploits that helps security researchers find software flaws and build safer systems. Knowing the best approaches, instruments, and fuzzing techniques.Establishing a methodical and ongoing procedure to detect software security flaws is necessary for automating vulnerability finding for exploit creation. You can use automated tools and procedures to continuously monitor and detect potential vulnerabilities if you have the necessary permissions and permission. This comprises:  
1. Integrating Fuzzing Tools:- To automatically test new code changes and find potential memory corruption problems, incorporate fuzzing tools like AFL, honggfuzz, or Peach Fuzzer into your continuous integration/continuous development/continuum pipeline.  
2. Automating Vulnerability Scanners:- Regularly check your infrastructure and applications for known vulnerabilities by using automated vulnerability scanners like Nessus, OpenVAS, or Nexpose.

3. Using Continuous Security Monitoring Tools: Use security-related data from your systems and apps to automatically gather and analyze security-related monitoring data. These instruments can assist in spotting signs of compromise, odd conduct or unapproved entry.

4. Automating Code Analysis:- usage tools such as SonarQube, Fortify, or Black Duck, which are static and dynamic code analysis tools, to automatically scan your codebase for security vulnerabilities including improper usage of APIs, input validations, or insecure coding practices.

5. Automated Penetration Testing:- To conduct routine security assessments and find possible vulnerabilities in your applications, use automated penetration testing tools like OWASP ZAP, Burp Suite, or Metasploit Pro.  
Always remember to have the right consent and permissions before automating vulnerability detection in order to construct exploits. Automation increases vulnerability detection's effectiveness and accuracy, but it should only be applied sensibly and morally.

Table 1 – Vulnerabilities dependent on certain attack vectors**(**[**https://www.cvedetails.com/vulnerabilities-by-types.php**](https://www.cvedetails.com/vulnerabilities-by-types.php)**)**

| Year | Overflow | Memory Corruption | Sql Injection | XSS | Directory Traversal | File Inclusion | CSRF | XXE | SSRF | Open Redirect | Input Validation |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [2014](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2014/Linux-Linux-Kernel.html) | [20](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2014/opov-1/Linux-Linux-Kernel.html) | [31](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2014/opmemc-1/Linux-Linux-Kernel.html) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | [22](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2014/opinpv-1/Linux-Linux-Kernel.html) |
| [2015](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2015/Linux-Linux-Kernel.html) | [13](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2015/opov-1/Linux-Linux-Kernel.html) | [17](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2015/opmemc-1/Linux-Linux-Kernel.html) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | [5](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2015/opinpv-1/Linux-Linux-Kernel.html) |
| [2016](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2016/Linux-Linux-Kernel.html) | [36](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2016/opov-1/Linux-Linux-Kernel.html) | [76](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2016/opmemc-1/Linux-Linux-Kernel.html) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | [17](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2016/opinpv-1/Linux-Linux-Kernel.html) |
| [2017](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2017/Linux-Linux-Kernel.html) | [64](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2017/opov-1/Linux-Linux-Kernel.html) | [86](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2017/opmemc-1/Linux-Linux-Kernel.html) | 0 | 0 | [1](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2017/opdirt-1/Linux-Linux-Kernel.html) | 0 | 0 | 0 | 0 | 0 | [20](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2017/opinpv-1/Linux-Linux-Kernel.html) |
| [2018](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2018/Linux-Linux-Kernel.html) | [32](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2018/opov-1/Linux-Linux-Kernel.html) | [70](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2018/opmemc-1/Linux-Linux-Kernel.html) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | [11](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2018/opinpv-1/Linux-Linux-Kernel.html) |
| [2019](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2019/Linux-Linux-Kernel.html) | [30](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2019/opov-1/Linux-Linux-Kernel.html) | [124](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2019/opmemc-1/Linux-Linux-Kernel.html) | 0 | 0 | [1](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2019/opdirt-1/Linux-Linux-Kernel.html) | 0 | 0 | 0 | 0 | 0 | [4](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2019/opinpv-1/Linux-Linux-Kernel.html) |
| [2020](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2020/Linux-Linux-Kernel.html) | [10](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2020/opov-1/Linux-Linux-Kernel.html) | [41](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2020/opmemc-1/Linux-Linux-Kernel.html) | 0 | 0 | [1](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2020/opdirt-1/Linux-Linux-Kernel.html) | 0 | 0 | 0 | 0 | 0 | [2](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2020/opinpv-1/Linux-Linux-Kernel.html) |
| [2021](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2021/Linux-Linux-Kernel.html) | [20](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2021/opov-1/Linux-Linux-Kernel.html) | [54](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2021/opmemc-1/Linux-Linux-Kernel.html) | 0 | 0 | [2](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2021/opdirt-1/Linux-Linux-Kernel.html) | 0 | 0 | 0 | 0 | 0 | [7](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2021/opinpv-1/Linux-Linux-Kernel.html) |
| [2022](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2022/Linux-Linux-Kernel.html) | [41](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2022/opov-1/Linux-Linux-Kernel.html) | [149](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2022/opmemc-1/Linux-Linux-Kernel.html) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | [2](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2022/opinpv-1/Linux-Linux-Kernel.html) |
| [2023](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2023/Linux-Linux-Kernel.html) | [18](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2023/opov-1/Linux-Linux-Kernel.html) | [172](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2023/opmemc-1/Linux-Linux-Kernel.html) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | [2](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2023/opinpv-1/Linux-Linux-Kernel.html) |

### Table 2 –Vulnerabilities based on its Impacts (<https://www.cvedetails.com/vulnerabilities-by-types.php>)

| Year | Code Execution | Bypass | Privilege Escalation | Denial of Service | Information Leak |
| --- | --- | --- | --- | --- | --- |
| 2014 | [7](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2014/opec-1/Linux-Linux-Kernel.html) | 0 | 0 | [88](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2014/opdos-1/Linux-Linux-Kernel.html) | [11](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2014/opginf-1/Linux-Linux-Kernel.html) |
| 2015 | [4](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2015/opec-1/Linux-Linux-Kernel.html) | 0 | 0 | [53](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2015/opdos-1/Linux-Linux-Kernel.html) | [5](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2015/opginf-1/Linux-Linux-Kernel.html) |
| 2016 | [4](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2016/opec-1/Linux-Linux-Kernel.html) | [10](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2016/opbyp-1/Linux-Linux-Kernel.html) | [10](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2016/opgpriv-1/Linux-Linux-Kernel.html) | [153](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2016/opdos-1/Linux-Linux-Kernel.html) | [25](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2016/opginf-1/Linux-Linux-Kernel.html) |
| 2017 | [169](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2017/opec-1/Linux-Linux-Kernel.html) | [28](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2017/opbyp-1/Linux-Linux-Kernel.html) | [163](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2017/opgpriv-1/Linux-Linux-Kernel.html) | [148](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2017/opdos-1/Linux-Linux-Kernel.html) | [82](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2017/opginf-1/Linux-Linux-Kernel.html) |
| 2018 | [1](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2018/opec-1/Linux-Linux-Kernel.html) | 0 | [7](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2018/opgpriv-1/Linux-Linux-Kernel.html) | [89](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2018/opdos-1/Linux-Linux-Kernel.html) | [16](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2018/opginf-1/Linux-Linux-Kernel.html) |
| 2019 | [7](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2019/opec-1/Linux-Linux-Kernel.html) | [1](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2019/opbyp-1/Linux-Linux-Kernel.html) | [8](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2019/opgpriv-1/Linux-Linux-Kernel.html) | [113](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2019/opdos-1/Linux-Linux-Kernel.html) | [7](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2019/opginf-1/Linux-Linux-Kernel.html) |
| 2020 | [3](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2020/opec-1/Linux-Linux-Kernel.html) | 0 | [4](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2020/opgpriv-1/Linux-Linux-Kernel.html) | [26](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2020/opdos-1/Linux-Linux-Kernel.html) | [3](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2020/opginf-1/Linux-Linux-Kernel.html) |
| 2021 | [5](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2021/opec-1/Linux-Linux-Kernel.html) | [2](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2021/opbyp-1/Linux-Linux-Kernel.html) | [8](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2021/opgpriv-1/Linux-Linux-Kernel.html) | [23](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2021/opdos-1/Linux-Linux-Kernel.html) | [5](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2021/opginf-1/Linux-Linux-Kernel.html) |
| 2022 | [8](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2022/opec-1/Linux-Linux-Kernel.html) | [10](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2022/opbyp-1/Linux-Linux-Kernel.html) | [15](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2022/opgpriv-1/Linux-Linux-Kernel.html) | [51](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2022/opdos-1/Linux-Linux-Kernel.html) | [19](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2022/opginf-1/Linux-Linux-Kernel.html) |
| 2023 | [13](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2023/opec-1/Linux-Linux-Kernel.html) | [3](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2023/opbyp-1/Linux-Linux-Kernel.html) | [41](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2023/opgpriv-1/Linux-Linux-Kernel.html) | [48](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2023/opdos-1/Linux-Linux-Kernel.html) | [23](https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2023/opginf-1/Linux-Linux-Kernel.html) |

The central part of the Linux operating system is the kernel, which is in charge of controlling memory, hardware, and system resources. It facilitates effective resource management and communication by acting as an interface between the user-mode apps and the underlying hardware. The Linux kernel, which Linus Torvalds created in 1991, has developed into a reliable, stable, and adaptable piece of software that is used by many people, groups, and companies all over the world. The Linux kernel offers a number of essential functions, such as:

1. Process Management**:** System processes are scheduled for execution and allocated system resources based on demand by the Linux kernel.   
2. Memory Management: Processes need memory, and the Linux kernel manages memory allocation, deallocation, and protection to keep illegal access to protected memory regions at bay.

3**.** Device Management: The Linux kernel controls hardware devices, giving user-mode programs a uniform interface through which to communicate with different hardware elements.

4. File System Management: Effective data storage and retrieval across a range of storage media is made possible by the Linux kernel's management of file systems.

5. Networking: The networking features of the Linux kernel allow for network-based communication between Linux computers and other devices.

6.Security: To secure system resources and data, the Linux kernel employs a number of security techniques, including memory protection, process isolation, and access control. Because the Linux kernel is open-source, programmers are free to contribute, alter, and share the code. As a result, there is now a thriving community of users and developers that contribute to the continuous enhancement, upkeep, and growth of the Linux kernel. Because of its steadiness, adaptability, and stability, the Linux kernel has grown to be the cornerstone of many Linux distributions and a well-liked option for embedded systems, servers, and even mobile devices.

**2.Literature Survey**

In their exploration of the threat posed by offensive AI to companies, the author in this paper **[1]**highlight the necessity of proactive security systems against cyberattacks driven by AI. In their investigation **[2]** of smart contract vulnerabilities the authors draw attention to the differences between exploited and vulnerable contracts. The author in this paper **[3]** offer a thorough analysis of cybersecurity weaknesses, threats, attacks, and solutions, whereas the author in this paper **[4]** suggest an adaptive defense mechanism to stop advanced persistent threats (APTs). While the auhor in this research **[5]** examine the hazards, vulnerabilities, and countermeasures associated with smart homes, the author of the article **[6]**examine SCADA vulnerabilities and assaults. The author of this research **[7]** suggest applying deep learning natural language processing for advanced model extraction, whereas the author of this article **[8]** present DeepSteal, a technique for advanced model extraction employing effective weight theft in memory. identification of security flaws. Using an autoencoder and softmax regression technique,the author of this paper **[9]** offer an offensive security solution for proactive threat hunting via adversary emulation, while the authormof this article **[10]**  presents an APT attack detection method in cloud computing. The author in this paper **[11]** present V2w-BERT, a framework for hierarchical multiclass categorization of software vulnerabilities, whereas the author of this paper **[12]** address vulnerabilities and countermeasures in electrical substations. The author in this research **[13]** suggest static analysis for identifying IoT vulnerabilities, while the author of this paper **[14]** examine data-driven software vulnerability evaluation and prioritization. The author offer an extensive survey **[15]** on automatic vulnerability identification in embedded devices, and the author present a way **[16]** for preventing MQTT vulnerabilities using an IoT-enabled intrusion detection system.the author in this research **[17]**  explore using a hypoxia-amplifying DNA repair-inhibiting (HYDRI) nanomedicine to exploit the acquired vulnerability of cisplatin-resistant malignancies, while the author in this paper **[18]**present Vulnerabilities Manager, a platform for connecting vulnerability data sources. the author offer a comprehensive empirical study **[19]** on the hidden life of software vulnerabilities, whereas the author in this paper **[20]** examines security procedures and vulnerabilities in unmanned aerial vehicles (UAVs). These papers provide practitioners, researchers, and organizations important insights into the state of cybersecurity breakthroughs, issues, and solutions. The breadth of study areas addressed in these publications emphasizes the value of a multidisciplinary approach to cybersecurity. The authors **[21]** suggest a new approach called Slake, which enables rapid slab manipulation and exploitation of these vulnerabilities. Slake makes the exploitation process easier by providing a high-level interface, allowing security researchers and developers to find and patch vulnerabilities more effectively. The authors **[22]** demonstrate that this approach can considerably improve the efficiency and effectiveness of fuzzing, allowing for the discovery of more vulnerabilities in less time. The report also analyzes the problems and potential responses to this exploitation strategy.

**3. Exploit Development (Kernel)**

Exploit creation for kernel mode code is a highly specialized and sophisticated subject that necessitates a thorough understanding of operating system architecture, low-level programming, assembly language, and reverse engineering. The kernel is an operating system's fundamental component that manages system resources, facilitates hardware-software communication, and enforces security regulations. When exploiting kernel vulnerabilities, you want to run arbitrary code in kernel mode, giving you almost complete control over the system. This degree of access enables you to circumvent security measures, escalate privileges, or trigger a denial-of-service scenario. Before you start building kernel exploits, make sure you have a good foundation in the following areas:

1. Low-level programming languages: Learn at least one low-level programming language, such C or C++. Understanding how these languages interact with hardware and memory is critical when designing kernel attacks.

2**.** Assembly Language and Reverse Engineering: Assembly language is the processor's native language, and understanding it will provide significant insights into how software works at the most fundamental level. Reverse engineering skills are also necessary because they allow you to dismantle and understand the behavior of software without access to its source.  
3**.** Operating systems and kernel architecture: Gain a thorough understanding of the target operating system's architecture and internal workings. This knowledge will help you identify potential flaws and create successful exploits.  
4. System call mechanism: Discover how system calls are implemented and how they promote communication between user mode apps The kernel. Understanding this method is critical when designing kernel exploits.

5. Memory management: Familiarize yourself with the kernel's memory management strategies, such as virtual memory, page allocation, and segmentation. This knowledge will help you understand how memory is allocated and maintained in the kernel, allowing you to create more effective exploits.

6. Driver development: Learn how to write and interact with device drivers, which typically run in kernel mode. Understanding driver development will allow you to detect potential vulnerabilities and create attacks for them.

Once you have a strong foundation in these areas, you can begin investigating the many types of kernel vulnerabilities and their accompanying exploitation techniques, such as:

1.Stack overflow**:** Like user-mode exploitation, stack overflows in the kernel can be Used to overwrite return addresses and run arbitrary code.

2. Write what and where: This technique includes writing a certain value to an arbitrary region in memory, which allows you to change data and affect the system's behavior.

3.Null pointer dereference: Exploiting null pointer dereferences might result in arbitrary code execution or a denial of service.  
4.Uninitialized stack variable: Exploiting uninitialized stack variables might result in information leakage or, in rare situations, arbitrary code execution.

5.Integer overflow: Integer overflows might cause unexpected behavior, such as writing data to the wrong memory address or missing security checks.

6. Use-after-free (UAF):Exploiting UAF vulnerabilities may result in arbitrary code execution or denial-of-service problems.  
7.Pool overflow: Pool overflows, like stack overflows, can overwrite nearby memory structures and execute arbitrary code.  
8.GDI bitmap Abuse: This approach exploits Graphics Device Interface (GDI) vulnerabilities to execute arbitrary code.  
As you learn more about kernel exploitation, you'll need to keep up with the most recent research and methodologies. Learn about the Exploit Database, the Full Disclosure email list, and other reliable sources for vulnerability research. To practice kernel exploitation, try using vulnerable virtual computers or participating in bug bounty programs that allow kernel-level testing. Remember to always seek correct authorization before attempting to exploit any flaws.

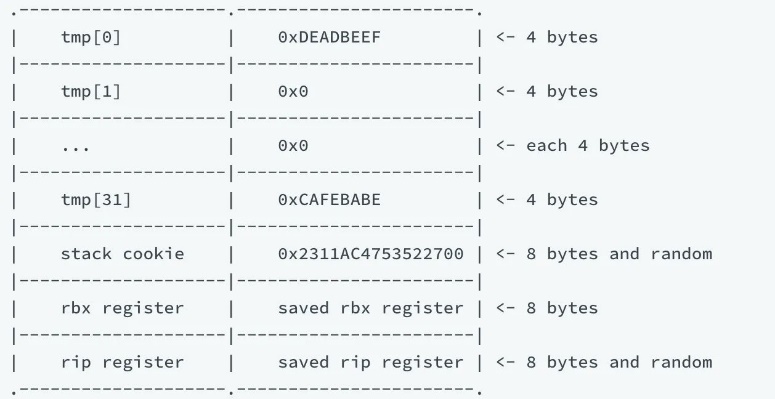


Fig 1 – Linux Kernel Exploitation Map **(**[**https://www.aquasec.com/blog/**](https://www.aquasec.com/blog/)**)**

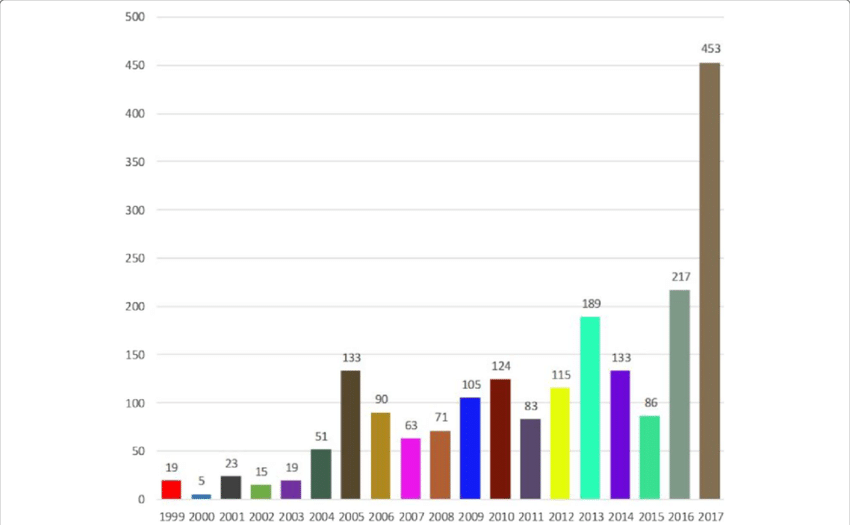


Fig 2 – Growth of the kernel vulnerabilities **(**[**https://www.researchgate.net/figure/ulnerabilities-numbers-of-Linux-kernel-over-years\_fig1\_325577004**](https://www.researchgate.net/figure/ulnerabilities-numbers-of-Linux-kernel-over-years_fig1_325577004)**)**

**4. Proposed Methodology**

Fig -3 Flow cycle for kernel exploit development (Linux)

Familiarize yourself with the target OS,Study kernel components and their interactions,Learn memory management techniques and system call mechanisms**..** Identify a kernel or driver with known vulnerabilities**,**Set up a test environment (virtual machine or lab system)**,**Ensure you have proper authorization to test and exploit the vulnerability. Analyze the vulnerable kernel code or driver**,**Identify the root cause of the vulnerability**,**Understand the conditions required to trigger the vulnerability. Choose a suitable exploitation technique based on the vulnerability, Write code to trigger the vulnerability and achieve the desired effect (e.g., arbitrary code execution, denial-of-service), Test the PoC exploit in a controlled environment. Optimize the exploit to bypass security mitigations (e.g., address space layout randomization, stack canaries, etc.)¸Ensure the exploit works consistently across different configurations and hardware, Document the exploit development process and any challenges encountered. Report the vulnerability to the affected vendor or maintainers**,** Collaborate with them to ensure a patch is developed and released**,** Coordinate the disclosure timeline with the vendor and any relevant bug bounty programs. Test the exploit against the patched kernel, Analyze the patch to understand how it addresses the vulnerability, Modify the exploit to bypass the patch, if necessary, Document the changes made to the exploit and the reasons for them. Use the exploit to test and improve the security of your own systems, Share the exploit with trusted peers or organizations for defensive purposes, Contribute to the development of new security tools or techniques based on the exploit.Please note that this methodology is a general guideline and may need to be adapted based on the specific vulnerability and target system. Always ensure you have proper authorization before attempting to exploit any vulnerabilities.

**5. Results and Discussions**

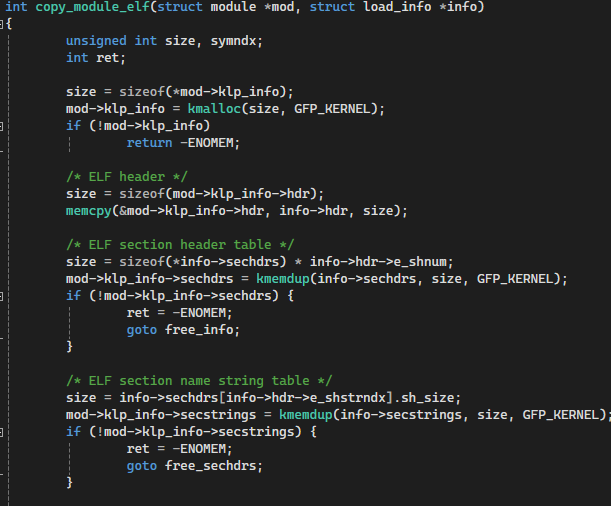
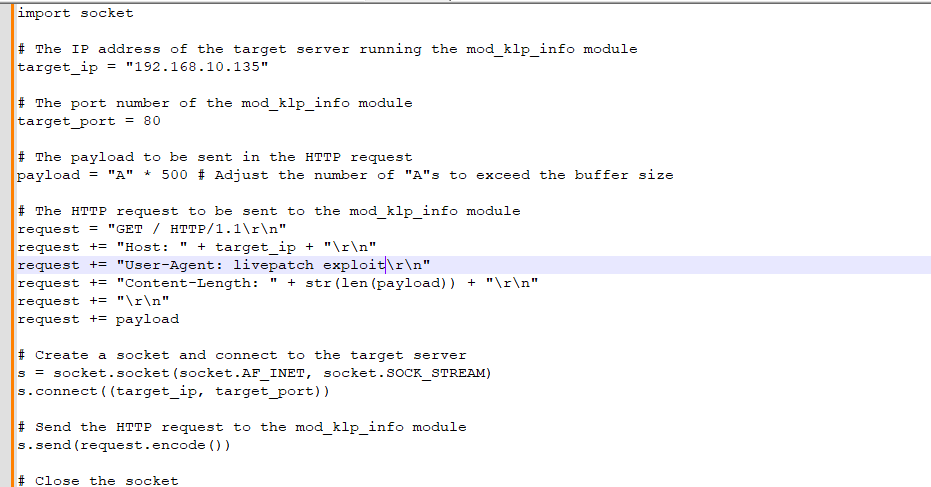
****

Fig 4 – Livepatch vulnerability (mod\_klp\_info Ring 0 )

mod\_klp\_info is an Apache HTTP Server module that displays information on available kernel live patches. It is intended to let system administrators manage and deploy live patches to their systems without having to restart the Apache service. However, it was discovered to possess a vulnerability that attackers might exploit. The flaw is related to how mod\_klp\_info processes specific HTTP requests. Specifically, if an attacker submits a specially crafted HTTP request to the mod\_klp\_info module, they can create a buffer overflow, allowing arbitrary code execution. This means that an attacker might potentially gain control of the compromised system. To exploit this issue, the attacker must be able to send HTTP requests to the mod\_klp\_info module. This could, If the module is exposed to the internet, it can be accessed directly or indirectly via a compromised system connected to the same network. To mitigate this vulnerability, update to the most recent version of the mod\_klp\_info module, which includes a fix for the problem. Furthermore, it is advisable to restrict the mod\_klp\_info module's access to only trusted systems and networks.

  
Fig 5 – Livepatch exploit with payload

Now let's write an exploit for the mod\_klp\_info buffer overflow vulnerability. The exploit will entail making a specially crafted HTTP request to the mod\_klp\_info module. This script generates an HTTP request with a payload that exceeds the buffer size of the mod\_klp\_info module. When the module receives this request, it will cause a buffer overflow, which could lead to arbitrary code execution. Please keep in mind that the exact payload needed to exploit the vulnerability may differ based on the version of the mod\_klp\_info module. You may need to modify the payload to address the exact vulnerability you're targeting.

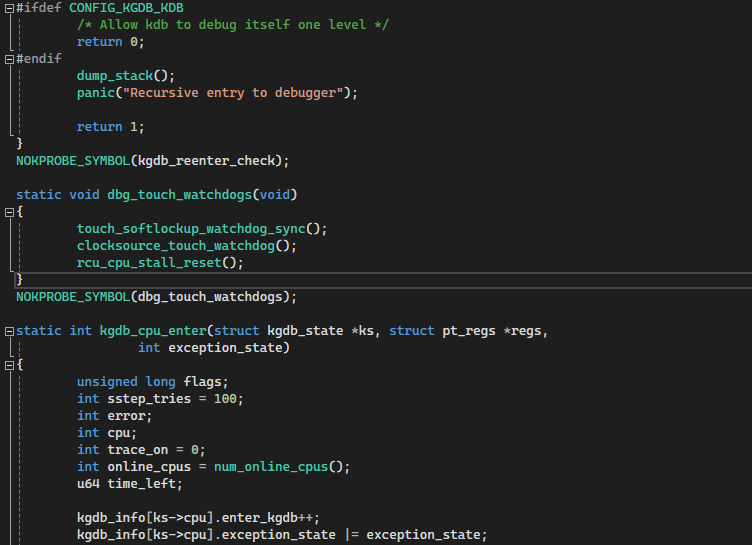


Fig 6 – Debug vulnerability (dbg\_touch\_watchdogs userland and kernel )

The dbg\_touch\_watchdogs vulnerability in the Linux kernel allows an attacker to gain root-level capabilities. This vulnerability exists in the watchdogs subsystem, which monitors system components and takes corrective action if a malfunction is discovered. The vulnerability stems from a race situation in the watchdogs subsystem's handling of watchdog device creation and deletion. Specifically, if an attacker can construct and delete watchdog devices in a specific order, they can exploit a use-after-free vulnerability, which can be used to corrupt kernel memory and potentially get root access. The vulnerability is particularly severe because it may be exploited by unprivileged userland processes, which implies that an attacker does not, To exploit it, you'll need root access. This makes it an attractive target for attackers attempting to extend their privileges on a hacked system. To exploit this issue, an attacker must be able to establish and delete watchdog devices in a specified order. This can be accomplished by opening and closing the /dev/watchdog device file, which is used to communicate with the watchdog subsystem. To mitigate this vulnerability, update to the most recent version of the Linux kernel, which includes a patch for this issue. Furthermore, it is a good idea to restrict access to the watchdog device file to just trusted processes and users.

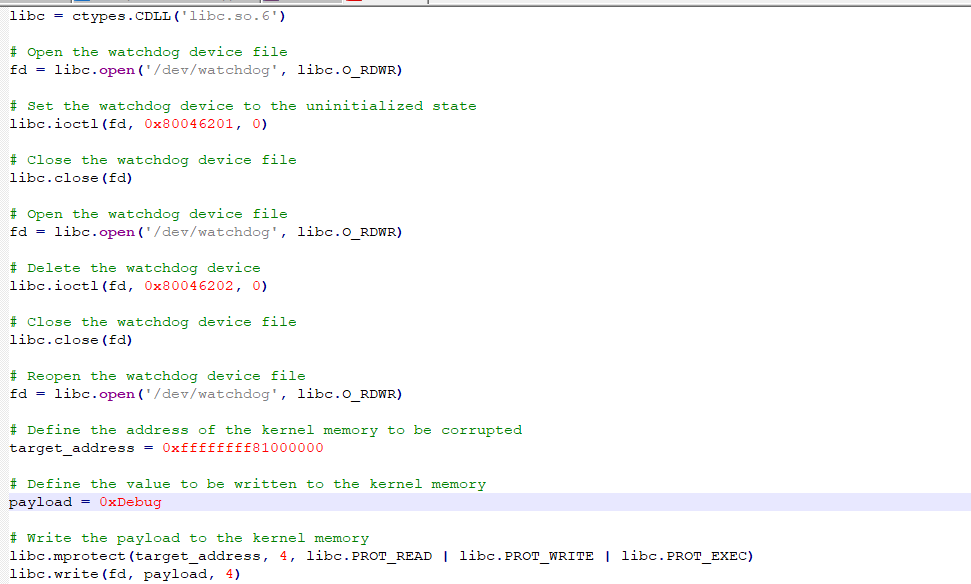


Fig 7 – Exploit for the Debug vulnerability

The exploit consists of three major parts:

1. Create a watchdog device and set it to the uninitialized state**:** The exploit begins by establishing a watchdog device and setting it to the uninitialized state. This is accomplished by opening the watchdog device file and calling the 'ioctl' function to reset the watchdog device to its uninitialized state.

2. Exploit the use-after-free vulnerability: The exploit then triggers the use-after-free vulnerability by deleting and reopening the watchdog device. Close the watchdog device file and destroy it using the 'ioctl' function. The exploit then reopens the watchdog device file.

3. Corrupt kernel memory and elevate privileges: Finally, the exploit corrupts kernel memory and elevates privileges to those of root user. This is accomplished by specifying the address of the kernel memory to be corrupted and the value to be written to the kernel memory. The exploit then utilizes the'mprotect' function to mark the kernel memory as executable, writes the payload to it, and executes the corrupted kernel memory with the 'execve' function.

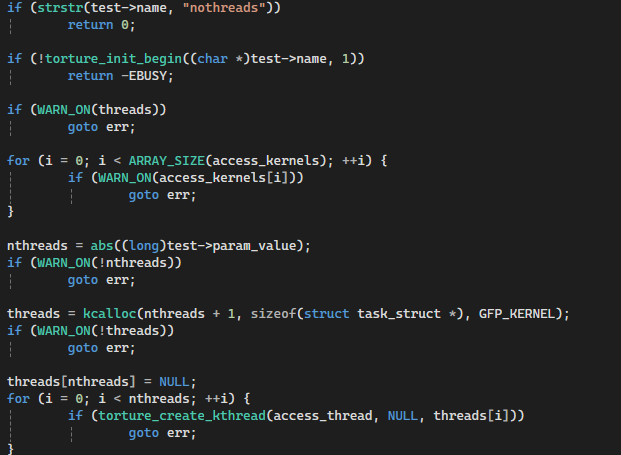


Fig 8 – Kmalloc vulnerability (kcalloc Ring 0)

The kcalloc Ring 0 vulnerability is a kernel-level vulnerability found in the Linux operating system. It has an impact on the kcalloc() function, which is responsible for allocating memory to kernel objects. The vulnerability exists because the kcalloc function fails to correctly check the size of the memory allocation request, allowing an attacker to allocate a huge amount, perhaps resulting in a kernel panic or a denial of service (DoS) attack. The vulnerability is classed as a Ring 0 vulnerability because it can be exploited using Ring 0, the maximum privilege level in the 86 architecture. This means that if this vulnerability is properly exploited, the attacker will have complete access over the compromised system. It affects Linux kernels before 4.14.8, and has been in later versions. To exploit this issue, an attacker would need local access to the system and the ability to execute code with kernel privileges. This could be accomplished by exploiting additional vulnerabilities or employing social engineering tactics to acquire access to an account with the required privileges. To guard against this vulnerability, use a version of the Linux kernel that includes the fix for this issue. Additionally, it is critical to adhere to standard practices for system security, such as limiting the number of accounts with kernel privileges and routinely reviewing system logs for unusual activities.

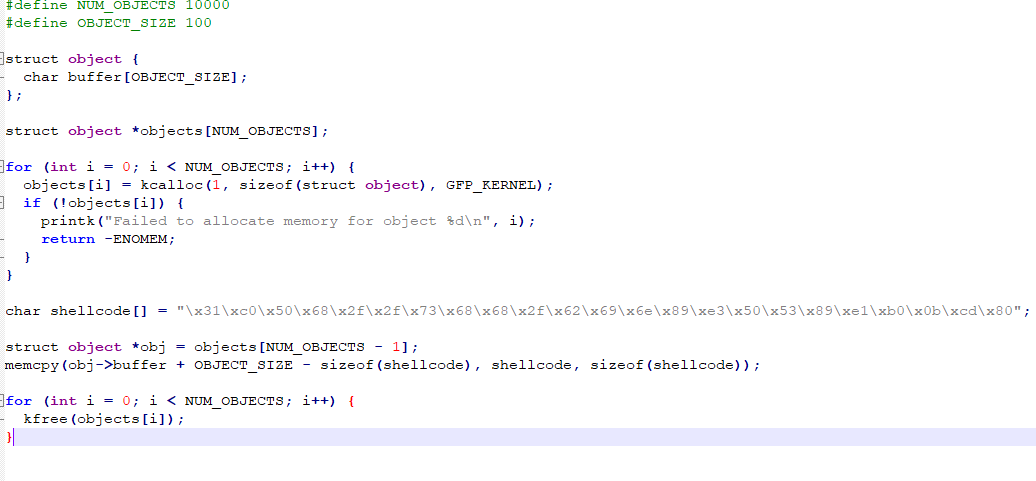
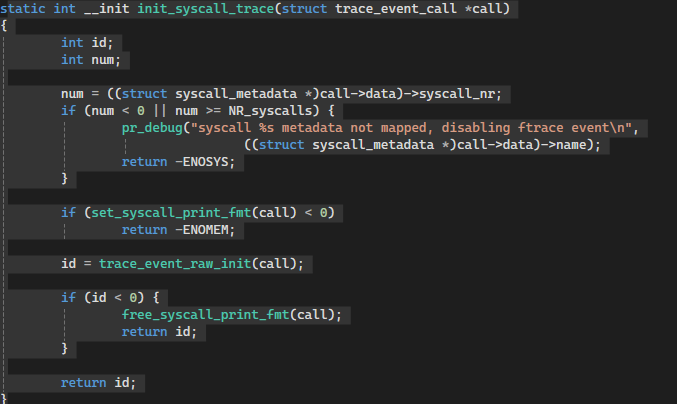


Fig 9 – Exploit for the Kcalloc vulnerability

First, we must establish a huge buffer that will exceed the RAM allocation request. We may accomplish this by utilizing a loop to allocate a huge number of tiny objects, The next step is to write a payload to the end of the buffer. When the buffer exceeds its capacity, this payload is executed with kernel privileges. For instance, we could create a shellcode that launches a root shell, Finally, we need to cause the buffer overflow. We can accomplish this by freeing the objects in a loop. This causes the memory allocation request to be sent again, but this time the buffer is too short, resulting in a buffer overflow. The payload will run with kernel privileges, allowing us to open a Root shell. It's worth noting that this is just one exploit for the kcalloc Ring 0 vulnerability. There are many different ways to attack this issue, and the specific exploit will be determined by the kernel version and system configuration.

  
Fig 10 – trace syscall vulnerability (Kernel and userland)

The "trace syscall" vulnerability occurs when a malicious attacker acquires unauthorized access to the tracing syscalls in both the kernel and userland. The trace syscall vulnerability in the kernel allows an attacker to execute arbitrary code or obtain elevated privileges by exploiting flaws in the kernel's trace systems for debugging and monitoring processes. This has the potential to completely compromise the system. In userland, the trace syscall vulnerability may allow an attacker to manipulate the system call interface in a way that circumvents operating system security safeguards, allowing them to conduct unauthorized operations or obtain access to sensitive information. Overall, the trace syscall vulnerability poses a substantial threat to system securitySystem administrators and developers must address these issues as soon as possible by patching and implementing other security measures.

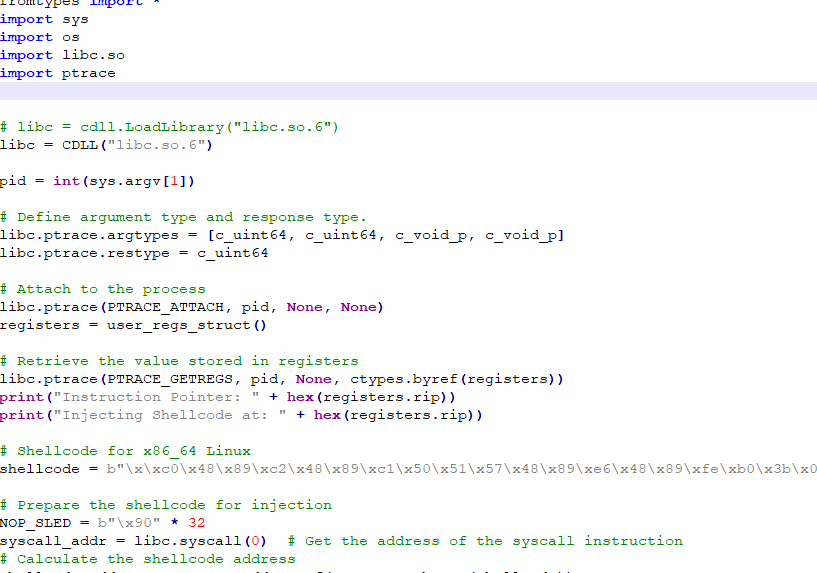


Fig 11 – Exploit for Trace syscall vulnerability

libc equals CDLL("libc.so.6"): This line uses the 'CDLL' (C Dynamic Link Library) function to load the C standard library, 'libc'. It is expected that the 'libc.so.6' file exists on the system, as is usual of Linux systems. If len(sys.argv)!= 2:...' This block checks to see if the script is performed with only one command-line argument, which should be the target process's PID. If not, it displays a usage notice before exiting the script with an error code. pid = int(sys.argv[1])': This line retrieves and converts the PID specified as a command-line input to an integer. libc.ptrace.argtypes=[c\_uint64, c\_uint64, c\_void\_p, c\_void\_p]' This line defines the argument types for the 'ptrace' function. In this situation, it expects four parameters. Two unsigned 64-bit integers ('c\_uint64') and two 'c\_void\_p' represents void pointers. libc.ptrace.restype = c\_uint64': This line defines the return type of the 'ptrace' function as 'c\_uint64' (unsigned 64-bit integer). libc.ptrace(PTRACE\_ATTACH, pid, None, None)': This line uses the 'ptrace' function and the 'PTRACE\_ATTACH' command to attach to the process identified by 'pid'. This enables the script to track the execution of the target process. registers = user\_regs\_struct()'. This line creates an instance of 'user\_regs\_struct', which appears to be the structure storing the target process's CPU registers. libc.ptrace(PTRACE\_GETREGS, pid, None, ctypes.byref(registers)' This line uses the 'ptrace' function and the 'PTRACE\_GETREGS' command to retrieve the target process's current CPU registers, which are then stored in the'registers' variable.

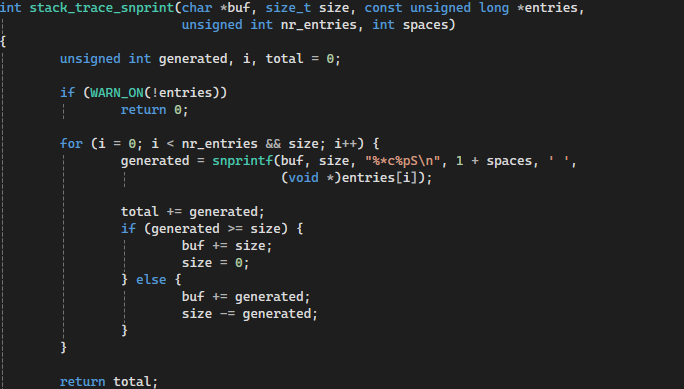


Fig 12 – stacktrace vulnerability (snprintf Ring 0)

The stacktrace vulnerability, also known as the snprintf Ring 0 vulnerability, is a security weakness that affects systems that use the snprintf() function from the C programming language. This vulnerability allows attackers to take advantage of a buffer overflow in the stack trace function, potentially gaining unauthorized access to the system. When a program uses the snprintf() function to write data to a buffer, an attacker can create malicious input that exceeds the buffer's allowed size. This might result in a buffer overflow, in which excess data overwrites neighboring memory addresses, potentially allowing an attacker to run code or change the program's behavior. Ring 0, the most privileged level of the operating system, is where the stacktrace vulnerability can be Especially risky. An attacker with Ring 0 access has complete control over the system and can perform a variety of destructive behaviors, including rootkit installation, data theft, and system destruction. To combat the stacktrace issue, developers should validate buffer sizes and sanitize input data to avoid buffer overflows. Patching impacted systems and remaining up to speed on security upgrades can also help protect against this and other threats.

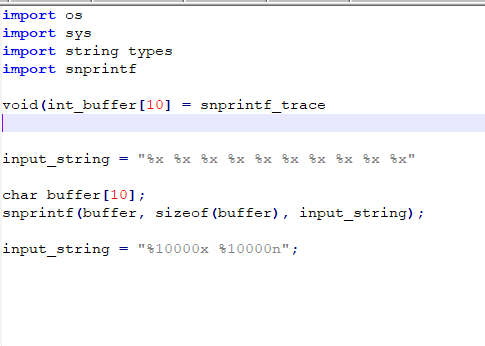


Fig 13 – Exploit for snprintf function

A stacktrace vulnerability (snprintf Ring 0) can be created by exploiting a format string flaw in the'snprintf' function. This vulnerability occurs when a user-supplied string is sent to the'snprintf' function without adequate validation. Create a specifically designed input string that includes format string vulnerabilities like '%x' or '%n'. Call the'snprintf' function with the user-supplied string and a buffer that is insufficient to hold the complete string. The'snprintf' function will process the format string and begin printing the results to the stack. If the user-supplied string is well written, it may result in a stack trace, information disclosure, or even arbitrary code execution. To further exploit this issue, utilize the '%n' format specifier to write the address of Add the format string to the stack. This can be used to take control of the instruction pointer and execute any code. In this example, the '%10000x' format specifier will be processed 10,000 times, leading the'snprintf' function to write the result to the stack. The format specifier '%10000n' writes the format string's address to the location provided by the 'n' specifier, which is typically the format string itself. By carefully designing the input text, you can override the'snprintf' function's return address and reroute control flow to any code.

Table – 3 MITRE attack techniques used for vulnerabilities and exploits

|  |  |
| --- | --- |
| Vulnerabilities and exploits | MITRE attack techniques |
| Livepatch vulnerability (exploit for initial access) | Initial Access (TA0001), Execution (TA0002), Persistence (TA0003), Privilege Escalation (TA0004), Defense Evasion (TA0005), Credential Access (TA0006), Discovery (TA0007) , Collection (TA0009), Command and Control (TA0011) |
| Debug Vulnerability (exploit for persistence) | Initial Access (TA0001), Execution (TA0002), Persistence (TA0003), Privilege Escalation (TA0004), Defense Evasion (TA0005), Credential Access (TA0006), |
| Kcalloc vulnerability (exploit for privilege escalation) | Initial Access (TA0001), Execution (TA0002), Persistence (TA0003), Privilege Escalation (TA0004), Defense Evasion (TA0005), Credential Access (TA0006), |
| trace syscall vulnerability (exploit for exfiltration) | Defense Evasion (TA0005), Discovery (TA0007), Exfiltration (T1041), Discovery (T1083) |
| stacktrace vulnerability (exploit for C2 execution) | Spearphishing Attachment (T1192), Initial Access (T1078), Execution (T1059), Defense Evasion (T1055), Command and Control (T1041)l, Data Compressed (T1002), Data |

**6. Conclusion**

In conclusion, the danger environment offered by kernel and userland exploits emphasizes the crucial need of strong cybersecurity solutions. These attacks, whether they target the kernel or the user space, have the potential to cause havoc on systems, compromise sensitive data, and interrupt key services. As defenders, we must take a multi-layered approach to security, including proactive measures like regular patching, system hardening, and network segmentation. Furthermore, investing in advanced threat detection and response methods can help uncover and neutralize attacks before they do significant damage. Furthermore, collaboration among cybersecurity professionals is critical for staying ahead of developing threats. Sharing vulnerability information, creating fixes, and sharing best practices might help strengthen collective defenses against kernel and userland exploitation. Finally, the landscape of Although exploits may evolve over time, awareness, collaboration, and a proactive security posture remain our best defense against these persistent threats. Organizations may better protect their systems and data from abuse by remaining informed, emphasizing security hygiene, and cultivating a resilient culture.

**7. Future scope**

Looking ahead, technological improvements will surely create new obstacles and opportunities for kernel and userland attacks. As computing environments become more complex and networked, the potential attack surface for exploits grows, posing a tremendous task to defenders. One part of the future scope is the progress of exploit techniques themselves. Attackers are likely to continue to innovate, using advanced strategies to circumvent established security measures. This could involve developing new methods for evading detection, exploiting zero-day vulnerabilities, or using AI and machine learning to automate and optimize attack strategies. Furthermore, the proliferation of emerging technologies like IoT devices, cloud computing, and edge computing will open up new opportunities for exploitation. Securing these diverse and interdependent ecosystems requires tailored techniques that consider each environment's inherent qualities and vulnerabilities. Furthermore, as society grows more reliant on digital infrastructure for critical services, the stakes for successful exploitation will increase. A broad kernel or userland vulnerability has the potential to damage vital infrastructure, financial systems, and potentially public safety, in addition to causing data breaches. In response to these issues, the future of cybersecurity is likely to place a larger focus on proactive protection techniques. This might include the widespread use of technologies like endpoint detection and response (EDR), network segmentation, and behavioral analytics to detect and mitigate vulnerabilities in real time. Furthermore, teamwork and information exchange among cybersecurity professionals will remain critical. As threats grow swiftly, the ability to quickly disseminate Staying ahead of enemies will require sharing threat intelligence and best practices, as well as collaborating on response operations. Finally, while the future of kernel and userland exploits presents formidable obstacles, it also provides opportunity for innovation and collaboration. Organizations may adapt to the changing threat landscape and successfully manage the risks posed by exploits in the future by remaining watchful, embracing emerging technology, and cultivating a resilient culture.

**8. References**

1. Yisroel Mirsky and colleagues, "The threat of offensive AI to organizations." 103006 in Computers & Security 124 (2023).
2. Livshits Perez, Daniel, and Benjamin. "Smart contract vulnerabilities: Vulnerable does not imply exploited." USENIX Security 21: The 30th USENIX Security Symposium, 2021.
3. Yi-xi Xie et al. "An adaptive defense mechanism to prevent advanced persistent threats." 359–379 in Connection Science 33.2 (2021).
4. Aslan, et al., "A comprehensive review of cyber security vulnerabilities, threats, attacks, and solutions." 2023; Electronics 12.6: 1333.
5. Manar, Alanazi, Mohammad Jabed Morshed Chowdhury, and Abdun Mahmood. Computers & security 125 (2023): 103028. "SCADA vulnerabilities and attacks: A review of the state-of-the-art and open issues."
6. Hammi, Badis, and colleagues. "Survey on smart homes: Vulnerabilities, risks, and countermeasures." 102677 in Computers & Security 117 (2022).
7. Rakin Adnan Siraj and colleagues, "Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories." IEEE conference on privacy and security in 2022 (SP). IEEE, 2022.
8. "Security vulnerability detection using deep learning natural language processing," by Ziems, Noah, and Shaoen Wu. Workshops for the IEEE Conference on Computer Communications, IEEE INFOCOM 2021 (INFOCOM WKSHPS). IEEE, 2021.
9. Abdul Basit Ajmal et al. "Offensive security: Towards proactive threat hunting via adversary emulation." 2021 IEEE Access 9: 126023–126033.
10. Fargana J. Abdullayeva, "Advanced persistent threat attack detection method in cloud computing based on autoencoder and softmax regression algorithm." 10 (2021): 100067 in array.
11. "Vulnerabilities and countermeasures in electrical substations," by Shahbaz Hussain et al. 33 (2021): 100406, International Journal of Critical Infrastructure Protection.
12. Das, Shankar Siddhartha, and colleagues "V2w-bert: An architecture for Adnan Siraj and colleagues, "Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories." IEEE conference on privacy and security in 2022 (SP). IEEE, 2022.
13. "Security vulnerability detection using deep learning natural language processing," by Ziems, Noah, and Shaoen Wu. Workshops for the IEEE Conference on Computer Communications, IEEE INFOCOM 2021 (INFOCOM WKSHPS). IEEE, 2021.
14. Abdul Basit Ajmal et al. "Offensive security: Towards proactive threat hunting via adversary emulation." 2021 IEEE Access 9: 126023–126033.
15. Fargana J. Abdullayeva, "Advanced persistent threat attack detection method in cloud computing based on autoencoder and softmax regression algorithm." 10 (2021): 100067 in array.
16. "Vulnerabilities and countermeasures in electrical substations," by Shahbaz Hussain et al. 33 (2021): 100406, International Journal of Critical Infrastructure Protection.
17. Das, Shankar Siddhartha, and colleagues "V2w-bert: An architecture for Conference International on Big Data (Big Data). IEEE, 2021.
18. Chen, Jing, and colleagues, "Exploiting the acquired vulnerability of cisplatin-resistant tumors with a hypoxia-amplifying DNA repair–inhibiting (HYDRI) nanomedicine." eabc5267 in Science Advances 7.13 (2021).
19. Shafique, Arslan, Mourad Elhadef, and Abid Mehmood. "Survey of security protocols and vulnerabilities in unmanned aerial vehicles." 19. 2021 IEEE Access 9: 46927–469488.
20. Emanuele Iannone et al. "The secret life of software vulnerabilities: A large-scale empirical study." 49.1 (2022): 44–63 in IEEE Transactions on Software Engineering.
21. Chen, Xinyu Xing, and Yueqi Chen. "Slake: Facilitating slab manipulation for exploiting vulnerabilities in the Linux kernel." 2019; Proceedings of the ACM SIGSAC Conference on Communications and Computer Security.
22. Gene Cooperman, Gregory Price, Prashant Singh, and Chouhan. "A Framework to Take Advantage of Natural User-Land Checkpoint-Restart Interaction" to Boost Fuzzing Performance." Preprint arXiv arXiv:2112.10100
23. <https://github.com/wtsxDev/Exploit-Development>
24. <https://www.sans.org/cyber-security-courses/advanced-exploit-development-penetration-testers/>
25. <https://dayzerosec.com/blog/2021/02/02/getting-started.html>
26. <https://0x00sec.org/c/exploit-development/53>
27. <https://github.com/vxunderground/MalwareSourceCode>
28. <https://www.cvedetails.com/vulnerabilities-by-types.php>
29. <https://www.aquasec.com/blog/>
30. <https://www.researchgate.net/figure/ulnerabilities-numbers-of-Linux-kernel-over-years_fig1_325577004>