Linux Incident Response - understanding the heap and the stack

Taz Wake | Halkyn Consulting Ltd | Originally published 2023-12-18

Introduction

In Linux, effective memory management puts the fun into the fundamentals of robust software development. This high-level article looks at the heap and stack, two critical components of memory management, and examines their interactions with 64-bit registers, security implications, and real-world applications.

Stacking with the stack

The stack, operating on the Last In, First Out (LIFO) principle, is a structured area of memory crucial for managing function calls in Linux. Each time a function is called, a stack frame is created, containing function parameters, local variables, and the return address. The stack's automatic allocation, managed by the compiler, and its fixed size predetermined at the program's start, make it a fast and efficient memory resource. *However, its fixed size can lead to "stack overflow" if not managed carefully.*

The Stack and Function Calls

In function calls, the stack plays a vital role. For instance, when a function 'A' calls another function 'B', a new stack frame for 'B' is placed on top of 'A''s. After 'B' completes execution and returns, its stack frame is discarded, revealing 'A''s frame. This structured and temporary nature of the stack makes it ideal for handling function calls.

Heaping it onto the heap

In contrast to the stack, the heap is a dynamic, less structured memory area used for global or long-lived memory allocations. Unlike the stack, memory on the heap must be manually allocated and deallocated, typically using functions like **malloc()** and **free()**. This flexibility is essential for handling data that extends beyond the scope of a single function call but introduces complexity and potential fragmentation issues.

Dynamic Memory Allocation

The heap is particularly useful for allocating large data structures or objects whose size cannot be determined at compile time. For example, a program might use **malloc()** to dynamically allocate memory for an array based on user input. This memory remains allocated until explicitly freed, allowing for greater flexibility but requiring careful memory management.

Comparing the Heap and Stack

The stack and heap serve different purposes in memory management. The stack's automatic, fast, and fixed-size nature makes it suitable for small, temporary variables. In contrast, the heap's manual, slower, and dynamically-sized nature is better suited for larger or longer-lived data.

Basic Security Measures

Linux employs security measures like Data Execution Prevention (DEP) and Address Space Layout Randomisation (ASLR) to safeguard the heap and stack. DEP prohibits code execution from non-executable memory segments, mitigating stack-based buffer overflow attacks. ASLR randomises memory addresses of the stack, heap, and other critical structures, making it difficult for attackers to predict where code or data resides in memory.

Wait, security exists? What?

There are a lot of security measures that can be implemented in Linux applications (or by the Kernel) to protect the heap and the stack. Some common examples include:

1. **Canaries**: StackGuard, a notable implementation of stack protection, uses canaries to prevent buffer overflows. These canaries are special values placed in memory that, if altered, indicate a buffer overflow. There are different types of canaries:

- **Terminator Canaries**: Contain characters like NULL, CR, LF, and EOF, which terminate most string operations.
- **Random Canaries**: Generated randomly at program start and are more difficult for an attacker to predict.
- **Random XOR Canaries**: These are random canaries XOR-scrambled with parts of the control data, making it harder for attackers to bypass the protection.

These canaries, especially when used in conjunction with other security measures, can significantly reduce the risk of successful buffer overflow attacks by terminating the program if a canary value is altered.

2. **Bounds Checking**: This compiler-based technique adds runtime bounds information for each block of memory and checks all pointers against these bounds. Bounds checking can be performed at pointer calculation time or at dereference time. Implementations may use a central repository describing each memory block or 'fat pointers', which contain both the pointer and data describing the region they point to.

3. **Tagging**: This involves marking certain memory areas as non-executable or non-allocated to prevent them from containing executable code or being affected by buffer overflows. The NX bit hardware feature supported by Intel, AMD, and ARM processors is an example of tagging used to detect buffer overflows.

4. **Address Space Randomization (ASLR)**: ASLR randomly rearranges the address space locations of key data areas of a process, making it harder for attackers to predict

the location of executable code or other critical data areas. This unpredictability makes it difficult for attackers to successfully execute buffer overflow attacks.

5. **Data Execution Prevention (DEP)**: DEP flags specific areas of memory as non-executable, preventing an attack from executing code in these regions. This is particularly useful in mitigating stack-based buffer overflow attacks, where the attacker tries to execute code in the stack area of memory.

6. **Structured Exception Handler Overwrite Protection (SEHOP)**: SEHOP helps to prevent malicious code from exploiting the Structured Exception Handling system in operating systems, thus thwarting a common technique used in buffer overflow attacks.

Understanding Stack-Based Buffer Overflows

Stack-based buffer overflows occur when data exceeds the buffer's capacity on the stack, potentially overwriting adjacent memory, leading to data corruption or altered execution flow. These are critical vulnerabilities that can be exploited to execute arbitrary code, often leading to system compromise.

The Complexity of Heap-Based Attacks

Heap-based attacks exploit dynamic memory management vulnerabilities like heap overflows or use-after-free errors. These attacks are more intricate than stack overflows but can lead to severe consequences such as arbitrary code execution and instability in applications.

Memory Management in 64-bit Linux Systems

In 64-bit Linux systems, memory management benefits from larger address spaces provided by 64-bit registers. This enhancement allows for more efficient management of both heap and stack, especially in memory-intensive applications, by addressing a significantly larger memory space.

Real-World Applications and Implications

In practical scenarios, like a Linux-based web server, the stack and heap are used in tandem. The stack manages individual client requests and operations, while the heap is employed for storing session data or large datasets. The effective management of these memory structures is crucial for handling multiple simultaneous connections efficiently.

Advanced Memory Management Techniques

Modern Linux systems also employ advanced memory management techniques like memory mapping and shared memory. These methods leverage the capabilities of 64-bit architectures for more effective and flexible memory management, essential for high-performance applications.

About Halkyn Consulting

Halkyn Consulting Ltd is a UK-based security and risk management consultancy providing 24/7 incident response, digital forensics, threat hunting, virtual security leadership (vCISO) and security compliance support to organisations worldwide. We deliver clear, practical, vendor-independent advice based on real-world threats — no hype, no upsell.

More free security resources are available at www.halkynconsulting.co.uk/security-resources.

Web: www.halkynconsulting.co.uk | **Email:** info@halkynconsulting.co.uk | **Tel:** +44 1244 940858