Journal of Integrated Studies

*Article*

# The Predictable Unpredictable: Undermining Linux ASLR Through Memory Disclosure

**Robert Ingram**

## Abstract

Address Space Layout Randomization (ASLR) is designed to mitigate memory corruption exploits such as buffer overflows by randomizing the locations of critical memory regions. While ASLR significantly increases the difficulty of predicting memory addresses, research has shown that it is not an infallible defense. This paper investigates the practical effectiveness of Linux's ASLR implementation in preventing buffer overflow attacks, with particular emphasis on how its protections can be bypassed through memory disclosure. This paper combines a literature review of recent work on ASLR's strengths and weaknesses, including cases where ASLR paradoxically aids exploitation, systematic bypass methods, and complementary defenses with an experimental evaluation on a 64-bit Ubuntu Linux system. The experiment involved implementing a vulnerable C program and recording memory layouts with ASLR enabled and disabled. Results demonstrate that while ASLR introduces meaningful entropy across stack, heap, and code segments (approximately 16–19 bits), these protections are fully neutralized when memory addresses are disclosed. Observations confirm that memory leaks convert ASLR from a probabilistic barrier into a deterministic vulnerability, enabling precise attacks such as Return-Oriented Programming (ROP).

**Keywords**: Address Space Layout Randomization, Buffer Overflow, Memory Disclosure, Exploit Mitigation

## Introduction

Address Space Layout Randomization (ASLR) is widely recognized as a "de-facto standard exploit mitigation" technique in modern software (Jang, 2022). Its fundamental concept involves unpredictably randomizing memory layouts of critical regions, such as the stack, heap, and dynamic-link libraries (DLLs), to significantly increase the difficulty of memory exploitation (Brizendine & Rimal, 2025). This randomization prevents attackers from reliably predicting the memory locations of target code or data, thereby raising the bar for various memory corruption attacks, including buffer overflows (Brizendine & Rimal, 2025). Popular operating systems like Linux, OSX, and Windows have adopted ASLR by default for their applications, acknowledging its proven efficacy and practicality as a core software exploit mitigation technique (Brizendine & Rimal, 2025). Especially in 64-bit systems, ASLR makes it infeasible to predict virtual memory addresses, compelling attackers to acquire stronger exploitation capabilities, such as information leakage, to succeed (Jang, 2022).

Despite its widespread adoption and general effectiveness, ASLR is not impenetrable, and its protective capabilities are often limited by inherent design flaws and platform-specific vulnerabilities. Recent research, such as "Predictable Paths: Novel ASLR Bypass Methods and Mitigations," highlights advanced ASLR bypass methods on modern Windows systems, demonstrating how predictable internal structures can be exploited to disclose base addresses of system DLLs (Brizendine & Rimal, 2025). This underscores that even high entropy ASLR implementations are not impervious to sophisticated attacks that leverage memory disclosure vulnerabilities.

My motivation in this paper is to explore the practical effectiveness of ASLR and to demonstrate how attackers can potentially bypass its protections. I aim to answer the research question: "How effective is modern ASLR implementation in preventing buffer overflow exploits compared to a non-randomized memory layout?". To achieve this, my study will include both a review of recent academic research on ASLR's strengths and weaknesses, as well as a proof-of-concept exploit run in a 64-bit Ubuntu Linux System. This experiment will involve implementing a vulnerable C program, developing exploits with and without ASLR enabled, and then testing and evaluating the success rates of these exploits to quantify ASLR's protective capabilities.

## Background

### Key Concepts

#### *High Entropy ASLR*

An advanced form of ASLR is High Entropy ASLR, primarily implemented in 64-bit Windows systems (Brizendine & Rimal, 2025). This enhancement dramatically increases the level of entropy in compiled binaries by leveraging the significantly larger 64-bit address space, leading to a much greater range of possible memory locations for base addresses. For instance, it can provide up to 24 bits of entropy for base addresses, making brute-force attacks virtually infeasible. High Entropy ASLR also randomizes stacks, heaps, and critical system DLLs like Kernel32.dll, Kernelbase.dll, and NTDLL.dll. Additionally, the "Force Relocate Images" feature in Windows further strengthens ASLR by ensuring all DLLs are relocated at runtime, even those not initially compiled with ASLR support.

### *Buffer Overflow*

A buffer overflow is a memory corruption vulnerability that occurs when a program attempts to write more data into a fixed-size buffer than it can hold, leading to the overwriting of adjacent memory locations (Brizendine & Rimal, 2025). In heap-based buffer overflows, this vulnerability occurs within the heap area. Common causes include miscalculating buffer lengths (e.g., using signed integers with negative inputs) or failing to limit data input. Exploitation can involve corrupting either the metadata of the memory allocator or adjacent application heap data. A specific type of buffer overflow is NULL poisoning, often resulting from an off-by-one error, where the overwritten value is a NULL byte. This can be a reliable primitive to pivot the stack frame for further attacks if stack canaries are absent or bypassed.

### *Code Reuse Attacks*

Code Reuse Attacks (CRAs) are a class of control flow hijacking technologies that do not inject new malicious code but instead re-use existing code snippets (known as "gadgets") from the operating system or application to form malicious payloads (Brizendine & Rimal, 2025). Before launching an attack, adversaries must identify and chain these gadgets. Gadgets are typically short sequences of instructions ending with control flow altering instructions like ret (for ROP), or jmp and call (for JOP).

### *Return-Oriented Programming*

Return-Oriented Programming (ROP) is a practical exploitation technique that allows adversaries to circumvent Data Execution Prevention (DEP) [1]. It achieves this by chaining together existing code fragments, where the ret instruction is used to pop the next instruction pointer from the stack (Brizendine & Rimal, 2025). By manipulating the stack to contain a

sequence of gadget addresses, an attacker can orchestrate arbitrary computation without injecting new executable code, as the gadgets are repurposed legitimate code already present in executable memory region.

### *Return-to-libc*

Return-to-libc is an earlier form of code reuse attack. It predates ROP and typically involves redirecting program execution to functions within the standard C library (libc) to perform malicious actions, often bypassing non-executable stack protections (Brizendine & Rimal, 2025). Shacham et al. demonstrated a brute-force attack on PaX ASLR using return-to-libc techniques, highlighting ASLR's vulnerability on 32-bit systems with insufficient entropy (cited in Brizendine & Rimal, 2025).

## ASLR Implementation, Impact, and Challenges

In Linux, early implementations like the PaX kernel patch (2001) introduced address randomization for the stack and mapped libraries (PaX Team, 2003). Modern Linux systems, such as Ubuntu 22.04, utilize ASLR, though some implementations, particularly with Linux folios, can see entropy drop to as low as 19 bits, making bypasses feasible within minutes on x86_64 machines (PaX Team, 2003). Further protections in Linux include Position Independent Executables (PIE), which randomize the base address of the executable itself, and Relocation Read-Only (RELRO), which makes the Global Offset Table (GOT) non-writable after symbol resolution, preventing certain overwrite attacks (Brizendine & Rimal, 2025).

## Memory Layout With and Without ASLR

To clarify the fundamental mechanism of ASLR, consider the virtual memory layout of a process:

### *Without ASLR*

In a system without ASLR, the major memory segments such as the code segment (.text), data segment, heap, stack, and loaded libraries are loaded at fixed, predictable virtual memory addresses. This means that upon every execution of a program, these segments will reside at the exact same set of addresses. An attacker can, therefore, hardcode these addresses into an exploit, reliably targeting specific functions or data structures.

### *With ASLR*

When ASLR is enabled, the base addresses of these key memory segments (code, data, heap, stack, and shared libraries/DLLs) are randomized at runtime. This means that with each execution of the program, these segments are mapped to different, unpredictable starting addresses within the virtual address space. While the relative offsets between objects within a single segment often remain constant, the absolute starting addresses of the segments themselves change. For example, a library like libc will be loaded at a different base address each time the program runs, making it impossible for an attacker to reliably jump to a hardcoded address within it without first discovering its current randomized location through an information leak.

## Literature Review

### ASLR's Efficacy and Inherent Limitations

ASLR's practicality is well-proven, and its adoption across major operating systems like Linux, macOS, and Windows is widespread, including in embedded software (Jang, 2022). For 64-bit systems, ASLR is particularly effective, making it infeasible for attackers to predict virtual memory addresses without additional exploitation capabilities like information leakage (Brizendine & Rimal, 2025).

However, despite its general efficacy, ASLR possesses inherent limitations and can, in certain circumstances, be circumvented or even ironically aid attackers. Jang (2022) introduces the concept of "BadASLR," which refers to rare edge cases where ASLR counter-intuitively assists in successful memory exploitation. These cases include aiding free chunk reclamation in heap spraying (Type-I), supporting stack-pivoting in frame-pointer null poisoning (Type-II), reviving the exploitability of invalid pointer bugs (Type-III), and introducing wild-card ROP gadgets (Type-IV) (Brizendine & Rimal, 2025). The paper notes that BadASLR Type-I and Type-III cases have been found in real-world vulnerabilities, particularly in smaller, non-interactive applications like multimedia/document parsers (Brizendine & Rimal, 2025). In contrast, BadASLR scenarios are deemed very unlikely in complex, interactive software such as browsers or kernels (Brizendine & Rimal, 2025).

Brizendine and Rimal (2025) corroborate that while ASLR is a preeminent mitigation, its effectiveness is often curtailed by inherent design flaws and platform-specific vulnerabilities. Their work demonstrates that even High Entropy ASLR, designed to provide a higher level of security, is "not impervious" and can be susceptible to advanced attacks. The authors highlight that raising entropy alone does not guarantee acceptable security, revealing a critical need for more comprehensive, potentially OS-level, improvements in randomization and access controls. Li et al. (2023) further emphasize that no ASLR implementation can completely hide all code and code pointers, leading to potential bypasses through sophisticated code probing technologies. While ASLR scrambles memory to make object addresses unknown, adversaries can still obtain target code addresses and forms either directly or indirectly (Li et al., 2023). This collective body

of work underscores that while ASLR is a powerful defense, it requires complementary measures and architectural enhancements to address its inherent limitations and prevent its circumvention.

## Code Reuse Attacks (CRAs) and ROP as Exploitation Techniques

Code Reuse Attacks (CRAs), particularly Return-Oriented Programming (ROP), are a primary concern in memory security, actively exploited by adversaries to bypass modern defenses such as Data Execution Prevention (DEP) and ASLR (Jang, 2022). Jang (2022) introduces ROP as a practical exploitation technique that re-uses existing code segments by chaining together small pieces of code, or "gadgets," often terminating with a ret instruction. The paper even identifies a specific BadASLR scenario (Type-IV) where ASLR itself could paradoxically introduce more diverse "wild-card ROP gadgets" in x86/x64 position-independent code environments due to randomized inter-segment distances and instruction encoding. However, it notes that this scenario is largely theoretical and unrealistic in practice due to the common availability of sufficient gadgets elsewhere.

Brizendine and Rimal (2025) detail ROP as a powerful attack method where attackers repurpose executable code fragments (gadgets) to construct a "ROP chain" for malicious payloads. By hijacking the instruction pointer (e.g., RIP in x 64 systems), attackers can redirect execution to this chain, achieving arbitrary code execution without injecting new code, thus circumventing DEP. Historically, ASLR was introduced to safeguard against return-to-libc attacks, which later evolved into more sophisticated ROP techniques. The authors' work directly employs ROP chains to achieve their ASLR bypass, demonstrating how specific ROP gadgets can systematically traverse predictable Windows internal structures to disclose critical DLL base addresses. This highlights the ongoing "arms race" between attackers and defenders, as code-

reuse attacks continue to evolve with techniques like Just-in-Time code reuse (JIT-ROP), Jump-oriented Programming (JOP), and Counterfeit Object-Oriented Programming (COOP).

Li et al. (2023) further establish that CRAs are control flow hijacking technologies that rely on adversaries preparing a "gadget chain" from existing system code snippets. They emphasize that under ASLR, especially fine-grained ASLR, the addresses and forms of code blocks become largely invisible to attackers, necessitating "code probing" to identify available gadgets. The paper's proposed defense, AntiRead, directly addresses this challenge by aiming to prevent code that has been read from being used as gadgets. AntiRead achieves this by randomizing the memory layout and disabling the execution permission of probed code pages in their original space, while providing a readable, non-executable copy for legitimate applications and an executable copy in a new address space for legal calls. This approach acknowledges the fundamental role of CRAs and ROP in modern exploitation and seeks to disrupt the attacker's ability to construct functional gadget chains.

## Memory Probing and Information Leakage as ASLR Bypass Mechanisms

A recurring theme across the sources is the critical role of memory probing and information leakage as a primary means for attackers to circumvent ASLR's unpredictability. Jang (2022) states that ASLR's effectiveness in randomizing virtual memory addresses compels attackers to "additionally equip themselves with a stronger exploitation capability – information leakage". Such leakage, for example, through arbitrary memory read capabilities (e.g., unbounded array indexing), allows attackers to dynamically discover memory addresses of crucial data structures. A single leaked pointer can then de-randomize an entire memory segment due to constant relative offsets. BadASLR Type-I (aiding free chunk reclamation) and Type-III

(reviving invalid pointer exploitability) cases found in practice exemplify scenarios where ASLR's randomization, paradoxically, creates opportunities for attackers to exploit memory layout information once a degree of control or knowledge is gained.

Brizendine and Rimal (2025) present a novel ASLR bypass methodology explicitly centered on memory disclosure vulnerabilities. Their work demonstrates how attackers can "weaponize memory disclosure vulnerabilities" to overcome even High Entropy ASLR. The core of their bypass involves using ROP to exploit predictable internal Windows structures, such as the Process Environment Block (PEB) and its associated module lists (InMemoryOrderModuleList, InLoadOrderModuleList, InInitializationOrderModuleList). By traversing these structures via fixed offsets, they reliably disclose the base addresses of critical system DLLs (Kernel32.dll, Kernelbase.dll, NTDLL.dll). This act of disclosure effectively "negates ASLR's protections" by revealing the randomized base addresses, thereby expanding the attack surface for ROP. They emphasize that a "single disclosed pointer can help de-randomize a large portion of memory space," even on 64-bit systems with high entropy.

Li et al. (2023) directly address "code reading" as a fundamental method for memory probing used by adversaries to "get available gadgets" in the face of ASLR. They outline three primary attack vectors leveraging code reading. First, gradually moving from leaked data segments to code segments (simulated by HeartBleed, which can disclose 64KB data at a time and identify code by binary forms like function headers). Second, directly reading code via a leaked code pointer and recursively following indirect addresses (typical in JIT-ROP attacks). Finally, obtaining target addresses by reading code pointers stored in continuous memory, such

as the Global Offset Table (GOT) and Virtual Tables (Vtable) through their relative addresses in the Procedure Linkage Table (PLT).

AntiRead's design aims to neutralize these probing techniques by ensuring that any code acquired through reading is either randomized or non-executable, thus preventing its use in constructing gadget chains for CRAs (Li et al., 2023). The authors highlight that this includes protecting closed-source objects and dynamically loaded code against these information leakage attacks.

## Proof-of-Concept Exploitation

To evaluate the practical effectiveness of ASLR in mitigating memory-based attacks, I conducted a controlled experiment on a modern Linux system. A custom C program was developed to expose key memory locations (code, global, stack, and heap) and to read its own memory map via /proc/self/maps. This approach not only allowed us to observe address variability across multiple executions but also demonstrated a critical weakness in ASLR: information disclosure. With the right knowledge, an attacker could compute precise offsets for code-reuse attacks such as Return-Oriented Programming (ROP). By comparing ASLR-enabled and ASLR-disabled states, I quantified address entropy and highlighted the limitations of ASLR when memory disclosure is possible.

## Experimental Setup

My experiment was conducted on a 64-bit Ubuntu 22.04 Linux system with the kernel version 6.2. Both ASLR-enabled (default) and ASLR-disabled states were tested to compare the effect of memory randomization. The ASLR configuration was controlled via the /proc/sys/kernel/randomize_va_space interface:

- ASLR enabled: cat /proc/sys/kernel/randomize_va_space returns 2 (full randomization).
- ASLR disabled: executed sudo sh -c 'echo 0 > /proc/sys/kernel/randomize_va_space' to disable all randomization.

All tests were executed on a single-user system to reduce environmental noise, and the system was rebooted between tests to ensure consistent initial conditions.

## Vulnerable Program Implementation

A small C program, aslr_info_leak.c, was implemented to include the following components:

1) Global Variable (global_var): Represents data stored in the .bss or .data segment.
2) Stack Variable (stack_var): A local variable inside main() to represent stack memory layout.
3) Heap Pointer (heap_ptr): Allocated via malloc() to monitor heap randomization.
4) Function Entry Address (main): Used to monitor code segment randomization.

The program prints the addresses of these elements and subsequently displays the process memory map by reading /proc/self/maps. This allowed observation of both segment addresses and memory layout structure, including shared libraries.

**Figure 1**

*Full Source Code of aslr_info_leak.c*

```c
#include <stdio.h>
#include <stdlib.h>

int global_var = 42;

void print_proc_maps() {
    FILE *f = fopen("/proc/self/maps", "r");
    if (!f) {
        perror("fopen");
        exit(1);
    }

    char line[256];
    printf("=== Memory map of this process ===\n");
    while (fgets(line, sizeof(line), f)) {
        printf("%s", line);
    }
    fclose(f);
}

int main() {
    int stack_var = 123;
    void *heap_ptr = malloc(16);

    printf("Address of main: %p\n", (void*)main);
    printf("Address of global_var: %p\n", (void*)&global_var);
    printf("Address of stack_var: %p\n", (void*)&stack_var);
    printf("Address of heap_ptr: %p\n", heap_ptr);

    print_proc_maps();
    return 0;
}
```

## Data Collection Procedure

The procedure involved multiple iterations to capture variability:

1) Compile the program using gcc -O0 -o aslr_info_leak aslr_info_leak.c to disable compiler optimizations that could influence memory layout.
2) Run the program 10 times with ASLR enabled and record addresses of main, global_var, stack_var, and heap_ptr.
3) Run the program 10 times with ASLR disabled and record the same addresses.
4) Store and tabulate addresses to compute observed ranges and entropy estimates for each memory segment.

Each execution was independent, with a brief pause between runs to ensure fresh allocations and avoid potential memory reuse artifacts.

### Metrics

The primary metrics evaluated were: Address Variability (the difference between minimum and maximum observed addresses across multiple runs for each memory segment) and Effective Entropy (calculated using the formula: *Entropy (Bits) = log2 (Max Address – Min Address)*)

This metric quantifies the uncertainty an attacker faces in predicting the runtime location of a memory segment.

## Results

The results of my experiment illustrate the impact of ASLR on memory layout randomization in a Linux environment. I executed the vulnerable program multiple times under two conditions: ASLR enabled (default) and ASLR disabled (via echo 0 > /proc/sys/kernel/randomize_va_space). Memory addresses of the main function, a global variable (global_var), a stack variable (stack_var), and a heap-allocated pointer (heap_ptr) were recorded for each execution.

### ASLR Enabled

When ASLR was active, memory addresses varied significantly across runs:

- Code (main): Ranged from 0x562730b3031f to 0x55c6ec5e231f
- Global variable (global_var): Ranged from 0x562730b33010 to 0x55c6ec5e5010
- Stack variable (stack_var): Ranged from 0x7ffc4baef99c to 0x7ffed3f261ec
- Heap pointer (heap_ptr): Ranged from 0x562749ffd2a0 to 0x55c70d1ee2a0

These results demonstrate the probabilistic nature of ASLR, each memory segment is independently randomized, making repeated exploitation attempts less predictable.

Disabling ASLR produced completely static memory addresses across multiple executions:

- Code (main): 0x55555555531f
- Global variable (global_var): 0x555555558010
- Stack variable (stack_var): 0x7fffffffdd8c
- Heap pointer (heap_ptr): 0x5555555592a0

The deterministic addresses confirm that ASLR is the key mechanism introducing entropy in memory layouts.

## Observed Entropy

I estimated the effective entropy in each segment using the range of observed addresses (see Table 1). These values indicate meaningful randomization, though the entropy is finite and can be effectively neutralized if memory addresses are leaked.

**Table 1**

*Observed Memory Entropy Under ASLR*

| Memory | Observed memory entropy under ASLR | | |
|---|---|---|---|
| | *Min Address* | *Max Address* | *Observed Entropy (bits)* |
| Stack | 0x7ffc4baef99c | 0x7ffed3f261ec | ~19 |
| Heap | 0x562749ffd2a0 | 0x55c70d1ee2a0 | ~18 |
| Code (main) | 0x562730b3031f | 0x55c6ec5e231f | ~19 |

## Discussion

### Practical Implications of ASLR

The experimental results confirm that Linux ASLR introduces non-trivial randomness across stack, heap, and code segments. However, the experiment also demonstrates that ASLR's protective effect can be entirely bypassed if memory disclosures are available. In my scenario, access to /proc/self/maps or direct address leaks converts ASLR from a probabilistic defense into a deterministic map of memory locations, allowing attackers to reliably predict addresses of stack, heap, and code segments.

My observations reflect principles described in "Badaslr: Exceptional cases of ASLR aiding exploitation". Specifically, leaked heap addresses resemble Type-I BadASLR scenarios, in which ASLR's heap randomization inadvertently facilitates reuse of free chunks, a common technique in heap spraying (Jang, 2022). Similarly, leaked global and heap addresses resemble Type-III cases, where ASLR can revive the exploitability of invalid pointer references by mapping previously inaccessible virtual memory addresses into valid, attacker-controlled regions. While my experiment did not execute full exploits, the deterministic addresses illustrate how BadASLR conditions could be instantiated in a controlled setting.

The /proc/self/maps experiment demonstrates an explicit form of information disclosure that undermines Address Space Layout Randomization (ASLR). In this experiment, the program reads its own memory map, revealing the exact base addresses of its executable, heap, stack, and shared libraries (e.g., libc). Although ASLR successfully randomizes these addresses on every execution, the disclosure renders this protection moot: with full knowledge of library load

addresses, an attacker could compute precise offsets for code-reuse attacks such as Return-Oriented Programming (ROP).

This finding parallels the core insight of Predictable Paths, which introduces a universal ASLR bypass on Windows by exploiting implicit disclosures (Brizendine & Rimal, 2025). Rather than relying on a single explicit leak, Predictable Paths systematically walks deterministic internal structures (like the Process Environment Block) to derive library addresses. Both approaches highlight the same critical weakness: ASLR's effectiveness hinges entirely on preventing an attacker from learning memory layout. Once addresses are disclosed, whether explicitly, through /proc/self/maps or other debugging interfaces, or implicitly, via predictable memory structures, the randomization guarantees of ASLR collapse.

By demonstrating a straightforward and easily reproducible explicit leak, this experiment complements Predictable Paths's findings, reinforcing that ASLR cannot be evaluated solely by its entropy level. Even a high-entropy randomization scheme becomes ineffective when address layout information is accessible. Together, these results underscore the need for complementary defenses such as hardened /proc permissions, pointer authentication, execute-only memory (XOM), and memory isolation to eliminate or reduce avenues of disclosure.

The effectiveness of AntiRead is further emphasized. ASLR alone cannot prevent an attacker from leveraging read-accessible memory disclosures. AntiRead-style execute-only memory would prevent such disclosures from revealing usable addresses, demonstrating a complementary mitigation strategy that addresses a fundamental limitation of ASLR.

## Summary of Findings

1) ASLR introduces meaningful entropy, but this entropy is finite and segment-dependent.

2) Memory disclosure nullifies ASLR protections, converting probabilistic defenses into deterministic attack vectors.
3) Integration with additional mitigations is necessary, including stack canaries, RELRO, and AntiRead-inspired mechanisms, to ensure defense-in-depth.

Together, these observations illustrate that ASLR is a critical but insufficient mitigation when used in isolation, particularly in environments where memory addresses may be disclosed.

## Conclusion

This study comprehensively demonstrates the practical effectiveness and critical limitations of Linux ASLR in preventing memory exploitation, particularly when faced with memory disclosure vulnerabilities. Through controlled experiments comparing ASLR-enabled and ASLR-disabled executions on a 64-bit Ubuntu system, I quantified the segment-specific entropy introduced by ASLR and unequivocally showed that even meaningful randomization is entirely neutralized by memory disclosures. Specifically, access to /proc/self/maps was shown to convert ASLR from a probabilistic defense into a deterministic map of memory locations, enabling precise targeting for advanced code-reuse attacks like Return-Oriented Programming (ROP).

These findings corroborate and substantiate theoretical insights from recent academic research, particularly the concepts of 'BadASLR,' 'Predictable Paths,' and 'AntiRead.' My practical demonstration reinforces that ASLR alone, despite its widespread adoption and perceived robustness, cannot provide a resilient defense against sophisticated attacks that leverage information leakage. The core vulnerability lies in its inability to prevent an attacker from gaining knowledge of randomized memory layouts.

Therefore, to achieve truly effective mitigation against memory exploitation, ASLR must be deployed not in isolation but as an integral component of a layered security architecture. This necessitates its combination with complementary techniques such as execute-only memory (XOM), robust stack canaries, and RELRO. Future advancements in memory randomization should also explore more aggressive access control mechanisms and 'code reading' prevention strategies.

A comprehensive defense-in-depth approach is paramount, one that explicitly accounts for both probabilistic and deterministic attack scenarios, acknowledging that the 'unpredictable' can quickly become 'predictable' through information disclosure. Only through such a multi-layered and proactive strategy can systems truly withstand the evolving landscape of memory-based exploits.

## References

Brizendine, B. & Rimal, B. P. (2025). Predictable paths: Novel ASLR bypath methods and mitigations. *IEEE Access, 13,* 102784-102802. https://doi.org/10.1109/ACCESS.2025.3578602

Jang, D. (2022). BadASLR: Exceptional cases of ASLR aiding exploitation. *Computers & Security, 112,* 102510. https://doi.org/10.1016/j.cose.2022.102510

Li, Y., Cai, J., Bao, Y., & Chung, Y. (2023). What you read is what you can't execute. *Computers & Security, 132,* 103377. https://doi.org/10.1016/j.cose.2023.103377

PaX Team. (2003). *Address Space Layout Randomization (ASLR).* http://pax.grsecurity.net/docs/aslr.txt

## Appendix A: System and Software Environment

- Operating System: Ubuntu 24.04.3 LTS (WSL2)
- Kernel Version: 5.15.167.4-microsoft-standard-WSL2
- Architecture: x86_64
- Compiler: GCC 13.3.0 (Ubuntu 13.3.0-6ubuntu2~24.04)
- Build Tools: GNU Make 4.3
- ASLR Setting: /proc/sys/kernel/randomize_va_space = 2 (enabled by default; later set to 0 for testing)

## Appendix B: Source Code

The experiment used a custom C program, aslr_info_leak.c, to display address information for program components and system memory mappings. Key features of the program:

**Prints addresses of:**

- main() function
- A global variable
- A stack variable
- A dynamically allocated heap pointer
- Outputs /proc/self/maps to show memory layout

**Builds Command:**

*gcc -O0 -o aslr_info_leak aslr_info_leak.c*

## Appendix C: Experiment Design

The experiment aimed to demonstrate the effects of Address Space Layout Randomization (ASLR).

1. Verified system ASLR settings via /proc/sys/kernel/randomize_va_space.
2. Compiled and executed aslr_info_leak multiple times with ASLR enabled to observe randomized memory addresses.
3. Disabled ASLR (echo 0 > /proc/sys/kernel/randomize_va_space) and re-ran the binary to observe deterministic addresses.
4. Collected memory maps for each run to illustrate changes.

## Appendix D: Collected Data (Representative Runs)

| Run | ASLR Setting | Main Addr | Global Var Addr | Stack Var Addr | Heap Ptr Addr |
|---|---|---|---|---|---|
| 1 | Enabled | 0x562730b3031f | 0x562730b33010 | 0x7ffed3f261ec | 0x562749ffd2a0 |
| 2 | Enabled | 0x558f5ef5d31f | 0x558f5ef60010 | 0x7ffc9ece928c | 0x558f99f112a0 |
| 3… | Enabled | 0x55c6ec5e231f | 0x55c6ec5e5010 | 0x7ffc4baef99c | 0x55c70d1ee2a0 |
| 10 | Disabled | 0x55555555531f | 0x555555558010 | 0x7fffffffdd8c | 0x5555555592a0 |

Additional full /proc/self/maps data can be provided for reproducibility.

## Appendix E: Required Software

- Ubuntu 24.04 or equivalent Linux distribution
- GCC (tested on 13.3.0)
- Access to /proc filesystem
- Root privileges (required for toggling ASLR)

*Robert Ingram is a technology consultant finishing his Bachelor of Science in Computing and IT at Athabasca University. He has worked in multiple technology fields such as cybersecurity, mobile app development, data warehousing, business intelligence, and enterprise software. His research interests include military and defence intelligence, and he aspires to work in Canada's defence sector. Outside of work, Robert enjoys running, admiring the Lexus IS350, and caring for his pet tree frog Linguine.*