Poster: Selective Exploit Mitigation Under Modular Software Distribution: A Case Study on the N00B Bounds Checker

Thomas Frans
thomas.frans@student.kuleuven.be
KU Leuven
Sint-Katelijne-Waver, Belgium

Adriaan Jacobs
adriaan.jacobs@kuleuven.be
DistriNet, KU Leuven
Ghent, Belgium

Ruben Sturm
ruben.sturm@kuleuven.be
DistriNet, KU Leuven
Ghent, Belgium

Stijn Volckaert
stijn.volckaert@kuleuven.be
DistriNet, KU Leuven
Ghent, Belgium

Abstract

Memory errors remain a leading source of security vulnerabilities in C and C++ software. *Bounds checkers* can automatically mitigate them, but have failed to gain wide adoption so far due to concerns of legacy compatibility and performance overhead. The recent N00B bounds checker (presented at CCS'26) shows how to address these concerns through a novel legacy-compatible pointer layout, which is crucial for interoperability with non-hardened code.

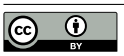
However, practical deployment challenges still remain. Like many bounds checkers, N00B assumes a *monolithic* build and deployment process where it generates the final executable and can completely control process layout and linking. Real software distribution is *modular* instead, with composable and independently built dynamic libraries, packages, and containers. A realistic deployment of N00B-hardened code therefore requires seamlessly integrating N00B- and non-N00B-hardened components, and dynamically adapting to the requirements of hardened components without imposing restrictions on the deployment of non-hardened components.

We explore such modular deployment of bounds checking in existing software distribution methods, and propose concrete solutions that bring N00B closer to practical adoption. To that end, we implement runtime support that allows N00B-hardened components to benefit from full bounds checking regardless of the threading, memory-allocation, or hardening status of the surrounding process, in large part via transparent and composable linker and loader extensions that prepare the process environment at startup.

Our proposal could generalize beyond N00B, and we hope it helps inspire conversations about similar adoption barriers for other exploit mitigations, or in other forms of software packaging (e.g. containers). We aim to discuss these possibilities with CCS attendees at the poster session.

CCS Concepts

• **Security and privacy → Systems security**; *Software security engineering*.



This work is licensed under a Creative Commons Attribution 4.0 International License.
CCS '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846445

Keywords

Bounds checking, memory safety, software distribution, systems security

ACM Reference Format:
Thomas Frans, Adriaan Jacobs, Ruben Sturm, and Stijn Volckaert. 2026. Poster: Selective Exploit Mitigation Under Modular Software Distribution: A Case Study on the N00B Bounds Checker. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3830454.3846445

1 Problem Statement and Overview

Memory-corruption bugs have long caused the majority of high-severity vulnerabilities in major software projects [9]. They occur in code written in memory-unsafe languages such as C and C++, where invalidated pointers can corrupt or leak memory and let attackers hijack control [12]. Memory-unsafe code implements many of the most popular web servers, operating system kernels, and language runtimes, and many popular dependencies of nominally memory-safe programs. Hence, a program written in a memory-safe language such as Rust often still calls into native libraries written in unsafe languages, the vulnerabilities in which may be used to bypass and corrupt the safe program subset [8].

Bounds checkers are compiler-assisted mitigations that catch these errors at run time, before an attacker can exploit them [2, 4, 10]. A new proposal called N00B manages to do so without requiring total program recompilation [4], and while staying fully compatible with uninstrumented external code. This allows for seamless *selective* hardening of (polyglot) applications with memory-unsafe components, or to lower overall overhead by hardening only the most security-critical unsafe code [14].

In practice, however, integrating with existing software distribution and linking mechanisms to leverage this design advantage turns out to be challenging. State-of-the-art exploit mitigations like N00B often assume a monolithic software build, distribution, and deployment process, but mainstream software distribution and deployment (e.g. via package managers) is *modular* instead, with individually packaged and reusable components assembled into full applications. An individual hardened component controls neither its position in the dependency hierarchy nor the initialization order,

and cannot reliably impose the process-wide constraints on memory layout, threading, or symbol-resolution that state-of-the-art defenses commonly require [2].

For instance, despite its selective application potential, N00B imposes several early and process-wide requirements on applications if any N00B-hardened code (in any transitive dependency) is to be executed. For one, N00B's current prototype heap allocator lacks any thread synchronization, making it compatible only with purely single-threaded programs. In addition, N00B expects all non-hardened objects to be allocated above a minimum address so it can efficiently ignore them during bounds checks; otherwise it misreads their pointers as hardened and may raise false out-of-bounds errors. Finally, all heap allocations should be served by N00B's allocator, so that as many pointers as possible carry bounds even if the first hardened code is loaded much later (e.g., a `dlopen`-ed plugin), which conflicts with default linking policy.

We want to explore a more modular, transparent deployment methodology for exploit mitigations. Here, we consider N00B specifically, as a case study and to prototype a general architecture that can satisfy common mitigation requirements. Concretely, we contribute a transparent program loader (`N00Bloader`) that sets up N00B's process environment before any application code runs, letting an unmodified program use N00B-hardened shared libraries (Section 2); a thread-safe implementation of `N00Balloc` to allow N00B-hardened components in multithreaded programs (Section 3); an evaluation on a realistic use case showing effectiveness and negligible performance impact (Section 4); and a discussion of generalizability to other deployment methods and mitigation types (Section 5).

Our code is available at https://github.com/adriaanjacobs/N00B.

2 A Cooperative Program Loader

In general, an application may combine N00B-hardened components with many that are unmodified and unaware of N00B. For the hardened components' checks to be effective, the whole process must meet N00B's requirements [4]:

R1 All heap allocations should be served by `N00Balloc`.

R2 All non-hardened objects must be placed above N00B's minimum address.

Monolithic N00B (and other bounds checkers) trivially meet these requirements by controlling the linking or loading stage of the final executable. In a modular setting, however, with transitive hardened and non-hardened dependencies, no single hardened component can reliably do so.

Our approach builds around the observation that a program loader is particularly well suited to implement these requirements. For one, it can directly override symbol resolution, regardless of original linking order (addressing **R1**). Secondly, as the earliest user-space code after startup, it could reserve the entire region below the N00B compatibility address early on, so that every later allocation (global, stack, or via `mmap` or `brk`) is served from above it by the kernel (addressing **R2**). Finally, the program loader is configurable at the *user* end, which is the only party that has a complete view of the assembled application and its hardened components, and can thus flexibly adapt to the requirements of the application without breaking the modularity of the distribution chain (see Section 5).

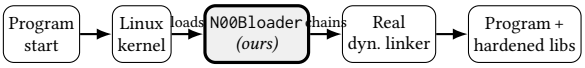


Figure 1: The `N00Bloader` runs ahead of the ordinary dynamic linker, reserves the N00B region, and preloads `N00Balloc`, so that an unmodified program can load N00B-hardened shared libraries.

To implement this behavior in existing program loaders, maintaining custom patches for different versions on each operating systems, even within UNIX, is tedious. Hence, for prototyping, we implement a small *chain loader* [5] called the `N00Bloader` that replaces the system loader, and is just sophisticated enough to load the actual system loader, which then loads the application and all of its dependencies. In this way, the `N00Bloader` is able to reserve the N00B-hardened memory region as early as possible, and can instruct the system loader to preload `N00Balloc`. We detail this process below.

Reserving the region. Program loaders are invoked in two ways: either explicitly by the user, with the application to load as the command line argument (CLI mode), or implicitly by the kernel, when specified as the ELF interpreter of an executable (interpreter mode). This distinction matters when reserving the N00B-hardened region, since it results in a different memory layout at startup. In CLI mode, only the program loader exists in memory at startup (static binary loaded by the kernel), and the loader should itself map the executable to load. We hard-code the loader's base above the N00B compatibility address with a linker directive, so it can reliably reserve the entire region immediately at startup.

In interpreter mode, the kernel maps both the specified ELF interpreter (the `N00Bloader`, in our case) *and* the invoked executable itself in memory already at startup. Since the executable is built independently, we cannot control its base address, so the kernel may place it inside the N00B region (subject to ASLR). This is problematic, since all global objects of the executable will incorrectly appear to have N00B-hardened addresses. We sidestep this by reducing every invocation to the CLI case: even when started as the interpreter, the `N00Bloader` simply reserves the N00B region (potentially overwriting the mapped executable) and explicitly invokes the system loader with the target executable as its command line argument. The system loader then maps it above the reserved region.

The N00B region reservation persists for the entire process, so no later mapping lands below the compatibility address, whether from the system loader, the program, or a direct `mmap` or `brk` call. We adapt `N00Balloc` to operate inside the reserved region (Section 3).

Preloading `N00Balloc`. The dynamic loader resolves each symbol to its first encountered definition, so depending on library traversal order it may bind `malloc`-related symbols to the C library instead of `N00Balloc`. We therefore *preload* `N00Balloc` for any application with a (transitive) N00B-hardened dependency, by passing the `preload` option to the system loader.

This requires editing the argument list the kernel left on the process stack, which is delicate because the same stack also holds the `N00Bloader`'s own call frames. The `N00Bloader` therefore splits the initial stack, keeping the argument, environment, and auxiliary-vector block in place while relocating its live frames to a fresh stack

outside the region and fixing up the affected pointers, before it finally hands control to the system dynamic linker. We also add auxiliary-vector variables that delineate the end of the reserved N00B region, so that N00Balloc can reliably stay inside this region.

3 Making N00Balloc a viable default

The N00Bloader makes N00Balloc the process-wide heap allocator, so that as many pointers as possible carry N00B-encoded bounds even when only a subset of the application is hardened. Since that subset may touch any object, minimizing the pointers left without bounds maximizes the security benefit.

As the default allocator, N00Balloc must be a reasonably fast, general-purpose, thread-safe allocator. The N00B prototype was not thread-safe, and not evaluated against existing allocators on dedicated benchmarks. We changed the implementation so each thread has its own thread-local N00Balloc instance, from which it can serve most allocation requests without synchronizing with other threads. We push objects freed by another thread to a separate atomic free list in the owning thread's allocator, which the owning thread reclaims in a single atomic exchange once its own freelist becomes empty, inspired by mimalloc's freelist sharding [7].

Finally, because the N00Bloader reserves the whole N00B region up front, the allocator cannot mmap fresh arena space. Instead, it activates pages from the reservation by manipulating their permissions using mprotect.

4 Evaluation

We evaluate on an Intel Core Ultra 9 285K with 64 GiB of DDR5-4800 and a Linux kernel with Intel LAM support, with hyper-threading, frequency scaling, and ASLR disabled.

Allocator benchmark. On mimalloc-bench [6], the rebuilt N00Balloc beats the original prototype on every single-threaded benchmark but one, and stays close to the glibc allocator across the multithreaded benchmarks (figures in poster).

Actix use case. As a realistic selective-hardening use case in a setting with modular dependency management, we build a minimal Rust Actix Web server that terminates TLS through OpenSSL [13] (written in C), and load-test it with wrk [3] fetching a static 4 KiB response over TLS (12 client threads, 120 connections, a 12-thread server, five runs averaged). We build a hardened OpenSSL shared library with N00B, ABI-compatible with native OpenSSL, and load it as the OpenSSL component when launching our Actix server through the N00Bloader. Compared to a fully unprotected version, our hardened version shows 2.7% lower throughput.

5 Discussion

This paper attempts to marry the process-wide restrictions of exploit mitigations with their selective application to modular software components, and finds that the program loader may be well-positioned to bridge the two. Still, open questions remain.

For one, how should the loader customization itself be communicated and distributed? A hardened library could declare a dependency on a specific (chain) loader, which the package manager then sets as ELF interpreter when installing any executable that transitively depends on it. Alternatively, if mainstream loaders gained plugin support, hardened ELF objects could request a plugin during ELF traversal via metadata. This would be a more general form of the way modern loaders detect and reconcile CFI-hardened and legacy objects [11], for which they all had to be patched.

Secondly, how and when could users choose a hardened version of a particular dependency? The security-overhead trade-off is highly context-dependent, so users may want per-dependency, per-execution control. This can be approximated today by prepending a directory of hardened libraries to LD_LIBRARY_PATH. Package maintainers could then provide both hardened and default builds.

Finally, we also see a similar modularity problem appearing for mitigation deployment at the container level. These bundle applications with their dependencies into a single container image, e.g. via Docker. To flexibly obtain hardened images, we want to explore whether hardened dependencies could be injected into otherwise unmodified containers, e.g., via layer rebasing and overlays [1], without continuously rebuilding fresh images.

Acknowledgments

This research is partially funded by the Internal Funds KU Leuven and by the Cybersecurity Research Program Flanders.

References

[1] Cloud Native Buildpacks Project. 2026. Cloud Native Buildpacks. https://buildpacks.io/.
[2] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. Association for Computing Machinery, New York, NY, USA, 132–142. doi:10.1145/2892208.2892212
[3] Will Glozer. 2026. wrk: a HTTP benchmarking tool. https://github.com/wg/wrk.
[4] Adriaan Jacobs, Carlo Ramponi, Jonas Roels, Bruno Crispo, Silviu Vlasceanu, Mahmoud Ammar, and Stijn Volckaert. 2026. N00B: Bounds Checking for the Masses. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26)*. Association for Computing Machinery, New York, NY, USA.
[5] Stephen Kell. 2021. Chain loading, not preloading: the dynamic linker as a virtualization vector. https://www.humprog.org/~stephen/blog/2021/01/04/.
[6] Daan Leijen. 2026. mimalloc-bench: Suite for benchmarking malloc implementations. https://github.com/daanx/mimalloc-bench.
[7] Daan Leijen, Benjamin Zorn, and Leonardo De Moura. 2019. Mimalloc: Free list sharding in action. In *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*. Springer, 244–265.
[8] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. 2022. Cross-Language Attacks.. In *NDSS*. 1–18.
[9] Matt Miller. 2019. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape, 2019. *BlueHatIL 2019* (2019). https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf
[10] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 245–258. doi:10.1145/1542476.1542504
[11] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '19)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. doi:10.1145/3337167.3337175
[12] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (SP)*. 48–62. doi:10.1109/SP.2013.13
[13] The OpenSSL Project. 2026. OpenSSL. https://openssl-library.org/.
[14] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. 2015. High system-code security with low overhead. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 866–879.