

---

# The Rise and Fall of Nim

By: \$t@\$h, QVLx Labs.    Date: 01.01.2024

---

Crackers sure do know how to ruin a good thing. And by good thing I mean Nimlang.

In the realm of programming languages, Nim stands out for its unique blend of efficiency, expressiveness, and flexibility. Developed to offer the best of both worlds: the speed and power of statically typed languages like C and the simplicity and readability of dynamically typed languages like Python. Nim has carved out a niche in the software development landscape. However, its strengths have also made it attractive to cybercriminals, leading to a surge in its use for malicious purposes. Lets look at Nim in-depth, examine its features, see why it's gaining popularity among crackers, and determine implications on security.

## *Nim's Language Design*


Nim's syntax and language design are key to understanding its appeal. It adopts a syntax that is highly readable while also allows for high degree of expressiveness. Readability is achieved through Python-like syntax, while static typing system adds to its performance efficiency.

The language supports multiple paradigms, including imperative, object-oriented, and functional programming, making it a versatile tool for a wide range of applications. Nim's design also emphasizes compile-time mechanisms, including a macro system.

The Nim compiler plays a central role in its performance. It converts Nim code into optimized C, C++, Objective C, or JavaScript, leveraging the mature ecosystems and optimizations of these languages. This approach allows Nim programs to achieve execution speeds comparable to C and C++, which is a significant advantage over interpreted languages.

The compiler has various performance optimizations, including but not limited to:

- Inline Expansion: Reducing function call overhead.
  - Dead Code Elimination: Removing code that does not affect the program.
  - Loop Unrolling: Increasing the efficiency of loops in certain contexts.
  - Constant Folding: Precomputing constant expressions at compile time.
-



Nim's ability to compile to multiple back-ends, including C, C++, and JavaScript, makes it a powerful tool for cross-platform development. This cross-compilation capability is crucial for writing software that can run on a wide range of hardware and software platforms. Moreover, Nim provides system-level access, which is essential for low-level programming. This includes direct access to memory and system calls, capabilities that are often leveraged in systems programming, embedded systems, and device drivers.

Nim's meta-programming capabilities are one of its most powerful features. The language's templates and macros enable developers to write highly abstract and complex code at compile-time. This feature is particularly useful for creating domain-specific languages and high-level abstractions.

In terms of concurrency, Nim provides several models, including message passing and shared memory approaches. These models allow for the efficient use of multi-core processors and are crucial in modern software development. Not unique to Nim though.


Memory safety is another critical aspect of Nim. While it offers garbage collection for ease of use, it also allows for manual memory management, giving developers the flexibility to optimize performance and control resource usage closely.

Nim's approach to memory management is a critical aspect of its design. It employs a unique garbage collection system that combines reference counting with a mark-and-sweep algorithm. This hybrid approach allows for efficient memory management, crucial in avoiding memory leaks and dangling pointers. This feature of Nim prevents common software vulnerabilities, which is good for humanity but also good for crackers trying to hide or preserve their malware. A double-edged sword we see with Rust as well. Additionally, Nim provides the option for manual memory management, giving developers more control over resource allocation and deallocation, which is particularly important in systems programming and high-performance applications.

Nim's type system is designed to ensure safety and robustness. It enforces strict type-checking at compile time, significantly reducing the risk of type-related errors at runtime. This is particularly important in the context of security, as many vulnerabilities arise from improper handling of types. Nim's support for generics and polymorphism further enhances its type safety, allowing for more flexible and reusable code while maintaining strict type adherence.

Nim's standard library is comprehensive, covering a wide range of functionalities, from basic string manipulation to complex network programming. This extensive standard library, along with third-party libraries available in Nim's package manager, Nimble, provides developers with a robust ecosystem for software development. This ecosystem is a double-edged sword, as it not only aids legitimate developers but also provides cyber attackers with a wealth of resources for crafting malicious software.

Nim's capabilities in network programming are of particular interest in the context of cyber attacks. The language provides powerful tools for network communication, including support for



HTTP, TCP, and UDP protocols. This makes it a useful tool for developing network-centric malware, such as botnets and command-and-control (C2) servers. The ease with which networked applications can be developed in Nim, combined with its performance efficiency, makes it an attractive choice for cyber attackers looking to deploy distributed attacks.

The Nim compiler's code generation techniques are a key factor in its efficiency. It employs advanced optimization strategies to generate highly efficient machine code. This includes aggressive inlining, loop optimizations, and sophisticated dead code elimination. These optimizations not only improve the performance of Nim programs but also make the analysis of compiled binaries more challenging, a feature that is exploited by cyber attackers to evade detection.


### *Nim in Cybersecurity: Offensive and Defensive Perspective*

The same features that make Nim an attractive choice for software developers also make it appealing for cyber attackers. Its ability to generate optimized, cross-platform code is particularly useful for creating malware that can operate on different systems. From an offensive perspective, Nim is used to develop malware that is harder to detect and analyze. The language's ability to evade traditional detection methods, such as signature-based antivirus software, makes it a potent tool for cyber attackers. Signature-based techniques need to catch up a bit to meet the growing threat.

The use of Nim in cyberattacks is a growing trend. Nim-based malware, such as NimzaLoader, demonstrates the language's capabilities in creating sophisticated cyber threats. These include advanced evasion techniques, the ability to execute complex attack chains, and the use of polymorphic and metamorphic code to avoid detection. Have seen several multi-stage attack campaigns where Nim was used to develop the initial payload, which then downloaded further Nim-based modules. These modules were designed to perform specific tasks, such as data exfiltration and lateral movement within the targeted network, demonstrating Nim's effectiveness in complex attack scenarios.

Cybersecurity researchers have identified an increasing number of malware samples written in Nim, exhibiting advanced evasion techniques. These techniques include polymorphic code generation, where the malware alters its code on each deployment to avoid signature-based detection, and the use of sophisticated encryption and obfuscation methods to conceal malicious payloads.

As Nim continues to be used in both legitimate software development and cyberattacks, the cybersecurity community faces the challenge of adapting to this dual-use technology. This involves developing new tools and strategies to detect and mitigate Nim-based threats, as well as educating developers and security professionals about the risks associated with the language.



In terms of defense, the rise of Nim-based malware presents new challenges. Traditional security tools are less effective against threats developed in Nim, requiring a shift to more advanced, behavior-based detection methods. Defending against Nim-based threats requires a multi-faceted approach. Security researchers are developing specialized tools and techniques for analyzing Nim binaries, including advanced static and dynamic analysis methods. Machine learning algorithms are also being employed to detect patterns indicative of Nim-based attacks. Additionally, there is a growing emphasis on educating cybersecurity professionals about the characteristics of Nim malware, enabling them to better identify and respond to such threats.

I've written a detector for Nim using ensemble learning as a starting point, but this work needs to be continued by larger companies with more resources. See QVLx detectors to get the idea.

As Nim continues to gain popularity, the challenge for the cybersecurity community is to balance the benefits of this powerful programming tool with the security risks it poses. This involves not only technical solutions but also a broader understanding of how programming languages like Nim can be used and misused. The future of Nim in the cybersecurity landscape will depend on the ongoing efforts to mitigate its misuse while harnessing its potential for positive innovation.

In conclusion, the emergence of Nim as a tool in cybercrime shows the evolving nature of cybersecurity threats. Nim's powerful features, while beneficial for legitimate software development, also make it an effective tool for cyber attackers. The cybersecurity community must continue to adapt to these changing threats, developing new strategies and tools to counter the malicious use of emerging technologies like Nim.

It will be interesting to see if crackers end up killing Nimlang for good. It picking up traction is welcomed, however, not if it's because it becomes the most widely-used language by bottom-feeders far and wide.