BlackBerry
Intelligent Security. Everywhere.

OLD DOGS NEW TRICKS: ATTACKERS ADOPT EXOTIC PROGRAMMING LANGUAGES

By the BlackBerry Research & Intelligence Team

An examination into the trend by threat actors and security researchers alike of leveraging new and uncommon programming languages to evade detection and hinder analysis.

CONTENTS

Foreword

> Malware authors are known for their ability to adapt and modify their skills and behaviors to take advantage of newer technologies. That tactic has multiple benefits from the development cycle and inherent lack of coverage from protective products. This paper will look into less prolific programming languages and their use in the malware space. It is critical that the industry and customers understand and keep tabs on these trends because they are only going to increase.
>
> – Eric Milam[1] , VP of Threat Research, BlackBerry

OVERVIEW

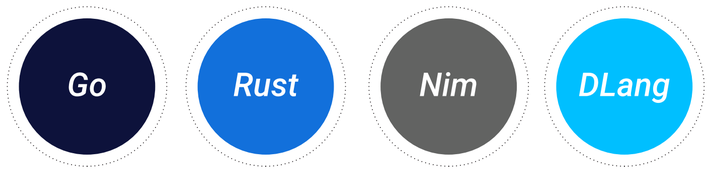
Malware authors have a reputation for being slow to change what works for them. But that is not always the case. Some malware groups have taken the opportunity to branch out and try new or "exotic" programming languages to address pain points in their development process or to try to evade detection by the security community.

The BlackBerry Research & Intelligence Team[2] chose four uncommon programming languages of interest to examine over the course of this work: Go, D, Nim and Rust. This choice was due in part to our detection methodology. We've identified an increase in their use for malicious intent, and we have seen an escalation in the number of malware families being identified and published that use these languages. These languages have also piqued our interest because they could be considered more developed and they have a strong community backing.

Although this trend is nothing new, BlackBerry aims to shed light on the state of the current threat landscape regarding these new and emerging languages. We'll cover both the reasons for their adoption and what areas we expect to see a further uptick in as this trend enters its next evolution.

And perhaps most importantly, we'll discuss ways both private individuals and corporations can address these growing risks.

FOUR UNCOMMON LANGUAGES OF INTEREST



INTRODUCTION

Technological advancements are one of the driving factors in modern society. New technologies can revolutionize lives, improve efficiency at an incredibly large scale and permanently alter the status quo of society. They also have the capacity to be misused by bad actors with ulterior motives or turned against the very purpose for which they were created.

For example, although the concept of email had been around since the advent of ARPANET in the 1970s, it didn't reach mainstream adoption until the explosion of the Internet in the mid-to-late 90s. With it came a deluge of email abuses such as the ILOVEYOU[3] computer worm in early 2000, which ran rampant and affected an estimated 10% of all Internet-connected computers at the time.

Though not exclusive to computer science, this trend of abusing new technology has been observed repeatedly with both new and uncommon programming languages. Even though the initial motivation for the creation of new programming languages is to achieve an improvement on existing languages and technologies, it is almost an eventuality that they will also be dissected by individuals or groups for malicious use. That could happen through security researchers creating a new proof of concept to help prevent future threats or a threat actor using the new language to develop a new malware variant.

From the use of Delphi and VB6 as a wrapper layer of malware, to a rewrite of the now-infamous BazarLoader[4] (named NimzaLoader) in the Nim programming language, we've seen history repeat itself. And we ask—why is this the case?

WHY USE UNCOMMON PROGRAMMING LANGUAGES?

WHY USE UNCOMMON PROGRAMMING LANGUAGES?

New languages are typically adopted as they improve upon a deficit in an existing language. Their creators could be in search of simpler syntax, performance boosts or more efficient memory management. Or the nature of the new language could better suit the environment in which it is to be used (for example, Internet of Things devices use lower-level languages such as C or assembly).

The user-friendly nature of some languages can also drastically improve both ease of development and the quality of life of the developer (for example, the pip package manager for Python or npm for Node.js).

But first, let's take a look at what got us to this point.

WHY USE UNCOMMON PROGRAMMING LANGUAGES?

The Old Guard

Delphi and VB6 have been prominent within the threat landscape since the early 2000s, when VB6 malware reached near-epidemic levels. As quoted on VirusBulletin[5], VB6 was well-known for being difficult to reverse engineer: "Visual Basic is widely considered to produce the most hated binaries in the history of reverse engineering – indeed, on mentioning this topic to some reverse engineers, they didn't know whether to laugh or to cry".

Although VB6 has dropped somewhat in popularity since its heyday of the 2000s, the Delphi programming language was still actively being used to pack and wrap commodity malware such as RemcosRAT and NanoCore until recently. This practice was mentioned in a FireEye report in 2018[6].

These languages have forged the path that newer languages now walk. History tends to repeat itself, and as such, we will be studying the latest evolution of this trend within this report.

Malware Analysis Tooling for the Uncommon Language

As we saw with VB6, certain languages can certainly hamper reverse engineering efforts. Malware analysis tooling does not always adequately support exotic programming languages. This failing can make analysis efforts a more tedious experience because the analyst must sift through unlabeled library code and rabbit-hole subroutines.

This challenge is often amplified when the binary is statically linked, where library routines are included within the binary by the linker, as opposed to being resolved dynamically during runtime. These library routines often have what appears to be garbled function names. This situation is due to the disassembler not being able to parse the language-specific metadata present within the binary or identifying language-specific string literals.

This garbling occurs with the languages of focus within this white paper, including Go, Rust, Nim and DLang. Binaries written in these languages can appear more complex, convoluted and tedious when disassembled, compared to their traditional counterparts based on C/C++/C#. This white paper will explore that trend in greater detail with respect to each language later.

In a similar vein, analysts could also be unfamiliar with the flow of execution of these new languages. There can be a steep learning curve to understand their intricacies or peculiarities. Malicious developers can abuse the analysts' lack of familiarity to make the task of reverse engineering more tedious (though not by any means impossible).

Thwarting Signature-Based Detection

Signature-based detection of malware depends on specific static characteristics being present within a file. These are qualities about the file that do not change and that do not require the file to be executed for someone to visualize them.

Hashes are an example of a static characteristic, which requires each byte to be identical within the target scope (that is, a hash of the whole file or a hash of a certificate, etc.). Signatures like YARA rules[7] have a set of static properties or characteristics named "conditions." Once they are met, the rule is fulfilled and can be seen to match or trigger.

When malware is authored in a new language, as opposed to what has been seen traditionally (for example, BazarLoader being rewritten in Nim), signatures written to detect the previous iteration will more than likely not match. New signatures will then have to be created to detect these variants. This signature creation is done either manually using human malware researchers or by using artificial intelligence (AI). This trend holds true with other languages and malware families as well.

WHY USE UNCOMMON PROGRAMMING LANGUAGES?

Additional Layers of Obfuscation

An argument could be made that in the case of more uncommon programming languages, the language itself acts as a layer of obfuscation. Each of these languages is relatively new and has little in the way of fully supported analysis tooling. As such, they can appear quite alien under the hood. It is because of their relative youth and obscurity that the languages themselves can have a similar effect to traditional obfuscation and be used to attempt to bypass conventional security measures and hinder analysis efforts.

We're seeing a growing number of loaders and droppers written in uncommon languages. These new first-stage pieces of malware are designed to decode, load and deploy commodity malware such as the Remcos and NanoCore Remote Access Trojans (RATs) as well as Cobalt Strike[8]. They have been used to help threat actors evade detection on the endpoint.

These complicating factors plus the languages' slow rate of adoption are largely why there are not many custom obfuscation techniques currently available for these languages in the threat landscape. That is not to say there are none currently available. One of the most prevalent among the languages studied within this paper is the "Gobfuscate" method. As the name suggests, this obfuscation is Go oriented.

A note from the developer describes how Gobfuscate works: "Gobfuscate manipulates package names, global variable and function names, type names, method names, and strings."

Gobfuscate has already been used in the wild in several Go-based malware variants such as Blackrota[9] and EKANS ransomware. It was also used in the recently unveiled ChaChi RAT[10] variant, which was uncovered in June 2021 by the BlackBerry Research & Intelligence Team.

Additional obfuscation methods include garble[11] for Go, denim[12] for Nim and obfstr[13] for Rust. We've not observed DLang-specific custom obfuscation methods in the wild yet.

WHY USE UNCOMMON PROGRAMMING LANGUAGES?

Malware and Software Engineering

Although they might be overlooked by the developer community, malware developers are at their core software engineers. We set out to uncover exactly what it is about these new languages that would entice software engineers to choose one over a more traditional programming language—no matter which side of the corporate fence they choose to sit on.

As we dug deeper, we discovered that each language has its own benefits and drawbacks for different scenarios: C is not object oriented, whereas C++ is. C++ is strongly typed, whereas Python isn't. Python is great for data science, but it is a less than ideal choice for devices with limited performance. In non-software-engineering terms, each language has areas of application where it excels and areas where it fails.

Nim, for example, can be compiled into several languages such as C, C++ and even JavaScript (yes, you read that right). DLang has many syntax improvements over C as well as being fully interoperable with (and syntactically similar to) C. Rust has very low overhead and is very efficient where performance is concerned, and Go is touted as C for the 21st century.

When choosing a language, a developer must weigh options such as the target environment, syntax, purpose and suitability of the language to the problem at hand. Furthermore, memory management, static vs. dynamic linking and codebase extensibility should all be major considerations as well as many others.

New languages often come with a higher degree of security consideration, offering features such as memory-safe programming by design. This functionality can protect the developer from introducing easily overlooked security holes that can result in memory-related bugs and vulnerabilities.

Why would threat actors be conscious of using these more secure languages, you might ask? Well, the answer is quite simple—they don't want to leave themselves open to exploitation. This problem was recently seen with EmoCrash[14], where security researcher James Quinn discovered that the infostealer malware Emotet[15] was vulnerable to a buffer overflow within the installation routine of the main binary. In doing so, Quinn developed EmoCrash to leverage this vulnerability and act as an Emotet "vaccine," preventing installation of the malware in the first place.

Additionally, the use of new languages can help to demonstrate that an individual, a development team or company is on the technological cutting edge. It shows that they are using the most modern, most efficient and most productive means of developing their products. However, doing so can come at a cost—be it financial or temporal.

Much like in the business world, developers with experience in these languages are hard to come by, and they can garner a higher salary. This requirement increases the overhead for such a project.

In a similar vein, training existing developers to write code in these languages can be a significant time investment. That is not always the case, which we'll discuss more later, but in a tight development pipeline, it can still cause a deficit. Within the threat landscape, these rules also apply, but there are still more reasons why security researchers and threat actors alike could benefit from using these uncommon languages.

We'll investigate these in the following pages.

WHY USE UNCOMMON PROGRAMMING LANGUAGES?

Cross-Compilation

According to the latest statistics[16] compiled by Statcounter, the operating system market share leader for the previous 12 months was Windows®, which holds 73.54% of the market. MacOS® follows with a 15.87% market share.

Modern-day organizations use a mixture of these two operating systems across departments for typical users' work. Back-end systems and infrastructure are often heavily centered on Linux® OS. Mobile devices have also seen an increase in use for work-related applications. This scenario presents attackers with a conundrum of sorts, having to potentially use different coding languages and different tools or malware to target the various operating systems depending on their target and goal.

In theory, cross-compilation provides attackers the option of authoring the same malware variant (containing the same or similar functionality) in one language and having it cross-compiled to target different architectures and operating systems. This approach would allow them to potentially cut down on the number of tools required to meet their goals and to widen the net of any malicious campaign.

This is not a new concept by any means. The infamous Adwind RAT[17] had the ability to target multiple operating systems due to it being written in platform-agnostic Java. The veritable plague of Mirai[18] botnet variants targeted a wide breadth of operating systems and architectures. In more recent times, there have been several instances of malware being written in Go and cross-compiled to be deployed in campaigns targeting various system types and organizations.

Another such example is the WellMess[19] malware that targeted Windows and Linux machines in mid-2020. This malware is believed to have been developed by APT29, aka Cozy Bear[20].

It is not only advanced persistent threat (APT) groups that have been following this trend. In January 2021, a new malware targeting cryptocurrency users dubbed ElectroRAT[21] appeared. Similar to WellMess, it was also developed in Go and was cross-compiled to target users of Windows, macOS and Linux with Trojanized applications as their means of infection vector.

Another such example is the WellMess variant that targeted Windows and Linux machines in mid-2020. This variant is believed to have been developed by APT29.

Security Software Detection

New developments in any industry generally lead to a wave of necessary changes and improvements in accompanying workflows and technologies. The cybersecurity sector is no different. The ever-changing threat landscape, along with the proliferation of malware written in what were once considered niche languages, means that security software vendors and developers must stay ahead of the curve or risk being overrun with new threats that they are unable to detect and mitigate.

This proactive approach for improving support of alternative or new technologies in the scope of cybersecurity requires running a cyclical workflow. It centers around repeated and efficient threat hunting for new entries, refining and testing new capabilities and then deploying product improvements to users.

This workflow is an expensive and technically difficult task because it largely depends on the quality and visibility of input data into the cycle. If the input data does not give insight to current problem areas, then it will be difficult to implement changes.

Vendors and developers must be clever because the pool for samples written in less common languages is small. Larger and more well-defined sample sets with wide feature coverage are critical for math model training[22]. They're also important for use in threat detection engines, the creation and testing of detection heuristics and the overall understanding of how threat actors are using these technologies. For example, a lackluster sample set can lead to higher trends of false positives and negatives. Insufficient test samples might make the effectiveness of generated detections less effective than expected.

Unfortunately, it is common to learn of deficiencies or holes in security after an incident occurs. This reactive approach works for most users in the general sense. But it comes with the notable expense paid by the first unprotected victim—the sacrificial lamb[23], as it were—that led to the discovery of the threat. The goal of any protective tooling is to reduce, ideally to zero, the number of successful attacks on the user-base being protected by such tools. Depending on a successful (or at least visible) breach or incident to start making improvements towards detection makes this method a non-starter.

LANGUAGE BREAKDOWNS

ONE TIMELINE TO BIND THEM ALL

Although there has been some notable malware written in Go, Rust, Nim and DLang since their inception, occurrences were few and far between. Most of what has been found was written in Go.

These uncommon programming languages are no longer as rarely used as once thought. Threat actors have begun to adopt them to rewrite known malware families or create tools for new malware sets.

Figure 1 is a timeline of some prominent examples of malware written in these languages throughout the last decade. This timeline illustrates the uptick in their usage, particularly that of Rust, Nim and D. It is worth noting that this is not an exhaustive list of malware families developed in these languages:

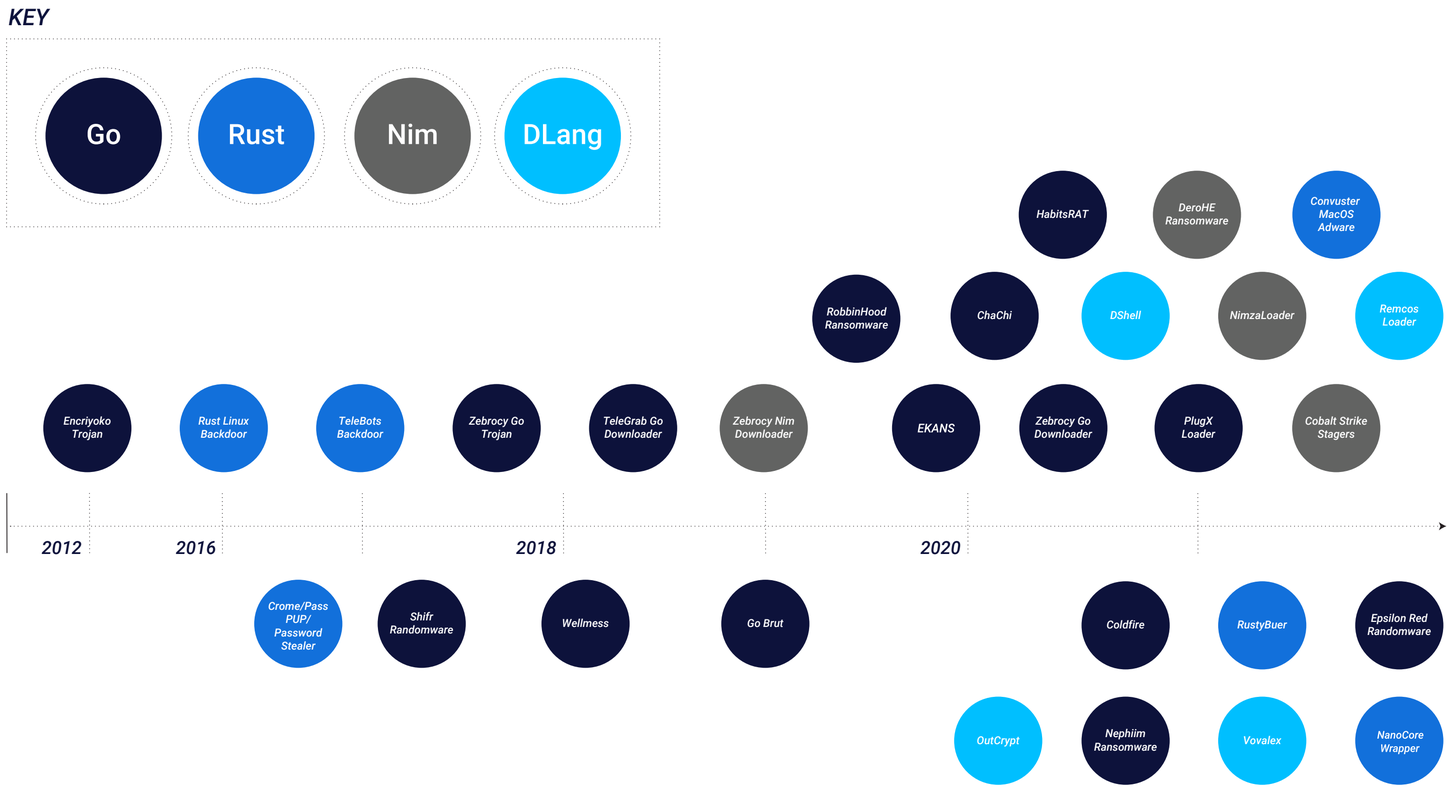


Figure 1: Timeline of prominent examples of malware written in the languages of Go, Rust, Nim and DLang.

DLANG

Overview

According to a note on the developer's website, "the general look of D is like C and C++. This makes it easier to learn and port code to D. Transitioning from C/C++ to D should feel natural. The programmer will not have to learn an entirely new way of doing things."

DLang, also known as D, first appeared in alpha form in 2001. Development on this language continued until the first stable release in January 2007.

Designed to be a multi-use programming language that follows a C-like syntax, it offers a performance level on par with C++. It aims to provide developers the means to author code quickly and efficiently, with a low learning curve. It's useful for a wide range of applications including web development, machine learning (ML), GUI applications, data analytics, kernel development and even AAA video game development.

Furthermore, it is possible to compile DLang code to target a variety of different architectures such as amd64, x86, PowerPC, AArch64, MIPS64 and Sparc. It can also be used on all major operating systems (OS) including Windows, Linux, macOS and even Android™ via various compiler supports.

As testament to its design and usefulness, DLang has recently been adopted by several industry titans[24] for various applications. This trend indicates it is likely we'll see further adoption and skillset development within the industry in the coming decade.

Major Features

DLang has a variety of qualities that make it appealing to malware authors:

- It has an easy learning curve.
- It can be cross-compiled to target various OSs and architectures.
- It's suitable for the building of lightweight and/or stand-aloneutilities.
- It includes multiple paradigm support, including object-orientated, structured and functional.
- It draws inspiration from C/C++.
- It's suitable for the development of a wide range of project and application types.

DLANG

The Current D Threat Landscape

Given the ease of use for C programmers, and the fact that DLang can be cross-compiled to target various OSs and architectures, this means that (in theory, at least) it is an ideal language to be abused for malice by threat actors.

At the time of writing, there have only been a handful of documented instances of DLang being used in the development of executables either by threat actors for malicious intent or by the security industry for use in Professional Services offerings. The first was a utility called “DShell” that was developed by FireEye for use in its red-teaming services.

DShell

The existence of DShell[25] was unwittingly unveiled to the public in the aftermath of a breach suffered by FireEye. A suspected but unnamed APT threat actor infiltrated its network in December of 2020, allowing FireEye’s tool to be placed into the hands of the criminal community.

As its name suggests, DShell is a DLang-compiled red-team tool that functions like a backdoor. It includes the ability to modify firewall rules, contains an encoded payload and can connect to a command and control (C2) channel.

Vovalex

The Vovalex[26] ransomware family first made an appearance in the wild in February of 2021. It is the first documented ransomware written in the DLang programming language.

Vovalex uses Trojanized versions of commonly used applications such as CCleaner as an infection vector. It typically drops and runs the installer of the Trojanized file so as not to arouse the victim’s suspicion. This Trojan runs as a sub-process of the installer, which gives the user the impression that everything is proceeding as expected.

In tandem, the Vovalex code begins its nefarious execution flow and searches the host for targeted files and directories. It then encrypts those files, appending a ".vovalex" extension to each one.

Vovalex is a relatively unsophisticated ransomware variant by today's standards. It doesn't appear to contain any of the functionalities that have become common in modern ransomware, including deleting shadow copies, terminating processes and services, spreading mechanisms or negatively impacting networking functionality. Despite its somewhat "vanilla" appearance, it can still cause significant damage to victims.

OutCrypt

The first mention of OutCrypt ransomware in the wild was in July 2020 in a tweet by the user @Amigo_A_. It was developed in DLang and dubbed "OutCrypt" due to its appending of the extension “. _out" to any encrypted files.

OutCrypt uses an unknown infection vector and has not been linked to any known attacks yet. It also does not appear to drop any ransom note or mention a ransom payment.

Upon execution, the malware begins to search through directories for files to encrypt. A copy of that file is made and then encrypted with an “. _out" file extension appended to it.

OutCrypt is unique in that during execution and subsequent file encryption, it lacks both common modern ransomware techniques and it fails to make ransom demands. It does not delete shadow copies or contain any networking functionality. It does not drop any ransom note or demand any ransom, and it does not contain or display any way to contact the attackers.

There are several references in the code to "testing," and it appeared in the wild under the name “dirtytest.exe,” which could point to it being a proof of concept or a variant under development. Furthermore, once OutCrypt is executed, it is possible to stop the execution of the malware by hitting the shortcut keys "Ctrl + C."

Nevertheless, an infection by this ransomware will leave a user's files unrecoverable. In this sense, it could be considered a destructor. The threat’s motive isn't financial, but to destroy or renders a user's files unusable, similar to the goals of a wiper malware.

DLANG

Interesting Samples from Our Own Hunting

RemcosRAT

First seen in the wild in 2016, RemcosRAT was developed and marketed by a German firm called BreakingSecurity as a Windows remote access utility. It is a relatively sophisticated RAT that has been widely abused in many campaigns since its release. It provides an attacker with an array of functionality to fully monitor and control any Windows OS from XP onwards, performing activities such as the following:

- Terminating processes
- Executing processes
- Opening a network connection
- Searching for files and directories
- Keylogging
- Activating a webcam
- Uploading and downloading additional files from the Internet
- Modifying existing files or folders
- Updating itself
- Playing or stopping audio

Earlier this year, the BlackBerry Research & Intelligence Team[27] uncovered the use of DLang as a wrapper and loader. It was being used to decode, load and deploy a RemcosRAT payload into the memory of the victim's computer.

As shown in Figure 2, a look at the strings from this threat reveals references to Phobos, the standard DLang runtime library, as well as the D compiler.

```
C:\D\dmd2\windows\bin\..\..\src\phobos\std\internal\cstring.d
C:\D\dmd2\windows\bin\..\..\src\phobos\std\utf.d
C:\D\dmd2\windows\bin\..\..\src\phobos\std\algorithm\iteration.d
C:\D\dmd2\windows\bin\..\..\src\phobos\std\algorithm\searching.d
C:\D\dmd2\windows\bin\..\..\src\phobos\std\array.d
C:\D\dmd2\windows\bin\..\..\src\phobos\std\format.d
C:\D\dmd2\windows\bin\..\..\src\phobos\std\conv.d
C:\D\dmd2\windows\bin\..\..\src\phobos\std\uni.d
C:\D\dmd2\windows\bin\..\..\src\phobos\std\stdio.d
```

Figure 2: References to Phobos revealed in strings from RemcosRAT.

RemcosRAT has been around for a while now and has appeared in numerous forms among numerous campaigns. This recent development could point to a potential trend where an obscure language such as DLang is being used to add a new "coat of paint" to an existing, powerful commodity malware. This strategy can result in a new threat being given a new lease on life, especially regarding attempts to bypass any existing detection mechanisms for it.

NIM

Overview

Nim is another notable language that is becoming increasingly common due to several features that make it stand out from the other options. Like more mature languages such as C, C++ and Java, Nim is statically typed and compiled. Andreas Rumpf began development in 2005 under the original “Nimrod” project name (Nim in Action[28]).

In 2008, version 0.6.0 of the project was published. This date marked the first release where the compiler was written and compiled in Nim rather than being developed in Pascal, as it had been in previous versions (see archived releases[29]). The current naming scheme of “Nim” was effective as of version 0.10.2[30].

Major Features

Nim was designed with the following three goals in mind:

Efficiency—Nim binaries are native and dependency-free, not requiring an underlying virtual machine or interpreter to convert and execute code. This setup leads to small and easily redistributable executables. Inspired by the likes of C++ and Rust, Nim offers deterministic memory management and compile-time memory safety checks for array bounding issues, overflows, null pointers and more to help ensure reliability.

Expressiveness—The language is designed to support multiple programming paradigms, including object-oriented programming. This design makes code reuse and metaprogramming easier and it allows for programs that can modify themselves at runtime. In addition, there is built-in support for binding to C, C++ and Objective C libraries with ease, allowing developers to make use of existing functionality already implemented in those languages.

Elegance—Nim's syntax is inspired by Python, Ada and Modula. It uses block indentation and allows for more human-readable code, creating a relatively low barrier to entry for new developers. Tracebacks are influenced by the Python implementation that contains useful information to aid in the debugging process.

Nim can be cross-compiled for all major operating systems such as Windows, Linux, BSD and macOS. These binaries can be statically or dynamically linked, depending on customization at the time of compilation. Nim also can produce JavaScript code, allowing for coalesced client and server development.

Like many other modern languages, Nim offers a built-in package manager, Nimble. It is designed to use Git and Mercurial repositories as package sources. This design helps with ease of access and supports package installation, publishing and configuration validation, among other functionalities.

NIM

Notable Malware

BazarLoader – NimzaLoader

In February 2021, threat actor TA800 distributed new malware in a phishing campaign. The samples were written in Nim, which inspired the name NimzaLoader (also known as Nimar Loader).

NimzaLoader is typically distributed through phishing emails that attempt to lure the user into clicking a link to a PDF, which downloads and executes the malware. Though the C2 servers for this campaign are no longer active, there is evidence that TA800 is using NimzaLoader primarily to download further malware such as Cobalt Strike as a secondary payload.

In the past, TA800 has been found predominantly using Trickbot[31] or BazarLoader (BazarBackdoor) in its attacks. It is uncertain whether NimzaLoader is a variant of BazarLoader, but there are enough distinct differences between the two that some analysts consider NimzaLoader to be in its own malware family.

Zebrocy & Nim Loader

Zebrocy is a family of malware first seen in 2015. It is usually delivered as an email attachment, and it targets embassies and ministries of foreign affairs in Eastern Europe and Central Asia.

Over the years, Zebrocy has been rewritten in several different programming languages. The first Nim downloader for it appeared in 2019. APT28 leverages a multi-language kill chain to enhance its detection evasion capabilities. The Zebrocy binary is written in Go, and since 2019, its downloader is written in Nim. APT28 has used uncommon languages repeatedly in its development processes.

DeroHE Ransomware

In January of 2021, the IObit forums were compromised and used to distribute a version of the DeroHE ransomware. Various forum users were emailed with offers for a free one-year subscription for IObit products. People who decided to click the link for the download, which was hosted on the compromised forum's site, would get a ZIP file with legitimate signed IObit files, plus one unsigned malicious DLL written in Nim as a parting gift.

Cobalt Strike

Cobalt Strike has become a popular tool among adversaries for command and control. As such, detecting and stopping the various loaders used to download Cobalt Strike beacons is a task many endpoint protection solutions have become very good at. Probably to counter the increasing effectiveness of these products, various loaders have been found that were written in Nim.

RUST

Overview

The Rust programming language project was started in 2006 by Graydon Hoare as a side project while working at Mozilla Research[32]. Within three years, Mozilla increased involvement and began sponsoring the project after it reached testing milestones and accomplished a level of maturity.

Since 2015, the Rust language has been an independent organization from Mozilla while still being a major sponsor and contributor. Over the years, Rust has grown and prospered, leading the language to be used in many major applications, such as Mozilla's own Firefox® web browser.

Other companies such as Microsoft, Amazon Web Services (AWS), Google, Facebook and Huawei are major sponsors and members of the Rust Foundation, which is a nonprofit organization that acts as a steward for the project[33]. As an example of vendor support, Microsoft has open-sourced and continues to support a Rust library[34] for interacting with the Windows API. This support allows for easy access to the functionality that Windows developers expect from more mature languages such as C++.

Despite being a relatively new programming language, Rust is a fan favorite among developers and has been voted as "most loved" in five Stack Overflow developer surveys, including the most recent 2020 edition[35]. Rust combines the power of low-level control with speed and memory efficiency, partially thanks to a lack of garbage collection. Garbage collection is centered around automatic memory management, present in languages like Java and Python, but it comes at a performance cost.

Rust also offers a notable improvement in memory safety over longer-lived languages such as C/C++. This feature has led to development efforts to rework existing portions of high-profile projects like the Linux kernel using Rust. Financial support from Google and the Internet Security Research Group has ensured development efforts go towards enhancing memory safety within the Linux ecosystem[36].

RUST

Major Features

Rust offers a way around some of the pain points common in other popular language choices. Python and other dynamically typed language developers will be all too familiar with "TypeErrors" during debugging. This is something that the statically typed variables in Rust avoid because these bugs are found during compile time. Developers can efficiently control memory usage and significantly increase performance[37] while not risking segmentation faults, use-after-free or buffer overflow situations that lead to errors and vulnerabilities.

According to Microsoft engineer Matt Miller, around 70%[38] of Windows' patched bugs over the last 12 years are memory safety-related issues. This trend has been shown in the modern threat landscape as well, where several high-profile incidents were caused by unsafe memory management within the affected application.

Critical vulnerabilities like CVE-2021-3156 that affects sudo (a *nix utility for privilege and account access management) allowed unprivileged users to escalate their account privileges through specific execution and arguments to the helper binary sudoedit. This vulnerability was due to a user-accessible heap overflow.

Additional safety constructs that Rust employs, such as the borrowing system, can lead to a drastic reduction in exploitation of services and tooling that continues to plague legacy and modern applications without these safeguards. In addition to increased protective benefits, the ownership model offers boosts to multiprocessing efficiency because it was developed with resource sharing and concurrency as a foundational goal.

The user and development experience for Rust receives high praise from the community[39]. Rust is installed and managed via Rustup, which is a single toolchain manager. It allows for language updates, release channel changes and (most notably) the ability to target other platforms and architectures for cross-compilation.

Rust has various levels of build support. Common x86, x64 and ARM architectures are fully supported along with others such as PowerPC, IBM Z/s390 and embedded-focused targets. Additionally, the Rust package manager, Cargo, resolves dependencies and invokes the build process to create Rust binaries.

Per the Cargo documentation[40], "it is only a slight exaggeration to say that once you know how to build one Cargo-based project, you know how to build all of them." This statement highlights the ease of use of the system. The Rust community is also strong, with a large presence of open-source and shared Rust libraries available at crates.io. There is also well-regarded documentation and many ways to find assistance from others.

RUST

Notable Malware

The features and capabilities of Rust that lead it to be popular in the community also mean that it was inevitable that threat actors would use the language to create malware as well. Threat actors have utilized Rust to develop new variants of existing malware, rewrite backdoors or loaders to add complexity to common malware and author entirely new malicious programs.

Convuster Adware

Convuster is Rust-based adware that targeted macOS systems in March 2021. Most malware seen on macOS systems is usually adware written in C, C++ or Swift. The exact method of how Convuster arrives on a device is unknown, but it is probably downloaded through other adware rather than directly by the user. Once on a victim's system, Convuster communicates with a (now inactive) server and uses built-in macOS tools to run.

RustyBuer

Buer is a malware loader originally written in C that was first seen being distributed through phishing campaigns in late 2019. The Buer Loader is often sold in underground marketplaces and used by malware as a service (MaaS) operators to download ransomware or Trojans.

Earlier this year, a new variation of the Buer Loader rewritten in Rust was found targeting more than 50 industry verticals. This new variant was dubbed RustyBuer.

Despite its rewrite in a new language, RustyBuer maintains compatibility with existing Buer backend C2 servers and panels. RustyBuer has been seen in more sophisticated phishing campaigns. Its rewrite into Rust makes it more likely to evade detection compared to its C-language predecessor.

TeleBots Downloader and Backdoor

TeleBots, believed to originate from a Russian threat actor, has been associated with attacks against Ukraine's critical infrastructure in the past. AlthoughTeleBots is usually seen using KillDisk malware, researchers were able to link new ransomware and updated tools—including a Rust Trojan downloader and backdoor—to the group in 2016 and 2017, respectively.

This group was seen distributing the Rust downloader through spearphishing emails[41] with attached Microsoft® Excel® documents that contain malicious macros. Once the user enables the macro, the Rust downloader is executed as the first stage of attack, which ultimately downloads a Python backdoor and KillDisk as the final stage.

In 2017, an enhanced version of the group's previously heavily used Python backdoor was also rewritten into Rust. The functionality of the newer Rust backdoor remains largely unchanged, and it receives commands from the TeleBot API like its predecessor. This setup points to the possibility that the rewrite was mainly performed to evade detection.

Early Linux Backdoor

In 2016, one of the earliest Rust malware samples was discovered by antivirus vendor Dr. Web[42]. This Linux backdoor using IRC was believed to be a proof of concept because the sample did not have the capability to spread to other victims and (at the time of the blog post) the associated IRC channel was not live. Dr. Web analysts noted the ease with which they could target other operating systems because Rust code can be cross-compiled for other OSs such as Windows and macOS.

RUST

Notable Malware

Interesting samples from our own hunting

The BlackBerry Research & Intelligence Team has found notable samples and frameworks that exemplify some of the recent developments pertaining to the Rust language and its use by malicious actors.

NanoCore Dropper

This Rust binary performs the important role of dropping an otherwise easily identified malicious binary onto a victim's device and initiating execution. There is nothing necessarily notable about the Rust binary other than the need for its existence, which shows that the hardest part of a malware campaign is sneaking past increasingly complex security systems.

As malware families such as NanoCore become more prevalent, the effectiveness of security software towards those targets will naturally increase. The modification of a particular layer of the execution sequence might be all that it takes to return a campaign to a functional state. There is little need to reengineer an entirely new approach if iterative changes lead to a longer term of success.

PyOxidizer

PyOxidizer[43] is another entry in a series of tools that attempt to make the package maintenance and distribution of Python code easier. It uses Rust to load and manage the execution of an embedded Python interpreter. The user does not need to install Python or sort out dependencies because the output of PyOxidizer is a Rust binary for the configured toolchain that contains all necessary components (including Python) bundled.

Threat actors commonly use other tools such as PyInstaller or Py2exe that offer native binaries to execute Python code on victim systems. PyOxidizer is another avenue that malicious actors unfortunately take to make the most of existing Python tooling wrapped in a package more unfamiliar to anti-malware software. In this case, that is the execution of Rust binaries.

One common downside for these types of utilities, including PyOxidizer, is that the resulting binary can be quite large. At times, they can be many megabytes long because the Python interpreter and any external dependencies must be packaged within the final binary.

Web Browser Credential Theft

Chromepass[44] is a utility that provides a Rust client designed to extract passwords and cookies from a victim's Chromium-based browser and communicate them back to an accompanying listening server. It is implemented with Python via PyInstaller. Rust does most of the heavy lifting on the client side, extracting the password and cookie data from the victim's browsers and sending it back to the attacker.

This tool advertises antivirus evasion as a core component, per the project README. This is another example of Rust being used to circumvent conventional defense strategies due to little support for less common build components.

GO

Overview

Go was developed by Google in 2007 by Rob Pike, Ken Thompson and Robert Griesemer. It was made public in 2009 and officially released in 2012. They sought to address the disconnect they saw between the older languages in use and the computing landscape reality. Their belief[45] was that "the problems introduced by multicore processors, networked systems, massive computation clusters and the web programming model were being worked around rather than addressed head-on."

Although it is obvious why Google wanted Go, many others have adopted it along the way, including Twitch, Uber, Docker and Soundcloud. Even though Go might not be the most "loved" language among developers, it does score a podium spot in "most wanted." The monolith that is Google, along with companies that work with it, could be a large factor as to why.

The Go website describes its purpose as "making it easy to build simple, reliable, and efficient software."

Simple—Go belongs to the C family, but with a more simplified syntax. This construct means that Go programs should be easier to read and learn than their C equivalents.

Reliable—Google has made a promise that there will be source-level compatibility for the language and a standard library across Go v1. For example, any code written in Go v1.1 need only be re-compiled rather than being re-written, for Go v1.16 or future versions.

Efficient—Go maintains C runtime efficiency and builds on compilation efficiency. Further, Go is fast. The secret is Goroutines[46], which are analogous to lightweight threads managed by the Go runtime. Go provides a set of APIs for concurrency that abstracts the developer away from many of the details and pitfalls.

Like the other languages mentioned above, Go can be cross-compiled to all major operating systems as well as Android, JavaScript and WebAssembly.

Software engineers and malware authors alike flocked to this language, not only for stylistic reasons. Google backing the project has also increased its popularity as well as the number of libraries that have been made available.

GO

Major Features

Go is an open-source, statically typed, compiled language. The compiler was originally written in C but was rewritten with Go in 2015. This approach qualifies the language as self-hosted (for example, the compiler is written in the same language as the language it is compiling).

Go has a syntax like C, but unlike C, it also offers garbage collection, structural typing and concurrency. And, like Rust, it also has memory safety. In Go, memory management is handled automatically at runtime, which helps to reduce common vulnerabilities caused by memory safety issues.

Go was designed for the era of computing marked by large, networked environments, where increases in core count were outpacing clock increases. The documentation page explicitly states, “It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language.” Go is in a sweet spot for usage complexity that exists somewhere between more mature compiled languages like C or C++ and interpreted languages like Python or Ruby.

Go uses an advanced package tool (apt)-like[47] package management system, which simplifies the installation of external packages and their dependencies by using the “go install” command. It also allows developers to easily publish their own packages publicly so they can be used by others.

Because Go statically links required modules, binaries tend to be very large. A simple “Hello World” weighs in at around 2 MB. This static linking has the added advantage of producing executables that are stand-alone and require no additional files from the running system, making distribution less complex.

Statically linking legitimate libraries can interfere with security software because the inclusion of the wide array of supporting library modules can make up most of the functionality (rather than the user code). Special considerations must be made in machine learning detection model training and heuristic development to avoid potential pitfalls that could lead to false negative or false positive convictions.

GO

Notable Malware

ElectroRAT

In the latter half of 2020 and into the early months of 2021, the cryptocurrency market was on a bull run spearheaded by Bitcoin. With this buzz came a spike in cryptocurrency-related scams designed to fleece investors of their holdings.

One such campaign was uncovered by researchers at Intezer in January 2021. This threat, called ElectroRAT, included bogus social media accounts, websites and a new malware RAT to tie it all together.

ElectroRAT was written in Go and arrived in the form of Trojanized versions of commonly used cryptocurrency-related applications. These applications were hosted on fake websites, which users were pointed to via advertisements and promotions on social media and online forums.

Its primary goal was to target and pilfer the victims' cryptocurrency wallet. But, like any good RAT, it also was capable of additional functionality such as screenshotting, keylogging, uploading and downloading files as well as executing commands from the victims' console.

ElectroRAT is an excellent example of a threat actor using one of the languages mentioned in this document to design and develop a previously unseen malware variant from the ground up, for a specific purpose or campaign. In this case, it was used to target cryptocurrency users and then it was cross-compiled to target all major desktop operating systems. This approach allowed it to maximize its target victim base, all while using one malware variant.

EKANS

EKANS or "Snake" is an obfuscated ransomware written in Go and unique for having specific industrial control system processes as its targets. It also distinguishes itself as being a rare instance where industrial operations are targeted not by a nation state, but by actors motivated by financial gain.

Zebrocy in Go

A rewrite of the Zebrocy payload in Go was first seen as early as 2018. Additionally, a Zebrocy downloader executable developed in Go was seen in the wild in October 2020. It was disguised as a Microsoft® Word document.

WellMess

WellMess is a family of malware, usually associated with APT29, seen targeting COVID-19 vaccine makers, among others. These implants are cross-compiled for PE and ELF, and they support HTTP, HTTPS and DNS communications. Newer variants support PowerShell capabilities once a connection is established.

Early Go Dropper/Ransomware

Generally considered the first sample of a Go malware Trojan, Encriyoko was first reported by Broadcom in late 2012. This threat attempts to masquerade as an Android rooting tool, "GalaxyNxRoot.exe." It drops two files: an information-stealing Trojan that exfiltrates system data to a remote location and a downloader that retrieves an encrypted file. The downloaded file is a ransomware that leverages the Blowfish algorithm to encrypt the victims' files.

GO

Go Analysis Roadblocks

As mentioned previously, due to their static linking, Go binaries are typically relatively large in size. Although this should be an easy file to digest, the sheer number of functions imported is high in comparison to the functionality of the binary seen in other languages.

A common anti-analysis method for Go is "Gobfuscation[48]," a tool that "manipulates package names, global variable and function names, type names, method names, and strings" by doing the following:

- Refactoring the GOPATH with the hashes of names.
- Hashing names of variables, structs, etc.
- Obfuscating strings by replacing them with functions.

The differences can be seen in Figure 3. The top screenshot is of an early ChaChi variant, and the bottom is a later variant after the threat actors implemented Gobfuscate.

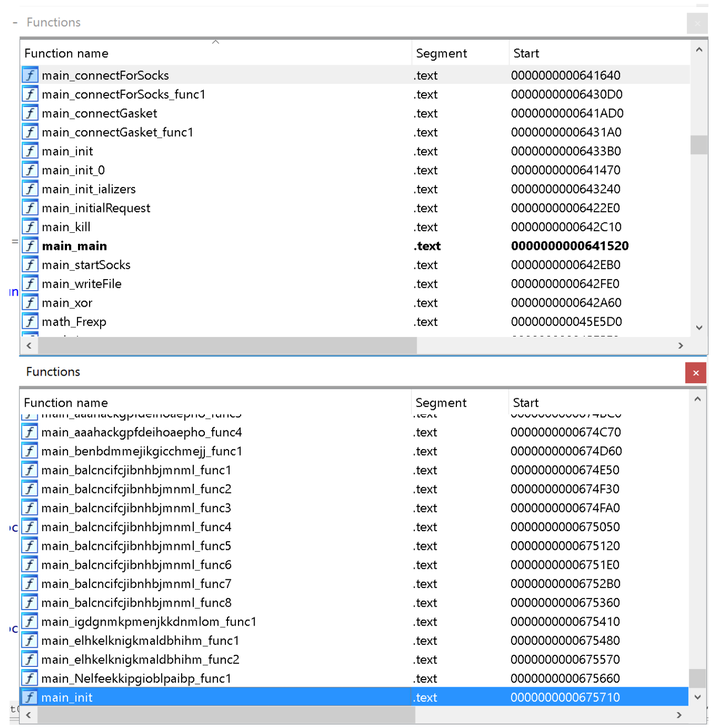


Figure 3: Top: Earlier ChaChi variant.
Bottom: Newer ChaChi variant with Gobfuscation implemented.

Although there are automated plug-ins for Binary Ninja[49] and Cutter[50], anyone using other tools like IDA Pro or Ghidra must perform the de-obfuscation process manually.

GO

Interesting Samples from Our Own Hunting

Cobalt Strike

Threat actors and adversary emulators alike have fallen in love with Cobalt Strike, making it perhaps the most popular framework in use today. Purpose-built to assist in infection, C2 operations and lateral movement, it is not surprising that payload beacons are being implemented in these more obscure languages. In our own hunting, a large portion of our resulting sample set turned up as positives for Cobalt Strike indicators. Go, with its robust support, cross-compatibility, and vast number of libraries, is currently an ideal choice for new offensive development efforts.

ChaChi

BlackBerry researchers have recently identified a RAT written in Go they named ChaChi[51]. The RAT is currently being used by the PYSA (Mespinoza) ransomware operators as part of their toolset. ChaChi is so named because of its use of Chashell and Chisel libraries, rather than writing fully custom components.

The ChaChi malware has been used to attack government authorities, healthcare organizations, educational institutions[52] and other private entities. The authors were very motivated to enhance the malware to stay ahead of detection, developing improvements in both code and tactic obfuscation along the way.

It also leverages the increasingly common “Gobfuscator” to muddle string, package and field names. Although ostensibly developed to prevent source code information disclosure, this tool has since been co-opted by malware creators to burden analysis efforts.

ChaChi had been active for over a year before discovery. The state of Go analysis, especially as it surrounds Gobfuscation, could have contributed to this delay.

DISASSEMBLY COMPARISON – HELLO, WORLD!

SIZE OVERVIEW FOR TEST “HELLO WORLD” SAMPLES

Development of software in various languages with the same workflow will result in notably different output binaries. A simple “Hello World” exercise in each of the languages discussed in this report leads to a wide array of differences in binary size and composition, build artifacts and metadata, and overall complexity. This disparity is true even without any additional layers of obfuscation or routines to raise the bar on analysis. As an additional note for comparison purposes, the same has been done for a C++ binary (see Table 1):

LANGUAGE	*"HELLO WORLD" BINARY SIZE*
D	877 KB
Nim	92 KB
Rust	438 KB
Go	2 MB
C++	51 KB

Table 1: “Hello World” binary size comparison between different programming languages.

SIZE OVERVIEW FOR TEST “HELLO WORLD” SAMPLES

The binaries were compiled in release configuration where applicable. We’ve done so to mimic a real-world use case, where analysts are not likely to be lucky enough to encounter malware with debug and symbol information.

As shown in Figures 4–19, all the test samples are using minimal code to write a string to STDOUT on the command line. The methods employed by each language to create the same terminal output led to very different outcomes in terms of the binaries.

Each exercise has the accompanying main function that performs the bulk of the user-defined code, along with a snippet of the total function list aggregated by IDA Pro v7.6. This process was done to show the variances between function naming, imported function counts, and the overall execution format.

This exercise shows that although the result of printing a string might be the same, analysis of these differing languages is not at all similar. They all have their own intricacies.

Note the variance in function count and naming schemes as well as the disassembly differences in programs that all accomplish the same output. In the real world, the scale of the code we encounter is far greater than a single print along with added layers of obfuscation to make analysis more complex and time-consuming for researchers.

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, world!\n";
    return 0;
```

Figure 4: C++ “Hello World” code contents.

```
; Segment type: Pure code
; Segment permissions: Read/Execute
__text segment para public 'CODE' use64
assume cs:__text
;org 100003B50h
assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing

; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public _main
_main proc near
push    rbp
mov     rbp, rsp
mov     rdi, cs:__ZNSt3__14coutE_ptr
lea     rsi, aHelloWorld ; "Hello, world!\n"
mov     edx, 0Eh
call    __ZNSt3__124__put_character_sequenceIcNS_11char_traitsIcEEEERNS_13basic_ostreamIT_T0_EES7_PKS4_m ;
xor     eax, eax
pop     rbp
retn
_main endp
```

Figure 5: C++ “Hello World” disassembly (main).

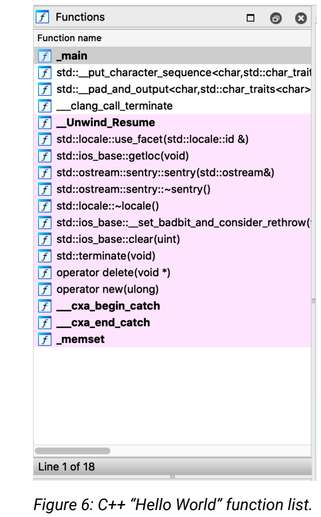


Figure 6: C++ “Hello World” function list.

SIZE OVERVIEW FOR TEST “HELLO WORLD” SAMPLES

```
import std.stdio;

void main()
{
    writeln("Hello, world!");
}
```

Figure 7: DLang “Hello World” code contents.

```
; Segment type: Pure code
; Segment permissions: Read/Execute
__text segment dword public 'CODE' use64
assume cs:__text
;org 100004354h
assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing

; Attributes: bp-based frame

public __Dmain
__Dmain proc near
; __unwind {
push    rbp
mov     rbp, rsp
lea     rdx, _TMP0      ; "Hello, world!"
mov     edi, 0Dh
mov     rsi, rdx
call    j___D3std5stdio__T7writelnTAyaZQnFNfQjZv
xor     eax, eax
pop     rbp
retn
; } // starts at 100004354
__Dmain endp
```

Figure 8: DLang “Hello World” disassembly (main).

Figure 9: DLang “Hello World” function list.

```
echo “Hello, world!”
```

Figure 10: Nim “Hello World” code contents.

```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public _main
_main proc near

var_8= qword ptr -8

push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     cs:_cmdLine, rsi
mov     cs:_cmdCount, edi
mov     cs:_gEnv, rdx
lea     rax, _PreMainInner
mov     [rbp+var_8], rax
call    _systemDatInit000
lea     rdi, [rbp+var_8]
call    _nimGC_setStackBottom
call    _systemInit000
mov     rax, [rbp+var_8]
call    rax
lea     rax, _NimMainInner
mov     [rbp+var_8], rax
lea     rdi, [rbp+var_8]
call    _nimGC_setStackBottom
mov     rax, [rbp+var_8]
call    rax
lea     rax, _nim_program_result
mov     eax, [rax]
add     rsp, 10h
pop     rbp
retn
_main endp
```

Figure 11: Nim “Hello World” disassembly (main).

```
; Attributes: bp-based frame

_NimMainInner proc near
push    rbp
mov     rbp, rsp
lea     rdi, _TM__xLHv575t3PG1lB5wK05Xqg_2
mov     esi, 1
pop     rbp
jmp     _echoBinSafe
_NimMainInner endp
```

Figure 12: Nim “Hello World” disassembly (inner main, echo call highlighted).

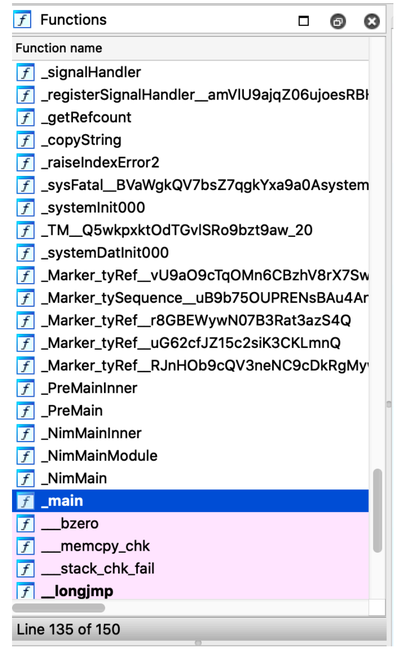


Figure 13: Nim “Hello World” function list.

SIZE OVERVIEW FOR TEST “HELLO WORLD” SAMPLES

```rust
fn main() {
    println!("Hello, world!");
}
```

Figure 14: Rust "Hello World" code contents.

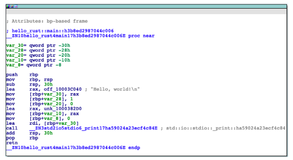


Figure 15: Rust “Hello World” disassembly (main).

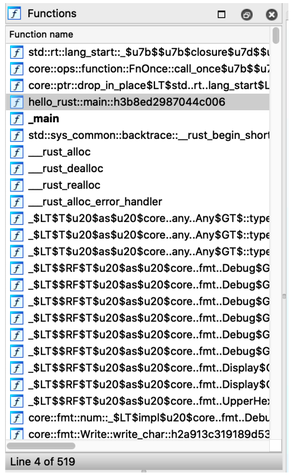


Figure 16: Rust “Hello World” function list.

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, world!")
}
```

Figure 17: Go “Hello World” code contents.

Figure 18: Go “Hello World” disassembly (main).

Figure 19: Go “Hello World” function list.

LANGUAGE ADOPTION BY THREAT ACTORS

LANGUAGE ADOPTION BY THREAT ACTORS

Although C-language malware is still the most widespread, threat actors such as APT28 and APT29 have been using these unconventional programming languages in their malware sets more often than other groups.

APT28 or Fancy Bear is a Russian state-sponsored group that has been operating since 2004. The group has frequently made headlines worldwide and is most notably known for allegedly hacking the United States' Democratic National Committee in an attempt to influence the 2016 presidential election. APT28 has been involved and associated with a wide range of attacks and malware families, but the Zebrocy malware family notably uses multiple uncommon programming languages within its kill chain.

The first sample of Zebrocy seen in 2015 was made up of three components: a Delphi downloader, an AutoIT downloader and a Delphi backdoor. Regardless of the programming language Zebrocy has been written in, the malware is spread through phishing campaigns that contain an initial Trojan that will attempt to communicate with a C2 server and execute a downloader to drop a malicious payload via an established backdoor. Though the malware has seen multiple rewrites and has evolved over time, the method of delivery via email attachment and general functionality remains largely the same.

In 2018, analysts linked a Go Trojan to APT28 and identified it as a rewritten version of the original Zebrocy Delphi downloader. In the years following, most recently in 2020, Go has proven to be an APT28 favorite because the other core components of Zebrocy—the backdoor payload and downloader—were also found rewritten into Go.

In 2019, a Nim downloader was found alongside the Go backdoor in the same Zebrocy campaign targeting embassies and ministries of foreign affairs in Eastern Europe and Central Asia. The group is still active and was last seen using the COVID-19 pandemic as a lure to deliver the Go downloader variant in late 2020.

Like APT28, APT29 (known as Cozy Bear) is also a Russian threat actor group found to be using Go in recent malware sets. The group is best known for its involvement in the SolarWinds compromise[53] in early 2020. APT29 was seen targeting Windows and Linux machines in 2018 with WellMess, a RAT written in Go and .NET.

The Go version of WellMess is the most prevalent and comes in both 32- and 64-bit variants as PE and ELF files, giving APT29 the ability to deploy it to more than one type of architecture and OS. The group typically gains access to a victim's network by performing vulnerability scans of an organization's external IP addresses and using public exploits against the vulnerable systems they encounter.

In 2020, APT29 was seen using a more sophisticated version of WellMess in attempts to steal information about COVID-19 vaccine development from multiple organizations located in the U.K., the U.S. and Canada. Although the newer variant is still written in Go, the threat group has added more complexity to the malware, including more network communication protocols and the ability to run PowerShell scripts post-infection.

Both threat actors are still active and have conducted some of the most impactful Russian cyberattacks to date. Recent activity suggests that these groups have been using the uncommon programming languages mentioned in this paper to add complexity to their malware, target multiple platforms and evade detection.

SECURITY COMMUNITY ADOPTION OF UNCOMMON LANGUAGES

SECURITY COMMUNITY ADOPTION OF UNCOMMON LANGUAGES

Developers and threat actors are not the only groups capitalizing on the popularity and benefits of these newer programming languages. In recent years, the security community has also adopted these languages and used them for their offensive advantages in implementations such as Red Team tools. Many of these tools are open-sourced or publicly available. They reference features such as cross-compilation and efficiency in their repositories as motives behind using these more uncommon languages.

In December 2020, FireEye reported that a sophisticated threat actor had gained unauthorized access to its Red Team tools. As a countermeasure, FireEye publicly released a statement along with a GitHub repository[54] containing detection signatures to help identify the stolen tools. In this repository, FireEye revealed that its Red Team had been using a combination of specially modified, publicly available tools as well as tools that were created in-house for the team. These were written in various languages including Go, DLang and Rust.

SECURITY COMMUNITY ADOPTION OF UNCOMMON LANGUAGES

DLang

Though these languages are now beginning to gain traction, DLang security tools are still uncommon. However, among the tools that were included in the disclosure, FireEye listed a backdoor named DShell written in DLang that was specifically developed for its Red Team.

Nim

Given that Nim is still relatively young in the world of programming languages, offensive tooling written in Nim is still a rarity. Currently, OffensiveNim[55] is one of the only major offensive toolsets available.

The GitHub repository for this tool contains detailed documentation, where its creator outlines their "experiments in weaponizing Nim" along with the reasoning behind selecting Nim as the language of choice. Several of the reasons included (such as cross-compilation and similarities to Python) overlap with the same features of Nim that appeal to both threat actors and developers.

Although OffensiveNim is not yet a complete framework like PowerShell Empire or Metasploit, the repository contains general offensive operations written in Nim and tips on how to use the tools. Additionally, the repository provides several ready-to-use examples, including the ability to run .NET code from memory, embed a ZIP file that is decompressed at runtime and various ways of running shellcode.

OffensiveNim appears to still be under development, with more examples currently listed as works-in-progress. It was last updated in June 2021.

SECURITY COMMUNITY ADOPTION OF UNCOMMON LANGUAGES

Rust

Although Rust is also new to the playing field, the language is slowly gaining popularity among the security community. It has been used in new offensive toolsets and spin-offs of preexisting popular tools such as DirBuster.

Matryoshka, another tool that was stolen in the FireEye breach, is a multi-stage Red Team tool written in Rust. This tool works by downloading a first-stage payload, running second-stage malware via a dropper and then installing the actual payload. Matryoshka utilizes process-hollowing to evade detection.

Feroxbuster[56] is described as a “simple, fast, recursive content discovery tool written in Rust” by its developer. The brute-force tool was combined with a wordlist to search for unlinked content in targeted directories. It is specifically designed to perform forced browsing, which is an attack where the aim is to search and access resources that are not referenced by the application but are still accessible.

Feroxbuster shares many similarities with other content discovery tools such as DirBuster and Gobuster. As evidence of the language’s increasing popularity, the tool was named Feroxbuster after the creator discovered that another Rust-written content discovery tool with the name they wanted already existed. Feroxbuster is updated frequently and was recently added to official Kali Linux repositories.

Go

Despite being the youngest language on our list, Go has been adopted widely by the infosec community, specifically by Red Teamers. Go has seen many Red Team tools rewritten or purpose-built just for it. Its speed and cross-compatibility are seen as huge pluses. There is a large spectrum of uses for offensive practitioners, from web brute-forcers to payload generators. Go-based tools are supported by large organizations for use in custom applications.

The FireEye Red Team tools disclosure showed that it had created a multi-platform Go RAT. Additionally, Go is leveraged for the Bishop Fox adversary emulation tool, Sliver. Further, Bishop Fox has put in the effort of forking another Go obfuscation tool called Garble, and it continues to add development effort for use in Sliver.

The popular C2 framework Merlin is completely written in Go for the purpose of being natively cross-platform. It’s clear that the security community sees value in the concurrency, efficiency and cross-compilation that Go offers.

CONCLUSIONS

OLD DOG – NEW TRICKS

Older malware written in traditional languages like C++ and C# is actively being given new life with droppers and loaders written in exotic languages, such as those mentioned in this work. Typically, the older malware will be stored in encrypted form within the first stage, using XOR, RC4, AES or other methods of encryption and encoding.

Once decoded, the binary is dropped to disk and executed (by a dropper) or injected into a running process and loaded into memory (by a loader). This is an attractive proposition for threat actors because they do not need to go to the lengthy effort of recoding the malware and instead can “wrap” it in one of these delivery methods.

Although we have seen Go used for these methods for some time now, we have observed this trend beginning to take effect with languages like D (with RemcosRAT), Rust (with NanoCore) and Nim (with Zebrocy and Cobalt Strike). Although wrappers and loaders are more cost-effective, some well-resourced threat actors are beginning to rewrite their existing malware using exotic languages. Examples of this trend are the switch from BazarLoader to NimzaLoader and from Buer to RustyBuer. The pseudonyms are used to track them by referencing the name of the language.

Existing signatures might have caught the second stage of a Dropper or Loader when using an existing well-known piece of malware, either when dropped to disk or loaded into memory. But these rewrites have the potential to hamper security solutions because existing static signatures will likely fail.

There might also be a less obvious reason for creation of these rewrites. When learning a new programming language, it is easier to recreate a solution you’ve built before. With an understanding of an existing solution, developers have only to concentrate on increasing their knowledge of the language, rather than battling the steeper learning curve of a new language and a new solution.

This approach could pave the way for malware developers with newfound experience in these languages to incorporate them as a more integral part of their toolkit.

DELPHI & VB6 – PASSING THE BATON

Although Delphi and VB6 helped to create the trend of using more uncommon languages to create malware, they have now in effect passed the baton to newer, more recent languages such as those mentioned within this report. As late as 2018, malware such as RemcosRAT and NanoCore were seen packed using a Delphi first stage. We have now observed these malware families being wrapped within the D and Rust languages.

There are several different factors responsible for this evolution. For example, developers skilled in Delphi/VB6 can be difficult to find. Newer languages bring general syntactical and quality-of-life improvements, not to mention there are fewer available detection capabilities for emerging technologies. Malware developed in Delphi or VB6 has not stopped entirely, however, with malware such as GuLoader[57] having been identified as recently as late 2019.

COBALT STRIKE AND SHELLCODE STAGERS – THE (NOT SO) NEW FRONTIER

Cobalt Strike has gained a high degree of infamy through its prevalence within the kill chain of many high-profile ransomware attacks and nation-state APTs. Cobalt Strike beacons are primarily used as a second-stage payload to facilitate lateral movement within the target network and to simulate the actions of an advanced adversary.

Cobalt Strike was developed with the intent of being used by security practitioners to strengthen their defenses against such advanced adversarial tactics. But pirated versions of the software (as well as the product source code) have been leaked online, meaning that it is no longer just security practitioners who now have access to this advanced tooling.

BlackBerry has seen a large uptick in use of initial stagers for Cobalt Strike being compiled using Go, and more recently in Nim. These initial stagers are the binary used to facilitate first-stage, initial access by reaching out to download the Cobalt Strike beacon from a TeamServer. This server is responsible for serving the beacons themselves.

It is important that defenders stay ahead of the curve in catching Cobalt Strike–related files written in these languages to enhance defensive capability against such a formidable threat.

DOES ADOPTION IN THE INDUSTRY MIRROR ADOPTION IN THE THREAT LANDSCAPE?

Since the dawn of computing itself, the success or failure of a new language depends upon its adoption within legitimate business.

A "thumbs up" from any industry titan can be significant to their adoption into the mainstream. As has been frequently observed with new programming languages and associated technologies, the rest of industry tends to follow where the industry titans lead.

This is not always the case, however. Many cutting-edge startups leverage new technologies that can (at least eventually) inversely influence market leaders.

Malware developers also contribute, inadvertently, to the growing trend. Being the first to break ground by pioneering a product (in this case, a new malware variant) in new and uncommon languages can be just as much of a goal and an ambition to a threat group as it would be to a legitimate business. This achievement can mean a greater level of kudos and reputation gain for developers, regardless of the color of the hat that they wear.

Another aspect to consider is that analysis tools and techniques are typically not developed by the security industry until there is a certain level of saturation of malware being written in a new language. Even if a language begins to pick up adoption within the business world, it can take time for the analysis tooling to reach a point where they are able to process these new languages in an adequate fashion, if they ever do—with VB6 and Delphi being a case in point.

BlackBerry findings show that DLang malware appears to be the least adopted language within the threat landscape, despite its adoption by several industry players over the last few years. DLang has seen an uptick in use in 2020 through mid-2021 in terms of the development of several types of malware. This could be the beginning of a new trend of DLang adoption within the threat landscape. However, it's important to note that correlation is not equal to causation.

GO IS BECOMING A “GO TO” INSTEAD OF A “GO WHERE?”

Based on research and trends within the current threat landscape, it appears that Go has matured to the point where it is now one of the "Go-to" languages for threat actors. This popularity is both at the APT and commodity level for the development of malware variants.

This assumption is based upon the fact that new Go-based samples are now appearing on a semi-regular basis, including malware of all types, and targeting all major operating systems across multiple campaigns.

Additionally, Go is a high-performance and well-liked language among developers, with a recent survey conducted by golang.org finding "ninety-one percent of participants[58] specifying that they would prefer to use Go for their next new project and eighty-nine percent specifying that Go is satisfactory for their current team."

More code means more analysis for a security researcher and/or an antivirus product. This requirement can mean that Go-based malware is generally a more arduous and time-consuming analysis proposition than a C- or C++-based sample. Furthermore, the first Go-based custom obfuscators have started appearing, such as "Gobfuscate" and "Garble," which add additional complexity to the task and overall make Go a very enticing prospect to threat actors of all levels.

These custom obfuscation techniques are also being actively leveraged by threat actors in the development and deployment of new Go-based malware variants, as the BlackBerry Research & Intelligence Team recently unveiled with its discovery of the ChaChi RAT variant.

A large number of Go-based malware variants have been documented in the wild, from APT-level samples such as APT28's Zebrocy loader, ransomware such as Epsilon Red and Snake, to previously undocumented variants such as ChaChi. The Go threat landscape has never been more active and is likely to continue on a similar trajectory in the future.

THREAT HUNTING EFFICIENCY THROUGH SMALL SAMPLE SETS

Steve Miller[59], former digital forensics researcher for the U.S. Department of Homeland Security, has hit on a point that is worth emphasizing[60]. In an Internet full of malware samples, uncovering the "who" behind a piece of malware can be like finding needles in a stack of needles. The act of hiding in a crowd is a technique that has been used by attackers to impede researchers and analysts alike since the dawn of computing. They often do so through the inclusion of benign, open-source code to conceal themselves among the rushes.

As we've described, these languages can come with several improvements once they're adopted into the software development lifecycle of a threat actor. Although this trend might sound bad for researchers, the inverse is also true. By using these languages for enhanced detection evasion, or for quality-of-life improvements, they also inadvertently aid us in our hunt for malicious samples.

Due to the relatively low number of compiled binaries in these languages, it is arguably easier to identify malicious samples. Thus, the needle-stack, as we could affectionately refer to it, is drastically reduced in size.

As the adoption of these languages increases, the needle-stack will too. Now is the time that researchers should look to make hay. After all, it's summer, and the sun is shining. Or, to paraphrase Steve Miller, it's time to flex[63].

IS DYNAMIC ANALYSIS MORE EFFECTIVE FOR THESE THREATS?

As stated previously, signatures for existing malware families that are based off static properties have little success in tagging the same malware once rewritten in these more obscure languages. In situations such as Buer and RustyBuer (as well as BazarLoader and NimzaLoader), new rules usually must be created to tag these tangentially related variants.

So, if static signatures are being broken each time a malware family is rewritten, is there much we can do to tag them?

We have a greater chance at catching these multi-language malware families using dynamic or behavioral signatures, signatures that tag behavior via sandbox output, or EDR or log data. These techniques can be far more reliable in such instances.

Although the codebase could be ported over to this new language and thus break the static indicators, the actions of the malware can often stay the same. This is especially true in situations where the malware is re-coded. In other circumstances such as shellcode loaders, which often inject into processes using a limited subset of Windows API calls, they can be identified using that limited subset.

The languages investigated in this report have bindings that allow them to interface with the Win32 API and use these API calls. In essence, they can use an almost-identical methodology to that of more traditional languages such as C++. Particular languages can use their own APIs in place of Win32 APIs. For example, they could use cryptographic libraries that would restrict the visibility of certain events. However, the use of these libraries within a binary can often be “signaturized” too.

By taking a step back from the implementation and looking at the core concept of how these pieces of malware interact with the system, threat researchers and software engineers alike can create more implementation-agnostic detection rules to be able to tag these dynamic behaviors if static signatures fail.

Dynamic signatures do not trump their static ilk by any means. Both are now necessary to have a comprehensive detection capability on the endpoint and beyond, and they should be used accordingly.

FINAL THOUGHTS

This report is intended to add new insight to the existing work of the security community on the topic of less-common programming languages and their application in malicious software and threat actor campaigns. It is important for defenders to further the discussion on the risk and effects of not defending against parts of the threat landscape that could seem obscure.

Malicious binaries written in languages like D, Rust, Go or Nim currently comprise a small percentage of the languages being used by bad actors in the world today. However, it is imperative that the security community stay proactive in defending against the malicious use of emerging technologies and techniques.

Programs written using the same malicious techniques but in a new language are not usually detected at the same rate as those written in a more mature language. The loaders, droppers and wrappers previous discussed are in many cases simply altering the first stage of the infection process rather than changing the core components of the campaign. This is the latest in threat actors moving the line just outside of the range of security software in a way that might not trigger on later stages of the original campaign.

This discrepancy in detections can be attributed to many factors. A smaller sample set for product testing, training and improvement along with a lack of supporting tooling are part of the equation. Many features that analysts and researchers have come to enjoy, and at times rely on for binary analysis, are simply not available during the early stages of a language's adoption (see Figure 20.)

The limited use of these more modern technologies in comparison to more mature workflows does not lend itself to an outpouring of market support, but these threats are active and continue to have a very real impact.

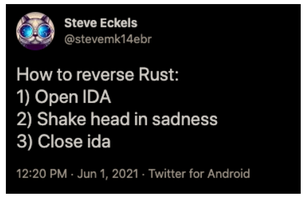


Figure 20: A comical take on reversing Rust binaries in IDA. (Source: @stevemk14ebr).

YARA RULE RELEASE

The following Yara rules were authored by the BlackBerry Research & Intelligence Team to catch the threats described in this document:

Mal_Infostealer_RemcosRAT

```
import "pe"

import "math"

import "hash"

rule Mal_InfoStealer_RemcosRAT

{

  meta:

    description = "Dlang wrapped RemcosRAT"

    author = "Blackberry Threat Research & Intelligence"

    strings:

    $f0 = {48 3A 2F 50 72 75 65 62 61 73 2F 43}

    $f1 = {43 43 52 59 50 54 45 52 42 4C 41 55}

    $DLang_Str1 = "C:\\D\\dmd2\\windows\\bin\\..\\..\\src\\phobos\\std\\utf.d" ascii wide

    $DLang_Str2 = "C:\\D\\dmd2\\windows\\bin\\..\\..\\src\\phobos\\std\\file.d" ascii wide

    $DLang_Str3 = "C:\\D\\dmd2\\windows\\bin\\..\\..\\src\\phobos\\std\\format.d" ascii wide

    $DLang_Str4 = "C:\\D\\dmd2\\windows\\bin\\..\\..\\src\\phobos\\std\\base64.d" ascii wide

    $DLang_Str5 = "C:\\D\\dmd2\\windows\\bin\\..\\..\\src\\phobos\\std\\stdio.d" ascii wide

condition:

// Must be MZ file

uint16(0) == 0x5a4d and

// Must be less than

filesize < 700KB and

// Must have exact import hash

pe.imphash() == "06f23da70e8da5f1231dae542708d4b9" and

      // Must have Strings

      all of ($f*) and 3 of ($DLang_Str*)   }
```

YARA RULE RELEASE

Mal_Ransom_OutCrypt

```
import "pe"

import "math"

import "hash"

rule Mal_Ransom_OutCrypt

{

  meta:

    description = "OutCrypt Ransomware"

    author = "Blackberry Threat Research & Intelligence"

  strings:

    $f0 = {B9 E0 79 46 00 B8 2A 00 00 00}

    $f1 = {BB 20 7A 46 00}

    $f2 = {B9 90 79 46 00 51 6A 13 FF 75 24 FF 75 20 BA 50 A7 46 00 52 E8 66 DA 00 00 83 C4 14 52 50 E8 68 19 00 00 8D 45 A8 E8 64 CD 00 00
8D 45 B4 E8 5C CD 00 00 C7 45 FC 01 00 00 00 8D 8D F4 FF FF FF 6A 01 51 68 90 70 46 00 E8 2A DA 00 00 83 C4 0C E8 02 00 00 00 EB 10}

    $f3 = {BA D0 9B 46 00}

    $f4 = "HESOYAMAEZAKMIRIPAZHAHESOYAMAEZAKMIRIPAZHA" ascii wide

  condition:

  // Must be MZ file

  uint16(0) == 0x5a4d and

  // Must be less than

  filesize < 700KB and

  // Must have exact import hash

  pe.imphash() == "a584e0e9fb9f4fbc415a1ef3c40e8812" and

      // Must have Strings

      all of ($f*)

}
```

YARA RULE RELEASE

Mal_Ransom_Vovalex

```
import "pe"

import "math"

import "hash"

rule Mal_Ransom_Vovalex

{

   meta:

      description = "Vovalex Ransomware"

      author = "Blackberry Threat Research & Intelligence"

   strings:

      $f0 = {52 45 41 44 4D 45 2E 56 4F 56 41 4C}

      $f1 = {6E 6F 74 65 70 61 64 00}

      $rans_note1 = "Send us a mail with proofs of transaction: VovanAndLexus@cock.li" ascii

      $rans_note2 = "README.VOVALEX.txt" ascii

      $rans_note3 = "VovanAndLexus@cock.li" ascii

      $rans_note4 = "Monero: 4B45W7V1sJAZBnPSnvcipa5k7BRyC4w8GCTfQCUL2XRx5CFzG3iJtEk2kqEvFbF7FagEafRYFfQ6FJnZmep5TsnrSfxpMkS" ascii

     $rans_note5 = "Send 0.5 XMR to this Monero wallet: 4B45W7V1sJAZBnPSnvcipa5k7BRyC4w8GCTfQCUL2XRx5CFzG3iJtEk2kqEvFbF7FagEafRYFfQ6FJnZmep5TsnrSfxpMkS" ascii

   condition:

   // Must be a 64-bit executable

   pe.is_64bit() and

         // Must have Strings

          all of ($f*) and 4 of ($rans_note*)

}
```

YARA RULE RELEASE

Mal_ShellcodeLoader_Go

```
rule Mal_ShellcodeLoader_Go

{

        meta:

                author          = "Blackberry Threat Research & Intelligence"

                description = "Tags Go Specific build tags and the presence of shell code headers"

        strings:

        $Go1 = "go.buildid" ascii wide

         $Go2 = "Go build ID:" ascii wide

                $shellcode_fiber_header_x86 = {fc e8 (89|82) 00 00 00 60 89 e5 31 d2}

                   $shellcode_fiber_header_x64 = {fc 48 83 e4 f0 e8 (c0|cc) 00 00 00}

        condition:

                uint16(0) == 0x5a4d

                and ($Go1 or $Go2)

                and ($shellcode_fiber_header_x86 or $shellcode_fiber_header_x64)

}
```

YARA RULE RELEASE

Mal_ShellcodeLoader_Nim

```
rule Mal_ShellcodeLoader_Nim

{

        meta:

                author          = "Blackberry Threat Research & Intelligence"

                description = "Tags Nim Specific function name and either shellcode headers or the presence of the string shellcode"

        strings:

                $nim_outOfMemHook = {6F75744F664D656D486F6F6B5F5F6B5A4E61413775314D665357355A656F47767738786700}

                $shellcode_fiber_header_x86 = {fc e8 (89|82) 00 00 00 60 89 e5 31 d2}

                   $shellcode_fiber_header_x64 = {fc 48 83 e4 f0 e8 (c0|cc) 00 00 00}

                   $shellcode = "shellcode" nocase

condition:

                uint16(0) == 0x5a4d

                and $nim_outOfMemHook

                and (

      ($shellcode_fiber_header_x86 or $shellcode_fiber_header_x64)

or $shellcode)

}
```

Endnotes

1 Milam, Eric. (2021, July 6). About Eric Milam. Retrieved from LinkedIn: https://www.linkedin.com/in/eric-milam/

2 BlackBerry Research & Intelligence Team, The. (2021, July 6). About The BlackBerry Research & Intelligence Team. Retrieved from BlackBerry: https://blogs.blackberry.com/en/author/the-blackberry-research-and-intelligence-team

3 Wikipedia. (2021, July 6). ILOVEYOU computer worm. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/ILOVEYOU

4 Bleeping Computer. (2020, Oct 12). BazarLoader used to deploy Ryuk ransomware on high-value targets. Retrieved from Bleeping Computer: https://www.bleepingcomputer.com/news/security/bazarloader-used-to-deploy-ryuk-ransomware-on-high value-targets/

5 Marschalek, Marion. (2014, July 10). Not old enough to be forgotten: the new chic of Visual Basic 6. Retrieved from Virus Bulletin: https://www.virusbulletin.com/virusbulletin/2014/07/not-old-enough-be-forgotten-new-chic-visual-basic-6

6 Muhammad, I., Ahmed, S., Vaish, A. (2018, Sept 20). Increased Use of a Delphi Packer to Evade Malware Classification. Retrieved from FireEye: https://www.fireeye.com/blog/threat-research/2018/09/increased-use-of-delphi-packer-to-evade-malware-classification.html

7 Wikipedia. (2021, July 6). YARA. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/YARA

8 Buber, Zohar. (2020, Nov 18). How to Identify Cobalt Strike on Your Network. Retrieved from Dark Reading: https://www.darkreading.com/threat-intelligence/how-to-identify-cobalt-strike-on-your-network/a/d-id/1339357

9 Yu, J. (2020, Nov 24). Blackrota, a heavily obfuscated backdoor written in Go. Retrieved from Netlab: https://blog.netlab.360.com/blackrota-a-heavily-obfuscated-backdoor-written-in-go/

10 BlackBerry. (2021, June 23). PYSA Loves ChaChi: a New GoLang RAT. Retrieved from BlackBerry: https://blogs.blackberry.com/en/2021/06/pysa-loves-chachi-a-new-golang-rat

11 GitHub. (2021, July 6). Burrowers/Garble. Retrieved from GitHub: https://github.com/burrowers/garble

12 GitHub. (2021, July 6). Moloch--/Denim. Retrieved from GitHub: https://github.com/moloch--/denim

13 GitHub. (2021, July 6). CasualX/Obfstr. Retrieved from GitHub: https://github.com/CasualX/obfstr

14 Quinn, J. (2020, Aug 14). EmoCrash: Exploiting A Vulnerability In Emotet Malware For Defense. Retrieved from Binary Defense Blog: https://www.binarydefense.com/emocrash-exploiting-a-vulnerability-in-emotet-malware-for-defense/

15 Cylance Team, The. (2017, Dec 12). Cylance vs. Emotet Infostealer Malware. Retrieved from BlackBerry: https://blogs.blackberry.com/en/2017/12/cylance-vs-emotet-infostealer-malware

16 GlobalStats, Statcounter. (2021, June). Operating System Market Share Worldwide. Retrieved from Statcounter: https://gs.statcounter.com/os-market-share

17 Gatlan, Sergiu. (2019, Oct 1). New Adwind RAT Variant Used Against the US Petroleum Sector. Retrieved from Bleeping Computer: https://www.bleepingcomputer.com/news/security/new-adwind-rat-variant-used-against-the-us-petroleum-sector/

18 Crawley, Kim. (2019, Aug 27). Mirai Botnet Spawns Echobot Malware. Retrieved from BlackBerry: https://blogs.blackberry.com/en/2019/08/mirai-botnet-spawns-echobot-malware

19 (2018, July 6). Malware "WellMess" Targeting Linux and Windows. Retrieved from JPCERT https://blogs.jpcert.or.jp/en/2018/07/malware-wellmes-9b78.html

20 Wikipedia. (2021, July 6). Cozy Bear. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Cozy_Bear

21 Mechtinger, A. (2021, Jan 5). Operation ElectroRAT: Attacker Creates Fake Companies to Drain Your Crypto Wallets. Retrieved from INTEZER: https://www.intezer.com/blog/research/operation-electrorat-attacker-creates-fake-companies-to-drain-your-crypto-wallets/

22 Cylance Team, The. (2015, April). MATH VS. MALWARE – A Cylance Whitepaper. Retrieved from https://1c7qp243xy9g1qeffp1k1nvo-wpengine.netdna-ssl.com/wp-content/uploads/2015/04/Math-Vs-Malware-White-Paper.pdf

23 Skipper, Chad. (2016, July 01). No More Sacrificial Lambs. Retrieved from BlackBerry: https://blogs.blackberry.com/en/2016/07/no-more-sacrificial-lambs

24 D Language Foundation, The. (2021, July 8). Areas of D usage. Retrieved from DLang.org: https://dlang.org/areas-of-d-usage.html

25 Mandia, K. (2020, December 8). FireEye Shares Details of Recent Cyber Attack, Actions to Protect Community. Retrieved from FireEye: https://www.fireeye.com/blog/products-and-services/2020/12/fireeye-shares-details-of-recent-cyber-attack-actions-to-protect-community.html

26 Abrams, L. (2021, January 29). Vovalex is likely the first ransomware written in D. Retrieved from Bleeping Computer: https://www.bleepingcomputer.com/news/security/vovalex-is-likely-the-first-ransomware-written-in-d/

27 BlackBerry Research & Intelligence Team, The. (2021, July 6). About The BlackBerry Research & Intelligence Team. Retrieved from BlackBerry: https://blogs.blackberry.com/en/author/the-blackberry-research-and-intelligence-team

28 hPicheta, D. (2021). Nim in Action. Retrieved from: Mannings Publication, Co. https://livebook.manning.com/book/nim-in-action/chapter-1/5

29 Wayback Machine, The. (2016, June 23). Launching the 2016 Nim community survey. Retrieved from Nim Lang: https://web.archive.org/web/20160626002904/http:/nim-lang.org/news.html

30 Picheta, D. (2014, Dec 29). Version 0.10.2 released. https://nim-lang.org/blog/2014/12/29/version-0102-released.html

31 BlackBerry Cylance Threat Research Team, The. (2019, Sept 10). Threat Spotlight: TrickBot Infostealer Malware. Retrieved from BlackBerry: https://blogs.blackberry.com/en/2019/09/blackberry-cylance-vs-trickbot-infostealer-malware

32 Hoare, G. (2021, July 6). Graydon/Rust-prehistory. Retrieved from GitHub: https://github.com/graydon/rust-prehistory/commit/b0fd440798ab3cfb05c60a1a1bd2894e1618479e

33 Rust Foundation. (2021, July 6). Members. Retrieved from Rust Foundation https://foundation.rust-lang.org/members/

34 GitHub. (2021, July 6). Microsoft/Windows-rs. Retrieved from GitHub: https://github.com/microsoft/windows-rs%20)

35 Popper, B. (2020, May 27). The 2020 Developer Survey results are here! Retrieved from StackOverflow: https://stackoverflow.blog/2020/05/27/2020-stack-overflow-developer-survey-results/

36 Internet Safety Research Group. (2021, July 6). Linux Kernel. Retrieved from Internet Safety Research Group. https://www.memorysafety.org/initiative/linux-kernel/

37 Goulding, J. (2020, Jan 20). What is Rust and why is it so popular? Retrieved from StackOverflow: https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/

38 Cimpanu, Catalin. (2019, Feb 11). Microsoft: 70 percent of all security bugs are memory safety issues. Retrieved from ZDNet: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

39 Rust. (2021, July 6). Why Cargo Exists. Retrieved from The Cargo Book: https://doc.rust-lang.org/cargo/guide/why-cargo-exists.html

40 BlackBerry Research & Intelligence Team, The. (2020, Nov 11). The Art of Targeted Phishing: How Not to Get Hooked. Retrieved from BlackBerry: https://blogs.blackberry.com/en/2020/11/the-art-of-targeted-phishing-how-not-to-get-hooked

41 Dr. Web. (2016, Sept 8). Doctor Web discovers Linux Trojan written in Rust. Retrieved from Dr. Web: https://news.drweb.com/show/?i=10193&lng=en&c=14

42 Szorc, G. (2021, July 6). Indygreg/PyOxidizer. Retrieved from GitHub: https://github.com/indygreg/PyOxidizer

43 GitHub. (2021, July 6). Darkarp/Chromepass. Retrieved from GitHub: https://github.com/darkarp/chromepass

44 Pike, R. (2012, October 25). Go at Google: Language Design in the Service of Software Engineering. Retrieved from Google, Inc.: https://talks.golang.org/2012/splash.article46

45 Ramanathan, N. (2021, June 19). Goroutines. Retrieved from Go Lang Bot: https://golangbot.com/goroutines/

46 Debian Wiki. (2021, January 2). Apt. Retrieved from Debian Wiki: https://wiki.debian.org/Apt

47 Broadcom. (2012, Sept 18). Endpoint Protection. Retrieved from Broadcom: https://community.broadcom.com/symantecenterprise/communities/community-home/librarydocuments/viewdocument?DocumentKey=7a3cd022-0705-43fb-8c11-181ec86b2c74&CommunityKey=1ecf5f55-9545-44d6-b0f4-4e4a7f5f5e68&tab=librarydocuments

48 GitHub. (2021, July 6). StratisIOT/Gobfuscator. Retrieved from GitHub: https://github.com/StratisIOT/gobfuscator

49 Hankins, J. (2020, December 2). Automated string de-gobfuscation. Retrieved from Kryptos Logic: https://www.kryptoslogic.com/blog/2020/12/automated-string-de-gobfuscation/

50 Pimental, J. Automatic Gobfuscator Deobfuscation with EKANS Ransomware. Retrieved from GoggleHeadedHacker: https://www.goggleheadedhacker.com/blog/post/22

51 BlackBerry Research & Intelligence Team, The. (2021, June 23). PYSA Loves ChaChi: a New GoLang RAT. Retrieved from BlackBerry: https://blogs.blackberry.com/en/2021/06/pysa-loves-chachi-a-new-golang-rat

52 Osbourne, Charlie. (2021, June 23). ChaChi: a new GoLang Trojan used in attacks against US schools. Retrieved from ZDNet: https://www.zdnet.com/article/chachi-golang-a-new-go-trojan-focuses-on-attacking-us-schools/

53 Center for Internet Security, The. (2021, March 15). The SolarWinds Cyber-Attack: What You Need to Know. Retrieved from the Center for Internet Security: https://www.cisecurity.org/solarwinds/

54 Developers, FireEye. (2021, July 6) FireEye. Retrieved from GitHub: https://github.com/fireeye

55 Byt3bl33d3r. (2021, July 6). Byt3bl33d3r/OffensiveNim. Retrieved from GitHub: https://github.com/byt3bl33d3r/OffensiveNim

56 GitHub. (2021, July 6). Epi052/Feroxbuster. Retrieved from GitHub: https://github.com/epi052/feroxbuster

57 Wanve, U. (2020, June 25). GuLoader: Peering Into a Shellcode-based Downloader. Retrieved from CrowdStrike: https://www.crowdstrike.com/blog/guloader-malware-analysis/

58 Merrick, A. (2021, March 9). Go Developer Survey 2020 Results. Retrieved from The Go Blog: https://blog.golang.org/survey2020-results

59 BlackBerry (2021, June 23). PYSA Loves ChaChi: a New GoLang RAT. Retrieved from BlackBerry: https://blogs.blackberry.com/en/2021/06/pysa-loves-chachi-a-new-golang-rat

60 Miller, Steve. (2021, July 6). About Steve Miller. Retrieved from LinkedIn: https://www.linkedin.com/in/stevenumiller/

61 Miller, S. (2021, June 14). Tweet. Retrieved from Twitter: https://twitter.com/stvemillertime/status/1404532957604323329

62 Eckels, S. (2021, June 14). Tweet. Retrieved from Twitter: https://twitter.com/stevemk14ebr/status/1399777922743996417

Intelligent Security. Everywhere.

About BlackBerry: BlackBerry (NYSE: BB; TSX: BB) provides intelligent security software and services to enterprises and governments around the world. The company secures more than 500M endpoints including over 175M cars on the road today. Based in Waterloo, Ontario, the company leverages AI and machine learning to deliver innovative solutions in the areas of cybersecurity, safety and data privacy solutions and is a leader in the areas of endpoint security management, encryption, and embedded systems. BlackBerry's vision is clear—to secure a connected future you can trust.

For more information, visit BlackBerry.com and follow @BlackBerry.