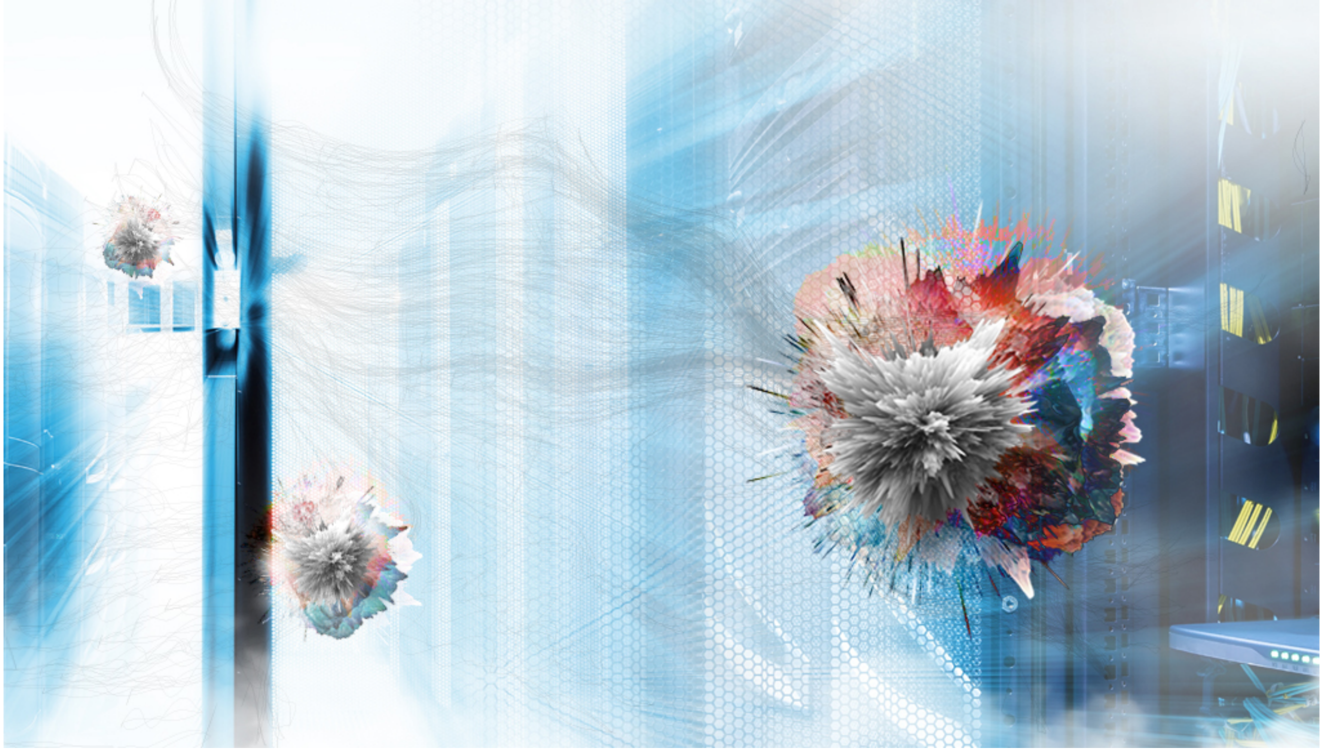
Malware Against the C Monoculture

research.checkpoint.com/malware-against-the-c-monoculture/

May 20, 2019



May 20, 2019
Research by: Ben Herzog

It's possible to write any program in any programming language; that's what Turing completeness means. Therefore, it's possible to write malware in any language, too. But in both cases, what's possible isn't always feasible. Performance issues, compatibility issues, the availability of third-party libraries and useful primitives — all of these can spell the difference between a huge success and a huge headache.

Malware authors face "product specifications" unlike any other developer. They need their product to run silently, reliably, and with minimum user interaction. As a result, they will go to great lengths to avoid an extra dependency that requires separate installation, or an extra dialogue box that asks "are you sure?". These things are usually just "nice-not-to-have"s, but for malware authors, they directly hurt the campaign's bottom line.

This contributes to somewhat of a monoculture in the malware landscape. A great majority of notable malware is written in C language, or at best C++. These may not be the most developer-friendly languages in the world (C language creator Dennis Ritchie sardonically noted in 2011 that C has "all the power of assembly language, combined with all the convenience of assembly language”), but the amount of example code and reference materials available for these languages is very great. More importantly, a well-built C binary looks just like a legitimate program to the untrained eye, and runs without a hitch or a dialogue box. If we started listing current or historical high-profile malware that’s written in C or C++, we wouldn’t stop.

Still, is this choice set in stone? How much of it is well-founded decision, rather than just attachment to the status quo? In this article, we explore that question via a review of some malware where the authors chose otherwise. We’ll separate the chosen languages into rough categories (dynamic, VM and static), and ask ourselves: Was this a good choice? Is this language — even this *kind* of language — a good fit for writing malware with, and why?

Easy, but There’s a Catch: Dynamic-Language Malware

There is no single well-defined quality that makes a programming language “dynamic” or “static”. These are more terms of art than rigid definitions. A programming language will tend to be called “dynamic” if it has enough of the following characteristics:

1. **Dynamically typed:** Types are only assigned to objects at run-time.
2. **Weakly typed:** Type conversions can happen without the developer explicitly invoking a conversion function.
3. **Garbage Collected:** Memory management happens automatically at run-time, and is invisible to the developer.
4. **Interpreted:** Language instructions are converted into machine instructions as a part of the execution process.

Some well-known examples of languages generally considered “dynamic” are Lua, Javascript, PHP, Perl, and Python (which isn’t weakly typed). These stand in contrast to “static” languages, which have traditionally had these characteristics:

1. **Statically typed:** Types are assigned to variables at compile time.
2. **Strongly typed:** Type conversions can only happen if the developer explicitly invokes a conversion function.
3. **Manual memory management:** The developer is responsible for specifying correct allocation and de-allocation of memory at compile time.
4. **Compiled:** Language instructions are all converted directly into machine instructions in a separate step, which must be completed before the program can be run.

This category includes C, C++, Pascal and Fortran.

On paper, dynamic languages should be very attractive to malware authors. When using them, the development process takes less time and is much less of a hassle — things “just work”, and quickly. There are no long compilation times, barely any time spent satisfying the type system, no boilerplate code to manage memory, and relatively little debugging before the code's first successful run from beginning to end. It's a much breezier experience than using "static" languages in general, let alone C.

That's not to say Dynamic languages are without drawbacks. Garbage collection has a performance cost; some run-time errors will get into production that a rigid type system would've caught; and dynamic code is empirically known to not "scale" as well when the project grows, due to the languages' lax guarantees and fountain of implicit magic. Still, malware authors don't typically care about these issues. They don't mind if the malware takes 3 minutes to run instead of 10 seconds; they'd rather ship quickly than enjoy any fancy-pants guarantees on esoteric edge cases; and they certainly aren't in it for the 'scaling' experience. From their point of view, you can live the dream of zero bugs, zero features, and zero technical debt – as long as you give zero priority to any of those aspects of software.

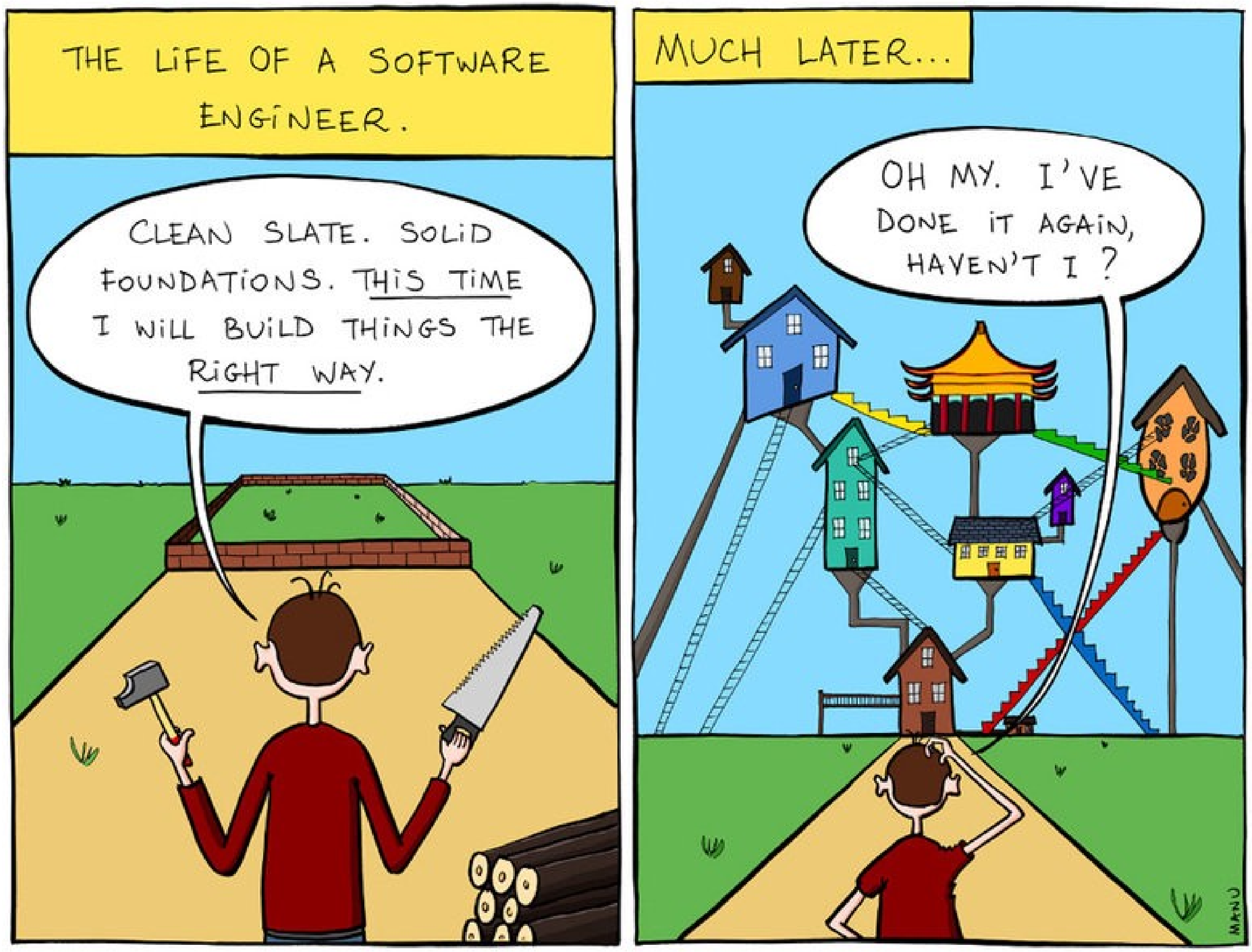
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"

The Joy of Dynamic Languages (Credit: Randall Munroe)

So why isn't all malware just written in dynamic languages? Generally, it's because:

1. Some actors have the above attitude to maintaining projects, and then others *don't*. They aim high, and their goal is to build a robust product and a lucrative brand identity, where they get to sit and watch as other criminals fork over cash to participate in their affiliate program or to use their product in a separate campaign. Feature logs, bug fixes, and technical debt are not mere hassles to these actors, but legitimate concerns.

2. Dynamic languages typically require an interpreter to function, which may not even be installed at the victim machine. Socially engineering someone to double-click a file is one thing, but socially engineering them to first install the Perl interpreter *and then* double-click the file is something else entirely.
3. Victims are used to running compiled binaries. When you hand them a compiled binary and name it frog_blender.exe, they'll just double-click it and get infected. The same gambit with an interpreted file will not work as easily. "Why does this file have a py extension instead of exe? Why is Windows asking whether I want to run or edit the file? Why do I need to pick a program to run it with?". A whole separate effort has to be made to socially engineer the victim into running the code at all.
4. In any language, some functions that are a malware author's bread and butter — such as reading and writing to the registry, working with remote processes, and sending HTTP requests — will be absent from the language's standard library. The difference is that in a compiled language, these dependencies can be baked into the malware at compile time, on the author's end; but in the case of an interpreted language, the malicious code looks for these dependencies at run time, on the victim machine, where they are almost certainly missing.


THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?
MANU

The Sorrow of Dynamic Languages (Credit: Manu Cornet)

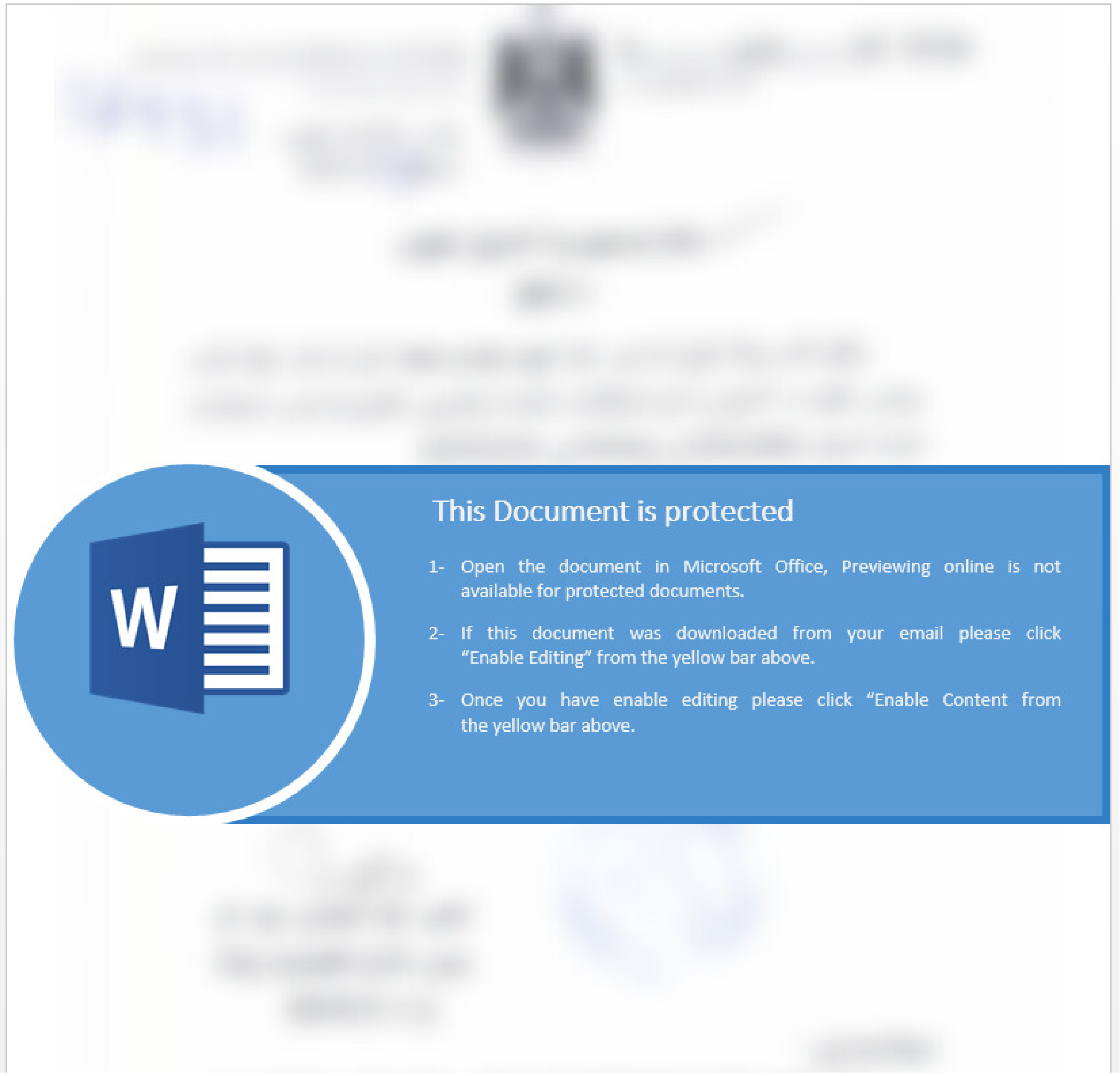
Dynamic code is typically much easier to analyze. Whole classes of anti-disassembly, anti-debugging, anti-VM, anti-Analyst's-will-to-live tricks are just out of reach when writing in a dynamic language. This is the result of several factors, and mostly the semantic gap between the interpreted script and the machine code that these techniques are typically written in. Also, interpreted malware doubles as its own source code, which is easier to understand and modify. Severely "obfuscated" scripts have been known to be de-obfuscated single-handedly by changing a single eval statement into a print.

Powershell Malware (MuddyWater, 2018; GhostMiner, 2018)

Powershell is a "task automation and configuration management framework" which Microsoft introduced in the year 2000. It is roughly analogous in function to Linux's bash, which means that if you want to do something, there's probably a command for it. Powershell is built on top of Microsoft's ".NET framework" — meaning that similarly to Java, it ultimately runs on a specific Virtual Machine, created for this purpose.

For a dynamic language, Powershell is unusually suited to writing malware, as it compensates for two of the drawbacks noted above. First, every version of Windows since XP ships with the Powershell interpreter included out of the box; second, all of Powershell's functionality is accessible immediately without the need to download third-party code. This leaves merely the concerns of poor scalability, weaker obfuscation and the lack of built-in social engineering. Actors willing to tolerate these issues, or to do some work and compensate for them, actually have the option to write their malware in a dynamic language, with all the resulting productivity gains.

One notable campaign that went the Powershell route is MuddyWater, a backdoor operation attributed to an Iranian threat actor that's been active since at least 2017. MuddyWater was mostly noted for an elaborate campaign targeted at the Saudi Arabian government, but has also been known to pick targets in the US and Europe. A year after the campaign's debut, in 2018, Turkish public companies in the finance and energy sectors were hit with a malicious document campaign that, as its payload, basically had a Powershell rewrite of previously-known MuddyWater tools. The payload itself was moderately obfuscated by BASE64 encoding, AES-encryption with a known key, and the hiding of some code fragments behind variables named after colorful English swear words. Once you got past that, the backdoor itself was minimalistic in function, and mostly used Powershell's facilities to easily perform tasks such as getting the victim's system architecture and privileges. When using C, these trivial tasks can require correctly chaining several calls from the Win32 API, which is notorious for functions such as the succinctly-named AccessCheckByTypeResultListAndAuditAlarmByHandleA that takes a mind-boggling 17 parameters.



This Document is protected

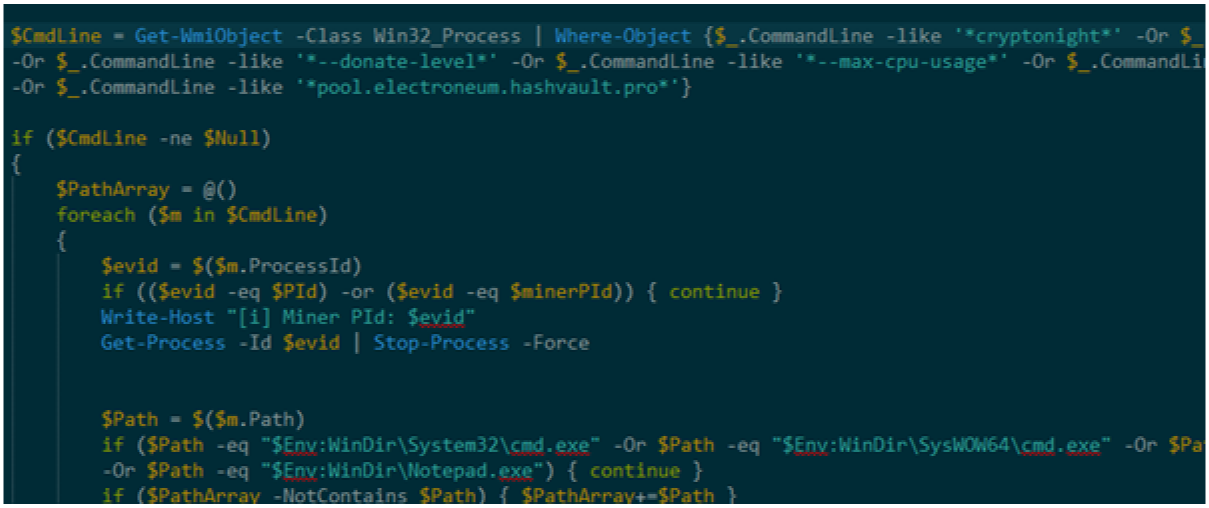
1- Open the document in Microsoft Office, Previewing online is not available for protected documents.

2- If this document was downloaded from your email please click "Enable Editing" from the yellow bar above.

3- Once you have enable editing please click "Enable Content from the yellow bar above.

MuddyWater decoy document, with the blurred logo of the Kurdistan regional government. (as initially outlined here)

Another example of Powershell malware was GhostMiner, a fileless threat discovered by researchers at Minerva at 2018 that used its hapless victims' CPU cycles to mine cryptocurrency. GhostMiner was a piece of technical art, combining ready-made pieces to create an unusually evasive infection chain. For its propagation phase, it acted akin to the old-school Morris Worm, probing random IP addresses and targeting them with an exploit for a certain 1-day vulnerability (in this case CVE-2017-10271, a vulnerability in Oracle's WebLogic server). Post-exploitation, it used two separate evasion techniques, both released as open-source tools by legitimate penetration testers, to reflectively load a malicious DLL from an obfuscated payload (Invoke-ReflectivePEInjection and Out-CompressedDll).



```
$CmdLine = Get-WmiObject -Class Win32_Process | Where-Object {$_.CommandLine -like '*cryptonight*' -Or $_
-Or $_.CommandLine -like '*--donate-level*' -Or $_.CommandLine -like '*--max-cpu-usage*' -Or $_.CommandLi
-Or $_.CommandLine -like '*pool.electroneum.hashvault.pro*'}

if ($CmdLine -ne $Null)
{
    $PathArray = @()
    foreach ($m in $CmdLine)
    {
        $evid = $($m.ProcessId)
        if (($evid -eq $PId) -or ($evid -eq $minerPId)) { continue }
        Write-Host "[i] Miner PId: $evid"
        Get-Process -Id $evid | Stop-Process -Force


        $Path = $($m.Path)
        if ($Path -eq "$Env:WinDir\System32\cmd.exe" -Or $Path -eq "$Env:WinDir\SysWOW64\cmd.exe" -Or $Pa
        -Or $Path -eq "$Env:WinDir\Notepad.exe") { continue }
        if ($PathArray -NotContains $Path) { $PathArray+=$Path }
```

Ghostminer code that removes competing miners from the victim system. (Credit: Minerva Labs)

One can see how both those choices were made to compensate for Powershell's weaknesses as a tool of the malware craft. Exploiting a vulnerability — rather than the human element — meant that the attack could proceed without socially-engineering anyone to run a Powershell script. This insight has not escaped other actors, and as a result, in recent years Powershell has seen wide use in "Fileless" malware built on the same principle. This extends to other contexts where code is already running on the victim machine, such as the tail of a malicious MS-Office macro attack (see for example here, as well as this quaint little backdoor which liaisons with its C&C server via DNS TXT entries). By using ready-made frameworks to handle evasion and obfuscation, the authors made sure that analyst frustration when coming across this threat was at least moderate.

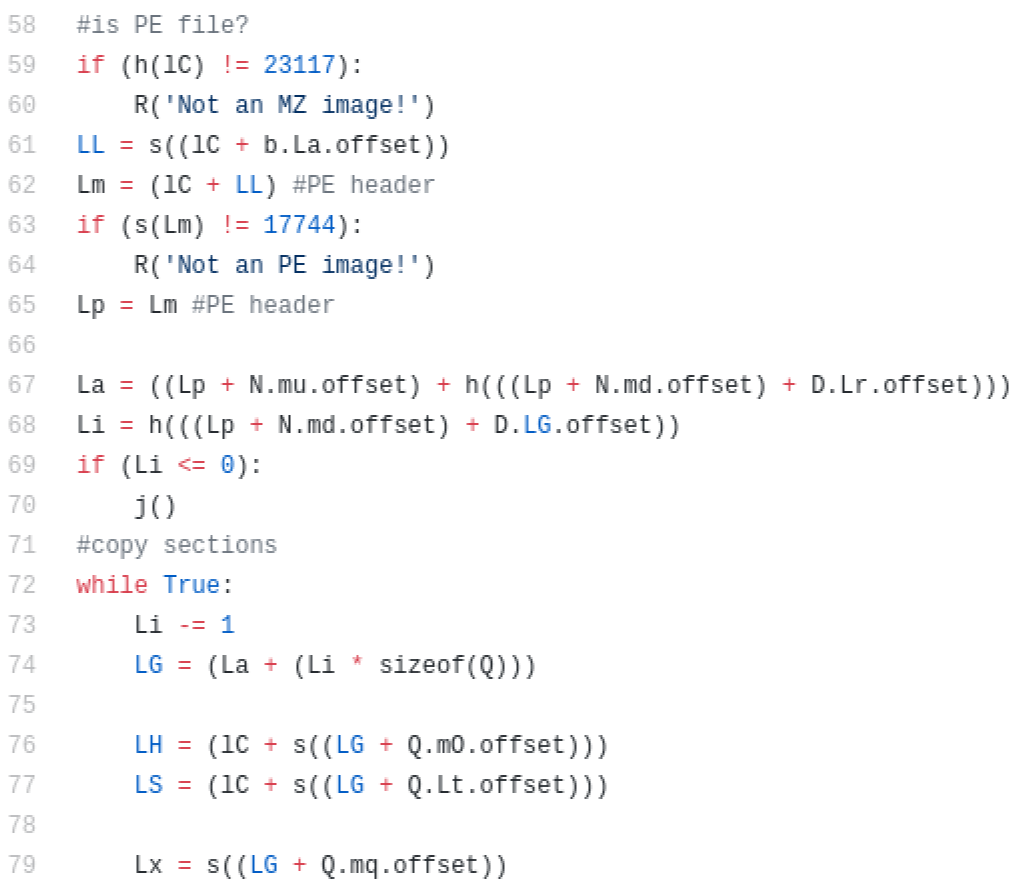
The only thing the attackers did not (and could not) compensate for was the language's unsuitability for commitment to large, maintenance-heavy projects. But maybe that's for the best: analysts suspect that GhostMiner's technical *tour de force* only managed to mine 1 XMR, equivalent to a grand total of about $200.

Python Malware (Pbot, 2018)

First released by Guido von Rossum in 1991, the Python programming language pivoted off its simple syntax and large standard library to become a sort of *lingua franca* for developers. It is one of the easiest programming language to learn, if not the easiest, and one of the most fully-featured, in that there's a library for nearly everything you might want to do. Python is the poster child for all the advantages of dynamic languages; if malware authors could just write all their malware in Python, all else being equal, they would.

Unfortunately for them, things are not that simple. With respect to writing malware, Python has all of Powershell's pitfalls, and then all the rest of the issues with dynamic languages on top of those. Python isn't pre-installed by default on Windows, and so has to be installed separately; also, many pieces of functionality that are essential to malware aren't even included with the Python standard library, and so require third-party modules. For instance, nearly every malware will have to interact with the Windows registry and send HTTP requests, but if you want to do that in idiomatic Python, you need to download a separate third-party module for each of those actions (such as winreg and requests, respectively).

All of these concerns make Python malware rare — but, apparently, not unheard of. For some malicious actors, the prospect of cutting down on 70% of their work with a few import statements is just too good to pass up, and the result is strange creatures like Pbot.



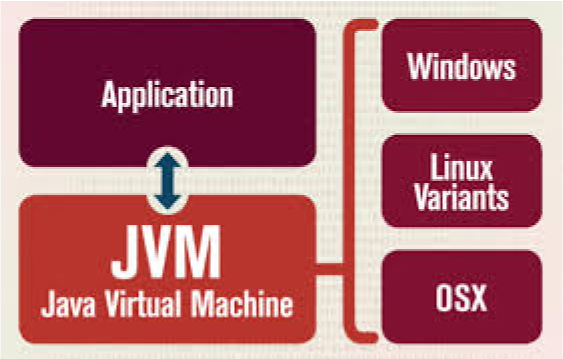
```
#is PE file?
if (h(lC) != 23117):
    R('Not an MZ image!')
LL = s((lC + b.La.offset))
Lm = (lC + LL) #PE header
if (s(Lm) != 17744):
    R('Not an PE image!')
Lp = Lm #PE header

La = ((Lp + N.mu.offset) + h(((Lp + N.md.offset) + D.Lr.offset)))
Li = h(((Lp + N.md.offset) + D.LG.offset))
if (Li <= 0):
    j()
#copy sections
while True:
    Li -= 1
    LG = (La + (Li * sizeof(Q)))

    LH = (lC + s((LG + Q.mO.offset)))
    LS = (lC + s((LG + Q.Lt.offset)))

    Lx = s((LG + Q.mq.offset))
```

Fragment of Pbot's loader code. (Credit: MalwareBytes)

Pbot is a piece of python Adware, noticed by Malwarebytes researchers in early 2018. It spends a lot of energy clearing several of the hurdles outlined above. For instance, at the time it was being spread via the RIG exploit kit, routing (again) around the requirement for social engineering. To deal with the fact that the victim may be missing the Python interpreter and/or the required libraries, Pbot does not deliver a raw .py script file, and instead delivers a large executable bundle which contains a complete Python environment and interpreter, including all the required dependencies. Bundles such as these can be created from Python packages using 3rd-party tools such as PyInstaller.

Pbot’s main functionality was a framework for injects and man-in-the-browser attacks. Without access to Pbot’s configuration, one could only speculate on the exact monetization model that authors had in mind; but given that most malware with such functionality uses it to steal banking credentials, and given that the malware came with a built-in whitelist of Russian banks to refrain from attacking, we can conclude that Pbot was probably a banking Trojan (the canonical example of such malware, ZeuS, was also known as Zbot).

By hitch-hiking on an exploit kit and using the executable bundle install, the authors of Pbot were able to skirt some of the inherent problems of Python malware, but not all. The most glaring drawback is the particularly weak obfuscation, which the authors didn’t do much to offset (even leaving in some comments and debug strings). It may also be worth mentioning that spam emails are still the most prolific attack vector, and that for cybercriminals, giving up on spam and relying solely on exploit kits is likely a bitter trade-off.



Schema of how an application written in a VM language (in this case, Java) can be run on multiple platforms.

Some of Column A, Some of Column B: VM Language Malware

Not all languages can be stuffed neatly into the “static” or “dynamic” pigeon-hole. There’s a notable set of programming languages which, on the one hand, are statically and strongly typed, and have an explicit compilation step; but on the other hand, are garbage-collected, and are compiled to bytecode that’s only translated to machine code at run time by a dedicated VM. You may rightly ask where’s the difference between this so-called “compilation” and the flow of a fully interpreted language like, for example, Python, which does after all have its own VM for exactly the same purpose. Mainly, the difference is that a VM language may easily lack a REPL (interactive prompt), whereas for a dynamic language, it’s a must-have feature; and that in a VM language, the bytecode translation must be invoked by the developer explicitly and both it and the VM itself generally have proper specifications, whereas in a dynamic language the conversion is invoked implicitly and is more of a mere implementation detail. (If you’re interested in a deeper dive into this shadowy corner, we recommend this blog post by Ned Batchelder). Some prominent examples of VM languages are Java, Scala and Kotlin, all of which run on Oracle’s JVM; as well as C# and VB.NET, all of which run on Microsoft’s CLR.

VM languages’ mishmash of characteristics puts them somewhere in the twilight zone between “static” and “dynamic”, with a resulting mix of pros and cons for malware developers. These languages provide an immediate productivity boost over C, and their bytecode can be run on any OS that has the appropriate VM installed; on the flip side, they introduce a glaring dependency in the form of the VM itself, as well as the challenge of getting the victim to run a non-PE file (there’s also the performance overhead, but again, that’s less important in the context of malware).

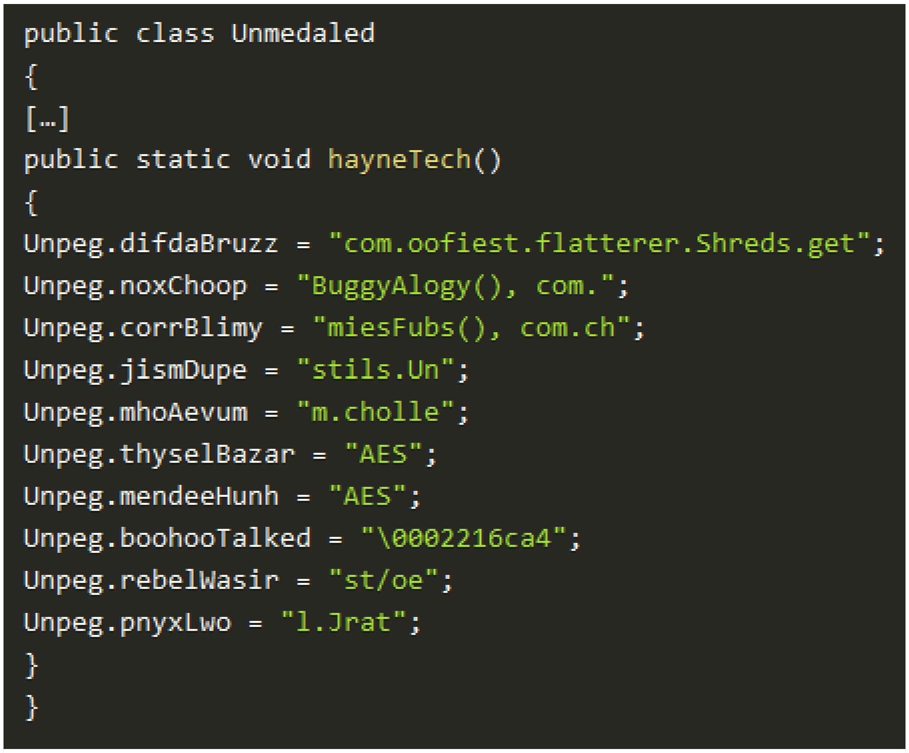
Java Malware (Jrat, 2012)

Java spent several years reigning as the most popular programming language on earth, and has a long and complicated history, fraught with patent lawsuits and corporate acquisitions. It is well-known for being an essential layer in the development stack of Android Apps, as well as a popular choice for server-side enterprise apps. Apart from its VM nature, Java is perhaps best-known for its strong adherence to Object Oriented Programming principles: In Java, everything is an object (as lampooned in Steve Yegge’s Execution in the Kingdom of Nouns).

Is Java a good choice for cybercriminals trying to write malware? That depends on how much they like Java’s Object-orientation, how much their campaign stands to gain from Java’s cross-platform reach, how much they care about obfuscation, and how much they are willing to let victim conversion rates suffer. The tagline goes “15 Billion devices run Java”, but the victim’s machine might not be one of them.

This apparently was not enough of a concern for the authors of JRat, a Remote Administration Tool discovered by Fortinet in 2016. JRat supports a variety of commands, among them deleting victim files, sending screenshots to its C&C server, killing processes and visiting URLs (this last one may be used for DDOS attacks and click fraud). Unlike with PBot, which included all its bundled dependencies, JRat only targets machines which already have Java installed.

JRat is obfuscated to a reasonable standard for a dynamic language; all the package and variable names are replaced with random gibberish, all the strings are split into many shreds that are only assembled at run-time, and some of the functionality is buried behind an AES-encrypted blob. While frustrating, these stumbling blocks still can’t hold a candle to the obfuscations and evasions one often encounters on classic-flavor C malware.



```
public class Unmedaled
{
[…]
public static void hayneTech()
{
Unpeg.difdaBruzz = "com.oofiest.flatterer.Shreds.get";
Unpeg.noxChoop = "BuggyAlogy(), com.";
Unpeg.corrBlimy = "miesFubs(), com.ch";
Unpeg.jismDupe = "stils.Un";
Unpeg.mhoAevum = "m.cholle";
Unpeg.thyselBazar = "AES";
Unpeg.mendeeHunh = "AES";
Unpeg.boohooTalked = "\0002216ca4";
Unpeg.rebelWasir = "st/oe";
Unpeg.pnyxLwo = "l.Jrat";
}
}
```

Jrat source containing obfuscated string shreds. (Credit: Fortinet)

Happily, judging by JRat itself, we are still far away from the specter of true cross-platform malware. The malware was not actually written fully in Java, and contained OS-specific components which were executed depending on the victim OS. Also, according to analysis by Trend Micro, the authors did not fully implement all of the malware's functionality for all operating systems, and instead showed clear "preferential treatment" to compatibility with Windows OS. This makes some sense, given that Windows OS is probably more familiar to the malware authors and has a significantly larger market share — but if we put ourselves in the authors' shoes, we can't help but see this choice as a missed opportunity.

We Fear It's the Future: Ergonomic Static Language Malware

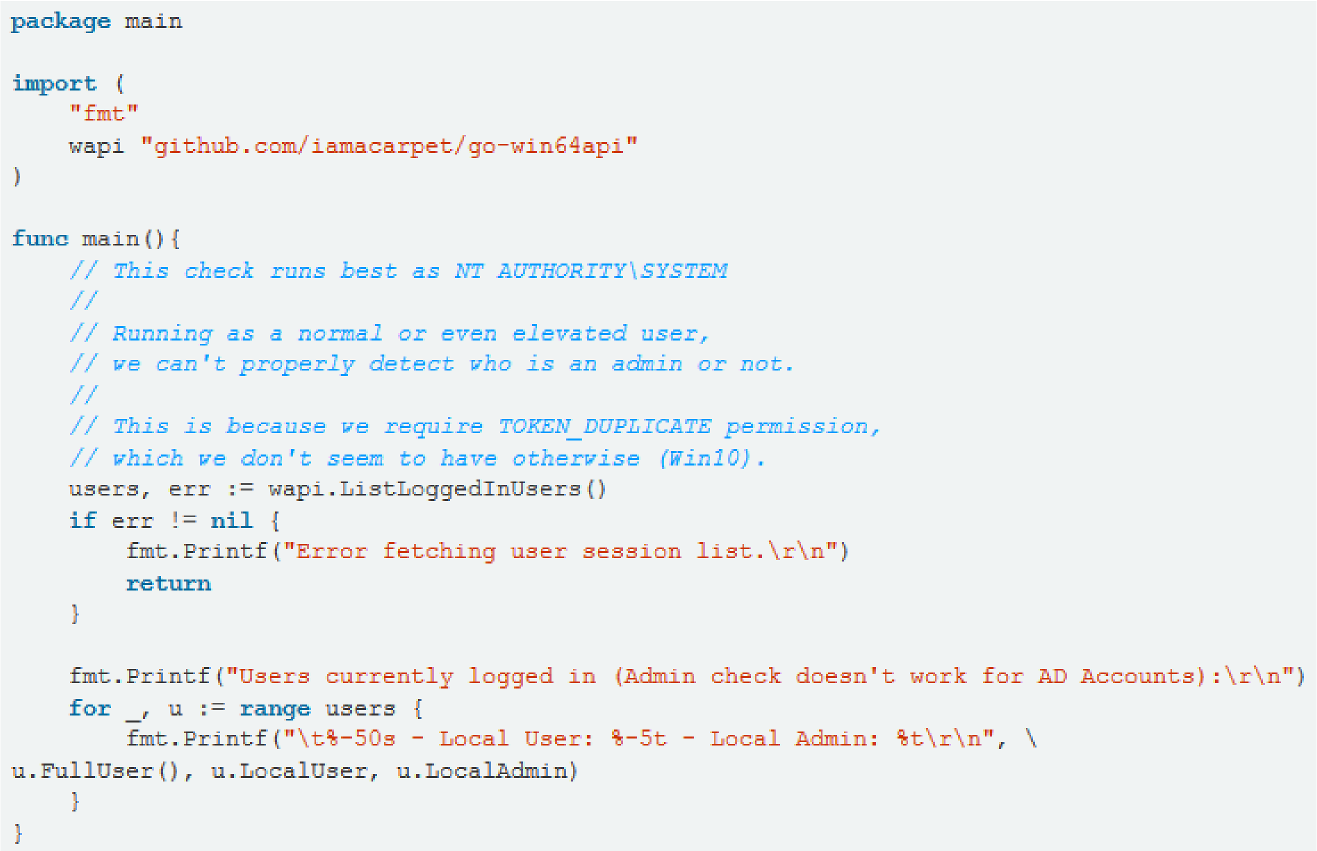
As mentioned above, ergonomics for static languages have been historically poor, especially so for C language. This deficiency gave birth to dynamic and VM-based languages, but also produced efforts to create new static languages with improved ergonomics. The most well-known effort in this direction is old reliable C++, which introduced exceptions, operator overloading, improved memory management and built-in support for Object Oriented Programming; yet during the years, has also inspired a litany of criticisms leveled at its ad-hoc patchwork design, Turing-complete template system and, most importantly, its manual memory management. The last 20 years therefore saw many efforts to address these issues by creating "a better C++", many of which failed to catch on and faded into obscurity.

For malware authors, the modern static languages in serious use today generally share the advantages and disadvantages of static languages, apart from some differences. First, the ergonomics tend to be much better, but still can't match the experience of writing in, say, Python. Second, these languages use stronger typing systems than C, which place more restrictions on conversions between types. This improves scalability with respect to project size, but means that some categories of clever tricks are right out, and the malware developer may have come to rely on those in their coding style. Third, these languages are typically new, with a smaller niche than C++, and so studying them is a costly, avoidable investment. Fourth, and related to the third: there is no large repository of leaked malicious code to pick and choose from, and the third-party libraries are sometimes not mature by the standard you'd come to expect from more established languages.

Golang Malware (Mirai, 2016; Zebrocy, 2019)

Golang, which was created at Google in 2007, puts a particular emphasis on readability, simplicity, built-in concurrency primitives, and a minimal feature set. The authors reportedly wanted the language specification to be "small enough to hold in a programmer's head", and opted to leave out many complex features (most notably generic programming facilities), which makes the learning curve for Golang a treat. Golang is therefore an opportunity for malware authors if they are willing to put in the moderate learning effort, and as long as they are also willing to roll up their sleeves and make do when otherwise they would have used a third-party library or copy-pasted code from the leaked Gozi sources.

Since 2012, the infosec world has been subjected to an intermittent drizzle of Golang malware. Among the notable examples, one can find this tool that co-opts the victim machine to assist in brute-forcing the credentials to phpMyAdmin and WordPress websites; WellMess, a simple backdoor that allows remote execution of shell commands on the victim machine; and we would be remiss not to mention the Mirai Botnet, a juggernaut made out of hundreds of thousands of IoT devices that were compromised via credential brute-force, and was infamously used to launch a gigantic denial of service attack against the blogging platform of investigative reporter Brian Krebs. (If you happen to have a spare hour, the follow-up investigation report by Krebs is really something.)



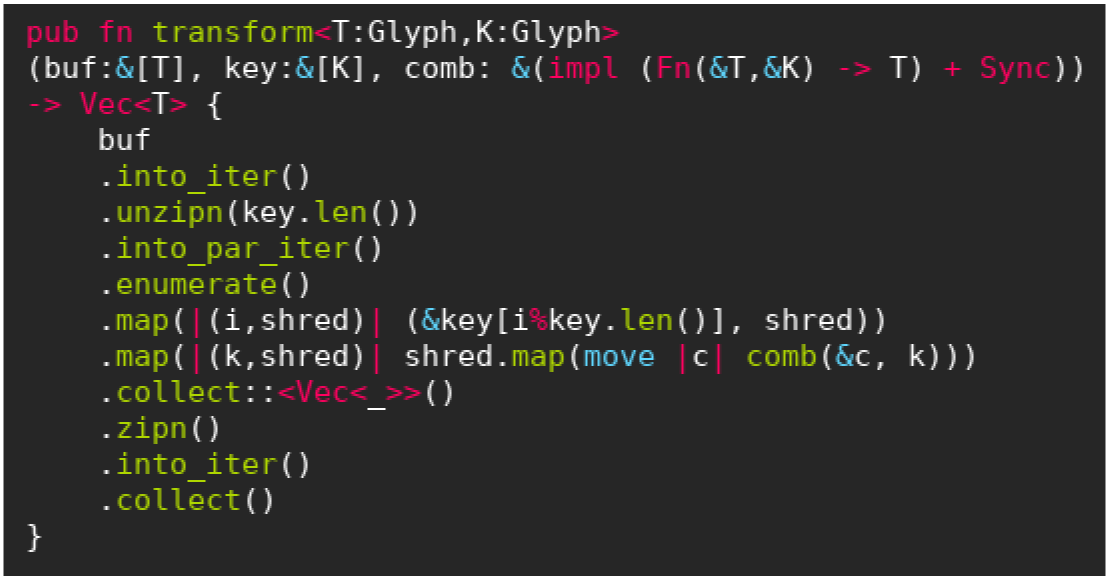
```
package main

import (
    "fmt"
    wapi "github.com/iamacarpet/go-win64api"
)

func main(){
    // This check runs best as NT AUTHORITY\SYSTEM
    //
    // Running as a normal or even elevated user,
    // we can't properly detect who is an admin or not.
    //
    // This is because we require TOKEN_DUPLICATE permission,
    // which we don't seem to have otherwise (Win10).
    users, err := wapi.ListLoggedInUsers()
    if err != nil {
        fmt.Printf("Error fetching user session list.\r\n")
        return
    }

    fmt.Printf("Users currently logged in (Admin check doesn't work for AD Accounts):\r\n")
    for _, u := range users {
        fmt.Printf("\t%-50s - Local User: %-5t - Local Admin: %t\r\n", \
u.FullUser(), u.LocalUser, u.LocalAdmin)
    }
}
```

This open-source Golang code to retrieve the list of logged-in users was used verbatim in Zebrocy. (Credit: Vitali Kremez)

Perhaps the most technically notable incident was in late 2018, when a new variant surfaced of the Zebrocy Downloader / Infostealer, a large piece of which was written in Golang. This piece of malware originated with the Sofacy Group (APT28), thought connected to the Russian military intelligence agency GRU. According to researchers at Kaspersky, this new variant was mainly used to attack central Asian governments, and rode on the tail of a complex infection chain that lured victims to run an executable file with an icon mimicking that of an MS-Word document. Post-infection, the malware displayed a decoy document to mitigate suspicion, which certainly implies some degree of author dedication.

Rust Malware (Exaramel Backdoor, 2017)

Rust language was originally conceived at Mozilla Research, and its main feature is a “borrow checker” that enforces thread and memory safety at compile time, removing the need for a run-time garbage collector and making the language viable for real-time and embedded systems. Apart from that, Rust has a complex type system, mind-boggling iterator adapters and many other such features directly inspired by Haskell and its functional ilk.

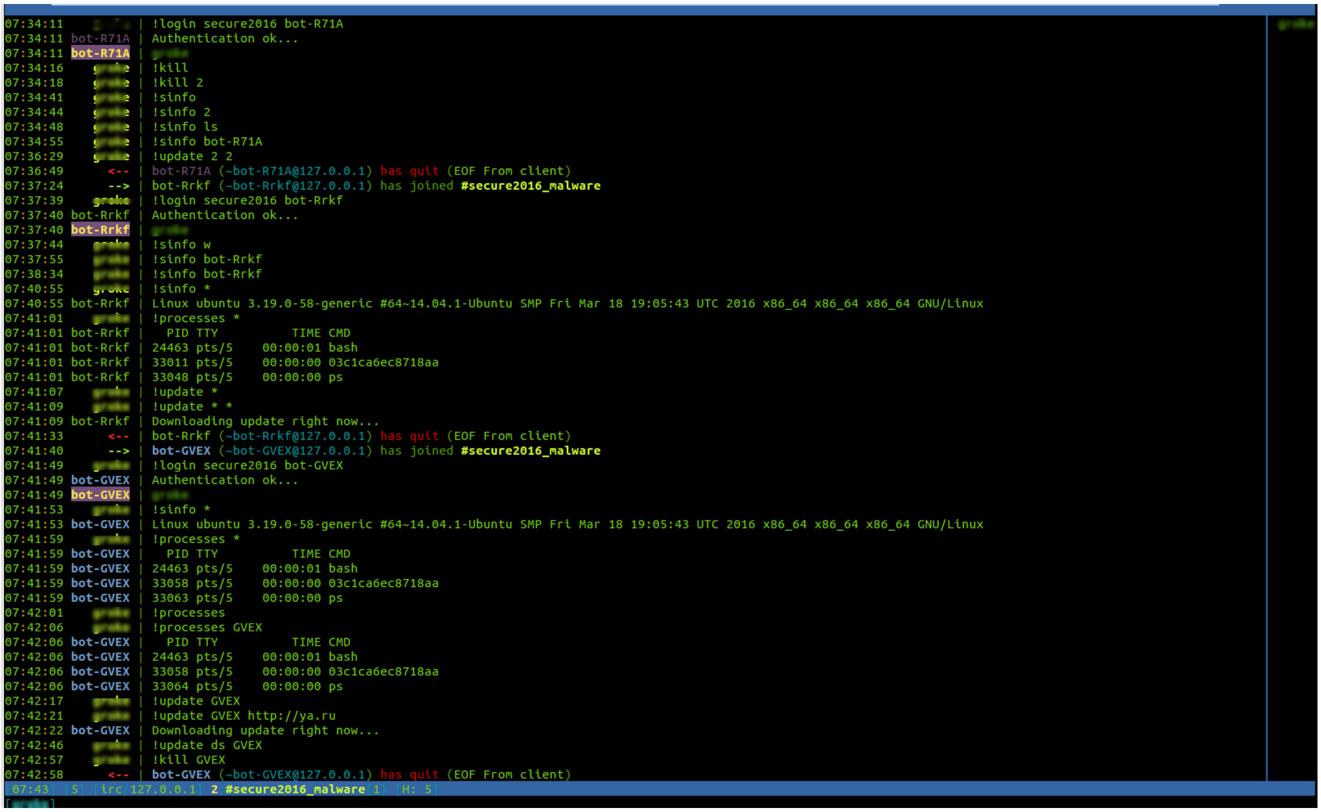


```
pub fn transform<T:Glyph,K:Glyph>
(buf:&[T], key:&[K], comb: &(impl (Fn(&T,&K) -> T) + Sync))
-> Vec<T> {
    buf
    .into_iter()
    .unzipn(key.len())
    .into_par_iter()
    .enumerate()
    .map(|(i,shred)| (&key[i%key.len()], shred))
    .map(|(k,shred)| shred.map(move |c| comb(&c, k)))
    .collect::<Vec<_>>()
    .zipn()
    .into_iter()
    .collect()
}
```

This Rust code performs a generalized rotating byte-wise encryption.

On the face of it, Rust is not a reasonable choice for malware authors. While practitioners hold the language in high regard, it has a notoriously difficult learning curve, and solves problems that cybercriminals don’t much care for. Maybe somewhere out there, there’s a malware author with a passion for shaving 5% off their malware’ running time, running it on embedded devices and implementing all the logic via zygohistomorphic prepromorphisms, but we suspect this person is the exception, rather than the rule.

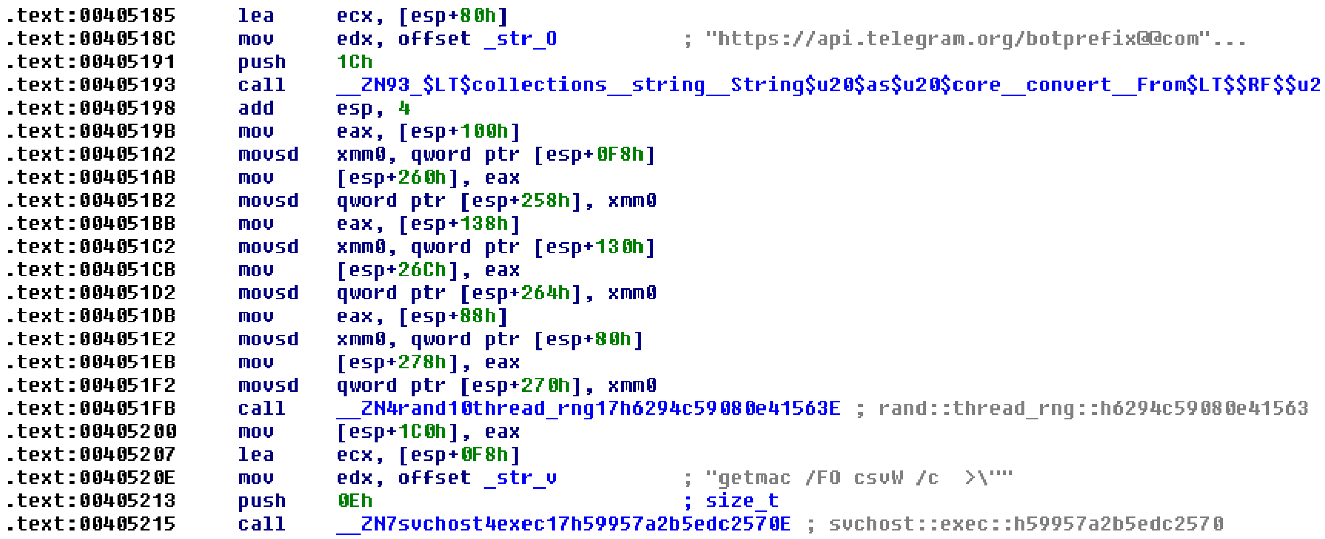
Imagine our surprise, then, when we first heard of Linux.Backdoor.IRC, originally discovered in 2016. It is a simple Rust backdoor that responds to commands via IRC protocol (some would call this “old-school”), and supports a total of 4 commands: collect & send OS information to the C&C server; collect and send information about running processes on the victim machine; connect to a specific chat channel; and self-destruct (delete self from file system). The analysts at Dr. Web, who discovered this curiosity, noted that the IRC channel was not active and the malware appeared to lack any mechanism for self-propagation, and thus came to believe that it was a prototype or a proof-of-concept. Whatever it was, when its authors were done, they apparently wiped a drop of sweat off their brow and said, “we’re not doing that again”.



```
07:34:11 [redacted] | !login secure2016 bot-R71A
07:34:11 bot-R71A | Authentication ok...
07:34:11 bot-R71A | [redacted]
07:34:16 [redacted] | !kill
07:34:18 [redacted] | !kill 2
07:34:41 [redacted] | !sinfo
07:34:44 [redacted] | !sinfo 2
07:34:48 [redacted] | !sinfo ls
07:34:55 [redacted] | !sinfo bot-R71A
07:36:29 [redacted] | !update 2 2
07:36:49      <-- | bot-R71A (~bot-R71A@127.0.0.1) has quit (EOF From client)
07:37:24      --> | bot-Rrkf (~bot-Rrkf@127.0.0.1) has joined #secure2016_malware
07:37:39 [redacted] | !login secure2016 bot-Rrkf
07:37:40 bot-Rrkf | Authentication ok...
07:37:40 bot-Rrkf | [redacted]
07:37:44 [redacted] | !sinfo w
07:37:55 [redacted] | !sinfo bot-Rrkf
07:38:34 [redacted] | !sinfo bot-Rrkf
07:40:55 [redacted] | !sinfo *
07:40:55 bot-Rrkf | Linux ubuntu 3.19.0-58-generic #64~14.04.1-Ubuntu SMP Fri Mar 18 19:05:43 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
07:41:01 [redacted] | !processes *
07:41:01 bot-Rrkf |   PID TTY          TIME CMD
07:41:01 bot-Rrkf | 24463 pts/5    00:00:01 bash
07:41:01 bot-Rrkf | 33011 pts/5    00:00:00 03c1ca6ec8718aa
07:41:01 bot-Rrkf | 33048 pts/5    00:00:00 ps
07:41:07 [redacted] | !update *
07:41:09 [redacted] | !update * *
07:41:09 bot-Rrkf | Downloading update right now...
07:41:33      <-- | bot-Rrkf (~bot-Rrkf@127.0.0.1) has quit (EOF From client)
07:41:40      --> | bot-GVEX (~bot-GVEX@127.0.0.1) has joined #secure2016_malware
07:41:49 [redacted] | !login secure2016 bot-GVEX
07:41:49 bot-GVEX | Authentication ok...
07:41:49 bot-GVEX | [redacted]
07:41:53 [redacted] | !sinfo *
07:41:53 bot-GVEX | Linux ubuntu 3.19.0-58-generic #64~14.04.1-Ubuntu SMP Fri Mar 18 19:05:43 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
07:41:59 [redacted] | !processes *
07:41:59 bot-GVEX |   PID TTY          TIME CMD
07:41:59 bot-GVEX | 24463 pts/5    00:00:01 bash
07:41:59 bot-GVEX | 33058 pts/5    00:00:00 03c1ca6ec8718aa
07:41:59 bot-GVEX | 33063 pts/5    00:00:00 ps
07:42:01 [redacted] | !processes
07:42:06 [redacted] | !processes GVEX
07:42:06 bot-GVEX |   PID TTY          TIME CMD
07:42:06 bot-GVEX | 24463 pts/5    00:00:01 bash
07:42:06 bot-GVEX | 33058 pts/5    00:00:00 03c1ca6ec8718aa
07:42:06 bot-GVEX | 33064 pts/5    00:00:00 ps
07:42:17 [redacted] | !update GVEX
07:42:21 [redacted] | !update GVEX http://ya.ru
07:42:22 bot-GVEX | Downloading update right now...
07:42:46 [redacted] | !update ds GVEX
07:42:57 [redacted] | !kill GVEX
07:42:58      <-- | bot-GVEX (~bot-GVEX@127.0.0.1) has quit (EOF From client)
[07:43] [5] [irc/127.0.0.1] 2:#secure2016_malware(1) [H: 5]
[[redacted]]
```

Screenshot of the IRC channel used to control Linux.Backdoor.IRC. (Credit: Dr. Web)

We would disregard Linux.Backdoor.IRC as a freak coincidence and/or urban legend, except that two years later it was followed by another, and rather more serious, instance of Rust malware: the Exaramel backdoor, used by the Telebots group. This group is thought to have been behind the high-profile NotPetya outbreak, as well as several attacks against critical infrastructure in Ukraine (which were dressed up as ransomware attacks with a highly unreasonable ransom demand). The group is thought to have connections with the BlackEnergy group, which was known for a similar *modus operandi*; the name “Telebots” is derived from its standard backdoor tool, which communicates with its C&C server via the Telegram bot API.



```
.text:00405185     lea     ecx, [esp+80h]
.text:0040518C     mov     edx, offset _str_0         ; "https://api.telegram.org/botprefix@@com"...
.text:00405191     push    1Ch
.text:00405193     call    __ZN93_$LT$collections__string__String$u20$as$u20$core__convert__From$LT$$RF$$u2
.text:00405198     add     esp, 4
.text:0040519B     mov     eax, [esp+100h]
.text:004051A2     movsd   xmm0, qword ptr [esp+0F8h]
.text:004051AB     mov     [esp+260h], eax
.text:004051B2     movsd   qword ptr [esp+258h], xmm0
.text:004051BB     mov     eax, [esp+138h]
.text:004051C2     movsd   xmm0, qword ptr [esp+130h]
.text:004051CB     mov     [esp+26Ch], eax
.text:004051D2     movsd   qword ptr [esp+264h], xmm0
.text:004051DB     mov     eax, [esp+88h]
.text:004051E2     movsd   xmm0, qword ptr [esp+80h]
.text:004051EB     mov     [esp+278h], eax
.text:004051F2     movsd   qword ptr [esp+270h], xmm0
.text:004051FB     call    __ZN4rand10thread_rng17h6294c59080e41563E ; rand::thread_rng::h6294c59080e41563
.text:00405200     mov     [esp+1C0h], eax
.text:00405207     lea     ecx, [esp+0F8h]
.text:0040520E     mov     edx, offset _str_v         ; "getmac /FO csvW /c  >\""
.text:00405213     push    0Eh                        ; size_t
.text:00405215     call    __ZN7svchost4exec17h59957a2b5edc2570E ; svchost::exec::h59957a2b5edc2570
```

A part of the Exaramel backdoor disassembly. (Credit: ESET)

The Exaramel backdoor was originally written in Python but was later Rewritten in Rust. According to ESET, it was distributed as a Trojan pretending to be an AV solution, and even communicated with domains that were forged to appear to be legitimate AV vendor domains. It supported a fairly sophisticated API, including the ability to create processes as a specific user, start and stop Windows services, and evaluate arbitrary VBS code. It was also bundled with an embedded credential stealer, which harvested passwords from a long list of web browsers and FTP clients.

We finally note that versions of this backdoor were spotted in the wild that were compiled to run on Linux OS, which implies that the Telebots group makes a point of being a cross-platform threat. This at least may explain why they rewrote their backdoor in a “modern C++ replacement” language. Outside of Rust and Golang’s 3rd party code ecosystems, to retarget your executable you’d either have to rely on a VM/interpreter at the victim end, or else go back and individually change OS API calls. As to why this group didn’t rewrite the backdoor in Golang instead, it is a mystery for the ages. Maybe they just really like generics.

Other Honorable mentions

- Linux.Grip (2005) reportedly used a snippet of the esoteric language Brainf*ck for generating encryption keys.
- Retefe (2013), a banking Trojan, has a history of unorthodox clones meant for non-Windows operating systems — including a clone written in Apple’s Objective-C, which was distributed circa April 2017 in a campaign that targeted the key demographic of Swiss macOS users.
- Gootkit (2014), also a banking Trojan, is written mostly in Javascript (Node.js), a language usually reserved for web development.

- LatentBot (2017), an all-purpose modular backdoor, as well as several packers that have been spotted in the wild, were written in Delphi — a dialect of Pascal, a language invented in 1970 for educational purposes.

Conclusion

As you've probably heard, the malware landscape is in constant evolution. Most actors just copy what's known to work, but some like to experiment — whether due to an actual need or sheer hipster spirit. The failures are then discarded as dead ends, while the successes are imitated. Some technologies are esoteric right now because they genuinely aren't a good fit for writing malware; others, because they haven't picked up credibility, or because threat actors shy away from their slightly different mix of pros and cons, and would rather deal with the devil they know.

It's difficult to say in advance which of these technologies, if any, will become the "next big thing" in the malware world. Some technologies, such as Golang and Powershell, seem particularly well-poised to do so (Powershell is already halfway there). Others, like Java and Rust, not so much. But for a technology to become fashionable in the cybercrime circles, all it takes is one ambitious author to produce one high-profile successful campaign that hinges on it.

Will we all still be looking at C malware all day long 10 years from now? 20? It's difficult to know. But maybe we should hope so. The current malware landscape, which is enough of a problem, came to be even as threat actors duly spent a bulk of their resources debugging segmentation faults, looking up function parameter order in the MSDN database and copy-pasting code from header to source files and back again. Maybe we don't *want* to find out what happens when all of that time suddenly becomes free time.