POLYSCRIPTING

Applying Moving Target Defense cybersecurity tactics to programming languages.

Author: Blue Gaston, Software Engineer

Bellevue, WA

TABLE OF CONTENTS

POLYVERSE

Introduction: Moving Target Defense

When it comes to programming, it is important to accept an essential fundamental truth: every piece of software is hackable. Ultimately, this means everyone is vulnerable. Given enough time and resources, a vulnerability can always be found, and an exploit can be crafted. What makes this attractive to a malicious actor is that a crafted attack can be applied across a wide surface area. With any given vulnerability, a hacker is able to execute an exploit across a range of machines that meet the criteria defined by a presupposed, assumed, and known attack vector. The effort-to-reward ratio is in their favor.

Exploits are cheap and widely available. While it is incredibly expensive to craft an exploit for every vulnerability, they can be built once and sold many times over because of the homogeneity of programs. Everyone runs the same programs, operating systems, machines, languages and databases. This includes those concocting attacks. This sort of identical access provides an advantageous roadmap to build malicious exploits, to find vulnerabilities and to carefully craft attacks that can be used at a large scale. It presents difficult problems and powerful opportunities within the security space.

Moving Target Defense (MTD) offers a solution that draws its inspiration from nature.

Genetic diversity is both a key to, and a result of the survival and evolution of organisms. All members of a population do not share the exact genetic makeup. If every human was a clone, the first deadly disease that came along would affect each individual the same way, essentially wiping out the human race. Think of a disease like a malicious hack. It needs to propagate and interact with the host's defenses in a certain manner in order to effectively spread. If every human was genetically identical, a disease able to successfully infect one person could similarly infect other humans with the same deadly consequence. Yet, this is not the case with organisms. A disease that is deadly to one individual, may not ail another with so much as a fever because of the diversity in their genetic makeup. The key here is that everyone possesses unique DNA, which is a key component to a species' survival.

What if computer programs shared this quality of having their own unique genetic makeup? This is the concept that MTD applies to cybersecurity. MTD is predicated on introducing unique components between machines, programs, binaries, and languages, thus limiting exploitation to when its makeup exactly matches the expected attack vector. As with infections, many attack vectors rely on being able to access certain anchor points or data. MTD aims to rearrange these anchor points so that an exploit is unable to adjust to nor account for the change, causing an attack to ultimately fail.

MTD is the practical application of nature's genetic diversity to technology. It creates a program that while identical in function, is entirely unique from any previous version of the program. For example, Polyverse's polymorphic version of Linux® is one such MTD solution. It relies on custom compilers to generate unique binaries that allow for the constant rearrangement of the aforementioned anchor points. By 'scrambling' these anchor points, the protected software programs and systems effectively become immune to all but the most targeted of memory exploits. Simply put, a malicious actor must choose to directly target your machine or server knowing that it is different from any with which they may have previously interacted. In the case of systems running polymorphic versions of Linux and adhering to a strategy of MTD, knowing that the attack vector, even if successfully enumerated, will not stay the same for long is an invaluable asset. In other words, the application's memory landscape is a constantly shifting moving target, making exploitation significantly more difficult, resource intensive, and time consuming.

With any given vulnerability, a hacker is able to execute an exploit across a range of machines that meet the criteria defined by a presupposed, assumed, and known attack vector. The effort-to-reward ratio is in their favor.

The tactics the polymorphic versions of Linux applies to compilers, a concept dubbed "Polyscripting" is now applying to language interpreters. Interpreted languages in web applications are ubiquitous and are used for critical tasks, such as information storage and retrieval, as well as providing seamless interactivity via an application's UI. These languages include PHP, JavaScript and SQL and provide commonplace, easily identifiable, and exploitable areas of publicly distributed web applications. One such exploitation is code injection attacks.

Code Injection Attacks

It is easy to point fingers when it comes to security breaches. Whether it's deprecated legacy code, a zero-day vulnerability, or a forgotten patch, people make mistakes and things happen. These breaches continue to happen, even as the industry focuses on budding new technologies like artificial intelligence, quantum computing, and blockchains in order to stay secure. SQL injection continues, and WordPress vulnerabilities that allow code injection are being taken advantage of. Data is consistently corrupted and stolen and ransomware is a constant plague on both the private and public sectors.

Code injection is an incredibly powerful tool that hackers employ to accomplish their goals. It is an attack vector allowing a malicious actor to run their own code on a server or website belonging to a separate entity. Often, it is used as a backdoor to access information or to change and to corrupt data. Some of the most devastating breaches in history have relied on simple code injection. For example, the Equifax breach relied on code that was injected through an unprotected deserialization call. There are certain methods to meticulously guard against code injection, such as input sanitization, code signing and whitelisting. Despite the techniques that exist to thwart code injection, such attacks continue to occur at an increasingly alarming rate. September 2018 alone saw numerous noteworthy code injection attacks:

- Scarma Labs published a white-paper before blackhat 2018 that described a PHP vulnerability that has gone unpatched and unreported for over a year since they first notified various services of the issue, WordPress, the most used CMS on the internet, as of a few weeks after the reports, had still not issued a fix for the vulnerability which allows code injection.
- A zero-day bug allowed hackers to access CCTV surveillance cameras, and subsequent code injection and remote code execution allowed hackers to gain access to user accounts as well as change passwords.
- A Remote Code Execution vulnerability existed in the widely popular Duplicator WordPress plugin that affected many users, this was patched September 5th, 2018.

Needless to say, this exploit is hardly a thing of the past.

Equifax is probably the most potent example of code injection that led to an incredibly devastating remote code execution attack. This mega-breach resulted in potentially 143 million Americans' most sensitive personal information being exposed. Equifax utilized Apache Strut's as its framework for creating Java web applications. The parser this uses—Jakarta—contained the security flaw. This flaw was patched prior to the breach, but the patch was never applied.

The Jakarta parser had a feature that allows you to deserialize XML into Java objects. A simplified version looks like this:

```
<object class="io.polyverse.Person">
       <field name="Name">Archis</field>
       <field name="City">Seattle</field>
</object>
```

All someone had to do was try to instantiate an internal object:

```
<object class="java.system.Exec">
       <field name="Command">/bin/rm</field>
       <field name="Params">-rf</field>
</object>
```

The Struts vulnerability allowed any and all objects to be instantiated by default when no whitelist/blacklist was provided. The hackers were able to inject code and execute it remotely.

This is part of a practice that Polyverse calls DevSecOps. Safe defaults by developers that prevent dangerous execution paths from being followed. The aforementioned flaw was widely exploited despite a corrective patch being published the same day the vulnerability was announced to the public. An extreme, but all too real example of someone capitalizing on an exploit of this nature.

1 https://cdn2.hubspot.net/hubfs/3853213/us-18-Thomas-It's-A-PHP-Unserialization-Vulnerability-Jim-But-Not-As-We-....pdf
2 https://threatpost.com/zero-day-bug-allows-hackers-to-access-cctv-surveillance-cameras/137499/
3 https://www.wordfence.com/blog/2018/09/duplicator-update-patches-remote-code-execution-flaw/

Polyscripting: An Introduction

Rather than endlessly stressing about patching and attempting to juggle all of the vulnerabilities exposed via your application's attack surface, Polyscripting removes the prerequisite mechanics that allow such attacks to occur. This ensures that even when safeguards prove ineffective, the attack vector was previously undiscovered, or a patch was not applied in a timely manner, the attack will simply not work.

Applying the idea of Moving Target Defense, the question to ask is what kind of homogeneity allows for malicious code injection? What makes code injection and remote code execution possible as a whole? What information does a malicious actor have to gather that allows them to exploit a third party's assets?

There are two assumptions made during this kind of attack: First, that malicious code can be injected into the system, and second, that the malicious code can be remotely executed.

Polyscripting negates that second assumption. Today, remote code execution and code injection attacks are possible because a hacker can write injectable code, upload it to a server, and execute it. In this scenario, the server understands the hacker's code in the exact same way it understands valid code because they are written in the same language, with the same syntax. This allows the attacker to derive value from the injection. The hacker's roadmap relies on the successful execution of their code. If a server contains a PHP interpreter, then it has the capacity to parse and execute any PHP code.

What if that wasn't the case? If a server was unable to execute injected code, then this attack vector as a whole would be rendered ineffective. Without impacting functionality, Polyscripting gives each website a unique instance of a programming language. This kind of diversity renders that second crucial assumption, that the attacker will be able to execute the code they have injected, false.

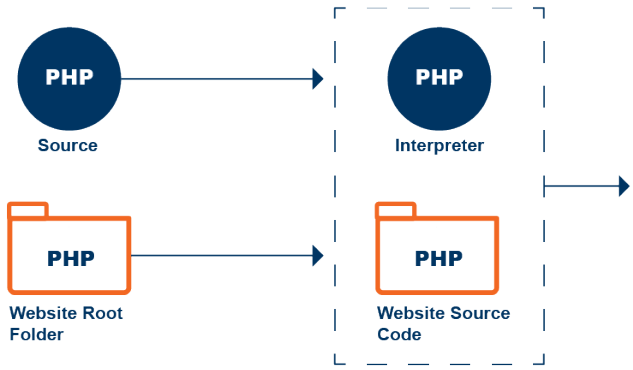
Polyscripting takes a programming language and scrambles (explained later but understanding scrambling as randomization will suffice for now) the syntax and grammar within the source for that language before the interpreter is compiled. The output is a dictionary that is used to transform all necessary source code before it runs in production. This results in an application that has its own unique implementation of a language, as well as the matching interpreter. The new interpreter no longer understands the original syntax and grammar of the original language. It will only execute the source code that matches the newly generated unique interpreter. Additionally, this process can be repeated on demand, adding additional layers of defense, making time an ally to a system's defenses through the use of regular intervals at which the interpreter and source code undergo polyscripting. This process emulates a moving target, remapping the application's address space so frequently that proper enumeration, crafting, and execution of an exploit becomes impractically difficult. This schews the effort-to-reward ratio so that it is no longer in a hacker's favor.

It comes down to cause and effect. Whether the cause of code injection is exploiting broken deserialization methods, a legacy vulnerability in a plugin, or an unknown language vulnerability, the responsibility to guard against these falls on the programmer. However, hackers are creative, and even the "most securely written" of programs get hacked. Just look at Facebook, Playstation, Equifax or Target. All companies with massive security teams that genuinely put in the research, time, and effort to stop the cause of these attacks, yet they still happen. Polyscripting is a way to stop this effect. Normally, the effect of a successful code injection attack would be the execution of the malicious code, with Polyscripting a syntax error gets thrown and no malicious code is run; stopping the malicious effect.

Standard Workflow

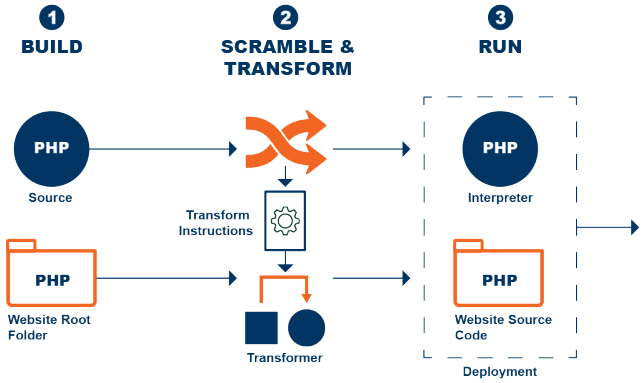
In a basic workflow for a standard website running PHP, the PHP interpreter is compiled and loaded onto the web server. The website's source code is also pushed to the same server. The PHP interpreter then parses and interprets the source code before sending the result elsewhere: to a user, browser, database, etc.

At a very basic level, this is a two-step process:



Polyscripting Workflow

Polyscripting only adds one additional layer to this deployment process. The PHP source code gets scrambled to the polyscripted version and the websites source code gets scrambled to match the unique instance of PHP that was generated. The interpreter for the language (PHP) is changed at compile time and, ideally, the scrambled dictionary is only accessed and only exists before being deployed to a web server.



Language Scrambling

The process of scrambling a language is beautifully simple. The make-up of a programming language is contained within its syntax and grammar. The keywords and syntax of a language are defined and compiled to make up the words and ordering of word-tokens that a language understands. Programs are then parsed based on this lexical syntax to generate the grammar the further defines a language.

as	→	GBkxiVw
eval	→	yrSFsQ
extends	→	EFiKYzjoWraK
for	→	ENCdPd
foreach	→	tVlGNMOD
function	→	jIoTEgRX
isset	→	KvCeJT
while	→	lzkWyBPn

The values of the keywords themselves are arbitrary in any given language. Keywords are defined for the convenience of those writing the code. If you think of these words as just a means to write a language, the values themselves are random. Where "echo" is defined in the lexical grammar, a replacement could be defined with any randomized value. If you replace "echo" in the lex file with "foo" and then run the code: foo "hello world," it will echo the string given. However, if you try to run the code: echo "hello world", a syntax error will be thrown. The language no longer understands "echo" but treats the command "foo" as it would previously have treated "echo".

The first step of Polyscripting is to replace all the keywords within a lex file and scramble them to randomized strings. Since the source code will only run scrambled on the deployment server, the development code will all be written in the standard language. During the process of scrambling, a dictionary will also be built with the instructions to transform the source code to the matching scramble.

Scrambled Lex File

The result of scrambling these keywords is a language interpreter that understands only unique strings as its reserved keywords. While no longer understanding the original keywords. "Use" is now an unparsable command, but nhZjBhADl will be linked to the same functionality. Below is a snippet from the PHP lex file before and after scrambling.

```
<ST_IN_SCRIPTING>"require_once" {
    RETURN_TOKEN(T_REQUIRE_ONCE);
}
<ST_IN_SCRIPTING>"namespace" {
    RETURN_TOKEN(T_NAMESPACE);
}
<ST_IN_SCRIPTING>"use" {
    RETURN_TOKEN(T_USE);
}
<ST_IN_SCRIPTING>"insteadof" {
    RETURN_TOKEN(T_INSTEADOF);
}
<ST_IN_SCRIPTING>"global" {
    RETURN_TOKEN(T_GLOBAL);
}
<ST_IN_SCRIPTING>"isset" {
    RETURN_TOKEN(T_ISSET);
}
```

```
<ST_IN_SCRIPTING>"DjOoLJi" {
    RETURN_TOKEN(T_REQUIRE_ONCE);
}
<ST_IN_SCRIPTING>"DvFxkKF" {
    RETURN_TOKEN(T_NAMESPACE);
}
<ST_IN_SCRIPTING>"nhZjBhADl" {
    RETURN_TOKEN(T_USE);
}
<ST_IN_SCRIPTING>"eEvjMu" {
    RETURN_TOKEN(T_INSTEADOF);
}
<ST_IN_SCRIPTING>"RrWrdyPApXtLc" {
    RETURN_TOKEN(T_GLOBAL);
}
<ST_IN_SCRIPTING>"KvCeJT" {
    RETURN_TOKEN(T_ISSET);
}
```

If a malicious actor was able to get a piece of code injected within a website that has been polyscripted, accessing that code will result in a syntax error. Not only does this stop the attack, but it also acts as a means of detection and notification for attacks.

Transforming Source Code

The process of scrambling the language is, by its very nature, similar to the process of transforming it. In order for an interpreter to understand the code it is parsing; it needs to be transformed to the proper scramble. While scrambling the interpreter a JSON file is also built that contains a dictionary of the tokens to the scrambled values. This dictionary of values will act as instructions to transform the application's source code. However, this dictionary does not sit on the server since scrambling and transforming take place prior to deployment. This effectively makes the transformation an irreversible operation for the attacker. Without the dictionary, the output is meaningless, and the attacker has no context.

Unlike varying types of encryption, there is no key or secret value necessary to understand the scramble or for the program to run properly. The default becomes the secure, Polyscripted state. After scrambling and transforming, the dictionary can be deleted and the Polyscripted code will still run identically to the language from which it was derived. Unlike obfuscation, Polyscripting isn't simply making source code more difficult for someone to read. A site with obfuscated code will still run the language normally, including injected code. Polyscripting scrambles the language itself; it changes the actual makeup of a language, the actual definitions contained within a language's pre-compiled source code.

Of course, it is worth noting that there are exceptions to this. Any dynamically generated code will need to go through the process of scrambling. That means, for example, if you are running WordPress and want to download a plugin, that plugin will not immediately be recognized. For security, you will need to install the plugin during the initial build of the site and before the scrambling process. Alternatively, the plugin can access the transformation dictionary directly during installation allowing for more flexibility in this process, but the co-location of the transformation dictionary and the application creates a new attack vector.

Instructions and Scrambling

The process of transformation traverses the source code of an app and uses the instructions to change the syntax to match the proper scramble. Much like the behavior of the interpreter will not be affected, the scrambling of the source code will not affect how the output and behavior of the code. The transformation only changes the way the way that tokens will be recognized by the parser.

```
root@22f91c7eca8c:/php/tests# bat vuln.php
       File: vuln.php
   1   #This is a vulnerable PHP file.
   2   <?php
   3   $superSecret = "!!!SecretMessage123!!";
   4   echo ">> ";
   5   $flexibleInput = fgets(STDIN);
   6   eval($flexibleInput);
   7
root@22f91c7eca8c:/php/tests# /php/php vuln.php
#This is a vulnerable PHP file.
>> echo "\n<<$superSecret>>\n"; # All your secrets belong to us.

<<!!!SecretMessage123!!>>
root@22f91c7eca8c:/php/tests# /php/php ../tok-php-transformer.php -p /php/tests/vuln.php
Polyscript from dir /php/tests/vuln.php to dir:/php/tests/vuln_ps.php
root@22f91c7eca8c:/php/tests# bat vuln_ps.php
       File: vuln_ps.php
   1   #This is a vulnerable PHP file.
   2   <?php
   3   $superSecret = "!!!SecretMessage123!!";
   4   ZivQazwxMBCS ">> ";
   5   $flexibleInput = fgets(STDIN);
   6   nGzVHz($flexibleInput);
   7
root@22f91c7eca8c:/php/tests# /polyscripted-php/bin/php vuln_ps.php
#This is a vulnerable PHP file.
>> echo "\n<<$superSecret>>\n"; # All your secrets belong to us.

Parse error: syntax error, unexpected '"' in /php/tests/vuln_ps.php(6) : eval()'d code on line 1
```

An interpreter parses a language by identifying the role of each part of the code. Given certain rules within the interpreter (in fact, the very rules that are changed during polyscritping) it is able to recognize and tokenize certain values. By using those exact rules contained within the interpreter the transformer simply parses each PHP file, but replaces the original token values with the scrambled ones provided by the instructions.

The language has a source of truth within it: its scanner and parser. If we use these exact methods to transform the language to the scrambled version, it ensures that it is being parsed exactly as it will be when being executed. Because of this the logic of the code does not change.

Put simply, the transformation process is done in such a way as to not affect the output of the code itself. Though the keywords are changing, the functionality of the instructions and the programming language remains the same.

PHP as Proof of Concept

Polyscripting is an elegant solution to a real problem. Polyverse's current R&D team is working on developing a usable open-source version of Polyscripting that scrambles PHP. The project is freely available on Github under an MIT license. The purpose of this project is to demonstrate a moving target defense strategy in a real and meaningful way. Polyverse strives to make cybersecurity

simple and manageable. PHP is only the first of many languages, and the team wants to apply the same simple concept to other programming languages.

This then begs the question: if the goal of Polyscripting is to apply the concepts across a wide spectrum of vulnerable server-side languages, why start with PHP? The answer is pretty simple: because people use it. Over a quarter of the internet is using WordPress to build out their websites. WordPress is—by a significant margin—the most used CMS in the world. All while being open source. It is also written in PHP. Not to mention the other CMS players that use PHP. Regardless of the critiques it endures, PHP is widely used because of this kind of popularity. It is also an open source interpreted language with a grammar and syntax that is accessible and easily manipulated, which is ideal for an open-source proof of concept like Polyscripting.

Its popularity also makes PHP a heavily targeted language. The previously mentioned exploits utilize PHP vulnerabilities to inject malicious code. To further compound the issue, millions of sites run antiquated versions of PHP that are no longer supported that contain well-known vulnerabilities. To update an entire code base is a task many are unable to take on due to a lack of resources, whether financial, chronological, or otherwise, subsequently leaving their product vulnerable to various threats.

PHP is the perfect language for demonstrating Polyscripting. Not only because of the ease of implementation and its widespread use, but because Polyscripting has the potential to solve meaningful problems that application's utilizing PHP encounter.

Polyscripted WordPress

Polyverse is a Gold level sponsor for the 2018 WordCamp conference in Seattle, WA. Though the main Polyverse product is the polymorphic version of Linux, Polyverse is sponsoring the event to showcase Polyscripting. It may seem like an odd choice given that Polyscripting does not relate to our keystone product, and it is an open-source tool. With the end goal being to move from theoretical concept, to actually stopping real-world attacks, we applied Polyscripting to WordPress so others could utilize our very latest security practices in tandem with one of their most commonly used tools.

It is an idea that is powerful even in its infancy, but as more people use and improve it, it has the potential to solve a significant problem.

To try out the WordPress demo and build a WordPress site leveraging Polyscripting as a defense mechanism checkout the open source repo: https://github.com/polyverse/ps-wordpress.

It is Polyverse's mission to create simple to use tools. With Polyscripting, WordPress can be deployed the same way as one might normally do so. This entails building out the source code, scrambling the language and code, and running it. The Polyscripted Wordpress container bundles all of this and makes deploying an instance of Polyscripted Wordpress just as effortless as utilizing the official Docker images to do so.

This is the most secure way of running Polyscripted WordPress.

However, even in this case, though not as secure, a site still reduces its attack surface and increases the effort it would takes to craft a successful code injection attack.

Conclusion

Polyscripting has the potential to be a powerful tool to defend against code-injection attacks. Though scrambling keywords is powerful, there are many other ways to increase the effectiveness of Polyscripting. Scrambling more than just keywords, but also built-in PHP functions, is a feature that would increase Polyscripting's effectiveness and is a likely addition in the near future. Similarly, scrambling more than the language tokens, but also the grammar and the Abstract Syntax Tree of the language will add an entirely new layer of security to any language Polyscripting is applied to. Polyverse is creating a new standard to expect from programming languages —Polyscripting capabilities.

For more information, contact

sales-us@polyverse.com

or visit our website

https://polyverse.com