The Hitchhiker's Guide to Malicious Third-Party Dependencies

Piergiorgio Ladisa, Merve Sahin, Serena Elisa Ponta, Marco Rosa, Matias Martinez, Olivier Barais

The Hitchhiker's Guide to Malicious Third-Party Dependencies

Piergiorgio Ladisa
SAP Security Research
Mougins, France
University of Rennes 1/INRIA/IRISA
Rennes, France
piergiorgio.ladisa@sap.com
piergiorgio.ladisa@irisa.fr

Merve Sahin
SAP Security Research
Mougins, France
merve.sahin@sap.com

Serena Elisa Ponta
SAP Security Research
Mougins, France
serena.ponta@sap.com

Marco Rosa
SAP Security Research
Mougins, France
marco.rosa@sap.com

Matias Martinez
Universitat Politècnica de Catalunya - Barcelona Tech
Barcelona, Spain
matias.martinez@upc.edu

Olivier Barais
University of Rennes 1/INRIA/IRISA
Rennes, France
olivier.barais@irisa.fr

ABSTRACT

The increasing popularity of certain programming languages has spurred the creation of ecosystem-specific package repositories and package managers. Such repositories (e.g., npm, PyPI) serve as public databases that users can query to retrieve packages for various functionalities, whereas package managers automatically handle dependency resolution and package installation on the client side. These mechanisms enhance software modularization and accelerate implementation. However, they have become a target for malicious actors seeking to propagate malware on a large scale.

In this work, we show how attackers can leverage capabilities of popular package managers and languages to achieve arbitrary code execution on victim machines, thereby realizing open-source software supply chain attacks. Based on the analysis of 7 ecosystems, we identify 3 install-time and 4 runtime techniques, and we provide recommendations describing how to reduce the risk when consuming third-party dependencies. We provide example implementations that demonstrate the identified techniques. Furthermore, we describe evasion strategies employed by attackers to circumvent detection mechanisms.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

Open-Source Security, Supply Chain Attacks, Malware Detection

ACM Reference Format:
Piergiorgio Ladisa, Merve Sahin, Serena Elisa Ponta, Marco Rosa, Matias Martinez, and Olivier Barais. 2023. The Hitchhiker's Guide to Malicious Third-Party Dependencies. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '23), November 30, 2023, Copenhagen, Denmark*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605770.3625212

SCORED '23, November 30, 2023, Copenhagen, Denmark

1 INTRODUCTION

Software modularization is a fundamental practice in modern software development, as it enables the division of complex systems into more manageable components and promotes software reusability. In this context, Open-Source Software (OSS) plays a central role by offering a wide range of pre-built and reusable modules that developers can leverage to enhance productivity. To streamline the use of OSS, package repositories and managers for specific programming languages have been established. These repositories serve as centralized databases where developers can easily discover, download, install, and manage open-source modules. On the client side, package managers are tools that assist downstream users in automatically handling the resolution and installation of required packages, including their dependencies. These mechanisms are popular among developers, but the full automation they provide can involve potentially risky processes that remain transparent to the users. In fact, package repositories have become a fruitful target for attackers seeking to propagate malware on a large scale [45, 46, 56]. OSS supply chain attacks are an increasing trend and demonstrated to happen through multiple attack vectors [28, 34]. The severity of this issue is highlighted by the suspension in May 2023 of new user registrations and package uploads on PyPI due to a significant increase in malicious activities [15].

Given the significance of OSS, which can account for up to 98% of codebases [31], and the widespread adoption of package management practices by both private and public organizations, enhancing the security of the software supply chain has become imperative. This is necessary to protect the interests of the community and ensure national security [22, 26, 42].

In this work, we investigate how to achieve Arbitrary Code Execution (ACE) through the distribution of malicious packages to downstream users. Moreover, we delve into the evasion techniques that attackers may employ to circumvent detection measures. We set out to answer the following research questions.

RQ1: How can 3rd-party dependencies, distributed via package managers, achieve ACE on downstream projects during the third-party package life-cycle phases (i.e., install, runtime)?

RQ2: What are the strategies adopted by attackers to evade the detection of malicious code?

To answer these questions, we first explore the functionalities of package managers for popular languages that allow to trigger execution at install-time and runtime. Then, we identify where attackers can hide malicious code to increase the chances of achieving its execution when the downstream project is built, tested, or run. Based on the analysis of 7 ecosystems, we identify 3 install-time techniques and 4 runtime techniques. Finally, we describe the various techniques employed by attackers to evade detection.

For the identified techniques, we provide practical guidance to downstream users on how to prevent ACE and to security analysts on how to analyze malicious packages. In addition, we release the source code of example implementations[1] available for the covered programming languages and their reference package manager, i.e., Python (pip), JavaScript (npm), Ruby (gem), PHP (composer), Rust (cargo), Go (go), and Java (mvn). This release aims to support fellow researchers in devising protective measures against such threats. We believe this may also help penetration testers to test whether the offensive techniques explored in this work may be achieved within their organizations.

The remainder of the paper is organized as follows. Section 2 provides background information on 3rd-party dependencies and OSS supply chain attacks. Section 3 presents our approach. Section 4 answers to RQ1, showing techniques to achieve ACE at install-time and runtime. Section 5 answers to RQ2, describing evasion techniques. Section 6 discusses related work. Finally, Section 7 presents our conclusions.

2 BACKGROUND

Modern software development extensively relies on 3rd-party dependencies. Dependencies are classified as *direct* when they are explicitly declared within a downstream application. In turn, direct dependencies may themselves have additional dependencies, referred to as *transitive* dependencies. Figure 1 shows the lifecycle of a 3rd-party dependency (both direct and transitive) from the perspective of a downstream project. We can distinguish two main phases: *install* and *run*. During the installation phase, the package manager on the client side fetches the dependency (i.e., a package) from the package repository and extracts it locally. If the package distribution is of *source* type (i.e., it contains only the source code), it is built for the target architecture of the downstream users. On the other hand, if the distribution is pre-built, the retrieved package can be used directly. Once the 3rd-party dependency is installed, it can be run within downstream projects as part of the main application or during the tests (in case of *test dependencies*).

OSS supply chain attacks encompass the insertion of malicious code into open-source components, enabling the distribution of malware to downstream consumers [28, 34]. To execute a successful OSS supply chain attack, the attacker must fulfill mainly three requirements [40]: (1) make a malicious package accessible to downstream users; (2) ensure that the downstream users actively engage with the malicious package (e.g., by installing it); (3) ensure that the downstream users eventually execute the malicious code contained within the package. While the taxonomy of attacks in [28] mostly covers the first two steps of this process, our work focuses on the last step, i.e., ensuring that downstream users eventually execute the malicious code.

[1] https://github.com/SAP-samples/risk-explorer-execution-pocs

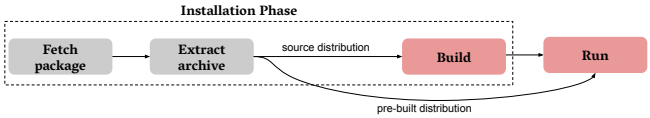

Figure 1: Lifecycle of a 3rd-party dependency (both direct and transitive) from the perspective of a downstream project.

Throughout this paper, we utilize the terminology Tactics, Techniques, and Procedure(s) (TTPs) as defined by the MITRE ATT&CK framework [7]. In our context, tactics refer to the main objectives of an attack, which in our work entails achieving ACE during installation time and runtime. Techniques encompass the specific methods employed by attackers, while procedures outline the practical implementation details followed by attackers.

3 APPROACH

To address both RQ1 and RQ2, we adopt the approach in Figure 2 and we rely on two primary data sources.

The first is the Backstabber's Knife Collection (BKC) [32], a dataset containing malicious packages from past attacks. This dataset spans various programming languages, i.e., JavaScript (npm), Python (pip), PHP (composer), Ruby (gem), and Java (mvn). We conduct a comprehensive manual analysis of these packages.

The second source is the Risk Explorer [29], which offers a comprehensive taxonomy and an extensive dataset of references related to OSS supply chain attacks. In this instance, we analyze grey literature materials to extract technical details from previous attacks, identifying any novel or advanced techniques that may not be present in the BKC.

To address RQ1, we first study the known attacks and malicious packages in [29] and [32] to identify the underlying root causes to achieve ACE. This involved examining the features provided by the language or the package managers that were exploited to achieve ACE by means of 3rd-party dependencies. To evaluate the presence and similarity of functionalities across diverse programming languages, we conduct a *comparative analysis of package managers' functionalities*. This examination involves scrutinizing the documentation of package managers associated with the selected programming languages to determine if similar functionalities exist and if they pose similar risks. In expanding this comparative analysis beyond the languages covered in [29] and [32], we include Go (go), chosen for its explicit focus on addressing supply chain attacks [48], and Rust (cargo), due to its increasing popularity [14, 50]. Upon identifying functionalities that could potentially lead to ACE, we proceed to develop a set of runnable implementations. These implementations provide minimal examples of each specific exploitation technique, showcasing the root cause of the issues and the different code locations susceptible to housing malicious content.

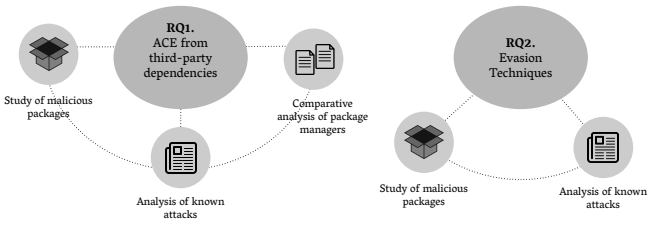

Figure 2: Approach followed to answer the research questions.

To address RQ2, we build upon the evasion techniques identified in [32] and supplement them with evasion strategies documented in the grey literature references from [29]. In this pursuit, we also refer to the literature on code obfuscation [19, 20, 38, 53] to identify additional evasion techniques that share similarities with those already exploited but have not been observed in the wild (to the best of our knowledge). We adopt the categorization of evasion techniques proposed by Schrittwieser et al. [38], which includes *data obfuscation*, *static code transformation*, and *dynamic code transformation*.

Our work presents an extensive overview of both observed and potential ACE mechanisms and evasion tactics. This information can help defenders in recognizing malicious packages and devising mitigation strategies within the software supply chain. However, we do not claim to be exhaustive, as there could be other (unknown) techniques for triggering the execution of malicious code and evading detection.

4 RQ1: ARBITRARY CODE EXECUTION STRATEGIES

In this section we answer RQ1, describing techniques that 3rd-party dependencies may employ to attain ACE when they are installed or run in the context of downstream projects. Building on Ohm et al.'s analysis [34], these techniques are enhanced through insights from grey literature and package manager documentation.

Table 1 provides a summary of the 7 techniques we have identified for achieving ACE. Table 2 maps these techniques to the selected languages/package managers and indicates whether each technique is applicable in a particular ecosystem. We provide example implementations for all these techniques.

4.1 (I) Install-Time Execution

We identify 3 techniques to achieve ACE when downstream users/projects install a 3rd-party dependency using popular package managers (i.e., npm, composer, pip, and gem).

(I1) Run command/scripts leveraging install-hooks [34]. This technique involves the execution of code by hooking the install process of 3rd-party dependencies in different stages, using specific keywords that package managers may provide.

JavaScript (npm). In Node.js, the *package.json* file contains both metadata information (e.g., name, version) and the list of dependencies related to a project [10]. It also provides installation hooks through the *scripts* property: keys for different lifecycle phases (e.g., *pre-install*) can be used to trigger the execution of the scripts provided as values [10]. The *package.json* file is processed by the package manager to install and resolve the dependencies for a specific package. During the installation process, scripts are executed according to their definition in the corresponding property to perform additional actions (e.g., to compile code) [52].

Procedure. To achieve ACE when installing a package through `npm install`, the package has to leverage the installation hooks, namely: *pre-install*, *install*, *post-install*, *preprepare*, *prepare*, *postprepare*, and *prepublish* (deprecated). An example is shown in Listing 1. At this stage, attackers have the possibility to directly execute malicious shell commands, or to invoke external scripts which must be included within the package.

Listing 1: (I1) Example implementation for JavaScript using installation hooks in package.json

```
{
 "name": "example",
 "version": "1.0.0",
 ... continues ...
 "scripts": {
  "pre-install": "** COMMANDS **"
 }
}
```

Recommendation(s). The `npm install` command provides the `--ignore-scripts` option, to avoid the execution of any script during installation [9]. If this solution is not viable, it is important to carefully review the content of the installation scripts. This review should be performed for all dependencies being installed, both direct and transitive. However, considering the large number of dependencies that can be required for a single npm package, automation is necessary. Currently, there are no ready-made solutions to address this problem, but academia has started proposing solutions [35, 52] to mitigate install-time attacks for JavaScript (npm). Still, works discussing malicious packages in npm [33, 35, 40, 52] primarily discuss *pre-install*, *install*, and *post-install* hooks, rather than extensively exploring the abovementioned hooks.

PHP (composer). In PHP, the popular package manager is *Composer*, which supports two types of package distributions: *dist* and *source* packages. Dist packages are pre-built packages distributed in a binary format. When installing a dist package, Composer skips the build process and directly uses the pre-built package. By default, dist packages are consumed over source packages. Source packages contain the source code of a package and must be built by the client. The *composer.json* file (equivalent to *package.json* for npm packages) contains the build instructions and offers installation hooks, using the *scripts* property, to execute additional commands within the installation process.

Procedure. To achieve ACE when installing a package using the `composer install` command, the following installation hooks can be defined in the `script` property of the *composer.json* file [12]: *pre-install-cmd*, *post-install-cmd*, *pre-autoload-dump*, and *post-autoload-dump*. The implementation is similar to the case of JavaScript (cf. Listing 1). If the package does not include the *composer.lock* file, which records the exact versions of the installed dependencies, the additional installation hooks *pre-update-cmd* and *post-update-cmd* will be executed during the installation (otherwise, they are executed only when `composer update` is run). As for npm packages, attackers may directly insert shell commands in *composer.json*, or invoke external scripts which must be included in the package.

How to achieve Arbitrary Code Execution

Package Life-Cycle Phase	Technique	Example
(**I**) Install-time	(**I1**) Run command/scripts leveraging install-hooks [34]	In JavaScript (npm), insert a shell command as value of the key `pre-install` in *package.json* (cf. Listing 1).
	(**I2**) Run code in build script [34]	In Python, insert code in *setup.py* (cf. Listing 2).
	(**I3**) Run code in build extension(s)	In Ruby (gem), insert code in build extension, such as the *extconf.rb* file (cf. Listing 5).
(**R**) Runtime	(**R1**) Insert code in methods/scripts executed when importing a module [34]	In Python, insert code in *__init__.py* file (cf. Section 4.2).
	(**R2**) Insert code in commonly-used methods	In Python, insert code in `setup()` method of `distutil.core`.
	(**R3**) Insert code in constructor methods (of popular classes)	In Python, insert code in `Dataframe()` constructor method of package typosquatting `pandas` package.
	(**R4**) Run code of 3rd-party dependency as build plugin	In Java (mvn), insert code in maven plugin that is executed when building downstream project.

Table 1: Techniques that a 3rd-party dependency may use to achieve ACE on downstream during its lifecycle.

	ACE Technique(s)						
	Install-time			Runtime			
Ecosystem	**I1**	**I2**	**I3**	**R1**	**R2**	**R3**	**R4**
JavaScript (npm)	✓			✓	✓	✓	
Python (pip)		✓		✓	✓	✓	
PHP (composer)	✓				✓	✓	
Ruby (gem)			✓	✓	✓	✓	
Rust (cargo)		✓			✓	✓	
Go (go)				✓	✓	✓	
Java (mvn)					✓	✓	✓

Table 2: Comparative analysis about applicability of available techniques per each language (and related package manager). Empty cells in the table indicate that a particular technique is not applicable to a specific language. We provide implementations for all techniques for each programming language.

Recommendation(s). Unlike `npm install`, the `composer install` command does not have an option to skip the execution of any script [4]. As mentioned earlier, the hooks *pre-update-cmd* and *post-update-cmd* are skipped if *composer.lock* exists. The *pre-autoload-dump* and *post-autoload-dump* scripts can also be skipped by using the *--no-autoloader* option. However, there is no built-in way to skip *pre-install-cmd* and *post-install-cmd* when using `composer install`. Therefore, it is necessary to examine the content of such installation hooks to ensure they do not execute any malicious code. Moreover, it would be important to evaluate the extension to Composer of approaches like [35, 52]. Recall that the aforementioned procedure only applies to source packages. Thus, it is crucial to prioritize dist packages for all required dependencies (both direct and transitive) and promote their availability.

(I2) Insert code in build script [34]. This technique involves the execution of code contained in scripts used by package managers during the installation of 3rd-party dependencies distributed as source code (as they need to be built before usage).

Python (pip). Python packages may be distributed as source or binary distributions. Source distributions (*sdists*) include an installation script, the *setup.py* file, to define the metadata (e.g., name, version), configurations for building and packaging, and additional actions that may be required. The *pip* package manager uses *setup.py* as a source of information to install and manage Python *sdists* packages. When running `pip install`, the *pip* tool executes the *setup.py* file of the package being installed.

Procedures. To achieve ACE when installing a package through `pip install`, an attacker can directly add malicious Python commands to the *setup.py* file. An example is shown in Listing 2 (malicious instruction in line 4).

Listing 2: (I2) Example implementation for Python sdist packages through code in setup.py

```
1 from setuptools import setup
2
3 # Any Python code will be executed, for example:
4 import os; os.system("..COMMANDS..")
5 setup(name='foo',version='1.0', ...)
```

Another way to achieve ACE when installing a package through `pip install` is to leverage the `cmdclass` property that allows the customization of the tasks performed. Commonly used cmdclass commands available in setuptools include `install` and `build`. Listing 3 demonstrates how to customize the install command class to obtain ACE at installation time. First, it is necessary to import the `install` method from the `setuptools.command` module (cf. line 2). Then, a subclass of `install` (`ExampleClass`, line 4) is created inside the *setup.py* file, that must implement the `run` method (line

6), which is executed by default at package installation. Malicious code inside the `run` method is thus executed.

Listing 3: (I2) Example implementation for Python sdist packages through cmdclass commands in setup.py

```
from distutils.core import setup
from setuptools.command.install import
    install  # Required import

class ExampleClass(install):
 def run(self):
  install.run(self)
  # Any Python code will be executed, for example:
  import os; os.system("**COMMANDS**")

setup(name='foo', ..., cmdclass='install': ExampleClass)
```

Recommendation(s). Achieving ACE primarily depends on consuming source distributions of Python packages for which the installation script is executed during `pip install`. On the contrary, pre-built packages (i.e., *bdists*) do not include installation scripts, nor execute code during the installation process. The challenge with bdists is that they need to be produced for each target architecture, i.e., a different version of the same Python package has to be built and published for each target architecture possibly willing to install it. Whenever bdists are available, they are the default choice of package managers. Therefore, when selecting a package to install, packages with bdist distributions should be prioritized. The same logic must be applied to all the direct and transitive dependencies as they are transparently installed together with the selected package. `pip` allows to ignore sdist dependencies via the option `--only-binary :all:` [11]. Yet, packages without binary distributions will fail to install, possibly preventing the successful outcome of the installation process. If sdists are required (either directly or indirectly), it is crucial to verify whether they include a *setup.py* file and ensure that it does not contain any malicious code.

Rust (cargo). In Rust, the *Cargo.toml* file is used to specify package metadata as well as its direct dependencies. Cargo (i.e., package manager for Rust) uses such a file when running `cargo install`. To install a package, cargo also builds the package itself as well as all its direct and transitive dependencies. In addition, cargo provides the flexibility to include build scripts within a package that are compiled and executed just before the package is built [1]. By default, the cargo build system will search for a `build.rs` script in the root directory of the project. This behavior can be overridden by specifying a different path to the build script in the `build` field of the `Cargo.toml` configuration file. Such a feature allows to perform tasks such as building 3rd-party non-Rust code.

Procedure. To achieve ACE when installing a package through `cargo install`, the attacker can include a malicious build script in a package (i.e., a crate) they distribute, that will eventually be executed by the cargo build system. Listing 4 shows how to trigger the execution of the commands in line 5 using the *build.rs* script.

Listing 4: (I2) Example implementation for Rust leveraging *build.rs*

```
use std::process::Command;

fn main() {
 # Any arbitrary Rust code can be executed, for example:
  let output =
     Command::new("sh").arg("-c").arg("**COMMANDS**").output();
}
```

Recommendation(s). To ensure the security of consumed 3rd-party dependencies, it is recommended to check for the presence of build scripts, i.e., *build.rs* or the one specified within the `build` field in the *Cargo.toml* file. This applies to both direct and transitive dependencies. Reviewing the content of these scripts is crucial to identify and mitigate potential malicious code.

(I3) Run code in build extension(s). This technique involves executing extensions of 3rd-party dependencies that are necessary for their build process.

Ruby (gem). The *.gemspec* file contains the metadata and dependencies for a Ruby package (i.e., a gem). Such a file is used by the RubyGems package manager to install, build, and distribute a package. Gems may include extensions that are built and executed at installation time (i.e., when running `gem install`) [6]. Note that extensions are also executed when manually building a 3rd-party dependency through the `gem build` command.

Procedure. To achieve ACE when installing a package through `gem install`, two files needs to be manipulated: the *.gemspec* and extension files. As shown in Listing 5, an attacker may define a build extension in *.gemspec* (line 5 in Listing 5a) and include it in the gem. Malicious code in the extension file will be executed (line 4 in Listing 5b).

Listing 5: (I3) Example implementation for Ruby gems leveraging build extensions

```
Gem::Specification.new do |s|
s.name        = "example"
s.version     = "1.0.0"
... continues ...
s.extensions = ["extconf.rb"]
end
```

(a) Content of the *.gemspec* file for the project

```
require "mkmf"

# Any arbitrary Ruby code will be executed, e.g.:
exec("**COMMANDS**")

# Needed to finish the extension without errors
create_makefile("")
```

(b) Content of *extconf.rb* file

Recommendation(s). The `gem install` command do not provide an option to ignore extensions. In order to prevent such a scenario, gems should be checked to verify if they declare extensions in *.gemspec*. If present, extensions should be analyzed to verify that they do not contain malicious code. Such a review has to be performed both for direct and transitive dependencies.

4.2 (R) Runtime Execution

Malicious code can be executed at runtime through various techniques. We identify four scenarios (cf. Table 1) where this is more likely to occur: compromising the code executed during the import

of external modules, manipulating popular methods, manipulating constructor methods, or leveraging build plugins.

(R1) Insert code in methods/scripts executed when importing a module [34]. Certain programming languages execute code when an import statement is processed, even before the code from the imported module is actually used (for Javascript, Python, and Ruby even if the imported module is never used). Thus, runtime execution of 3rd-party code may occur when developers import an external module. Below, we examine the procedures for programming languages susceptible to this technique.

JavaScript (npm). In Node.js, the `main` attribute [8] in the *package.json* file determines the entry point script (e.g., `index.js`) for a package when it is imported into an application (e.g., using `require`). Attackers can inject malicious code into the entry point script such that it will be executed at runtime when the 3rd-party module is imported, thus achieving ACE.

Python (pip). Regular packages in Python are typically implemented as a directory containing an `__init__.py` file [13]. Attackers may insert malicious code inside these files since their execution is triggered implicitly when packages are imported using the `import` statement.

Ruby (gem). Ruby automatically executes code at runtime upon a `require`, `require_relative` or `load` statement. Thus, attackers may insert malicious code within the `.rb` file from which the module is imported.

Go (go). There are several precautions taken by Go against software supply chain attacks [48]. It does not run any code on installation and there are no installation hooks, preventing the techniques of Section 4.1. Moreover, unlike other programming languages analyzed in this work, Go lacks a centralized package repository. Instead, packages are directly downloaded from their source code repositories. In Go, dependencies can execute code upon import in two ways: *(i)* by defining an `init()` method [5, 24], and *(ii)* by initializing a variable with an anonymous function [24]. This allows the code of a 3rd-party dependency to be executed with higher order of precedence, before the code of the importing application. Additionally, Go automatically removes unused dependencies by default, but prefixing the import with an underscore symbol (e.g., `_ "foo"`), retains even unused dependencies. Attackers can exploit these techniques to craft a malicious dependency that achieves ACE at runtime during its import [25].

(R2) Insert code in commonly-used methods. Attackers may target popular methods within 3rd-party dependencies that they distribute to downstream users. These methods are attractive because they are commonly used, increasing the likelihood that downstream users will invoke and rely on them. For example, this technique is used in the Java malicious package `com.github.codingandcoding:servlet-api-3.2.0` [32, 44], where the malicious code is contained in the `doGet()` method of the `HttpServlet` class.

(R3) Insert code in constructor methods (of popular classes). Attackers may target constructor methods as suitable places to insert malicious code. In fact, constructors are called during object instantiation, making them potential entry points for malicious activities. For instance, in a typosquatted version of the Python `pandas` package [28, 34], malicious code could be included in the constructor methods of the widely-used class `Dataframe()`.

Regarding Java, it is also noteworthy to mention that attackers may not only include malicious code in constructor methods but also in instance and static initializers [3]. Instance initializers are executed every time a class instance is created [2]. Static initializers are automatically executed (just once) when a class is initialized, i.e., when a class instance is created, a static method of the class is called, or a static field of the class is assigned or used (excluding constants) [2].

(R4) Run code of 3rd-party dependency as build plugin. This technique consists of running a 3rd-party dependency as a plugin within the build of a downstream project.

Java (mvn). In Maven, plugins enable developers to define tasks to be performed. Plugins are defined in the Maven configuration file of the project (i.e., *pom.xml*) and they can be bound to phases of the build process (e.g., compile, test, package), thus customizing and extending its functionality. Attackers can inject malicious code into Maven plugins as a means to achieve ACE during the build process or to infect the built artifact to spread malicious code. For example, this technique is exploited by the malicious Java package `com.github.codingandcoding:maven-compiler-plugin-3.9.0` [44].

Remarks. When using 3rd-party dependencies in the tests of a project, the same techniques described in Section 4.2 to achieve ACE apply. Additionally, attackers may ship packages with malicious code in the test routines [34]. This is out of scope in our work, since we focus on the execution of malicious code contained in 3rd-party dependencies during the installation, build, test, or runtime of downstream projects. As a result, malicious code in tests of 3rd-party dependencies is not executed as those tests are not executed in any stage of the lifecycle of the downstream project.

From the attacker's viewpoint, install-time techniques provide greater advantages than runtime techniques. In the former, victims simply need to install the package to initiate the malicious code execution, which might explain the prevalence of malicious packages executing code at install-time [34, 55]. Conversely, with runtime techniques, victims not only have to install the third-party dependency but also actively trigger the carrier of the malicious code (e.g., invoking the constructor as demonstrated in the R2 technique).

5 RQ2: EVASION TECHNIQUES

In this section, we present various techniques that aim to make the detection of malicious source code more challenging. We classify these techniques into the three categories of software obfuscation — namely, *data obfuscation*, *static code transformation*, and *dynamic code transformation* — as proposed by Schrittwieser et al. [38].

The covered techniques include both those that have been observed in real-world attacks (e.g., in BKC [32]) and those that are theoretically viable based on state-of-the-art techniques in code obfuscation [19, 20, 38, 53]. Such techniques can also be combined and used in conjunction to further challenge the detection and analysis of malicious code.

While we strive for comprehensiveness, it is important to note that this list is not exhaustive and may evolve over time as new techniques emerge.

5.1 Data Obfuscation

Malicious code often incorporates hard-coded strings that serve various purposes for attackers. These strings can include, e.g., URLs or IP addresses that point to attacker-controlled servers, shell commands, or paths to sensitive files [30, 41]. Since the analysis of these strings can provide insights into the attacker's infrastructure, techniques, and intended actions, attackers often employ obfuscation techniques to evade detection and conceal information that could lead to their identity. Thus, in this category, we encompass techniques that alter the way static data (i.e., strings) is stored within source code to conceal it from analysis and enhance its complexity [20, 38].

Encoding. One commonly used technique involves encoding strings in non-human-readable formats (e.g., base64, hex) and decoding them at runtime [34]. From an attacker's perspective, this technique is easy to implement and can effectively evade detection through simple string scanning techniques that look for sensitive words (e.g., `bash`). However, strings encoded in common formats are relatively easy to de-obfuscate.

Compression. Attackers can employ compression algorithms (e.g., *gzip*) to obfuscate strings [34]. The compressed strings would then be decompressed at runtime to retrieve their original content.

Encryption. Attackers may encrypt strings using common algorithms (e.g., AES) and decrypt them at runtime [34]. This approach increases the complexity of the analysis process, as the encrypted strings are not directly readable and require the decryption key to reveal their original content. The key used for decryption can be handled in different ways. In some cases, it may be stored locally within the package itself, making it relatively easy for security analysts to extract and decrypt the strings. To mitigate this risk, attackers may choose to retrieve the decryption key from a remote server at runtime. This approach adds an additional layer of complexity, as the decryption key is not readily available in the package. By relying on a remote server, attackers can dynamically control the availability and distribution of the decryption key, making it more challenging for security analysts to access and decrypt the strings.

Binary Arrays. A more advanced technique is to represent strings in a binary form and store them in binary arrays. Binary arrays provide a convenient way to store and manipulate binary data in a programmatic way. By leveraging binary arrays, attackers can perform operations such as bitwise operations, XOR operations, or custom encoding schemes to further obfuscate the strings [39]. To effectively detect and analyze such procedures, security practitioners need to employ advanced techniques capable of handling binary data and apply reverse engineering methods.

Reordering of Data. Another simple and effective technique is to split data into multiple pieces and re-aggregate them at runtime [20, 38]. Attackers can manipulate strings, such as URLs or shell commands, by splitting them into multiple chunks (e.g., assigned to multiple variables) and then concatenating them before providing them to the respective APIs or functions [34]. By breaking down the strings into smaller fragments and reassembling them dynamically, attackers can evade straightforward pattern matching techniques [38] that rely on static scanning of strings for detection. Security analysts need to employ more advanced detection that can handle dynamic string manipulation to effectively mitigate such attacks.

5.2 Source Code Transformations

To make it challenging for security analysts to understand the purpose and inner workings of malware, attackers employ various techniques to obfuscate the source code. The obfuscation process aims to create a complex and convoluted code structure that hinders reverse engineering and analysis.

Static Code Transformations. In this category, we cover techniques involving the transformation of source code that do not necessitate additional runtime modifications for execution [38].

Renaming Identifiers. To decrease the readability of code, attackers may employ the technique of renaming identifiers such as variable and function names [39]. By changing the names of these identifiers to arbitrary or nonsensical values, attackers make it challenging for analysts to understand the purpose and functionality of the code. An example of such a technique is shown in Listing 6.

Listing 6: Obfuscated code in the *setup.py* of the package *maratlib-0.2* [32]

```
import sys
l1l_cringe_ = sys.version_info [0] == 2
l1111_cringe_ = 2048
l11_cringe_ = 7
def l111_cringe_ (l111_cringe_):
 global l11l1_cringe_
 l111_cringe_ = ord (l111_cringe_ [-1])
 ll_cringe_ = l111_cringe_ [:-1]
 l111_cringe_ = l111_cringe_ % len (ll_cringe_)
 l1_cringe_ = ll_cringe_ [:l111_cringe_] + ll_cringe_ [
    l111_cringe_:]
... continues ...
```

Dead/Useless Code Insertion. Attackers may insert dead or useless code into their malicious code (e.g., `mplatlib-1.0` [32]) to deceive security analysts during the reverse engineering process and to make the latter more time-consuming [39, 54].

Dead code refers to portions of code that are intentionally added but never executed during the runtime of the program. It serves no functional purpose and may contain instructions or logic that are irrelevant to the actual operation of the malware.

Useless code, on the other hand, refers to code that may be syntactically correct and executable but serves no meaningful purpose in terms of the malware's functionality. As an example, it may consist of redundant or duplicated code, excessive comments, or superfluous operations.

Split Code into Multiple Files. Attackers may employ the technique of splitting malicious code into multiple files within the same package to obfuscate their activities and make detection and analysis more challenging. By distributing the malicious code across multiple files, it becomes harder for security analysts to identify and understand the complete scope of the malicious functionality.

Similarly, attackers employ the technique of leveraging second-stage payload(s) [34]. Instead of directly including the malicious code within the package, attackers only provide the initial code for fetching the actual malicious payload from external sources. By utilizing a remote server, attackers gain dynamic control over the availability and distribution of the second-stage payload, which increases the difficulty for security analysts to access the actual malicious code. This approach makes more complex the analysis process, as the initial code alone does not reveal the full extent of the malicious activities.

Hide Code into Dependency Tree. Another obfuscation technique used by attackers is the hiding of malicious code within the dependency tree of software packages. This technique involves including the actual malicious code in either the direct dependencies (e.g., as happened in the case of `event-stream` [27]) or the transitive dependencies (which is more effective) of the final package that will be distributed to downstream users in an OSS supply chain attack. Where to place the malicious code within the actual malicious dependency is explored in Section 4.

By embedding the malicious code deep within the dependency tree, attackers attempt to evade detection, as security scanning tools and analysts may have difficulties in thoroughly scanning every single dependency.

Split Code into Multiple Dependencies. Instead of placing all the malicious code in a single package, attackers may distribute it across multiple dependencies within the package ecosystem. This approach aims to make it more difficult to detect and analyze the malicious code since it is scattered across different packages. To the best of our knowledge, this technique has not been observed (yet) in real-world attacks.

In theory, a comprehensive scan of packages and their relationships within package repositories could potentially uncover such technique. In practice, this is challenging due to the vast number of packages and the continuous influx of new versions and updates.

Visual Deception. Attackers may employ visual deception techniques to make manual code review more challenging. One such technique involves adding excessive spaces, tabs, or other forms of whitespace to the code [43]. Compared with the other obfuscation techniques, this one capitalizes on human visual processing, as the excessive spaces can make it difficult for reviewers to spot anomalies or suspicious code patterns.

Another technique involves the use of Unicode homoglyphs or control characters [17]. The intent is to visually camouflage the malicious code, making it blend in with the surrounding code and potentially bypass casual inspection[2].

Polyglot Malwares and In-Line Assembly. Polyglot malwares employ multiple programming languages within a single malware instance. By combining different programming languages, attackers can exploit the unique characteristics of each language, thereby increasing the complexity of their malware. Moreover, certain programming languages allow the inclusion of assembly instructions directly in the source code. In-line assembly provides a means for direct interaction with system components and precise control over system resources. To the best of our knowledge, also this technique has not been observed (yet) in real-world attacks.

[2]This issue has started being addressed by some IDEs and compilers

Effectively analyzing and detecting the behavior of malware that employs multiple languages or incorporates in-line assembly requires advanced skills and expertise from security analysts and researchers. It demands a deep understanding of the intricacies of various programming languages and the low-level operations of the underlying system. Furthermore, traditional security mechanisms and tools may struggle to cope with the complexity introduced by polyglot malwares and in-line assembly.

Dynamic Code Transformations. In this category, we encompass techniques that involve transforming the code at runtime before its execution [38]. These techniques are not detectable through static analyzers.

Encoding, Compression and Encryption. Similar to strings, attackers can transform the malicious code itself using encoding, compression, or encryption techniques [34]. At runtime, the code is then decoded, decompressed or decrypted back to its original form before being executed. This is similar to the concept of *packing* for binary malwares [38, 47].

Steganography. Steganography consists of hiding information by embedding it within other data, such as images, audio files, or even seemingly harmless text files. Attackers may employ steganography as a technique to conceal malicious code within innocuous-looking files, as it happened with the case of the package `apicolor-1.2.4` in PyPI [18].

Dynamic Code Modification. Malicious 3rd-party dependencies can manipulate the behavior of commonly used methods by developers (e.g., built-in functions of a language like `System.out.println` in Java [23]) before their execution [38]. This tactic not only hides the invocation of malicious behavior within apparently harmless calls in the downstream application but also heightens the likelihood of such malicious code being triggered by the victim.

In dynamically typed languages (e.g., Python, JavaScript, Ruby), this functionality is already supported through monkey patching [37]. Listing 7 illustrates an instance of monkey-patching in Python, where the built-in function `print` is tampered with to execute shell command(s). In statically typed languages (e.g., Rust, Go), achieving this goal can be less straightforward and one possible way is through function/API hooking [49]. In the Java programming language, attackers can insert malicious code during runtime by altering the bytecode of a trusted Java class file that is utilized by downstream users. This could involve exploiting the Java instrumentation API or manipulating the classloading mechanism [51].

Listing 7: Example of monkey-patching the built-in function `print` in Python to make it execute shell command(s)

```
import os, builtins

original_print = print
def hacked_print(self):
 original_print(self)
 os.system("..COMMANDS..")
builtins.print = hacked_print
```

Remarks on warning suppression. To avoid alerting the victim or halting program execution due to exceptions, malware developers often employ a technique where they enclose the malicious code within a try block and associate it with an empty catch block. This technique is used to suppress any exception that may be thrown

during the execution of the malicious code [30]. In programming languages like Python and Java, the try-catch construct allows developers to handle exceptions gracefully and perform appropriate actions when an exception occurs. However, in the case of malware, the catch block is deliberately left empty, effectively silencing any exceptions that may occur.

By using an empty catch block, the malware ensures that any exceptions raised during its execution are not propagated or displayed to the user. This helps maintain stealth and prevents any error messages or abnormal program behavior that may alert the victim or raise suspicion.

From a security standpoint, the presence of empty catch blocks in code should raise suspicion and indicate potentially malicious activities. It is important for security analysts and developers to be vigilant and thoroughly analyze code for such suspicious constructs during code reviews and security assessments.

6 RELATED WORK

In this section, we present related works that focus on the security aspect of package managers in the context of OSS supply chain attacks.

Ohm et al. [34] analyze OSS supply chain attacks, investigating malicious packages in npm, PyPI, and RubyGems. They develop an attack tree, outlining techniques used in known malicious packages to trigger execution in software life cycle phases. Our work goes beyond existing malicious examples by investigating additional potential techniques and analyzing a wider range of programming languages.

Ladisa et al. [28] build upon Ohm et al.'s work [34] and conduct a systematic study to create a comprehensive taxonomy of how attackers inject malicious code into OSS. However, this study does not address the actual malicious content of packages. Our work complements this by analyzing the different execution and evasion techniques leveraged by malicious packages.

Okafor et al. [36] provide a systematic analysis of secure software supply chain patterns. In particular, they present a framework consisting of four stages of a software supply chain attack (i.e., compromise, alteration, propagation, and exploitation) and introduce three fundamental security properties (i.e., transparency, validity, and separation). Following their terminology, our work aims to understand the techniques used by attackers on package managers, in the compromise stage of an OSS supply chain attack.

Zimmerman et al. [57] examine the threats and risks faced by users of the npm ecosystem by investigating package dependencies, maintainers, and publicly reported security issues. In a similar fashion, Bagmar et al. [16] conduct a related study on the PyPI ecosystem. In contrast to these works, our work focuses on analyzing how dependencies can achieve code execution, not limited to npm or PyPI but also encompassing ecosystems with similar features. By exploring such mechanisms, we aim to provide a broader understanding of security risks across different ecosystems and provide runnable examples.

Duan et al. [21] examine the functionalities of npm, PyPI, and RubyGems, proposing a framework to assess functional and security features and detect malicious packages using program analysis. While their framework has a broad scope (e.g., covering authentication, signing features), our work has a practical approach on package manager features that can lead to ACE. Additionally, we extend our analysis to include other package managers (e.g., Composer, Cargo).

Wyss et al. [52] investigate the potential exploitation of install-time features in npm by attackers, and propose Latch as a solution. While they specifically focus on the *pre-install*, *install*, and *post-install* hooks, our work extends the analysis to identify additional installation hooks provided by npm. Furthermore, we explore similar attack vectors in other ecosystems beyond npm, broadening the scope of our investigation.

Zahan et al. [55] focus on npm, identifying six security weakness signals in software supply chain attacks. They analyze packages at scale to measure these signals and survey 470 package developers to assess their significance. While we share a focus on package manager security, our offensive approach explores how attackers exploit package managers across ecosystems for code execution, studying also evasion techniques.

7 CONCLUSIONS

In this work, we analyze 7 ecosystems and identify 7 different techniques usable by malicious 3rd-party dependencies to achieve ACE in downstream projects at install-time and runtime. Example implementations are provided to aid researchers and analysts in developing countermeasures. We also investigate evasion techniques employed by attackers to challenge detection.

In future work, we plan to extend our analysis to other ecosystems, develop recommendations for downstream users and security analysts, and conduct a study on developers' experiences when installing and using open-source packages. Moreover, we would like to further develop our analysis of programming languages in hybrid scenarios, like the usage of C extensions in Python and Ruby or the usage of the Java Native Interfaces in Java. Finally, it is relevant to perform a large-scale analysis that aims to estimate how significant the different techniques of ACE are for each ecosystem or the practical implications of using some of the countermeasures (e.g., how many installations fail in Python using the `--only-binary :all:` option).

Acknowledgements. We thank the reviewers for their constructive feedback, which has greatly contributed to the improvement of our work. This work is partly funded by EU grants No. 952647 (AssureMOSS) and No. 101120393 (Sec4AI4Sec).

REFERENCES

[1] [n. d.]. Build Scripts - The Cargo Book. https://doc.rust-lang.org/cargo/reference/build-scripts.html. [Accessed 30-Jun-2023].
[2] [n. d.]. Chapter 12. Execution — docs.oracle.com. https://docs.oracle.com/javase/specs/jls/se20/html/jls-12.html [Accessed 28-08-2023].
[3] [n. d.]. Chapter 8. Classes — docs.oracle.com. https://docs.oracle.com/javase/specs/jls/se20/html/jls-8.html. [Accessed 28-08-2023].
[4] [n. d.]. Command-line interface / Commands - Composer -- getcomposer.org. https://getcomposer.org/doc/03-cli.md#install-i. [Accessed 30-Jun-2023].
[5] [n. d.]. Effective Go - The Go Programming Language. https://go.dev/doc/effective_go. [Accessed 30-Jun-2023].
[6] [n. d.]. Gems with Extensions. https://guides.rubygems.org/gems-with-extensions. [Accessed 30-Jun-2023].
[7] [n. d.]. MITRE ATT&CK; -- attack.mitre.org. https://attack.mitre.org/. [Accessed 30-Jun-2023].

[8] [n. d.]. Modules: Packages. https://nodejs.org/api/packages.html. [Accessed 30-Jun-2023].

[9] [n. d.]. npm-install - npm Docs. https://docs.npmjs.com/cli/v9/commands/npm-install. [Accessed 30-Jun-2023].

[10] [n. d.]. package.json - npm Docs. https://docs.npmjs.com/cli/v8/configuring-npm/package-json#scripts. [Accessed 30-Jun-2023].

[11] [n. d.]. pip install - pip documentation v23.1.2 -- pip.pypa.io. https://pip.pypa.io/en/stable/cli/pip_install/#cmdoption-only-binary. [Accessed 30-Jun-2023].

[12] [n. d.]. Scripts - Composer -- getcomposer.org. https://getcomposer.org/doc/articles/scripts.md#scripts. [Accessed 30-Jun-2023].

[13] [n. d.]. The import system. https://docs.python.org/3/reference/import.html. [Accessed 30-Jun-2023].

[14] 2022. Stack Overflow Developer Survey 2022 -- survey.stackoverflow.co. https://survey.stackoverflow.co/2022/#technology-most-loved-dreaded-and-wanted. [Accessed 30-Jun-2023].

[15] 2023. PyPI new user and new project registrations temporarily suspended. https://status.python.org/incidents/qy2t9mjjcc7g?u=l1b53kd6n2rs. [Accessed 30-Jun-2023].

[16] Aadesh Bagmar, Josiah Wedgwood, Dave Levin, and Jim Purtilo. 2021. I know what you imported last summer: A study of security threats in thepython ecosystem. *arXiv preprint arXiv:2102.06301* (2021).

[17] Nicholas Boucher and Ross Anderson. 2023. Trojan Source: Invisible Vulnerabilities. (2023).

[18] Check Point Research. 2022. Check Point CloudGuard Spectral exposes new obfuscation techniques for malicious packages on PyPI. https://research.checkpoint.com/2022/check-point-cloudguard-spectral-exposes-new-obfuscation-techniques-for-malicious-packages-on-pypi/. [Accessed 30-Jun-2023].

[19] C.S. Collberg and C. Thomborson. 2002. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering* 28, 8 (2002), 735–746. https://doi.org/10.1109/TSE.2002.1027797

[20] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. *A taxonomy of obfuscating transformations.* Technical Report. Department of Computer Science, The University of Auckland, New Zealand.

[21] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[22] ENISA. 2022. ENISA Threat Landscape 2022. https://www.enisa.europa.eu/publications/enisa-threat-landscape-2022. [Accessed 30-Jun-2023].

[23] Geek. 2010. Tricky use of static initializer in Java - Override println — geekexplains.blogspot.com. http://geekexplains.blogspot.com/2009/05/tricky-use-of-static-initializer-in.html. [Accessed 28-08-2023].

[24] Paulo Gomes. 2019. Golang: stop trusting your dependencies! https://itnext.io/golang-stop-trusting-your-dependencies-a4c916533b04. [Accessed 30-Jun-2023].

[25] Michael Henriksen. 2021. Finding Evil Go Packages. https://michenriksen.com/blog/finding-evil-go-packages/. [Accessed 11-Jul-2023].

[26] The White House. 2021. Executive Order on Improving the Nation's Cybersecurity. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity. [Accessed 30-Jun-2023].

[27] Thomas Hunter II. [n. d.]. Compromised npm Package: event-stream — medium.com. https://medium.com/intrinsic-blog/compromised-npm-package-event-stream-d47d08605502. [Accessed 30-08-2023].

[28] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2023. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. *IEEE Symposium on Security and Privacy (SP)*, 1509–1526.

[29] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais, and Serena Elisa Ponta. 2022. Risk Explorer for Software Supply Chains: Understanding the Attack Surface of Open-Source Based Software Development. In *Proceedings of the ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED)*. 35–36.

[30] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais, and Serena Elisa Ponta. 2022. Towards the Detection of Malicious Java Packages. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED)*. 63–72.

[31] Frank Nagle, James Dana, Jennifer Hoffman, Steven Randazzo, and Yanuo Zhou. 2022. Census II of Free and Open Source Software—Application Libraries. *Linux Foundation, Harvard Laboratory for Innovation Science (LISH) and Open Source Security Foundation (OpenSSF)* 80 (2022).

[32] Marc Ohm. 2020. Backstabber's Knife Collection. https://dasfreak.github.io/Backstabbers-Knife-Collection. [Accessed 30-Jun-2023].

[33] Marc Ohm, Felix Boes, Christian Bungartz, and Michael Meier. 2022. On the Feasibility of Supervised Machine Learning for the Detection of Malicious Software Packages. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*. 1–10.

[34] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 23–43.

[35] Marc Ohm, Timo Pohl, and Felix Boes. 2023. You Can Run But You Can't Hide: Runtime Protection Against Malicious Package Updates For Node. js. *arXiv preprint arXiv:2305.19760* (2023).

[36] Chinenye Okafor, Taylor R. Schorlemmer, Santiago Torres-Arias, and James C. Davis. 2022. SoK: Analysis of Software Supply Chain Security by Establishing Secure Design Properties. In *Proceedings of the ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED)*. 15–24.

[37] Brian Pfretzschner and Lotfi ben Othmane. 2017. Identification of Dependency-Based Attacks on Node.Js. In *Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES)*. 1–6.

[38] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Comput. Surv.* 49, 1, Article 4 (apr 2016), 37 pages. https://doi.org/10.1145/2886012

[39] Savio Antony Sebastian, Saurabh Malgaonkar, Paulami Shah, Mudit Kapoor, and Tanay Parekhji. 2016. A study & review on code obfuscation. In *2016 World Conference on Futuristic Trends in Research and Innovation for Social Welfare (Startup Conclave)*. 1–6. https://doi.org/10.1109/STARTUP.2016.7583913

[40] Adriana Sejfia and Max Schäfer. 2022. Practical Automated Detection of Malicious npm Packages. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. 1681–1692.

[41] Michael Sikorski and Andrew Honig. 2012. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software* (1st ed.). No Starch Press, USA.

[42] Sonatype. 2022. 8th Annual State of the Software Supply Chain Report. https://www.sonatype.com/state-of-the-software-supply-chain. [Accessed 30-Jun-2023].

[43] Phylum Research Team. 2022. Phylum Discovers Dozens More PyPI Packages Attempting to Deliver W4SP Stealer in Ongoing Supply-Chain Attack. https://blog.phylum.io/phylum-discovers-dozens-more-pypi-packages-attempting-to-deliver-w4sp-stealer-in-ongoing-supply-chain-attack/. [Accessed 30-Jun-2023].

[44] Sonatype Security Research Team. [n. d.]. Sonatype Stops Software Supply Chain Attack Aimed at the Java Developer Community — blog.sonatype.com. https://blog.sonatype.com/malware-removed-from-maven-central. [Accessed 22-08-2023].

[45] The PyTorch Team. 2023. Compromised PyTorch-nightly dependency chain between December 25th and December 30th, 2022. https://pytorch.org/blog/compromised-nightly-dependency. [Accessed 30-Jun-2023].

[46] Bill Toulas. 2023. Malicious Lolip0p PyPi packages install info-stealing malware. https://www.bleepingcomputer.com/news/security/malicious-lolip0p-pypi-packages-install-info-stealing-malware. [Accessed 30-Jun-2023].

[47] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. 2015. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *2015 IEEE Symposium on Security and Privacy*. 659–673. https://doi.org/10.1109/SP.2015.46

[48] Filippo Valsorda. 2022. How Go Mitigates Supply Chain Attacks. https://go.dev/blog/supply-chain. [Accessed 30-Jun-2023].

[49] Bouke van der Bijl. 2015. Monkey Patching in Go. https://bou.ke/blog/monkey-patching-in-go/. [Accessed 30-Jun-2023].

[50] Steven Vaughan-Nichols. 2022. Linus Torvalds: Rust will go into Linux 6.1 -- zdnet.com. https://www.zdnet.com/article/linus-torvalds-rust-will-go-into-linux-6-1. [Accessed 30-Jun-2023].

[51] Jeff Williams. 2009. Enterprise Java Rootkits: "Hardly anyone watches the developers". In *BlackHat USA*.

[52] Elizabeth Wyss, Alexander Wittman, Drew Davidson, and Lorenzo De Carli. 2022. Wolf at the Door: Preventing Install-Time Attacks in Npm with Latch. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security (ASIACCS)*. 1139–1153.

[53] Hui Xu, Yangfan Zhou, Yu Kang, and Michael R. Lyu. 2017. On Secure and Usable Program Obfuscation: A Survey. arXiv:1710.01139 [cs.CR]

[54] Ilsun You and Kangbin Yim. 2010. Malware Obfuscation Techniques: A Brief Survey. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*. 297–300. https://doi.org/10.1109/BWCCA.2010.85

[55] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. 2022. What are weak links in the npm supply chain?. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE)*. 331–340.

[56] Karlo Zanki. 2022. IconBurst NPM software supply chain attack grabs data from apps and websites. https://www.reversinglabs.com/blog/iconburst-npm-software-supply-chain-attack-grabs-data-from-apps-websites. [Accessed 30-Jun-2023].

[57] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *28th USENIX Security Symposium (USENIX Security)*. 995–1010.