École Polytechnique Fédérale de Lausanne

Scripting Language Implementation Bug Analysis

by Milan Duric

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Chibin Zhang
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 2, 2024

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Chibin Zhang, who has supported me and provided guidance during the course of this project.

Lausanne, January 2, 2024 Milan Duric

Abstract

Scripting languages, such as Python, PHP and Ruby, enjoy a huge popularity among software developers across different areas of software engineering. While the security of individual scripitng languages has been, and still is, a subject of active research of both academia and the bug hunting community, little effort has been invested in identifying possible patterns between bugs across different scripting languages. Conclusions drawn from such a study would not only facilitate the secure design and implementation of future scripting languages, but would also accelerate the process of finding new bugs and vulnerabilities in existing scripting languages that were originally not considered by the study. In this project, we analyzed 60 different bugs in the implementation of Python, PHP and Ruby languages (20 bugs per langauge). Our study focuses solely on exploitable, low-level memory corruption bugs in the interpreters of these languages - CPython, Zend Engine and CRuby respectively. Through our analysis, we identified at least one recurring bug pattern across all three interpreters, which we refer to as *magic methods and callbacks with global side effects*. We demonstrate that all bug examples from this category share one of the two common, generic root causes, regardless of the scripting language they are affecting. Finaly, we argue that this bug pattern generically applies to other scripting language interpreters, by analyzing concrete examples of defense mechanisms in these interpreters.

Contents

Chapter 1

Introduction

Many scripting languages today have managed to establish themselves as universal, general-purpose programming languages. These programming languages offer emerging application scenarios and use-cases that range from general web application development on one side to machine learning and data science on the other. One of the main contributors to the popularity of these languages is the massive open-source ecosystem that accompanies each of them. Naturally, the increased popularity of scripting languages has also led to an increased interest in their security.

However, most of the current work on the security of scripting languages was primarily concerned with the security of the application code written in the scripting language, rather than the low-level execution engine and interpreter implemented in the native code. Admittedly, there are examples of recent work [21] that investigate the security impact of using native C extensions in scripting language runtimes. However, the focus of such research is on the native extensions and not on the interpreter core. On the other hand, existing work on scripting language interpreter security [10] focuses exclussively on individual interpreters and does not try to identify generic bug patterns that could apply to other interpreter implementations, possibly even from different languages. Moreover, the security of Javascript engines seems to have been studied to a much greater degree compared to other scripting language runtimes ([18], [23], [2], [8], [24], [6]). A more comprehensive study that cross compares bugs in different language interpreters would be beneficial for two reasons. Firstly, it would serve as a useful reference point when desiging and implementing new scripting languages and interpreters. Secondly, it would facilitate the bug hunting process for language security researchers, by allowing them to focus on language aspects that are more likely to be vulnerable.

In this project, we analyze bugs in the implementation of multiple interpreters from different languages. Our intuition is simple - given the existence of language features and constructs that are common for multiple languages, it is worthwhile to investigate whether the same bug in the implementation of a particular feature in one language is also present in the implementation of

an identical feature in another language. For this initial case study, we limit our analysis only to the standard interpreter implementations of three popular scripting languages: CPython [4], Zend Engine [16] and CRuby [5]. Additionally, we do not constrain our study by focusing on pre-defined list of common language features. Rather, we collect, locally reproduce and manually analyze 20 bug reports per studied interpreter and identify re-occuring language features as we progress.

After determining the set of re-occuring language features, we analyze the root cause of each bug. Ideally, we are looking for instances of the same root cause affecting the same language feature across different interpreters. However, we are also interested in identifying common root causes in general, regardless of whether they occur in the implementation of the same language features or not.

Our results, although inconclusive, indicate that common bug patterns in the native code of scripting languages do exist. Despite focusing only on a relatively small bug sample, we were able to identify at least one highly interesting language construct that could be repeatedly observed as the main culprit in the collected bug reports. This language construct, which we refer to as *magic methods and callbacks with global side effects* (see Section 5.1), appeared in 6 Python and 3 PHP bug reports. In all 9 reports, we observed only two distinct root causes of the bug. Furthermore, we show that bugs caused by simple syntax violations of the scripting language are still present. This indicates that automatic byte-level fuzzing techniques can still be effective against interpreter targets, in addition to more advanced token-level and grammar-based fuzzing techniques, which are normally used to catch bugs in more complex semantic violations (see Section 5.2). Finally, the results of the more general root-cause analysis across the entire bug sample indicate that the most common programming mistakes are related to *insufficient type checking, insufficient bound checking, absence of NULL guards* and *use of uninitialized or wrongly initialized variables*. These programming mistakes seemed to affect the native code of all 3 studied interpreters.

Chapter 2

Background

Scripting languages typically provide mechnanisms and methods for automatic memory management. Developers that write code in the scripting language usually cannot work on raw memory directly, as the language has been deliberately designed to lack support for such features. The entire complexity associated with memory management has been abstracted away from the developer. The full responsibility for both memory and type safety resides fully in the native C code that implements the language interpreter.

Although this design decision has allowed developers to write scripts that are free from a whole class of memory and type safety bugs that can be observed in C and C++, the problem of memory and type safety still exists in the native C code of the intepreter. One of the main postulates of scripting language design is that the intepreter should never crash ungracefully (for example via a Segmentation Fault), regardless of the nature of the input that the developer provided. Adhering to this principle becomes increasingly challenging when we consider the size of the codebase of projects such as CPython, PHP Zend Engine or CRuby, which is typically several hundreds of thousands lines of code.

2.1 Memory safety

Memory safety is one of the fundations of secure system software. We differentiate between spatial and temporal memory safety. A program is regarded as having no spatial memory safety violations if all memory accesses in the program are performed within the bounds of the underlying allocated object. On the other hand, a program is considered as having no temporal memory safety violations if all memory accesses in the program are performed at the time the underlying object is allocated, i.e., the allocated object is the same as when the pointer was created. Memory safety violations are a common culprit of control-flow hijack attacks.

C and C++ do not enforce memory safety at the language level. Existing techniques that aim to make these languages memory-safe, either by creating a dialect that includes only the safe features of the languages (Cyclone [11], CCured [15]), or by adding protection to unsafe features through compile-time instrumentation (SoftBound [12]), have unfortunately proven to be inadequate due to their overwhelming performance impact. Therefore, it usually remains the responsibility of the programmer to ensure his code does not violate memory safety, which is the challenge the programmers of CPython, Zend engine and CRuby intepreters also had to face.

2.2 Type safety

Type safety is a security property that indicates that each object in the code is accessed only according to its associated type. C and C++ are in principle weakly typed languages, because they allow the programmer to perform downcasts or unrelated casts to data types that may not correspond to the true runtime type of the concerned object. Such behavior can lead to memory safety violations and control-flow hijacks [9].

Chapter 3

Bug collection

Our study focused exclussively on bugs in the native code of the scripting language interpreters that cause memory corruption by violating either memory or type safety properties of the program. We started collecting memory corruption bugs by first looking at online *Common Vulnerabilities and Exposures* (CVE) databases. Our intuition was that bugs with assigned CVEs are more relevant from the security perspective than ordinary bugs that can be found in Issues and Pull Requests on the associated GitHub repositories.

The main CVE database we relied on was the *National Vulnerability Database* (NVD) [14] maintained by the *National Institute of Standards and Technology* (NIST) [13]. NIST provides a convenient API to query the NVD for CVEs using various query parameters. One of such query parameters is the *Common Platform Enumeration* (CPE). CPE is a structured naming scheme that aims to identify a software package. Each CVE has a set of CPEs assigned to it that are affected by the vulnerability concerned. The problem we encountered when querying the NVD is that most of the CVEs residing in the database have too coarse-grained CPEs assigned to it. Although CPE scheme was designed to pinpoint to the exact culprit of the vulnerability within a product, most CPEs that get assigned to CVEs only contain the vendor, product and the version identifiers.

CVE-2022-48560 and *CVE-2022-48565* both have *cpe:2.3:a:python:python:3.6.0:-:*:*:*:*:*:** assigned to them as affected Python interpreter version in the *NVD*. However, the former is a heap use-after-free vulnerability that affects the native C code in *Modules/_heapqmodule.c*, whereas the latter is an XML External Entity vulnerability in Python code in *Lib/plistlib.py*. Both files are part of the CPython project.

Example 1 – *CPE to CVE assignment discrepancies*

This naming deficiency made the entire process of querying the NVD very inefficient, as we were mainly pulling CVEs that do not affect the native code of the intepreter. Indeed, the CPython project contains both native C and Python code. The situation is quite similar for PHP and Ruby intepreters.

Project	CVEs Type 1	CVEs type 2	CVEs type 3
Python (CPython)	14	15	96
PHP (Zend)	97	39	567
Ruby (CRuby)	4	5	103

Table 3.1 – *Distribution of CVEs per interpreter. The second column (CVEs Type 1), indicates the number of CVEs with matching CWE and proof-of-concept link in* NVD*. The third column (CVEs Type 2), indicates the number of CVEs with matching CWE but* no *proof-of-concept link. The last column shows the number of remaining CVEs.*

In order to bridge this gap, we relied on *Common Weakness Enumeration* (CWE) scheme as an additional filter when querying the *NVD*. Thus, we gave priority to those CVEs that have one of 28 CWEs related to memory corruption flaws assigned to them. The list of used, relevant CWEs can be found in Table A.1. Futhermore, we also prioritized CVEs that had an "Exploit" tag associated with them in the NVD, which likely meant that link to a working proof-of-concept code was also available in the database.

Table 3.1 indicates the distribution of CVEs from *NVD*. Unfortunately, a huge proportion of available CVEs (*CVEs type 3*) are not related to memory corruption flaws and are hence highly unlikely to be affecting the native C code of the interpreter. On the other hand, many obtained CVEs that are related to memory corruption flaws (*CVE types 1 and 2*) are caused by vendorized modules that are included as bundles directly in the interpreter projects. In these cases, the actual vulnerability is in the included bundle, and not the native interpreter codebase.

CVE-2019-15903 is a heap-based buffer-overflow vulnerability in *libexpat*, an XML library, which is bundled in CPython under *Modules/expat/*. The CVE has a *CWE-125 (Out-of-bounds Read)* and Python CPEs corresponding to Python versions 2.7, 3.5, 3.6 and 3.7 assigned to it by the *NVD*. Therefore, the CVE falls in our priority group of *CVEs Type 1* in Table 3.1. However, as the actual bug is not in CPython native code, this CVE is not relevant for us.

Example 2 – *False positives in CVE type 1 group*

We analyzed all CVE entries for CPython from *CVE Type 1 group* and *CVE Type 2 group* from Table 3.1. In the end, only 5 of those CVEs made it to our final bug sample, as the rest did not match at least one of the aforementioned criterias. We also inspected 96 of the CVEs from the Type 3 group, in order to account for possible CWE labeling errors in NVD. However, we did not manage to find any new CVEs related to memory corrpution flaws in the native code of the CPython project.

The remaining 15 bugs in CPython, as well as all the collected bugs in PHP Zend project and CRuby, were collected by manually filtering Issues and Pull Requests in their respective GitHub repositories. This was a much faster and more effective process - all 3 repositories were well maintained and all issues had relevant tags and labels assigned to them. This, coupled with the generic content-based search functionality of Issues offered by GitHub, made it easy to construct fine-grained queries and filter relevant bugs.

'label:"Bug" label:"Category: Engine" is:closed ASAN' - The following issue filter on the php/php-src GitHub repository results in 4 bugs, 3 of which made it to our final bug sample. We deliberately gave priority to closed issues as they are more likely to have a patch which would allow us to better understand the bug. The *Category* label makes sure we only get results affecting the native part of the interpreter. Finally, we used various fuzzy strings to target memory corruption bugs.

Example 3 – *Collecting bugs on GitHub repositories*

3.1 Bug reproduction

We created a new Docker image for each bug in which we build the target version of the interpreter from source. Interestingly, the vast majority of analyzed bugs have been patched and backported to older interpreter versions. Therefore, we do not prepare our environment by simply pulling the latest changes from the branch corresponding to the interpreter version which was specified in the Issue description or CPE. On the contrary, we identify the commit which introduces the patch on the main (master) branch and then extract the commit hash of its first predecessor. This commit hash is then used to specify the version of the main (master) branch from which we build the interpreter.

Depending on the nature o the bug, we use 2 different sanitizers in order to instrument the binary. Specifically, if the bug is a signed integer overflow, we used the *Undefined Behavior Sanitizer* (UBSAN) [22], whereas in all other cases we rely on *Address Sanitizer* (ASAN) [19] to detect the memory safety violation and crash the binary early.

Example Docker files can be found in the Appendix B of this document, or on the attached Github repository, which also contains proof-of-concept code snippets that trigger the bugs.

Chapter 4

Results

4.1 Bug Types

Figure 4.1 shows the distribution of bug types in our bug sample. The most common types of bugs encountered were simple NULL pointer dereferences. As indicated in the image, we do not categorize NULL pointer dereference bugs under temporal memory safety violations, but define a separate category for them. We aim to differentiate between NULL pointer dereferences due to absence of NULL guards in the code from those artificially created by the execution environment or the compiler due to use of uninitialized pointers. We make this subtle distinction based on the associated patch - if the patch includes new code that properly initializes the pointer that previously caused a NULL pointer dereference (most probably because of the environment zeroing uninitialized memory before the program is run), then the bug falls under the use of uninitialized pointers category and is hence a temporal memory safety violation. Otherwise, if the patch adds simple NULL guard checks, then we classify the bug as an ordinary NULL pointer dereference.

Temporal memory safety violations constitute one third of the collected bug sample. As indicated in Figure 4.2, 80% of the bugs in this category were *heap use-after-free* bugs. Surprisingly, we did not encounter a single *heap double-free* bug. The second, and only other, type of temporal memory safety violation we encountered was *use of uninitialized pointers*. As previously stated, many of these bugs were manifested as NULL pointer dereferences, but we made a distinction between them based on the patch.

On the other hand, spatial memory safety violations account for one fifth of the bugs in our bug corpus. Not surprisingly, per Figure 4.3, 75% of these bugs were heap-based buffer overflows.

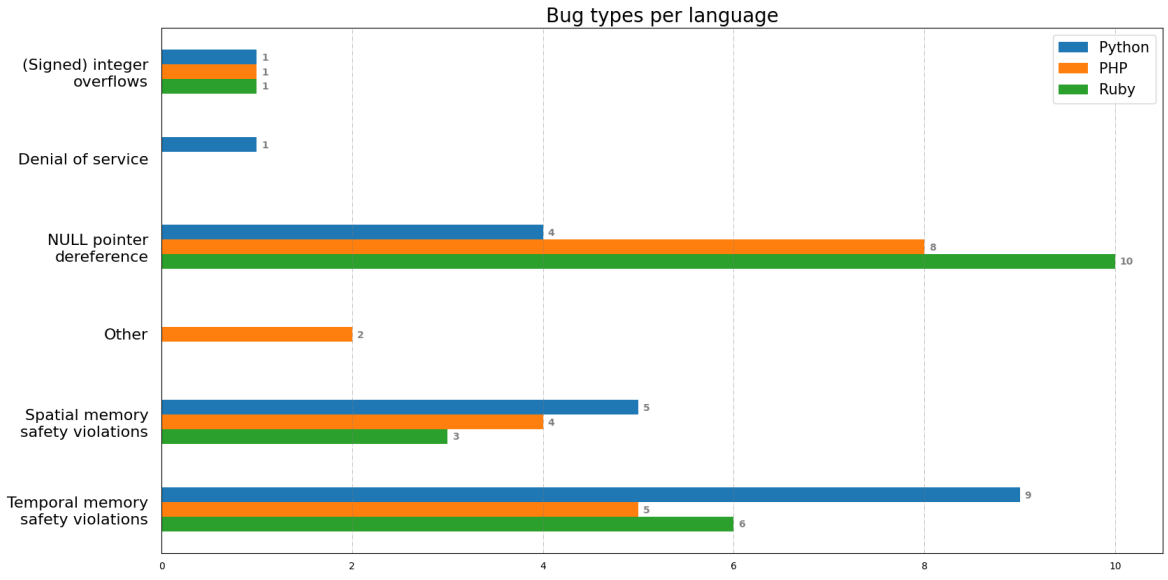

Figure 4.1 – Distribution of bugs per language

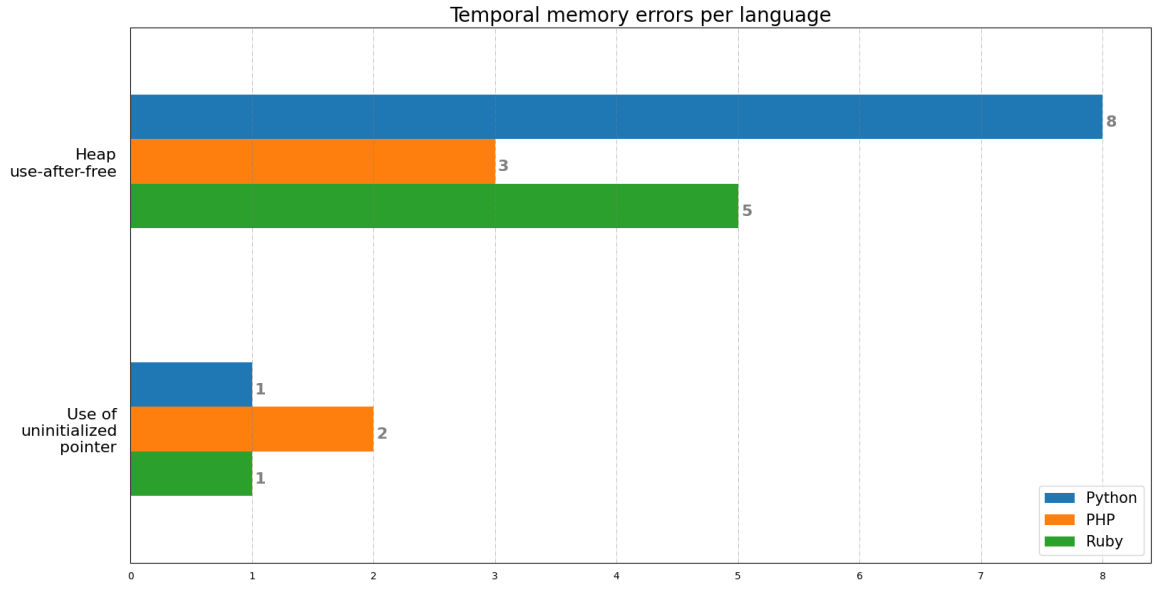

Figure 4.2 – Distribution of types of temporal memory safety violations

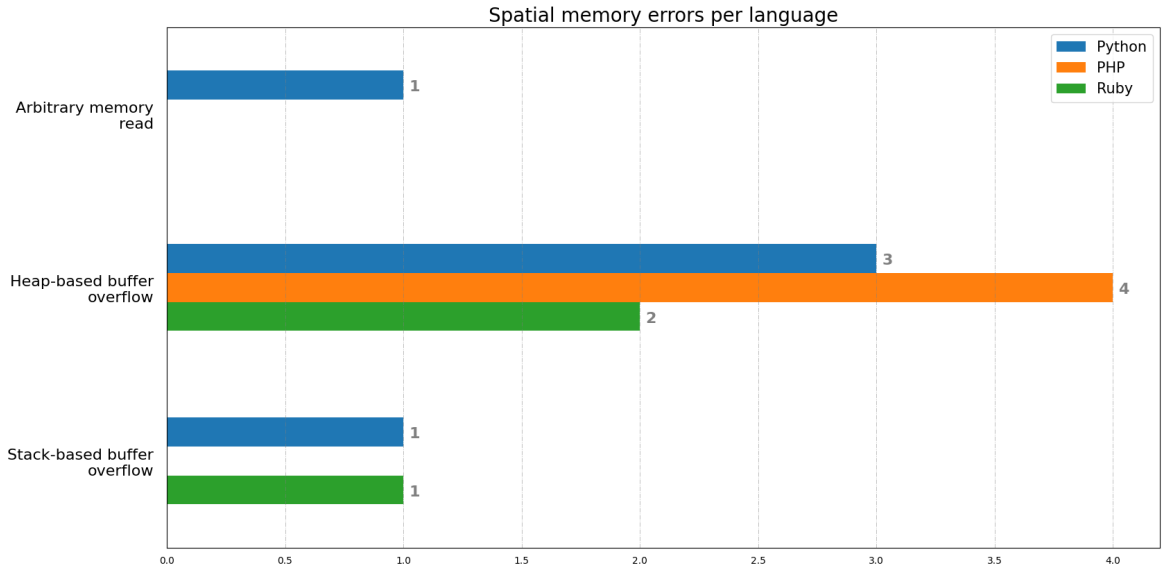

Figure 4.3 – Distribution of types of spatial memory safety violations

4.2 Language features

We labelled each bug in the bug corpus with the name of the high-level language feature in whose implementation the bug occurred. Some bugs had multiple labels attached to them. For example, many bugs from Section 5.1 occur due to pre-mature garabage collection occuring in the implementation of standard data structures of the language, such as collections. In this case, the bugs received both *Garbage collection* and *Standard data structures* labels. On the other hand, we separated bugs occuring in the syntax and semantic analysis of the target language (*Syntax errors* and *Semantic errors* labels) from bugs in the general error handling process (*General error handling* label), such as those ocurring when throwing a high-level language exception.

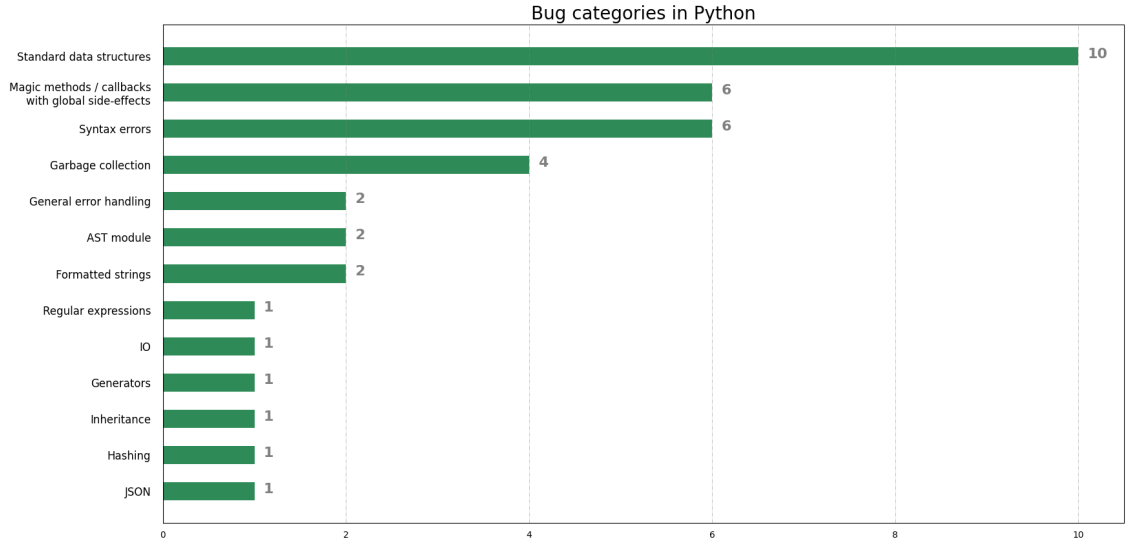

Figure 4.4 – Frequency of affected language features in CPython

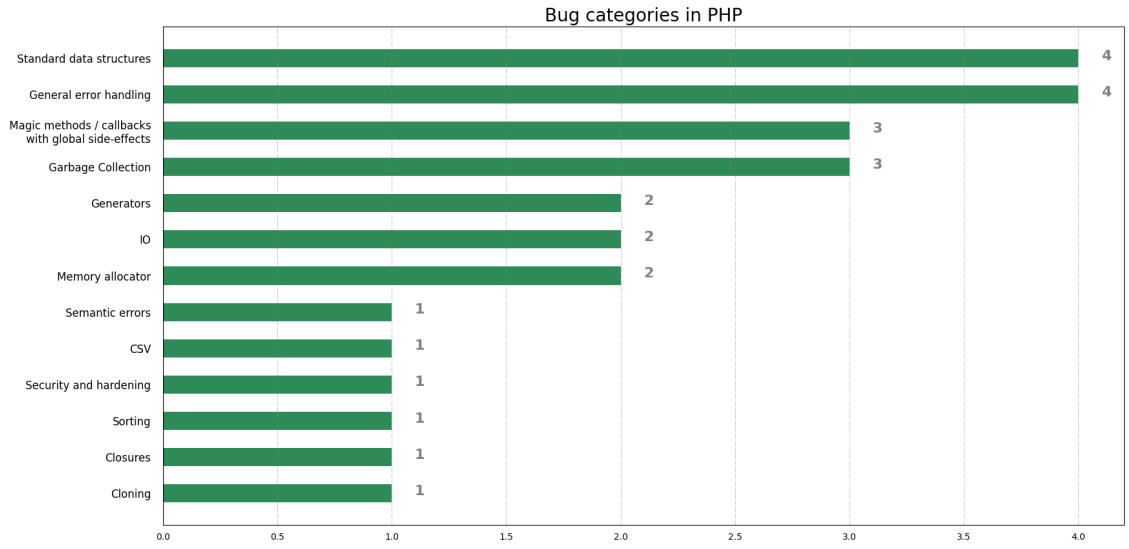

Figure 4.5 – Frequency of affected language features in Zend Engine (PHP)

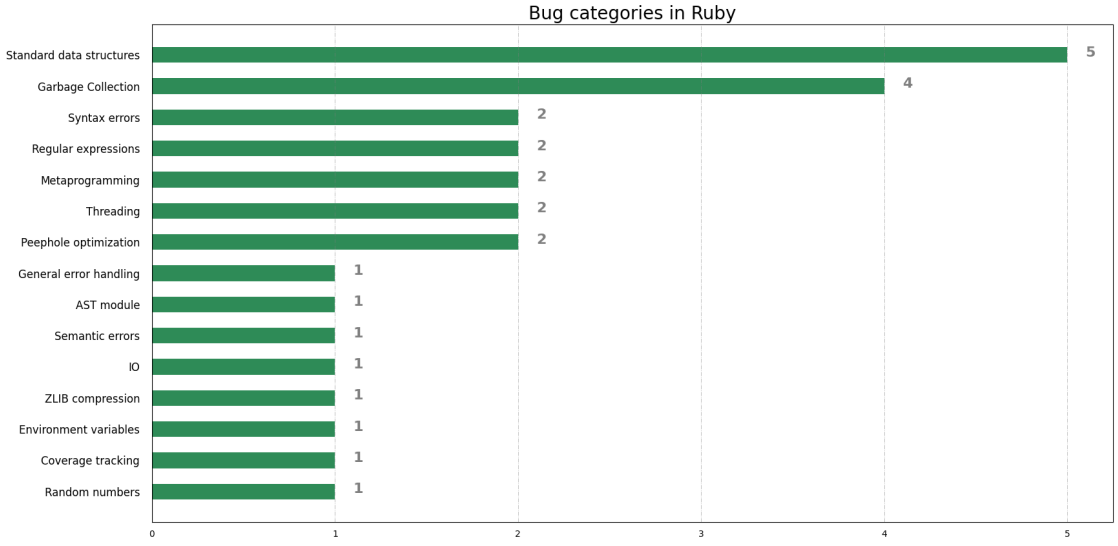

Figure 4.6 – Frequency of affected language features in CRuby

Chapter 5

Bug pattern analysis

5.1 Magic methods and callbacks with global side effects

One of the most interesting and repeatedly encountered bug patterns was *use of magic methods and callbacks with global side effects*. In these bugs, a magic method or a callback was used to make stateful modifications to objects that were later consumed by the code that called the magic method or callback in the first place. The calling code that would not account for the fact that the callback could exhibit global side effects would be vulnerable to different types of memory safety violations. Note that all the examples in this category are inherently pathological and exhibit control flow that is highly unlikely to be desirable by the script developer. Nevertheless, the interpreter must be resilient and robust enough not to crash on such input.

The most common way these bugs were exploited was by creating a custom class or data type and then implementing one of many of its magic methods. The next step would be to find a functionality that, when used, would temporarily transfer control flow to the malicious magic method. Code snippet 1 illustrates the problem.

However, this bug can also occur in an implementation of any language features that allows script developers to provide custom callbacks, not just magic methods. Code snippet 2 illustrates the problem.

We now look at the different root causes, at the native C code level, that led to the bugs in this category. We also discuss which languages were affected by these bugs in our study.

Code Snippet 1 Python Issue 91153: Magic method with interfering global side effects. Assignment to a particular index in the *bytearray* will implicitly call the *__index__* method of the *sneaky* class, in order to convert the object to an integer. In this example, the method empties the global array that is used as the target of the assignment. The code that called the magic method must account for possible modifications to the array.

```
ba = bytearray([0 for _ in range(10000)])

class sneaky:
    def __index__(self):
        ba.clear()
        return 1

ba[-1] = sneaky()
```

5.1.1 Root cause: Use of freed memory

Two thirds of the bugs (6 out of 9) in the *use of magic methods and callbacks with global side effects* category were caused due to not making sure that the objects, which had to be accessed after the invocation of the magic method or callback, remained allocated after returning from the callback. Bugs number 10168, 10169 and 11737 from the PHP corpus, as well as CVE-2022-48560, bug number 82791 and bug number 90773 from the CPython corpus fall into this category. In 5 out of 6 cases, the solution implemented in the patch was to artificially increase the reference count in the garbage collector for any pointer that could be tampered with by the callback and that had to be accessed after its return. Naturally, this was followed by immediate call to decrease the reference count once the callback returned, in order to prevent any memory leaks. Patch listing 1 provides a more detailed analysis of the CPython bug 82791. For the remaining bug, specifically PHP bug 10168, more invasive changes had to be applied in the patch. This meant re-designing the garbage collector to make it less aggressive by delaying its activity.

5.1.2 Root cause: Insufficient length checking

One third of the bugs (3 out of 9) in the *use of magic methods and callbacks with global side effects* category were caused due to computing lengths of arrays before invoking the magic method or callback, and then relying on that those lengths have not changed due to the side effects of the called method. All bugs in this category are from the CPython corpus, specifically bugs number 75011, 75348 and 91153. Note that ASAN reported a heap-based buffer overflow, a NULL pointer dereference and a heap use-after-free for these three bugs, respectively. The patches for these cases included reallocating the new list to account for the changes in the size made by the callback (bug 75011), moving the invocation of the callback before computing the index to the array (bug 91153)

Code Snippet 2 Python Issue 75348: Custom callback with interfering global side effects. *weakref.ref* creates a weak reference to the object referred to by *a*. The second parameter is the callback function which should be called when the object referred to by *a* is about to be finalized. However, this does not happen when the instruction *del a* is executed, but when the next object is allocated (this is a design decision specific to the older Python garbage collector), which occurs in the native implementation of the list slice operator. Hence, the callback is called by the code that implements the native list slice operator. The calling code must account for any global side effects caused by the callback.

```
import weakref

class A(object):
    pass

def callback(x):
    del lst[0]

keepali0e=[]
for i in range(1):
    lst=[str()]
    a=A()
    a.c=a
    keepali0e.append(weakref.ref(a,callback))
    del a
    while lst:
    vscode-ltex        keepali0e.append(lst[:])
```

and re-designing the garbage collector to make it less aggressive by delaying its activity (bug 75348). For a more detailed analysis of bug 75011, consult patch listing 2.

5.1.3 Affected languages

Although our bug corpus features only bugs in CPython and PHP from this category, we believe that any other scripting language that provides developers the possibility to define custom callbacks or magic methods in this way, has the potential to introduce bugs from this category. The most straightforward targets for these types of bugs are collection-like data structures that are implemented directly using native code. Thus, we decided to have a deeper look at *array* and *hash* implementations in both CRuby and MRuby, two of the most popular Ruby interpreters. Unfortunately, we did not manage to find any new bugs, however, we found occasional evidence in the code (see code snippet 3) that indicate the developers were actively paying attention to these types of bugs.

Patch 1 The listing shows the exploit (left side) and patch (right side) for CPython bug 82791 which caused a heap use-after-free. The extract from the patch is part of the native *list_count* function that implements the *count* method of the built-in list type. The *ob_item* pointer is the native C array that stores the elements of the list. The *value* pointer points to the argument of the *list_count* function, which is irrelevant in this case. The native method *PyObject_RichCompareBool* is responsible for executing the equality operation and hence invoking the implemented *__eq__* magic method, which will empty the Python list. As no other references are used to point to the elements of the list, the garbage collector will free the underlying memory. In order to prevent this, the patch had to artificially increase the reference count for each object of the Python list before calling *PyObject_RichCompareBool*. An important detail is the immediate decrementation of the reference count once the equality function finishes, which prevents memory leaks. Note that an identical bug affected other methods of the list type, such as *remove* and *index*.

```
class poc() :
    def __eq__(self,other) :
        list1.clear()
        return NotImplemented

list1 = [ poc() ]
list1.count(list1)
```

```
 Py_ssize_t count = 0;
 Py_ssize_t i;

 for (i = 0; i < Py_SIZE(self); i++) {
-    int cmp = PyObject_RichCompareBool(
         self->ob_item[i], value, Py_EQ);
+    PyObject *obj = self->ob_item[i];
+    Py_INCREF(obj);
+    int cmp = PyObject_RichCompareBool(
+        obj, value, Py_EQ);
+    Py_DECREF(obj);
     if (cmp > 0)
         count++;
     else if (cmp < 0)
         return NULL;
 }
 return PyLong_FromSsize_t(count);
```

Patch 2 The listing shows the exploit (left side) and patch (right side) for CPython bug 75011 which caused a heap-based buffer overflow. The extract from the patch is part of the native *_asyncio_Future_remove_done_callback* function that implements the *remove_done_callback* method of the Future class. The extract is part of a for loop that iterates over the list of callbacks in the Future object (omitted for brevity), and adds any non-matching callbacks to the *newList* array. The *item* pointer is the current callback in the list, whereas *fn* points to the argument of the *remove_done_callback* method (instance of *evil* class). The native function *PyObject_RichCompareBool* will execute the comparison between the *evil* instance and the current callback, which will implicitly call the implemented *__eq__* magic method. The magic method adds new callback to the callback list, which extends the duration of the for loop (this is not observable in the example). However, the *newlist* pointer was allocated before the loop started using the original size of the callback list. Therefore, in iteration 65, a heap overflow occurs during call to *PyList_SET_ITEM*

```
import asyncio

fut = asyncio.Future()
fut.add_done_callback(str)

for _ in range(63):
    fut.add_done_callback(id)

class evil:
    def __eq__(self, other):
        fut.add_done_callback(id)
        return False

fut.remove_done_callback(evil())
```

```
if ((ret = PyObject_RichCompareBool(fn,
    item, Py_EQ)) < 0) {
    goto fail;
}
if (ret == 0) {
-    Py_INCREF(item);
-    PyList_SET_ITEM(newlist, j, item);
-    j++;
+    if (j < len) {
+        Py_INCREF(item);
+        PyList_SET_ITEM(newlist, j, item);
+        j++;
+    }
+    else {
+        if (PyList_Append(newlist, item)){
+            goto fail;
+        }
+    }
}
```

.

5.2 Flaws in syntax and semantic analysis

A notable class of bugs occured when the script code would violate syntax rules of the language. Majority of these bugs come from CPython, namely bugs number 88562, 103718, 94360, 89571, 103824 and 104016. Particularly interesting are bugs number 88562 (see code snippet 4), 94360 and 103718, as they have been detected by vanilla AFL++ byte-level mutations ([7]), starting from an initial seed of Python files. It is known that byte-level fuzzers struggle to achieve greater coverage when fuzzing highly structured targets like interpreters. Nevertheless, they can still be effective in detecting bugs in error handling during parsing, as proven by the 3 previously mentioned examples.

On the other hand, we also encountered bugs that occured when blatantly violating semantics of the scripting language. Typical examples of these bugs would be bug number 9543 in PHP or bug number 14261 in CRuby. In the former case, the semantic violation was caused by trying to alias a built-in PHP type. In the latter case, the semantic error occurred because the Ruby code tried to use a reserved boolean literal as a variable name. Although the authors did not mention how they found these bugs, it seems likely that the automatic byte-level mutations employed by plain AFL++ were probably not the used technique, unlike in the previous case. On the contrary, it seems that an approach using token-level mutations, like the one described in [17], would be more effective in finding these kinds of bugs that occur during semantic violations.

5.3 Common root causes

In this section, we report most common root causes at the native C code level that led to the bugs in our corpus. We do not focus only on a particular high level language feature, like we did in Section 5.1, but maintain a more holistic approach and focus on the entire bug corpus.

5.3.1 Absence of NULL guards

As indicated in Figure 4.1, NULL pointer dereference bugs are the most prevalent in our bug corpus. They were all triggered by simple absence of NULL guards in the native C code. Some representative examples from this category are PHP bugs 7809 (see patch listing 3), 9709, 11178 and 11734, CPython bug 106263 and CVE-2019-5010, as well as CRuby bugs 14421 and 19173.

5.3.2 Use of uninitialized or wrongly initialized variables

A subcategory of temporal memory safety violations from Figure 4.2 were caused by using variables before they were properly initialized. Some notable examples from this category are PHP bugs 8083

and 9752, CRuby bug 13753 and CPython bug 106524 (see patch listing 4)

5.3.3 Insufficient bound checking

Several bugs were caused by improper bound checking of user-supplied arguments. This lack of sanitization would subsequently lead to spatial memory safety violations or integer overflows. Representative examples from this class are CVE-2014-4616 (leads to an arbitrary memory read primitive), CVE-2016-5636, bugs 104016 and 79103 (see patch listing 5) from CPython, bug 16376 from CRuby as well as bug 12151 from PHP.

5.3.4 Insufficient type checking

Finally, absence of type checks was also a common bug cause. Some notable examples from this category can be observed in the CRuby corpus, namely bugs number 15511, 18001 and 19793. Additionally, CPython bugs 85820 (see patch listing 6) and 89571 also fall into this category.

Code Snippet 3 Native code in *array.c* implementing *array.none?* function in CRuby. The function can be called with 3 different types of arguments, with each case being handled by one of the if/else blocks. If no argument is given, the function returns true if there are no truthy elements in the array, which is handled in the *else if* block. An important detail is the bound of the for loop in the *else if* block, which is not computed dynamically after each iteration, but is computed statically at the beginning of the function. This is not vulnerable because the control-flow is never transfered to the user-provided code within the for loop, so the array size will never change. This is not the case for the *if* and *else* blocks, which invoke *rb_funcall* and *rb_yield* functions, both of which can transfer control-flow to user-provided code. After monkey-patching the *if* block to also use variable *len* as the loop bound, we were able to construct pathological examples that trigger heap use-after-free errors.

```
static VALUE
rb_ary_none_p(int argc, VALUE *argv, VALUE ary)
{
    long i, len = RARRAY_LEN(ary);
    ...
    if (argc) {
        for (i = 0; i < RARRAY_LEN(ary); ++i) {
            if (RTEST(rb_funcall(argv[0], idEqq, 1, RARRAY_AREF(ary, i)))) {
                return Qfalse;
            }
        }
    }
    else if (!rb_block_given_p()) {
        for (i = 0; i < len; ++i) {
            if (RTEST(RARRAY_AREF(ary, i))) return Qfalse;
        }
    }
    else {
        for (i = 0; i < RARRAY_LEN(ary); ++i) {
            if (RTEST(rb_yield(RARRAY_AREF(ary, i)))) return Qfalse;
        }
    }
    return Qtrue;
}
```

Code Snippet 4 CPython bug 88562 (left side) and CRuby bug 14261 (right side). The Python snippet caused a heap use-after-free when reporting a syntax error caused by an unterminated multi-line string, whereas the Ruby code snippet caused a NULL pointer dereference when defining a variable using a reserved keyword. The Python bug was found using vanilla AFL with byte-level mutations. The technique used to find the Ruby bug was not reported, though we suspect token-level AFL would be effective in finding these types of bugs.

```
x = "ijosdfsd\
def blech():
    pass
```

```
x, true
```

Patch 3 PHP bug 7809 (exploit on the left side) and patch (on the right side). Function *zend_string_copy* cannot handle NULL arguments

```
<?php
class MySplFileInfo extends
SplFileInfo {
    public function __construct
       (string $filename) {}
}

$sfi = new MySplFileInfo("foo");
clone $sfi;
```

```
static zend_object *spl_filesystem_object_clone
    (zend_object *old_object) {
    ...
    switch (source->type) {
        case SPL_FS_INFO:
-           intern->path = zend_string_copy(
-               source->path);
-           intern->fname = zend_string_copy(
-               source->file_name);
+           if (source->path != NULL) {
+               intern->path = zend_string_copy(
+                   source->path);
+           }
+           if (source->fname != NULL) {
+               intern->fname = zend_string_copy(
+                   source->fname);
+           }
            break;
    ...
}
```

Patch 4 CPython bug 106524 (exploit on the left side) and patch (on the right side). The bug occurs in the Python regular expression engine when initializing a template with a negative or non-int group index. The problem was that *self->items* would remain uninitialized if the call to *PyLong_AsSsize_t* returned a negative value. The control-flow would transfer to *bad_template*, which would call *template_clear* (through *Py_XDECREF*), which would dereference an uninitialized pointer.

```
 import _sre
_sre.template("", ["", -1, ""])
```

```
static PyObject *
_sre_template_impl(..., PyObject *template)
{
     ...
+    memset(self->items, 0,
+      sizeof(self->items[0]) * n);
     for (Py_ssize_t i = 0; i < n; i++) {
         Py_ssize_t index = PyLong_AsSsize_t(
           PyList_GET_ITEM(template, 2*i+1));
         if (index == -1 && PyErr_Occurred()) {
             Py_DECREF(self);
             return NULL;
         }
         if (index < 0) {
             goto bad_template;
         }
         ...
     }
     return (PyObject*) self;
 bad_template:
     PyErr_SetString(PyExc_TypeError,
       "invalid template");
     Py_XDECREF(self);
     return NULL;
 }
```

Patch 5 CPython bug 79103 (exploit on the left side) and patch (on the right side). Argument *digestlen* is supplied by the user and needs to be sanitized to prevent overflows.

```
import hashlib
hashlib.shake_128().hexdigest(
   (-1)&2**64-1)
```

```
 static PyObject * _SHAKE_digest(
   SHA3object *self,
   unsigned long digestlen,
   int hex)
 {
     unsigned char *digest = NULL;
     SHA3_state temp;
     int res;
     PyObject *result = NULL;

+    if (digestlen >= (1 << 29)) {
+        PyErr_SetString(PyExc_ValueError,
+           "length is too large");
+        return NULL;
+    }
     ...
 }
```

Patch 6 CPython bug 85820 (exploit on the left side) and patch (on the right side). When allocating MemoryError classes (not shown in the patch subfigure), there is some logic in the native C code to use pre-allocated instances from a free list only if the type that is being allocated is not a subclass of MemoryError. Unfortunately, the same logic was not present in the native C code of the deallocator (shown in the patch subfigure), so the freelist would be populated even with subclasses of MemoryError. This would lead to the free list corruption and a heap use-after-free error. The patch verifies that the error is not a subclass of MemoryError before adding it to the custom free list.

```
import gc

def func():
    class TestException(MemoryError):
        pass

    try:
        raise MemoryError
    except MemoryError as exc:
        inst = exc

    try:
        raise TestException
    except Exception:
        pass

    inst = None

    try:
        raise MemoryError
    except MemoryError as exc:
        pass

    gc.collect()

if __name__ == "__main__":
    func()
    print("ok")
```

```
static void
MemoryError_dealloc(PyBaseExceptionObject *self)
{
-    _PyObject_GC_UNTRACK(self);
     BaseException_clear(self);

+    if (!Py_IS_TYPE(
+       self,
+       (PyTypeObject *) PyExc_MemoryError)) {
+        return Py_TYPE(self)->tp_free(
+           (PyObject *)self);
+    }

+    _PyObject_GC_UNTRACK(self);
     ...
```

Chapter 6

Conclusion and future directions

The core contribution of this study is the identification of a very specific bug pattern that seems to affect various scripting language interpreters. This bug pattern, which we refer to as *magic methods and callbacks with global side effects* (see Section 5.1), occurs when the developer of the native code overlooks the fact that the target of the control-flow transfer is actually a user-provided function which may exhibit arbitrary global side effects. Once the control-flow returns, various memory corruption errors can occur if the native code does not account for these global side effects.

The most logical next step would be to see if we can find any new bugs from this category. These bugs typically target native implementations of collection-like data structures and are usually not too difficult to spot during a manual source code review. However, they may also occur in less obvious places and language features, which was proven by CPython bug 75011. Therefore, it becomes highly important to enumerate all the language features and constructs that rely on any kind of user-provided callbacks in their native implementation.

Naturally, an automated approach towards detection of these types of bugs should be regarded as an ideal end goal. An initial, naive approach would probably be to manually transfer the 9 bug examples we have in this category into a target scripting lanugage, possibly even add some new examples (for example, replace a heap priority queue with another data structure of the language with a native implementation), and then feed those examples as initial seeds to a byte-level fuzzer like AFL++. A more sophisticated approach would go in the direction of defining a dedicated grammar that would automate the creation of such examples, as demonstrated successfully in pevious work ([1], [20], [3]). It remains to be seen if such an approach would be effective in finding new bugs from this category.

However, this does not mean that traditional byte-level fuzzers are no longer effective when fuzzing highly-structured targets such as scripting language interpreters. Out of 6 bugs that occured as a result of syntax violations in CPython, we concluded that at least 3 of them could be easily

reached by byte-level fuzz mutations, provided a decent initial seed corpus. Nevertheless, we also confirm the conclusions of previous studies, which state that most of the bugs in interpreters come after syntax analysis, and in order to detect them, grammar-aware fuzzing approaches are required.

Bibliography

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. "NAUTILUS: Fishing for Deep Bugs with Grammars". In: *Proceedings 2019 Network and Distributed System Security Symposium* (2019). URL: https://api.semanticscholar.org/CorpusID:69790362.

[2] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. "JIT-Picking: Differential Fuzzing of JavaScript Engines". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 351–364. ISBN: 9781450394505. DOI: 10.1145/3548606.3560624. URL: https://doi.org/10.1145/3548606.3560624.

[3] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. "One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 642–658. DOI: 10.1109/SP40001.2021.00071.

[4] *CPython*. https://github.com/python/cpython.

[5] *CRuby*. https://github.com/ruby/ruby.

[6] Robert Delhougne. *Concolic-Fuzzing of JavaScript Programs using GraalVM and Truffle*. SKILL 2021. 2021.

[7] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. "AFL++: Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[8] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, and Wei Huo. "SoFi: Reflection-Augmented Fuzzing for JavaScript Engines". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 2229–2242. ISBN: 9781450384544. DOI: 10.1145/3460120.3484823. URL: https://doi.org/10.1145/3460120.3484823.

[9] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. “HexType: Efficient Detection of Type Confusion Errors for C++”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2373–2387. ISBN: 9781450349468. DOI: 10.1145/3133956.3134062. URL: https://doi.org/10.1145/3133956.3134062.

[10] Chengman Jiang, Baojian Hua, Wanrong Ouyang, Qiliang Fan, and Zhizhong Pan. “PyGuard: Finding and Understanding Vulnerabilities in Python Virtual Machines.” In: *International Symposium on Software Reliability Engineering*. 2021.

[11] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. “Cyclone: A Safe Dialect of C”. In: *Usenix Annual Technical Conference*. 2002.

[12] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. “SoftBound: Highly Compatible and Complete Spatial Safety for C”. In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Dublin, Ireland, June 2009.

[13] *National Institute Of Standards and Technology (NIST)*. https://nist.gov/.

[14] *National Vulnerability Database (NVD)*. https://nvd.nist.gov/.

[15] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. “CCured: Type-Safe Retrofitting of Legacy Software”. In: 2005.

[16] *PHP-SRC (Zend Engine)*. https://github.com/php/php-src.

[17] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. “Token-Level Fuzzing”. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2795–2809. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/salls.

[18] Groß Samuel, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. “FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities”. In: *Network and Distributed Systems Security (NDSS) Symposium. 2023*. 2023.

[19] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. “AddressSanitizer: A Fast Address Sanity Checker”. In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, June 2012, pp. 309–318. ISBN: 978-931971-93-5. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[20] Prashast Srivastava and Mathias Payer. “Gramatron: Effective Grammar-Aware Fuzzing”. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2021. Virtual, Denmark: Association for Computing Machinery, 2021, pp. 244–256. ISBN: 9781450384599. DOI: 10.1145/3460319.3464814. URL: https://doi.org/10.1145/3460319.3464814.

[21] Staicu, Cristian-Alexandru, Sazzadur Rahaman, Ágnes Kiss, and Michael Backes. “Bilingual problems: Studying the security risks incurred by native extensions in scripting languages.” In: *Usenix Security Symposium*. 2023.

[22] *Undefined Behavior Sanitizer (UBSAN)*. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`.

[23] Wang, Junjie, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. “FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler.” In: *USENIX Security Symposium. USENIX. 2023*. 2023.

[24] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. “Automated Conformance Testing for JavaScript Engines via Deep Compiler Fuzzing”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 435–450. ISBN: 9781450383912. DOI: 10.1145/3453483.3454054. URL: https://doi.org/10.1145/3453483.3454054.

Appendix A

Bug collection

CWE	Description
CWE-120	Buffer copy without checking size of input
CWE-121	Stack-based buffer overflow
CWE-122	Heap-based Buffer Overflow
CWE-125	Out of bounds read
CWE-126	Buffer over-read
CWE-127	Buffer under-read
CWE-128	Wraparound error
CWE-129	Improper validation of array index
CWE-130	Improper handling of length parameter inconsistencies
CWE-131	Incorrect calculation of buffer size
CWE-134	use of externally-controlled format string
CWE-190	Integer overflow
CWE-191	Integer underflow
CWE-192	Integer coercion error
CWE-194	Unexpected Sign Extension
CWE-197	Numeric truncation error
CWE-242	Use of Inherently Dangerous Function
CWE-244	Improper Clearing of Heap Memory Before Release ('Heap Inspection')
CWE-401	missing release of memory after effective lifetime
CWE-415	Double-free
CWE-416	Use-after-free
CWE-457	use of uninitialized variable
CWE-466	return of pointer value outside of expected range
CWE-476	NULL pointer dereference
CWE-562	Return of Stack Variable Address
CWE-787	Out of bounds write
CWE-805	Buffer Access with Incorrect Length Value
CWE-1341	Multiple Releases of Same Resource or Handle

Table A.1 – *CWEs used for filtering the NVD*

Appendix B

Sample Docker files

Docker Build 1 Docker file for building CPython with Address Sanitizer based on a commit ID.

```
FROM debian:stable

ARG commit_id=1ef61cf71a218c71860ff6aecf0fd51edb8b65dc

RUN apt update -y && apt upgrade -y
RUN apt install -y libssl-dev zlib1g-dev libffi-dev libbz2-dev \
 libncursesw5-dev libgdbm-dev liblzma-dev libsqlite3-dev tk-dev \
 uuid-dev libreadline-dev wget unzip

RUN wget https://github.com/python/cpython/archive/${commit_id}.zip
RUN unzip ${commit_id}.zip
RUN /cpython-${commit_id}/configure --prefix=/root/localpython \
    --with-address-sanitizer --with-pydebug
RUN export ASAN_OPTIONS="detect_leaks=0" && make && make install
```

Docker Build 2 Docker file for building PHP Zend Engine with Address Sanitizer based on a commit ID.

```
FROM debian:stable
ARG commit_id=aff46d75e12659b9babc8aa4a74e820f5f29bd68
RUN apt update -y && apt upgrade -y
RUN apt install -y pkg-config build-essential autoconf bison re2c \
                    libxml2-dev libsqlite3-dev wget unzip

RUN wget https://github.com/php/php-src/archive/${commit_id}.zip
RUN unzip ${commit_id}.zip
RUN php-src-${commit_id}/buildconf
RUN php-src-${commit_id}/configure --prefix=/root/localphp \
    --enable-address-sanitizer --enable-debug
RUN export ASAN_OPTIONS="detect_leaks=0" && make && make install
```

Docker Build 3 Docker file for building CRuby with Address Sanitizer based on a commit ID.

```
FROM debian:stable
ARG commit_id=85ee4a65a22ebe6f3c65f0b10397bd6ebb976333
RUN apt update -y && apt upgrade -y
RUN apt install -y pkg-config build-essential autoconf bison \
 libyaml-dev gperf wget unzip ruby-full

RUN wget https://github.com/ruby/ruby/archive/${commit_id}.zip
RUN unzip ${commit_id}.zip
RUN ruby-${commit_id}/autogen.sh
RUN mkdir -p /ruby-${commit_id}/build

RUN cd /ruby-${commit_id}/build && ../configure --prefix="/root/localruby" \
    --disable-install-doc \
    cppflags="-fsanitize=address -fno-omit-frame-pointer" \
    optflags=-O0 \
    LDFLAGS="-fsanitize=address -fno-omit-frame-pointer"
RUN cd /ruby-${commit_id}/build && \
    export ASAN_OPTIONS="halt_on_error=0:use_sigaltstack=0:detect_leaks=0" && \
    make && make install
```