Bytecode Corruption Attacks Are Real — And How To Defend Against Them

Taemin Park, Julian Lettner, Yeoul Na, Stijn Volckaert, and Michael Franz

University of California, Irvine
{tmpark,julian.lettner,yeouln,stijnv,franz}@uci.edu

Abstract. In the continuous arms race between attackers and defenders, various attack vectors against script engines have been exploited and subsequently secured. This paper explores a new attack vector that has not received much academic scrutiny: bytecode and its lookup tables. Based on our study of the internals of modern bytecode interpreters, we present four distinct strategies to achieve arbitrary code execution in an interpreter. To protect interpreters from our attack we propose two separate defense strategies: bytecode pointer checksums and non-writable enforcement. To demonstrate the feasibility of our approach, we instantiate our attacks and proposed defense strategies for Python and Lua interpreters. Our evaluation shows that the proposed defenses effectively mitigate bytecode injection attacks with low overheads of less than 16% on average.

1 Introduction

Programs written in dynamic languages execute in a virtual machine (VM). This VM typically translates the program's source code into bytecode, which it then executes in an interpreter. Some VMs also include a JIT compiler that can compile the source code or bytecode into machine code that can be executed directly by the CPU. The VM usually guarantees that the execution of the script is type and memory safe by lifting the burden of managing the application memory and run-time types off the programmer.

Unfortunately, most VMs are implemented in type and memory *unsafe* languages (specifically, C/C++) which provide direct control over memory. Consequently, memory and type safety vulnerabilities often slip into the VM itself. Malicious scripts may exploit these vulnerabilities to leak information, inject malicious code, and/or hijack control flow. The JIT-ROP attack presented by Snow et al. [1], for example, showed that a single memory corruption vulnerability in a JavaScript VM allowed a malicious script to achieve arbitrary code execution, bypassing the VM's security mechanisms.

To make matters worse, dynamic code generation is naturally susceptible to various types of exploits. The memory region that contains the bytecode or machine code must be writable while the code is being generated. This weakens one of the most effective defenses against code injection, Data Execution Prevention (DEP), as the code cache does need to be both writable and executable (though not necessarily at the same time). Song et al. showed that it is possible to exploit the time window where JIT'ed code is writable to inject code stealthily [2]. While generating code, the VM also produces intermediate data such as bytecode and data constants which are used as input for subsequent code generation phases. Tampering with this intermediate data may also give an attacker arbitrary code execution capabilities, without having to directly hijack control-flow or corrupt the code section [3, 4]. Furthermore, these problems may be worse in bytecode interpreters than in JIT engines. Contrary to JIT'ed code, bytecode does not require page-level execute permissions as it is executed by an interpreter and not by the CPU. Malicious bytecode can therefore still be injected, even if DEP is enabled.

Several recently proposed defense mechanisms mitigate code injection attacks on VMs [2–4]. Frassetto et al. proposed to move the JIT compiler and its data into a secure enclave to defend against intermediate representation (IR) code corruption attacks [4], while Microsoft added verification checksums to detect corruption of temporary code buffers in the JavaScript engine of its Edge browser [3]. However, these defenses focus solely on protecting JIT-based VMs and overlook bytecode interpreters. This is not entirely surprising because there is the belief that the potential impact of a code injection-attack on a bytecode interpreter is limited. It is assumed that injected bytecode cannot access arbitrary memory addresses or call arbitrary functions, for example. We contradict this belief by showing that bytecode injection *is* a realistic attack vector with potentially high impact. Specifically, we present several attack strategies that may be pursued to achieve arbitrary code execution in a well-protected bytecode interpreter, *even if that interpreter employs a language-level sandbox to disable access to dangerous APIs and to introspection features.* Our attack allows scripts to perform operations or to interact with the host system in a way that normally would not be allowed by the sandboxed interpreter.

We implement our attack in the standard Python and Lua interpreters. Python and Lua are widely used to write add-ons and plugins for large applications. Bugs in these applications may allow remote attackers to execute arbitrary scripts (cf. Sect. 3). Attackers can also disguise malicious scripts or packages as benign software and distribute them through standard package managers and distribution channels where users may unknowingly download them [5]. By using the attack techniques presented in this paper, scripts downloaded through such channels can perform malicious actions even if the user executes them in an interpreter with a language-level sandbox (that normally prohibits such actions).

Finally, we present a simple and effective defense against bytecode injection that can be deployed without hardware support and with limited run-time overhead.

In summary, we contribute the following:

– We study the internals of modern bytecode interpreters, uncover several potential attack vectors, and show why bytecode corruption is challenging.
– We present an attack that enables arbitrary code execution in an interpreter by corrupting the bytecode and data caches. Our attack starts with an information disclosure step which infers the layout of the heap. Depending

on the layout of the heap, we pursue one of four different attack strategies when constructing the attack payload. We implement our attack in two different languages/interpreters with different architectures: CPython and Lua, stack-based and register-based VMs, respectively.
– We propose a defense that protects the integrity of the bytecode caches and evaluate a reference implementation for both interpreters. Our evaluation shows that the suggested defense successfully prevents all four of our attack strategies.

2 Background

Bytecode interpreters translate input programs into bytecode that encodes instructions for a virtual stack or virtual register machine. Most virtual stack machine instructions operate on data that is stored on a stack. An integer addition instruction, for example, typically pops the top two elements off the stack, adds them together, and pushes their sum onto the stack. By contrast, instructions for a register machine operate on data that is stored in addressable registers. An integer addition instruction for a register machine could load its input operands from registers `R1` and `R2`, add them, and store the result in register `R3`.

Regardless of the type of virtual machine that is being emulated, the size of the bytecode instruction set is small compared to a typical instruction set for a physical architecture. The latest version of the x86_64 instruction set contains well over a thousand instructions, whereas the latest version of the bytecode instruction set used in CPython contains just over a hundred instructions.

2.1 Bytecode Storage

DEP prevents both static and JIT-compiled machine code from being executed while it is writable and vice versa. DEP is ineffective for bytecode, however, because bytecode can be executed even when it is stored on non-executable pages. To prevent bytecode from being overwritten while it is being executed, the interpreter should mark the bytecode as read-only. This is generally not possible because most interpreters store bytecode on the heap, where it typically resides on the same pages as data that must remain writable at all times.

As a consequence of this design decision, it is possible to overwrite the bytecode even while it is executing. It is also easier to discover the location of the bytecode cache than the location of machine code. While the latter requires an arbitrary memory read vulnerability [1], we show that it is possible to discover the location of the bytecode cache with a vulnerability that can only reveal the contents of the heap.

2.2 Data Encapsulation

Interpreters typically encapsulate all program variables, including those of primitive types, into objects. Every object has a header which identifies the type of the encapsulated value. Figure 1, for example, shows two objects representing an integer and a string value. While the integer object only contains one field that stores the actual integer value, the string object has multiple fields to store different properties of the string.

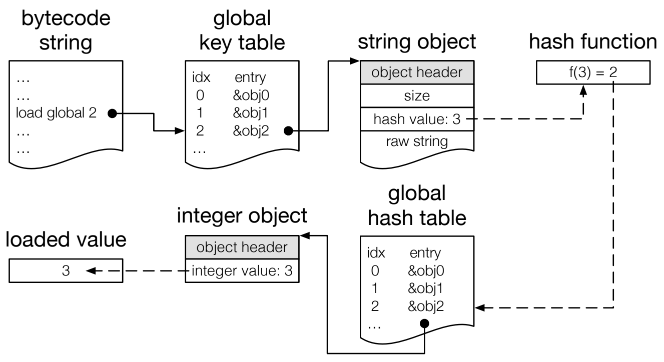

Fig. 1. Loading a global variable through a hash map-like table.

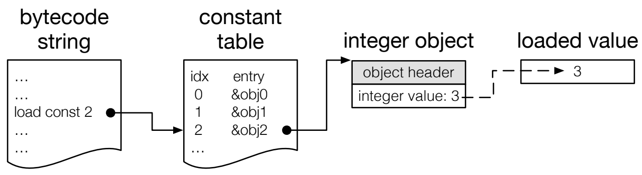

Fig. 2. Loading a value through the constant table.

2.3 Data Access

One of the most notable differences between machine instructions and bytecode instructions is how they access program data. Machine instructions typically encode register numbers, memory addresses, and word-sized constants as instruction operands. Bytecode instructions, by contrast, refer to memory addresses and constants via an indirection layer.

Figure 2 illustrates this difference by showing a bytecode instruction that loads a constant onto the stack. The constant is not embedded in the instruction itself. Instead, the instruction's operand denotes an entry in a constant table. This entry also does not contain the constant itself but instead refers to the address of the object that encapsulates the actual constant. These indirection layers limit the capabilities of an attacker who only can manipulate bytecode. Specifically, an attacker cannot load/store arbitrary constants/variables, call arbitrary functions, or access arbitrary memory locations without the help of built-in tables. To perform a system call, for example, there must be a table entry that refers to a function object, which in turn contains the address of a system call function. The attacker needs to manually inject this entry and the function object because they are unlikely to exist while the interpreter executes benign scripts.

Table 1 lists the supporting data structures and their related bytecode instructions for Python and Lua. When executing a bytecode instruction supported

Table 1. Supporting data structures in Python and Lua. Functions have their own constant and locals tables, but share globals tables.

Interpreter	Opcodes	Data Structures	Type
Python	`LOAD_CONST`	Constant Table	Array
	`LOAD_FAST` `STORE_FAST`	Fast Locals Table	Array
	`LOAD_NAME` `STORE_NAME`	Name Table	Hash Map
	`LOAD_GLOBAL` `STORE_GLOBAL`	Globals Table Builtins Table	Hash Map
Lua	`OP_LOADK`	Constant Table	Array
	`OP_MOVE`	Locals Table	Array
	`OP_GETUPVAL` `OP_SETUPVAL`	Upvalue Table	Array
	`OP_GETTABUP` `OP_SETTABUP`	Globals Table	Hash Map

by an array-typed data structure, the interpreter treats the bytecode's operand as an index into the array. The `LOAD_CONST` instruction, illustrated in Fig. 2, is one example of such an instruction.

Instructions supported by a hash map-typed data structure, such as `LOAD_GLOBAL`, shown in Fig. 1, access their target through a triple indirection. First, the interpreter uses the instruction's operand as an index into a key table containing strings. The interpreter loads the string that the instruction points to, hashes it, and uses the hash value as an index into a hash map table (i.e., the global hash table in this case). Then, the interpreter loads the object reference from the hash map table, and loads the data stored in this object.

2.4 Function Calls

Any function that is called from within a script has a function object associated with it. The function object contains a field indicating the type of the function (i.e., bytecode or native), as well as a pointer to the function's executable code (for native functions), or the function's bytecode object (which, in turn, points to the bytecode string).

To call a function from within bytecode, the caller first loads the function arguments onto the interpreter stack. Then, the caller loads the target function object from the name table (which is a hash map-like data structure). Next, the caller uses a function call bytecode instruction to prepare an argument list array, and to invoke the interpreter's call dispatcher function.

This dispatcher function receives pointers to the target function object and the argument list as its arguments. If the target function is a native C function, the dispatcher will call that C function with a context pointer as its first argument and a pointer to the argument list as its second argument. This context pointer is stored in the function object itself, and can therefore be overwritten by an attacker.

Calling Arbitrary C Functions The set of C functions that can be called by overwriting or injecting function objects on the heap is limited. The reason is that C functions normally expect to find their arguments in certain registers or stack slots, as stipulated in the platform's application binary interface standard.

However, bytecode interpreters pass arguments differently when calling C functions. Specifically, the aforementioned dispatcher function passes pointers to the context and to the argument list structure as the sole arguments to any C function. The context pointer is an implementation-specific pointer that can usually be controlled by the attacker. The argument list pointer, however, cannot be controlled by the attacker. Moreover, unless the C function is aware of the calling conventions used by the interpreter, it will not correctly extract the actual arguments from the argument list.

Consequently, the set of C functions that an attacker can call by corrupting function objects only includes functions that expect less than two arguments and functions that are aware of the calling conventions used in the interpreter.

2.5 Dangerous Interpreter Features

Most bytecode interpreters are designed under the assumption that the end-user will only run benign scripts on benign inputs. These interpreters therefore implement many features that could be abused if either the script or its inputs turn out to be malicious. Recurring examples of such features include the following.

The `eval` function. First introduced in LISP, the `eval` function present in many interpreted languages parses a string argument as source code, translates the source code into bytecode, and then executes the bytecode. Many remote code execution vulnerabilities in scripts are caused by allowing attackers to supply the string argument to `eval` (e.g., CVE-2017-9807, CVE-2016-9949, and scientific literature [6]).

Direct bytecode access. Many scripting languages, including Python and Lua, treat functions as mutable objects with a bytecode field that contains the raw bytecode instructions for their associated function. The script can read and overwrite this bytecode field, either directly or through an API. Python scripts, for example, can access bytecode through the `__code.__co_code` field that exists in every function object, whereas Lua scripts can use the `string.dump` and `load` functions to serialize the raw bytecode instructions for a given function object and deserialize raw bytecode instructions into a new function object respectively.

Dynamic script loading. Scripting languages often allow loading and execution of additional script code stored in the file system. Python, for example, supports the `__import__` function to load modules dynamically, whereas Lua provides the `require` function for this same purpose. An attacker that controls the arguments to these functions may be able to introduce malicious code into an otherwise benign script.

Native code support. Most bytecode interpreters including CPython and Lua support calling native code from the interpreted bytecode through a so-called Foreign Function Interface (FFI). The FFI allows the language to be extended with functionality that is not or cannot be made available within the scripting language itself. From a security perspective, the disadvantage of the FFI is that it can extend the attack surface of the interpreter. Like the interpreter itself, functions called through the FFI are often written in C or C++, which are neither type- nor memory-safe. Vulnerabilities in such functions therefore affect the entire interpreter.

Fully-featured APIs for accessing system resources. Python and Lua both expose APIs for creating, modifying, and deleting system resources such as files, sockets, threads, and virtual memory mappings. The reference interpreters for both languages impose no restrictions on how the script uses these APIs. Typically, the API invocations are only subject to access control checks by the OS itself, and the script therefore runs with the same privileges of its invoker.

2.6 Running Untrusted Scripts

If a bytecode interpreter is used to run untrusted scripts, it is often necessary to restrict or block access to the dangerous features described in Sect. 2.5 or even remove them altogether. Broadly speaking, there are two different approaches to restricting access to dangerous language/interpreter features.

Language-level sandboxing. A language-level sandbox restricts access to dangerous features by intercepting, monitoring, and (potentially) manipulating function calls within the interpreter itself. As an example, you can build a language-level sandbox for Java programs based on the Java Security Manager [7]. This Security Manager wraps calls to dangerous functions to perform fine-grained access control checks. Similarly, lua_sandbox wraps internal interpreter functions to disable script access to certain Lua packages and functions [8].

Language-level sandboxing can also be achieved through source code-level transformations. Caja, for example, transforms untrusted HTML, CSS, and JavaScript code to isolate it from the rest of a web page [9]. RestrictedPython similarly rewrites Python bytecode to restrict access to certain APIs [10].

Finally, one can just remove dangerous functionality from the interpreter altogether, which is viable if the sole purpose of the interpreter is to run untrusted scripts. An example of such a stripped-down interpreter is the Python runtime environment in Google App Engine [11], which does not support native code, does not support direct bytecode access, and does not contain certain system APIs (e.g., for writing to the file system).

The advantage of language-level sandboxes is that they can deploy fine-grained access control checks to not just the APIs for accessing system resources, but also to internal functions that can be invoked without interacting with the OS. The disadvantage is that language-level sandboxes lack a hardware-enforced boundary between the sandbox and the potentially malicious script or program. Malicious scripts can therefore escape from such sandboxes if any part of the interpreter contains an exploitable memory vulnerability.

Application-level sandboxing. Application-level sandboxes restrict access to system resources by interposing on the system calls made by the interpreter. Since 2005, Linux offers the `seccomp` API for this purpose, while older sandboxes could build on the `ptrace` infrastructure.

The advantage of application-level sandboxes over language-level sandboxes is that they are protected from the interpreter by a hardware-enforced boundary (enforced through the memory paging mechanism). The disadvantages are that they can only restrict access to system APIs, and not to internal interpreter functions. Ideally, an interpreter therefore uses both language-level sandboxing and application-level sandboxing techniques when running untrusted scripts.

3 Threat Model And Assumptions

The goal of our work is to achieve arbitrary code execution through injected bytecode and data. Our threat model therefore excludes attacks that corrupt static code or that introduce illegitimate control flow in the static code (i.e., ROP attacks). Our model is consistent with related work in this area [2, 4, 12].

Strong protection for static code. We assume that the target system deploys state-of-the-art protection for static code. Specifically, we assume that Address Space Layout Randomization (ASLR) is enabled and applied to the stack, heap, main executable, and all shared libraries. We assume that machine code-injection attacks are impossible because Data Execution Prevention (DEP) is enforced. We assume that code-reuse attacks are mitigated by fine-grained control-flow integrity [13–15].

Memory vulnerability. We assume that the bytecode interpreter has a memory vulnerability that allows attackers to allocate a buffer on the heap, and to read/write out of the bounds of that buffer. The CVE-2016-5636 vulnerability in CPython is one example that permits this type of buffer overflow. Note that we do *not* assume an arbitrary read-write vulnerability.

Interpreter protection. We assume that the interpreter deploys a language-level sandbox (cf. Sect. 2.6) that disables all dangerous features listed in Sect. 2.5. Consequently, we assume that the scripts cannot access or modify bytecode directly. We further assume that there is no application-level sandbox in place. If the interpreter does use an application-level sandbox, then our attack by itself does not suffice to escape from the sandbox. It could, however, serve as a useful building block for a sandbox escape attack.

Attacker. We assume that the attacker can provide the script to be executed by the protected interpreter. The attacker-provided script does not contain any malicious features that will be blocked by the language-level sandbox. We also assume that the attacker knows the version and configuration of the interpreter, and that the attacker can run this same version locally on a machine under his/her control.

3.1 Realism

The assumption that an attacker can provide the script to be executed by the victim is realistic. Many large applications can be customized through Python or Lua scripts, and have App Store-like distribution systems where developers can freely share their scripts with other users of the same application.

Numerous video games [16], including the hugely popular World of Warcraft (WoW), for example, allows users to write Lua scripts to customize the game interface. Developers can upload these add-ons to dedicated fan sites, where they are downloaded by millions of users. Another example is Python's package manager PyPi, where rogue packages have been known to appear [5]. Packages downloaded through PyPi can subsequently be used in any Python-compliant interpreter, including interpreters with a language-level sandbox.

These script distribution systems usually lack the developer verification and malware scanning features that are commonly employed by application stores for mobile platforms. It is therefore relatively easy to disguise a malicious script as a legitimate piece of software, and to distribute it to a lot of users.

An attacker could also inject malicious script code into other (benign) scripts. We studied recent CVEs from 2014 to 2018 and found several examples of vulnerabilities that permit such script injection attacks (e.g., CVE-2017-9807, CVE-2017-7235, CVE-2017-10803, CVE-2016-9949, CVE-2015-6531, CVE-2015-5306, CVE-2015-5242, CVE-2015-3446, CVE-2014-3593, CVE-2014-2331). The CVE-2017-9807 vulnerability in OpenWebif, for example, existed because OpenWebif called Python's `eval` function on the contents of an HTTP GET parameter. An attacker could exploit this vulnerability by submitting a full script as part of this parameter.

4 Attacking Bytecode

Our attack achieves arbitrary code execution in a bytecode interpreter by simultaneously overwriting the bytecode of a single function and the supporting data structures accessed by that function (e.g., the constant table). Overwriting just the bytecode generally does not suffice, because that would force us to reuse only existing constants and variables.

4.1 Attack Overview

The attack proceeds in five steps:

Preparation: We load a script that contains an attacker function, at least one target function, and a blueprint buffer for the injected code. Target functions are benign functions whose bytecode and supporting data structures are easily recognizable in memory. Each target function contains a unique sequence of harmless operations that is translated into an easily identifiable bytecode string. We also use a unique and large number of constants and local variables in each target function, which allows us to recognize the function's constant/locals tables.

The blueprint buffer contains the raw bytecode sequence the attacker wishes to inject. In most cases, we cannot inject the blueprint buffer as-is, because its bytecode attempts to load data from data structures we cannot overwrite. We therefore rewrite the blueprint buffer in a later step to ensure that it accesses the correct data.

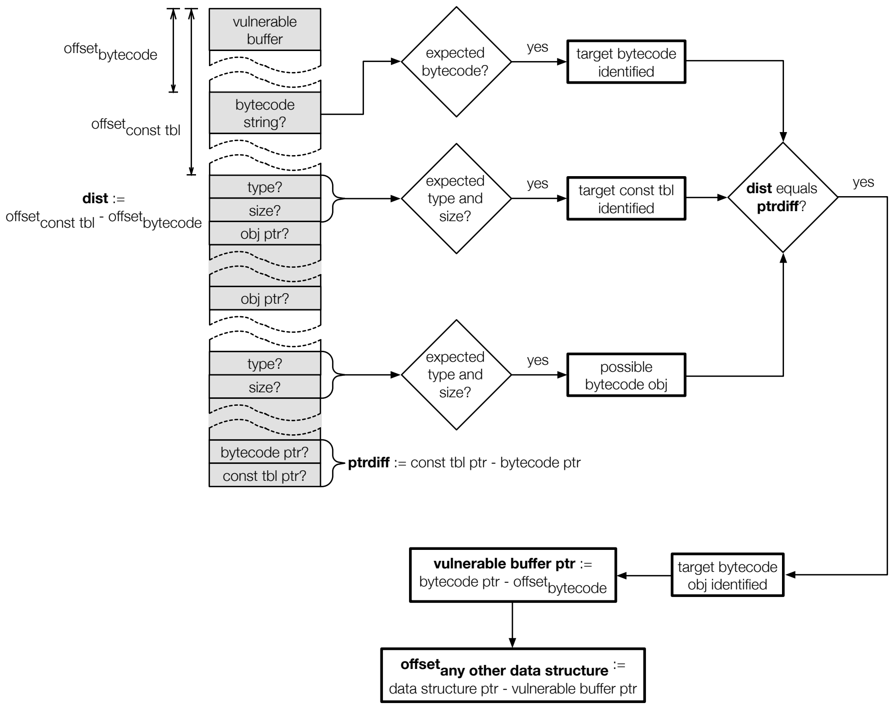
offset bytecode
offset const tbl
dist := offset const tbl - offset bytecode
vulnerable buffer
bytecode string?
type?
size?
obj ptr?
obj ptr?
type?
size?
bytecode ptr?
const tbl ptr?
ptrdiff := const tbl ptr - bytecode ptr
expected bytecode?
yes
target bytecode identified
expected type and size?
yes
target const tbl identified
dist equals ptrdiff?
yes
expected type and size?
yes
possible bytecode obj
target bytecode obj identified
vulnerable buffer ptr := bytecode ptr - offset bytecode
offset any other data structure := data structure ptr - vulnerable buffer ptr

Fig. 3. Overview of our heap layout inference step. By disclosing heap contents and identifying three data structures belonging to the same target function, we can subsequently follow pointers to other data structures by calculating their offset relative to the vulnerable buffer.

Note that the attack script itself looks benign to the interpreter. The script does not use introspection features, nor does it call any privileged APIs or perform privileged operations that are normally stopped by the interpreter's security mechanisms. The code we inject, by contrast, is not benign and does violate the interpreter's security policies, although it does so without being detected.

Heap layout inference: The goal of our second step is to infer the precise layout of a large portion of the heap. The layout information we infer includes the absolute positions (i.e., addresses) of the bytecode and supporting data structures of both the attacker function and at least one of the target functions.

We begin this step, which is illustrated in Fig. 3, by executing the attacker function. The attacker function allocates a buffer, and then leverages the buffer overflow vulnerability to read outside the bounds of that buffer, thereby leaking the contents of the heap. Based on these contents, we can determine the positions of a set of data structures relative to the vulnerable buffer. We do so as follows. First, we search for one of the target functions' bytecode strings. These are easily recognizable since the bytecode for each target function is known and remains the same across executions of the interpreter.

Once we have identified a bytecode string for a target function, we proceed to finding its constant table. The constant table is filled with pointers, which we cannot follow at this point because that would require an arbitrary read vulnerability. Therefore, we cannot examine the contents of a constant table to determine to which function it belongs. Instead, we read the type and size fields for any potential data structure we encounter. Lua uses a bitfield to encode data structure types, so constant tables have a fixed value in the type field. CPython's type fields are pointers to constant strings, which are always loaded at the same address (relative to the image base). In both cases, potential constant tables are easily recognizable.

Once we have determined that a data structure is a potential constant table, we read its size field. Our attacker script ensures that each target function has a large and unique number of constants in its constant table. We do this by declaring a local variable which stores a list of constant numbers in each target function. The size value of the constant table therefore uniquely defines to which function it belongs.

Having identified the bytecode string and the constant table for a specific target function, we now attempt to find the bytecode object for that function. Again, we can recognize potential bytecode objects based on their type field. Once we identify a potential bytecode object, we can determine if it belongs to the same function as the already identified bytecode string and constant table by verifying that the distance between the bytecode string pointer value and the constant table pointer value in the bytecode object matches the distance between the data structures we identified. If these distances match, we assume that we have found the right bytecode object, and that we now know the absolute addresses of the bytecode object, bytecode string, and constant table we disclosed.

At this point, we can follow any heap pointer to an address that is higher than that of the vulnerable buffer, and we can ultimately disclose the full layout of the heap area that follows the vulnerable buffer. We expect that the attacker will also be able to find at least one code pointer on the heap, thereby identifying the base address of the interpreter's executable code section. This is necessary to locate the C functions we wish to call in our attack. Recent work shows that this is a realistic expectation [17].

Attack strategy selection: Based on the heap layout information, we can select an attack strategy and inject the payload. The payload injection is subject to three constraints. First, we cannot write any data at addresses lower than that of the vulnerable buffer, because the vulnerability we are exploiting allows buffer read/write overflows, but not underflows. Second, for the same reason of the first constraint, the payload we inject must be contiguous. Third, we must be careful when overwriting the currently executing bytecode string or any of the data structures that may be accessed by the currently executing code, since doing so might crash the interpreter.

As a result of these three constraints, it is not always possible to overwrite the bytecode string and the constant table of a target function. We have therefore

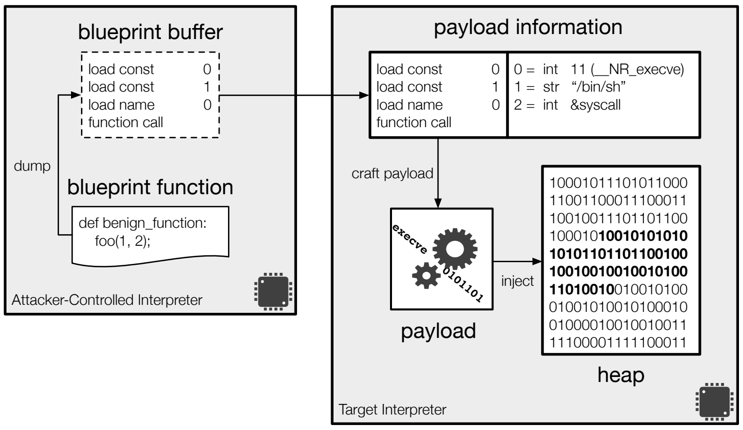

Fig. 4. The attacker creates a blueprint buffer by dumping a blueprint function. The blueprint buffer serves as the starting point for the attack payload construction.

devised multiple attack strategies, each targeting different data structures. We describe these strategies in Sect. 4.2.

Payload construction: We now craft the attack payload, which consists of a bytecode string, and a data structure containing references to the data the attacker wishes to use in the injected bytecode. We provide more details on the payload construction in Sect. 4.2.

Execution: At this point, we overwrite the bytecode and data structures we identified in the second step with the payload crafted in the fourth step. Finally, we transfer control to the target function we overwrote to trigger our injected code.

4.2 Crafting the Payload

We craft the payload based on an attacker-provided blueprint buffer. The blueprint buffer contains the raw bytecode string to be injected. The attacker must additionally provide information about the data to be used in the injected bytecode.

Figure 4 shows the process of creating a blueprint buffer, and converting it into an attack payload. The attacker begins by writing a blueprint function in an off-line instance of the interpreter. We assume that this off-line instance of the interpreter is controlled by the attacker, and that it provides access language introspection features (unlike the target interpreter the attacker wishes to attack).

The blueprint function has the desired attack semantics, but does not necessarily operate on the desired data. For example, if the attack should call a C function that is not normally exposed to the scripting language, then the attacker can just write a blueprint function that calls a different (accessible) function instead, dump the blueprint, and adjust the target of the function call while crafting the payload.

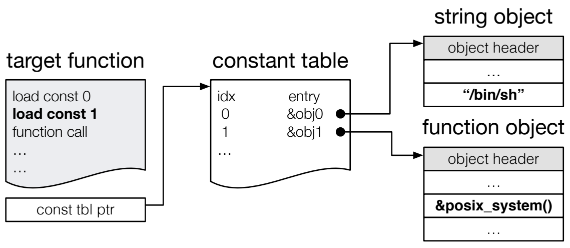

Fig. 5. The payload created based on Strategy 1.

Rewriting the Blueprint The attacker now rewrites the blueprint buffer into a concrete attack payload that works in the target interpreter. Depending on the inferred heap layout, the types of the disclosed data structures, and whether or not these data structures can be safely overwritten, the attacker can pursue one of four rewriting strategies. For simplicity, we explain our strategies using Python's bytecode convention. All strategies apply to the Lua interpreter as well, however, needing only trivial modifications.

Strategy 1: Overwriting a bytecode string and constant table The attacker chooses this strategy if the attacker has disclosed a constant table and the table's entry size is larger than or equal to the number of load instructions in the blueprint buffer. In the blueprint buffer, the function object is loaded from the *name table*. The attacker therefore needs to adjust the load instruction to a load from a constant table, i.e., `LOAD_CONST $id`. The attacker also needs to inject the objects with the prepared data and update the constant table so that each table entry points to the injected object. The resulting payload of Strategy 1 is shown in Fig. 5. `LOAD_CONST 0` loads a function argument, `"/bin/sh"`, then `LOAD_CONST 1` loads the function object overwritten with the address of *posix_system()* function which is a wrapper function in CPython that unboxes argument objects and calls C `system()` function with the unboxed arguments. Next, we use a `FUNC_CALL` instruction to call the injected C function. We could also call the `system()` function directly because it expects just one argument (cf. Sect. 2.4). In both cases, we are able to launch a system shell, which is normally not allowed in the sandboxed interpreter.

Strategy 2: Overwriting a bytecode string and hash map-like table If the attacker only found a hash map-like table or the target constant table size is too small to cover all the load instructions in the blueprint buffer, the attacker selects this strategy. Manipulating hash map-like structures is challenging, however, due to the multi-level indirections and the use of a hash function (see Sect. 2.3). The underlying idea of this strategy is to simplify the hash map manipulation by making the key table entries point to integer objects instead of string objects. This way, the attacker can access the hash map as if it were an array-like structure.

The implementation details can vary depending on how the interpreter accesses the hash map-like table. In CPython, the interpreter maintains a dedicated key table for all hash map-like structures. The `LOAD_GLOBAL` instruction fetches a key from the global key table, and then uses this key as an index into the global hash table. In this architecture, the attacker can overwrite the key table so that each key table entry points to an integer object written by the attacker. Lua, on the other hand, requires two bytecode instructions to load data from a hash map, one for loading the key and one for fetching the value. Moreover, Lua does not maintain dedicated key tables for any of its hash maps. Instead, the key can be loaded from any array-like table. The attacker can therefore convert an existing array-like tables into a key table and fill it with references to integer objects.

Similar to Strategy 1, the attacker replaces the bytecode dump of previous load instructions with that of the bytecode sequence for accessing hash map-like structures as described above. The attacker then changes entries in the hash map which point to attacker's objects.

Strategy 3: Overwriting a bytecode string and loadable object If the attacker is unable to update entries in any tables, he can shape his payload as a single function using this strategy. Instead of using existing tables, the attacker crafts a constant table and adjusts the bytecode and updates the data according as in Strategy 1. The attacker then prepares a bytecode object pointing to the adjusted bytecode buffer and attacker's constant table. To be able to load this bytecode object to the interpreter's stack (or to a register in a register-based machine), the attacker has to overwrite any loadable object with the bytecode object. To do so, the attacker prepares a unique constant object in the preparation step so that its data structure can be easily found in the heap layout inference step. The attacker then overwrites this constant object with the bytecode object, thereby the attacker's bytecode object can be loaded on the stack through the constant table. Based on this loaded bytecode object, the attacker makes a function object which itself becomes the attacker's payload to call the associated function. To do so, the target function's bytecode should be overwritten with two bytecode instructions. One is to create a function object using a bytecode object loaded on the interpreter's stack. The other is to call the function in the function object.

Strategy 4: Injecting bytecode and overwriting a bytecode object In Strategy 4, the attacker injects bytecode instead of overwriting the existing bytecode buffer. To this end, the attacker injects the bytecode on the heap and overwrites the bytecode pointer in the bytecode object with the address of the injected bytecode as shown in Fig. 6. Before injecting the bytecode, the attacker still needs to adjust the bytecode in the blueprint buffer and update the prepared data according to the available data structures (again, the same step as in Strategies 1 and 2).

5 Defense

We designed and prototyped a defense that thwarts the presented as well as other bytecode injection and overwrite attacks. The main goal of our defense

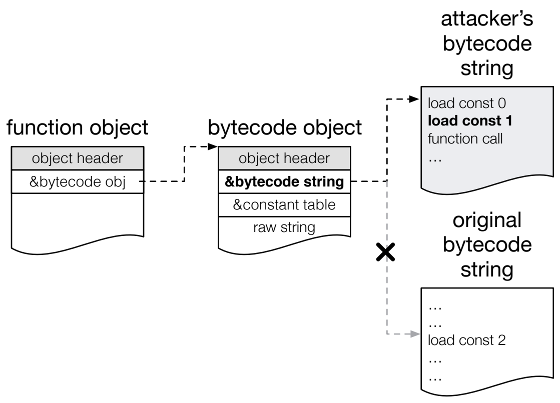

Fig. 6. The payload created based on Strategy 4.

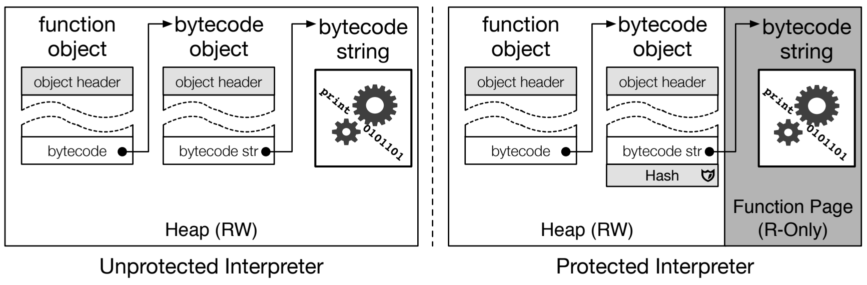

Fig. 7. Overview of our defense. We prevent bytecode strings from being overwritten by placing them in read-only memory. Bytecode injection attacks are prevented by verifying the bytecode pointer hash before executing a function.

is to protect the integrity of bytecode. The design of our defense is inspired by existing defenses against code cache corruption attacks [3, 18]. We propose two defense techniques: making bytecode strings read-only, and verifying bytecode targets during function calls. When combined, this effectively defeats all four of our attack strategies.

First, as shown in Fig. 7, we make all bytecode strings read-only so that the attacker cannot overwrite them. This specifically stops attack strategies 1 through 3, which overwrite the bytecode string of a target function. We implemented this feature by modifying the interpreter's memory manager and parser. Normally, when the interpreter parses a source function and translates it to bytecode, it allocates and stores that bytecode on the heap. We modified the interpreter to allocate a dedicated page for each function's bytecode string, and mark this page as read-only when the source function is fully translated.

This first defense technique prevents valid bytecode strings from being overwritten. However, it does not prevent bytecode injection attacks. Specifically, an attacker can still inject bytecode on the heap, and overwrite the bytecode string pointer in a bytecode object to point it to the injected bytecode instead. We implemented a second defense mechanism that prevents this type of attack. Concretely, we added a bytecode pointer verifier that checks the integrity of a function's bytecode pointer whenever it is called.

We extended the interpreter's parser to generate bytecode pointer checksums whenever it finalizes the translation of a source function into bytecode. We generate these checksums by calculating the hash of the concatenated value: $BytecodePointerValue||BytecodePointerLocation$.

As our hash function, we used the publicly available HighwayHash, which is an optimized version of SipHash. Both SipHash and HighwayHash are keyed hash functions. We generate a random hash key when the interpreter starts and prevent it from leaking by (i) keeping it stored in a dedicated CPU register at all times, (ii) using gcc's `-ffixed-reg` option to prevent reuse or spilling of that register, and (iii) customizing the hash function so it loads the hash key from the dedicated register and so it restores the old values of all registers that we might move the key into. Our bytecode pointer verifier recalculates and verifies the checksum whenever the interpreter invokes a bytecode function. The verifier effectively prevents strategies 3 and 4, which rely on a malicious function call, because the checksum verification will fail before the attacker's bytecode is executed.

6 Evaluation

We implemented our attack and defense for two commonly used bytecode interpreters: CPython 2.7.12 and Lua 5.3.2. We retrofitted a slightly altered version of a known heap buffer overflow vulnerability into CPython (CVE-2016-5636) and added a similar bug to Lua. We constructed an attack that launches a shell by calling `system("/bin/ls")`. We verified that all four of our proposed attack strategies succeed in both interpreters.

We also evaluated the run-time performance impact of our defense by running the Python Performance Benchmark Suite [19] for CPython and the Computer Language Benchmarks Game [20] for Lua. We ran these benchmarks on a GNU/Linux system running Ubuntu 14.04 LTS x64. The system has an eight-core Intel Xeon E5-2660 CPU and 64Gb of RAM. Fig. 8 and 9 show our results.

The run-time performance impact of the first part of our defense (making bytecode read-only) is generally negligible. Only `hg_startup`, `python_startup`, and `python_startup_no_site` slow down noticeably. These benchmarks measure the startup time of the interpreter, which is generally short, but do not measure the execution of any bytecode. The other benchmarks do include execution of actual bytecode.

In these other benchmarks, our checksum verification incurs run-time overheads of less than 16% on average. Since our checksum verification checks occur at every function call, the overhead is directly proportional to the number of

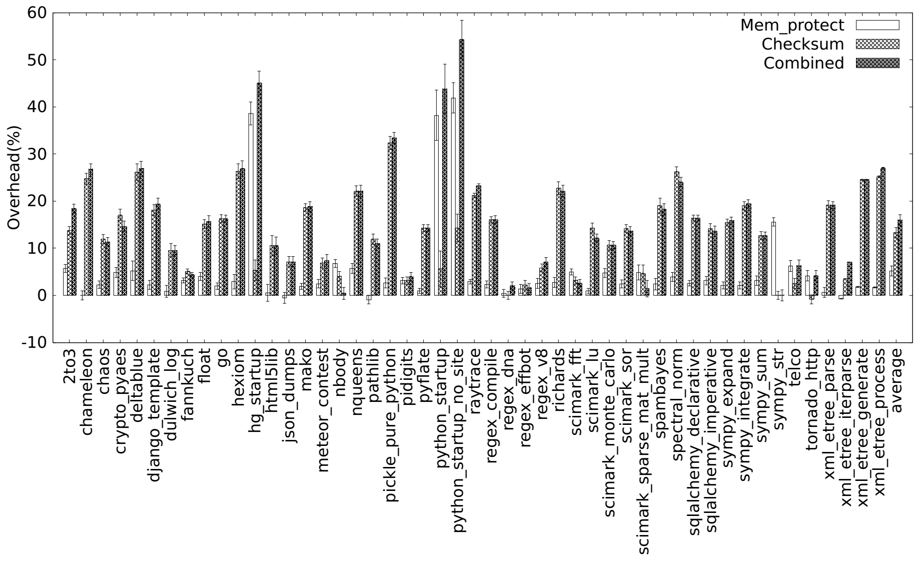

Fig. 8. Run-time overhead in the Python Performance Benchmark Suite.

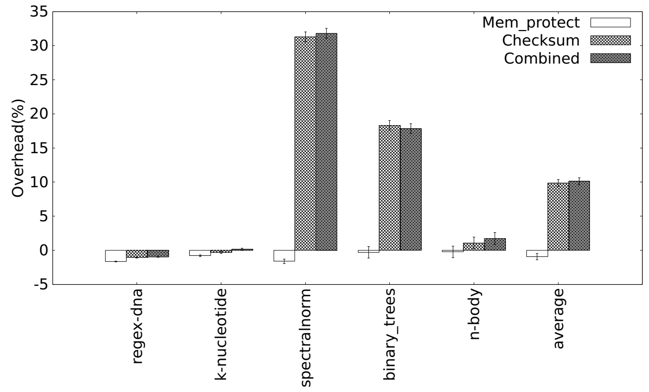

Fig. 9. Run-time overhead in the Computer Language Benchmarks Game for Lua.

function calls and returns. `spectralnorm`, and `binary_trees` benchmarks in Lua execute a significant number of recursive functions, which produces numerous function calls and returns, and thus high overhead.

7 Security Analysis

While our defense successfully stops all four of our proposed strategies, an attacker could still attempt to bypass it as follows:

Pure data-injection attacks: The current implementation of our defense only protects against bytecode overwrite and injection attacks. While this suffices to thwart all four of our proposed attack strategies, we do believe it might be possible to mount a pure data-injection attack that also achieves arbitrary code execution. In such an attack, the attacker would overwrite or inject new data to alter the behavior of a benign function without overwriting that function's bytecode.

To block these attacks, one can apply the same conceptual techniques we proposed in this paper to protect all of the interpreter's data structures. Immutable data structures such as constant tables can be moved to read-only pages, while mutable structures can be extended with verification checksums.

Partial state corruption attacks in multi-threaded scripts: The bytecode interpreters we evaluated parse and translate source functions into bytecode lazily (i.e., when the function is first called). Therefore, there is a time window after the interpreter has fully initialized for which source functions may be stored in a partially translated state in writable memory. Recent work by Song et al. [21] and Frassetto et al. [4] showed that it is possible to overwrite this partially generated state in interpreters that support multi-threaded scripting languages. Our defense is, in principle, also vulnerable to such attacks. To prevent such attacks, we could offload the parsing and bytecode translation to an external process, as was done by Song et al. [21].

Checksum forging: We protect pointers to bytecode strings with a verification checksum to prevent attackers from forging bytecode objects pointing to bytecode strings stored on the heap. If an attacker can create a bytecode object with the correct checksum, our defense would not be able to detect that the bytecode string it points to is stored in writable memory. We prevent such attacks by using a keyed hash function, and by storing the key in a dedicated register which is never leaked. We also prevent attackers from reusing correct bytecode pointers and checksums to redirect one bytecode function to a different, legitimate bytecode function. We achieve this by using not just the bytecode pointer itself, but also the location where it is stored as input to our hash function.

Checksum alternatives: As an alternative to our bytecode pointer checksum, we could have used a true HMAC (as was done in Subversive-C [18]), or a MAC-AES (as was done in CCFI [22]). We opted not to use an HMAC because our input (i.e., the concatenation of the bytecode pointer and its storage location) is fixed-length. An HMAC therefore does not increase security over our scheme. We did not implement the MAC-AES scheme used in CCFI because it requires many reserved registers, as opposed to just one register in our case.

8 Related Work

Most of the existing work in this area focuses on code-injection attacks and defenses for JIT-based VMs. The security of pure bytecode interpreters has received little attention in the academic community.

8.1 Direct Code Injection

Early JIT-based VMs either left the JIT-compiled code writable at all times, or mapped two copies (backed by the same physical memory pages) of the JIT-compiled code into the VM's virtual address space: one writable and one executable. In both of those cases, an attacker could simply inject code into the JIT code cache by locating it in memory and overwriting it. To prevent such attacks, all major browsers and high-performance JIT engines have now adapted Data Execution Prevention, and they generally only map one copy of the JIT code cache into virtual memory at any given time. The code cache is made writable only while code is being generated, and the JIT engine makes the cache non-writable while executing JIT-compiled code. We enforce the equivalent of Data Execution Prevention for bytecode by moving all bytecode strings to read-only memory. However, Song et al. showed that the timing window during code generation in which JIT'ed code is writable is sufficiently large to still inject code directly into the code cache [21]. The authors proposed to mitigate this attack by relocating the dynamic code generator to a separate process in which the code cache remains writable, while the original JIT process only maps a read-only view of the generated code.

8.2 JIT Spraying

The JIT spraying attack presented by Blazakis injects code indirectly [12]. Blazakis observed that JIT compilers copy constant integers unmodified into the JIT-compiled code region. An attacker can therefore embed instructions into constants used in a script to inject short code snippets into the JIT code cache. The injected code can be executed by jumping into the middle of instructions that encode the injected constants.

This attack was initially mitigated using a defense called constant blinding [23]. Constant blinding masks embedded constants by XORing them with a random constant value before they are used. However, this defense has since been bypassed by Athanasakis et al. [24], who showed that JIT spraying is also possible with smaller constants and that applying constant blinding to smaller constants is prohibitively expensive. Similarly, Maisuradze et al. demonstrated a variant of the JIT spraying attack that uses carefully placed branch statements to inject useful code into the code cache [25]. This attack cannot be mitigated by constant blinding at all.

As an alternative to constant blinding, Homescu et al. proposed to apply code randomization to JIT-compiled code [26]. With code randomization, it is still possible to spray the code cache with machine code that is embedded in constants, but the location of these constants becomes less predictable.

8.3 JIT Code Reuse

Snow et al.'s JIT-ROP attack showed that code randomization for JIT-compiled code can be bypassed by disclosing a pointer to the JIT code cache, and by recursively disassembling the code starting from the instruction pointed to by the disclosed pointer [1]. This technique allows attackers to discover the locations of injected constants, which they can then jump to to execute the embedded code. Execute-No-Read [27, 28] and destructive code read defenses [29, 30] normally prevent such code-disclosure attacks, but were proven ineffective in the context of JIT VMs [31].

8.4 Intermediate Data Structure Corruption

Theori proposed an attack that corrupts a temporary native code buffer in Microsoft's JavaScript engine (Chakra) [3]. This temporary buffer is used to store machine code before the JIT compiler emits it into a non-writable memory region. Microsoft subsequently added checksum verification logic to the JavaScript engine to verify integrity of the temporary buffer. Similar to this attack, Frassetto et al. corrupt intermediate representation (IR) code which the JIT compiler temporarily produces from bytecode to apply code optimizations and to generate machine code [4]. To defend against this attack, Frassetto et al. proposed a defense called JITGuard, which moves the JIT compiler and its data into an Intel SGX environment that is shielded from the application. JITGuard emits code to a secret region which is only known to the JIT compiler. Since the code is inaccessible to the attacker this also prevents code-reuse attacks targeting JIT'ed code. Our work bears similarity with these approaches in that we corrupt internal data structures of the VM to cause malicious code execution. Unlike the previous approaches, however, we corrupt *bytecode* which is considered more challenging to manipulate due to its restricted capabilities. Frassetto et al. mentioned in their discussion that corrupting bytecode is challenging and is out of scope [4].

Schuster et al. [32] and subsequent works [18, 33] presented whole-function code-reuse attacks that defeat strong CFI and code randomization defenses. These attacks exploit the dynamic dispatch mechanisms present in C++ and Objective-C resp. The general approach is similar to our attack strategies 3 and 4 which rely on a malicious function call. Our defense is also inspired by existing work [3, 18, 22] that uses hash checksums to verify the integrity of sensitive pointers, runtime metadata, and JIT code caches, respectively.

9 Conclusion

We presented an attack that achieves arbitrary code execution in bytecode interpreters that deploy language-level security mechanisms to prevent unauthorized access to files, sockets, or APIs. Our attack leverages a heap-based buffer overflow vulnerability in the interpreter to leak the contents of its heap, infer the heap layout, and overwrite or inject bytecode and program data. We also presented a defense that thwarts our attack by moving all bytecode to read-only memory pages, and by adding integrity checks to all bytecode pointer dereferences.

We evaluated our attack and defense on CPython 2.7.12 and Lua 5.3.2. Our evaluation shows that our defense incurs an average run-time overhead of less than 16% over a large set of Python and Lua benchmarks.

10 Acknowledgments

We would like to thank Stephen Crane, Bart Coppens, our shepherd Vasileios Kemerlis, and the anonymous reviewers for their valuable input.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124 and FA8750-15-C-0085, by the National Science Foundation under award numbers CNS-1513837 and CNS-1619211, and by the Office for Naval Research under award number N00014-17-1-2782.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, the Office for Naval Research, or any other agency of the U.S. Government.

References

1. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.: Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: IEEE Symposium on Security and Privacy (S&P). (2013)
2. Song, C., Zhang, C., Wang, T., Lee, W., Melski, D.: Exploiting and protecting dynamic code generation. In: Symposium on Network and Distributed System Security (NDSS). (2015)
3. Theori: Chakra JIT CFG bypass. `http://theori.io/research/chakra-jit-cfg-bypass` (2016)
4. Frassetto, T., Gens, D., Liebchen, C., Sadeghi, A.R.: JITGuard: Hardening just-in-time compilers with sgx. In: ACM Conference on Computer and Communications Security (CCS). (2017)
5. Willam Forbes: The PyPI Python Package Hack. `https://www.bytelion.com/pypi-python-package-hack` (2017)
6. Rieck, K., Krueger, T., Dewald, A.: Cujo: Efficient detection and prevention of drive-by-download attacks. In: Annual Computer Security Applications Conference (ACSAC). (2010)
7. Oracle Corporation: Securitymanager (java platform se 8). `https://docs.oracle.com/javase/8/docs/api/java/lang/SecurityManager.html` (2018)
8. GitBook: Lua sandbox library (1.2.7). `http://mozilla-services.github.io/lua_sandbox` (2017)
9. Google Developers: Introduction — caja. `https://developers.google.com/caja/` (2018)
10. GitHub: zopefoundation/restrictedpython: A restricted execution environment for python to run untrusted code. `https://github.com/zopefoundation/RestrictedPython` (2018)
11. Google Cloud: Google app engine: Build scalable web and mobile backends in any language on google's infrastructure. `https://cloud.google.com/appengine/` (2018)
12. Blazakis, D.: Interpreter exploitation: Pointer inference and JIT spraying. BlackHat DC (2010)
13. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: ACM Conference on Computer and Communications Security (CCS). (2005)
14. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing forward-edge control-flow integrity in GCC & LLVM. In: USENIX Security Symposium. (2014)
15. Niu, B., Tan, G.: Per-input control-flow integrity. In: ACM Conference on Computer and Communications Security (CCS). (2015)

16. Lua: Lua: uses. `https://www.lua.org/uses.html` (2018)
17. van der Veen, V., Andriesse, D., Stamatogiannakis, M., Chen, X., Bos, H., Giuffrida, C.: The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In: ACM Conference on Computer and Communications Security (CCS). (2017)
18. Lettner, J., Kollenda, B., Homescu, A., Larsen, P., Schuster, F., Davi, L., Sadeghi, A.R., Holz, T., Franz, M., Irvine, U.: Subversive-c: Abusing and protecting dynamic message dispatch. In: USENIX Annual Technical Conference. (2016)
19. Python Performance Benchmark Suite 0.6.1 documentation: The python performance benchmark suite. `http://pyperformance.readthedocs.io` (2017)
20. Alioth: The computer language benchmarks game. `http://benchmarksgame.alioth.debian.org` (2017)
21. Song, C., Zhang, C., Wang, T., Lee, W., Melski, D.: Exploiting and protecting dynamic code generation. In: Symposium on Network and Distributed System Security (NDSS). (2015)
22. Mashtizadeh, A.J., Bittau, A., Boneh, D., Mazières, D.: CCFI: Cryptographically enforced control flow integrity. In: ACM Conference on Computer and Communications Security (CCS). (2015)
23. Rohlf, C., Ivnitskiy, Y.: Attacking clientside JIT compilers. Black Hat USA (2011)
24. Athanasakis, M., Athanasopoulos, E., Polychronakis, M., Portokalidis, G., Ioannidis, S.: The devil is in the constants: Bypassing defenses in browser JIT engines. In: NDSS. (2015)
25. Maisuradze, G., Backes, M., Rossow, C.: What cannot be read, cannot be leveraged? revisiting assumptions of JIT-ROP defenses. In: USENIX Security Symposium. (2016)
26. Homescu, A., Brunthaler, S., Larsen, P., Franz, M.: Librando: transparent code randomization for just-in-time compilers. In: ACM Conference on Computer and Communications Security (CCS). (2013)
27. Backes, M., Holz, T., Kollenda, B., Koppe, P., Nürnberger, S., Pewny, J.: You can run but you can't read: Preventing disclosure exploits in executable code. In: ACM Conference on Computer and Communications Security (CCS). (2014)
28. Crane, S., Liebchen, C., Homescu, A., Davi, L., Larsen, P., Sadeghi, A.R., Brunthaler, S., Franz, M.: Readactor: Practical code randomization resilient to memory disclosure. In: IEEE Symposium on Security and Privacy (S&P). (2015)
29. Tang, A., Sethumadhavan, S., Stolfo, S.: Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In: ACM Conference on Computer and Communications Security (CCS). (2015)
30. Werner, J., Baltas, G., Dallara, R., Otterness, N., Snow, K.Z., Monrose, F., Polychronakis, M.: No-execute-after-read: Preventing code disclosure in commodity software. In: ACM Symposium on Information, Computer and Communications Security (ASIACCS). (2016)
31. Snow, K.Z., Rogowski, R., Werner, J., Koo, H., Monrose, F., Polychronakis, M.: Return to the zombie gadgets: Undermining destructive code reads via code inference attacks. In: IEEE Symposium on Security and Privacy (S&P). (2016)
32. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.R., Holz, T.: Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In: IEEE Symposium on Security and Privacy (S&P). (2015)
33. Crane, S., Volckaert, S., Schuster, F., Liebchen, C., Larsen, P., Davi, L., Sadeghi, A.R., Holz, T., Sutter, B.D., Franz, M.: It's a TRaP: Table randomization and protection against function reuse attacks. In: ACM Conference on Computer and Communications Security (CCS). (2015)