Institute of Computer Science, University of Tartu

Summary of "Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters"

Kaspar Kadalipp

Abstract

This is a summary of a paper published by Sean Heelan, Tom Melham and Daniel Kroening. This paper presents a system for automatically discovering primitives and constructing exploits using heap overflows in interpreters. Gollum differs from most other AEG solutions in that it is entirely greybox, relying on lightweight instrumentation and various kinds of fuzzing-esque input generation. It managed to produce exploits from 10 unique vulnerabilities in the PHP and Python interpreters.

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. 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. Focus is on heap overflow as it is among the most common type of vulnerability in interpreters.

2 Problem to solve

2.1 Practical Applicability

The goal is to construct a control-flow hijacking primitive or an absolute-write primitive. Essentially to hijack a function pointer on a heap or to write a value to an address you control. Primitive is a building block for an exploit, it's an input that uses a vulnerability trigger to provide a useful capability to an exploit developer.

To achieve this, firstly allocate the heap source and the destination pointer. Then manipulate the heap such that they are beside each other. Now after the heap overflow is triggered the destination gets overwritten and points to something else. So when this pointer gets called, instead of crashing it now points to a series of instructions.

2.2 Assumptions

(1) The heap starting state in known beforehand.

(2) Heap allocator is deterministic.

(3) User can provide the system with a means to break Address SpaceLayout Randomisation (ASLR).

(4) Control-flow integrity (CFI) is not deployed on the target binary.

3 System Overview

3.1 New Input Generation

Interpreters tend to come with large test suites, which can be used to mine patterns from existing code. To generate programs that prove them with the required capability, they inject the vulnerability triggers gotten from existing bug reports before and after every function call in existing test cases.

The goal is to cause corruption, ideally immediately before that corrupted data will be used. There's no point in injecting code immediately at the start or the end. At the start of the program, there are no objects created so there's nothing to corrupt. And at the end of the program, there is no code to execute afterward.

3.2 Heap Layout Manipulation

Now they have a large number of inputs that allocate and use objects, as well as trigger a vulnerability. But what the vulnerability corrupts depends on the heap layout, and the heap layout is variable. An input can corrupt pointers that are never used, collateral corruption can cause crashes or won't provide sufficient control over the values used in writes/calls. So given a heap source buffer and a destination buffer, the goal is to discover an input that places them adjacent to each other.

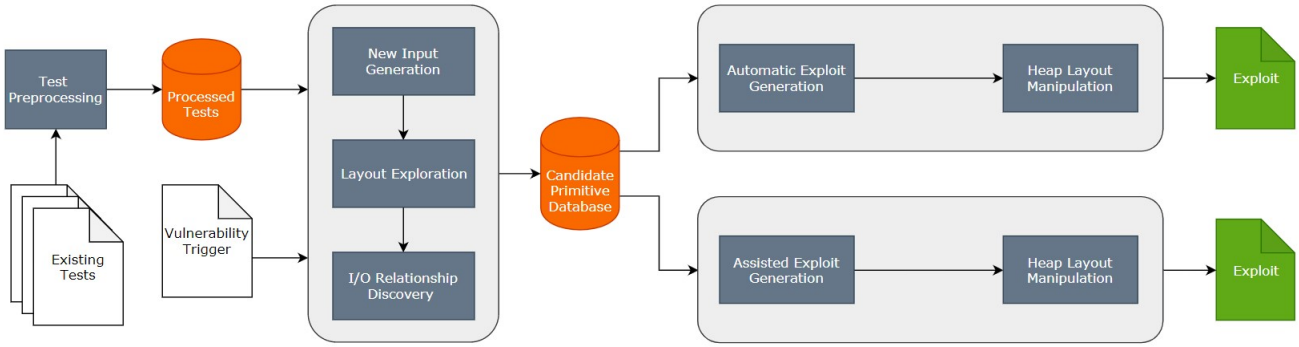

Figure 1: Gollum workflow model [1]

3.3 Layout Exploration

The primitive discovery is performed in a greybox manner, by running the application under different heap layouts and documenting crashes. Gollum utilizes a heap allocator ShapeShifter that allows for a particular heap layout to be selected. When overflow is detected, each live allocation is then scanned to see if it contains a pointer. The idea is to assume a heap layout and later only solve it if the layout proves to be useful.

3.4 Input-Output Relationship Discovery

Output from previous step gives a list of items, indicating if the program is run with certain layout then a crash will occur and the details are provided in the report. Here the goal is to figure out what bytes in the input influence the bytes that end up in memory. Having a control over bytes is sufficient for a large number of vulnerabilities. Traditional taint analysis engines struggle on language interpreters.

3.5 Exploitation Primitive Usage

Now they have a candidate primitive database (Figure 1) which contains a primitive category and I/O relationships linking input bytes to registers and memory. Gollum can automatically generate exploits from two primitive categories. First is IP-hijack, that allows Gollum to jump to an address in libc that spawns a /bin/sh shell. Second is arbitrary write, that allows Gollum to rewrite .got.plt address to spawn /bin/sh shell.

3.6 Evaluation

Entire system process so far has been run using the custom heap allocator to assume a heap layout holds. So now it's needed to find inputs to force this layout. They do this by extracting random fragments from the existing tests again and then run a genetic algorithm that searches for combinations of those fragments that achieve the layout [2].

For evaluation they took 10 vulnerabilities across Python and PHP interpreters. These vulnerabilities give complete control over a variable number of bytes after the heap allocated object and exploits were successfully generated for each one them.

3.7 Limitations

- Gollum cannot work on hard targets such as Google Chrome running on Windows 10.
- The vulnerability must be able to corrupt contiguous bytes in the memory. So it cannot handle vulnerabilities such as CVE-2019-6977 which allows a user to corrupt every 8th byte beyond the buffer overflow.

4 Conclusion

AEG as a monolithic task is daunting, but by breaking it down its possible to attack sub-problems and compose solutions. The authors demonstrated Gollum, a system for automatic exploit generation using heap-based overflows in interpreters. The main motivator was to put together a pipeline that was completely gray-box from end to end and that was shown to be feasible.

5 References

[1] Heelan, Sean & Melham, Tom & Kroening, Daniel. (2019). Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters. 1689-1706. 10.1145/3319535.3354224.

[2] Sean Heelan, Tom Melham, and Daniel Kroening. 2018. Automatic Heap LayoutManipulation for Exploitation. In27th USENIX Security Symposium (USENIXSecurity 18). USENIX Association, Baltimore, MD, 763–779.