Cybersecurity

RESEARCH Open Access

From proof-of-concept to exploitable

(One step towards automatic exploitability assessment)

Yan Wang[1,4,5,6], Wei Wu[1,3,4], Chao Zhang[2], Xinyu Xing[3], Xiaorui Gong[1,4*] and Wei Zou[1,4]

Abstract
Exploitability assessment of vulnerabilities is important for both defenders and attackers. The ultimate way to assess the exploitability is crafting a working exploit. However, it usually takes tremendous hours and significant manual efforts. To address this issue, automated techniques can be adopted. Existing solutions usually explore in depth the *crashing paths*, i.e., paths taken by proof-of-concept (PoC) inputs triggering vulnerabilities, and assess exploitability by finding *exploitable states* along the paths. However, exploitable states do not always exist in crashing paths. Moreover, existing solutions heavily rely on symbolic execution and are not scalable in path exploration and exploit generation. In this paper, we propose a novel solution to generate exploit for userspace programs or facilitate the process of crafting a kernel UAF exploit. Technically, we utilize oriented fuzzing to explore diverging paths from vulnerability point. For userspace programs, we adopt a control-flow stitching solution to stitch crashing paths and diverging paths together to generate exploit. For kernel UAF, we leverage a lightweight symbolic execution to identify, analyze and evaluate the system calls valuable and useful for exploiting vulnerabilities.
We have developed a prototype system and evaluated it on a set of 19 CTF (capture the flag) programs and 15 realworld Linux kernel UAF vulnerabilities. Experiment results showed it could generate exploit for most of the userspace test set, and it could also facilitate security mitigation bypassing and exploitability evaluation for kernel test set.

Keywords: Exploit, Vulnerability, Taint analysis, Fuzzing, Symbolic execution

Introduction

Due to the success of automated vulnerability discovery solutions (e.g., fuzzing), more and more vulnerabilities are found in real world applications, together with proof-of-concept (PoC) inputs. As a result, more and more human resources are spent on assessing vulnerabilities, e.g., identifying root causes and fixing them. It thus calls for solutions to automatically assess the severity and priority of vulnerabilities.

Vulnerability assessment, especially exploitability assessment, is important for both defenders and attackers. Attackers could isolate exploitable vulnerabilities and write exploits to launch attacks. On the other hand, defenders could prioritize exploitable vulnerabilities to fix first, and allocate resources accordingly. Moreover, defenders could learn from the exploits to generate IDS (Intrusion Detection System) signatures, to block future attacks.

A straightforward way to assess a vulnerability is *analyzing the program state at the crashing point*, i.e., the instruction leading to program crashes or security violations, which could be caught by a sanitizer (e.g., AddressSanitizer (Serebryany et al. 2012)). For example, Microsoft's `!exploitable` tool (!exploitable Crash Analyzer 2018) inspects all instructions in the crashing point's basic block, and searches for known exploitable patterns, e.g., control transfer instructions with tainted targets. `HCSIFTER` (He et al. 2017) takes an extra step to recover the data corrupted by heap overflow, enabling the program to execute more code after the crashing point, and thus provides more reliable assessments. However, these solutions rely on heuristics to determine the exploitability of vulnerabilities, and thus are inaccurate sometimes.

*Correspondence: gongxiaorui@iie.ac.cn
[1]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[4]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
Full list of author information is available at the end of the article

Springer Open

© The Author(s). 2019 **Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

Moreover, they could not provide exploit inputs to prove the exploitability.

The ultimate way to assess the exploitability of a vulnerability is *generating a working exploit*. But crafting an exploit is typically regarded as a time-consuming manual process requiring security knowledge.

Several prototype approaches to automatically generating exploits have been proposed. Sean Heelan proposed a prototype (Heelan 2009) in his thesis, using dynamic analysis and symbolic execution to generate exploits for classic buffer overflow vulnerabilities. AEG (Avgerinos et al. 2011) and Mayhem (Cha et al. 2012) provide end-to-end systems to discover vulnerabilities and automatically generate exploits when possible, for source code and binary respectively. Q (Schwartz et al. 2011) and CRAX (Huang et al. 2012) could generate exploits for binaries given PoC inputs. However, these solutions are insufficient and could only solve a small number of problems. For example, machines developed in CGC could only solve in total 26 out of 82 challenge programs in the Final Event. Most solutions could not exploit heap-based vulnerabilities.

For OS kernel which has higher complexity and scalability, it is not suitable for fully-automated exploit generation. This is mainly due to the fact that state-of-the-art program analysis techniques have many limitations. However, we can still use semi-automated techniques to facilitate exploitability evaluation by easing the process of exploit crafting.

There are several challenges need to be addressed for both fully-automated and semi-automated exploit generation:

Challenge 1: Exploit derivability issue As pointed in (Dullien and Flake 2011; Vanegue 2013), once memory corruption vulnerabilities are triggered, the victim program's state machine turns into a *weird (state) machine*. Exploitation is actually a process of programming the *weird machine* to perform unintended behavior. It is extremely important to set up the initial state of this weird machine in order to exploit it.

However, PoC inputs (e.g., provided by fuzzers) could corrupt some data and lead weird machines to non-exploitable initial states. For example, the program may exit soon after the crashing point due to some sanity checks. So, AEG solutions have to search for exploitable states not only in *crashing paths* taken by PoC inputs, but also in alternative *diverging paths*. In OS kernel, the diverging paths cause different kernel panic. Generating an exploit for a kernel UAF vulnerability also needs to vary the context of a kernel panic and explore exploitability in them.

This is known as *exploit derivability*, one of the core challenges of exploitation (Vanegue 2013). Few AEG solutions have paid attentions to this issue.

Challenge 2: Symbolic execution bottleneck Existing solutions heavily rely on symbolic execution to explore program paths (e.g., for vulnerability discovery), or perform reasoning (e.g., for test case and exploit generation). AEG (Avgerinos et al. 2011) and Mayhem (Cha et al. 2012) utilize symbolic execution to explore paths reachable from the vulnerability point and search for exploitable states, able to mitigate the aforementioned exploit derivability issue. However, symbolic execution has scalability issues and performs poorly in exploit generation.

First, it faces the *path explosion* issue when exploring paths, and consumes too many resources even when analyzing only one path. Second, it gets blind to certain exploitable states after concretizing some values. For example, it has to concretize symbolic arguments of memory allocations and symbolic indexes of memory access operations in a path, in order to model the memory states and enable exploring following sub-paths. But the concretized values could lead to non-exploitable memory states.

Challenge 3: Exploiting Vulnerability in OS kernel Kernel vulnerabilities have higher complexity than other vulnerabilities, and there is no solution can facilitate exploitability evaluation for them. Usually, vulnerabilities in OS kernel could lead to privilege escalation (Azad 2016) and critical data leakage (jndok 2016).

To solve the exploit derivability issue, we must search exploitable states in diverging paths not only crashing paths. However, symbolic execution which is heavily used in existing solutions has several severe challenges, and is not suitable for path exploration or exploitable state searching, especially for heap-based vulnerability or UAF in OS kernel So instead of symbolic execution, we use fuzzing to explore diverging paths.

First, we use dynamic analysis to analyze the vulnerabilities and collect some runtime information in the crashing path. In addition, we inspect corrupted memory objects (denoted as *exceptional objects*), and objects that can be used to locate the exceptional objects. Then we use oriented fuzzing to search alternative diverging paths for exploitable states based on the information collected before. Finally, we try to synthesize new *EXP inputs* to trigger both the exploitable states in diverging paths and vulnerabilities in crashing paths. In certain cases, we can directly generate working exploits. But it is not guaranteed. The complexity of OS kernel is far beyond the ability of current constraint solver. For OS kernel, it is not for the purpose of fully automating exploit generation. Rather, we leverage a *lightweight symbolic execution* to explore exploitability under different contexts.

Results We have build a framework `Revery`, able to generate working control-flow hijacking exploits for userspace programs. We also build a framework `FUZE`,

able to evaluate the exploitability of kernel Use-After-Free vulnerabilities.

We evaluated `Revery` it on 19 CTF (Capture The Flag) programs. It demonstrated that `Revery` is effective in triggering exploitable states, and could generate working exploits for a big portion of them. More specifically, `Revery` could generate exploits for 9 (47%) out of 19 programs, while existing open source AEG solutions could not solve any of them. Furthermore, it could trigger exploitable states for another 5 (26%) of them.

We implement `FUZE` on a 64-bit Linux system by extending a binary analysis framework and a kernel fuzzer. Using 15 real-world kernel UAF vulnerabilities on Linux systems, we then demonstrate FUZE could not only escalate kernel UAF exploitability but also diversify working exploits from various kernel panics. In addition, we demonstrate `FUZE` could even help security analysts to craft exploits with the ability to bypass broadly-deployed security mitigation such as `SMEP` and `SMAP`.

In summary, we have made the following contributions:

- We proposed an automated solution `Revery` able to transfer PoC inputs into EXP inputs, which could trigger vulnerabilities and enter exploitable states. It could also directly generate working exploits in certain cases.
- We designed `FUZE`, an exploitation framework that utilizes kernel fuzzing along with symbolic execution to facilitate kernel UAF exploitation as well as facilitating security mitigation circumvention.
- We proposed a novel *layout-oriented fuzzing* solution, to search for exploitable states in diverging paths, without symbolic execution.
- We have implemented a prototype of `Revery` and `FUZE`, and demonstrated its effectiveness in CTF programs and real world UAF vulnerabilities in Linux kernels..

Motivation example

In this section, we will illustrate the exploit derivability issue facing by automated exploit generation solutions, and present the overview of our solution `Revery`, with a running example demonstrated in Fig. 1.

The vulnerability

As shown in Fig. 1, there is a heap overflow vulnerability at line 10. The two objects `obj1` and `obj2` have the same size, and are likely to be allocated next to each other in the heap. If the vulnerability condition `vul` at line 9 is met, lengthy inputs could cause an overflow in the buffer `obj1->data`. As a result, objects (e.g., `obj2`) following this buffer will be corrupted.

Therefore, the statement at line 12 and 14 will read from and write to corrupted memory address respectively. If the corrupted pointer `obj2->ptr` points to invalid (e.g., nonexistent) memory, these two statements will cause crashes. If it points to valid memory, the statement at line 12 will execute normally (but result in wrong return value), while the statement at line 14 will further corrupt the target memory and cause Arbitrary Address Write (AAW).

From the perspective of exploitation, the statement at line 12 is non-exploitable, unless the returned value *res* affects control-flow in caller functions. But the statement at line 14 triggers an exploitable state. It causes an AAW primitive able to overwrite arbitrary targets, including the global function pointer `handler` which is invoked at line 15, and thus could cause control-flow hijacking at line 15.

Exploit derivability

As discussed in Vanegue (2013), exploit derivability is one of the core challenges of exploitation. More specifically, given a PoC input for a vulnerability, the program could be turned into a *weird machine*, but with a non-exploitable initial state. To successfully exploit the

```
1.  struct Type1 { char[8] data;                          };
2.  struct Type2 { int status;    int* ptr;  void init(){…};  };
3.  int (*handler)(const int*) = …;
4.  struct{Type1* obj1; Type* obj2;} gvar = {};
5.  int foo(){
6.    gvar.obj1 = new Type1;
7.    gvar.obj2 = new Type2;
8.    gvar.obj2->init();  // resulting different statuses
9.    if(vul)
10.     scanf("%s", &gvar.obj1->data);     // vulnerability point
11.   if(gvar.obj2->status)                // stitching point
12.     res = *gvar.obj2->ptr;             // crashing point
13.   else                                 // stitching point
14.     *gvar.obj2->ptr = read_int();      // exploitable point
15.   handler(gvar.obj2->ptr);             // hijacking point
16.   return res;
17. }
```

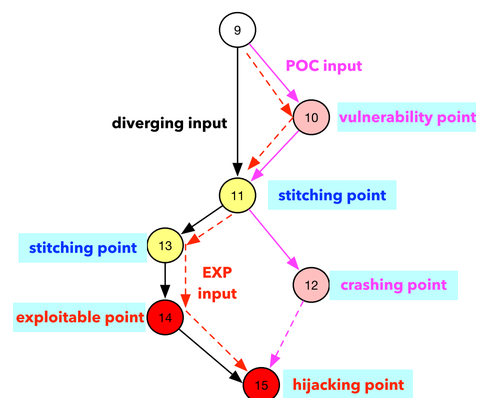

Fig. 1 An example heap overflow. The vulnerability at line 10 could overwrite the following object, i.e., obj2. PoC inputs would crash at line 12 and enter a non-exploitable state. Successful exploits will trigger the exploitable state at line 14

vulnerability, we have to search for exploitable states in alternative diverging paths, and lead the *weird machine* to exploitable.

As shown in the running example, assuming a PoC input proving the vulnerability at line 10 is provided (e.g., by fuzzers), it could overwrite the field `obj2->status` to non-zero, and overwrite `obj2->ptr` to invalid memory address, and cause a crash at line 12. So this PoC leads the *weird machine* to a non-exploitable initial state. A successful exploitation has to trigger the vulnerability (at line 10) and enter an exploitable state (e.g., at line 14).

For simplicity, we introduce several terminologies:

- **Crashing path:** the path taken by the PoC input, e.g., the path `9->10->11->12` in the example.
- **Crashing point:** the instruction where the program crashes or a security violation is caught by sanitizers, e.g., line 12.
- **Vulnerability point:** the instruction where the vulnerability (i.e., security violation) happens, e.g., line 10 in the example. A crashing path may have multiple security violations. The first violation point is denoted as the vulnerability point.
- **Exploitable point:** the instruction which could lead to a successful exploit, e.g., line 14 in the example. Exploitable points lead to exploitable states where the *weird machine* could work properly. In practice, arbitrary address read/write/execute instructions (AAR/AAW/AAX) are classical exploitable points.
- **Diverging path:** the path where exploitable states could be found, e.g., `9->11->13->14` in the example.
- **Hijacking point:** the instruction where the control-flow could be hijacked, e.g., line 15 in the example. They are special exploitable points. In the running example, it is a second-order exploitable point, caused by the first exploitable point in line 14.
- **Exploitation path:** the path taken by a successful exploit, e.g., `9->10->11->13->14->15` in the example.
- **Stitching points:** special instructions in the diverging path and crashing path, which could be stitched together to generate the exploitation path, e.g., line 11 and line 13 in the example. In practice, there may be numerous sub-paths between two stitching points to explore.

It is worth noting that, the crashing point (line 12) in the running example could reach to the hijacking point (line 15), but it is not exploitable. As aforementioned, this hijacking point is a second-order exploitable point, made by the exploitable point in line 14. Without the help of line 14, line 15 could not be exploited.

So, to conduct successful exploitations, we have to think outside the box made by the original PoC, and search for exploitable states in diverging paths. This is the intuition of our solution and the origin of the name `Revery`. To the best of our knowledge, existing AEG solutions paid few attentions to this *exploit derivability* issue.

Our solution: `Revery`

We proposed a novel solution `Revery`, to solve the *exploit derivability* issue and assess the exploitability of heap-based vulnerabilities. At the high level, `Revery` analyzes the vulnerability in detail, utilizes the vulnerability information to guide a fuzzer rather than symbolic execution to explore diverging paths and search for exploitable states, then synthesizes exploitation paths by stitching the crashing path and diverging path, and finally generates inputs to trigger both the vulnerability and exploitable states. As shown in Fig. 2, it has three major components.

Vulnerability analysis

`Revery` first analyzes the vulnerability in detail, similar to existing AEG solutions. It uses dynamic analysis to test target application with the provided PoC input. More specifically, it tracks the states of each pointer and memory object, and catches security violations along the crashing path. It could thus identify the vulnerability point, e.g., line 10 in Fig. 1.

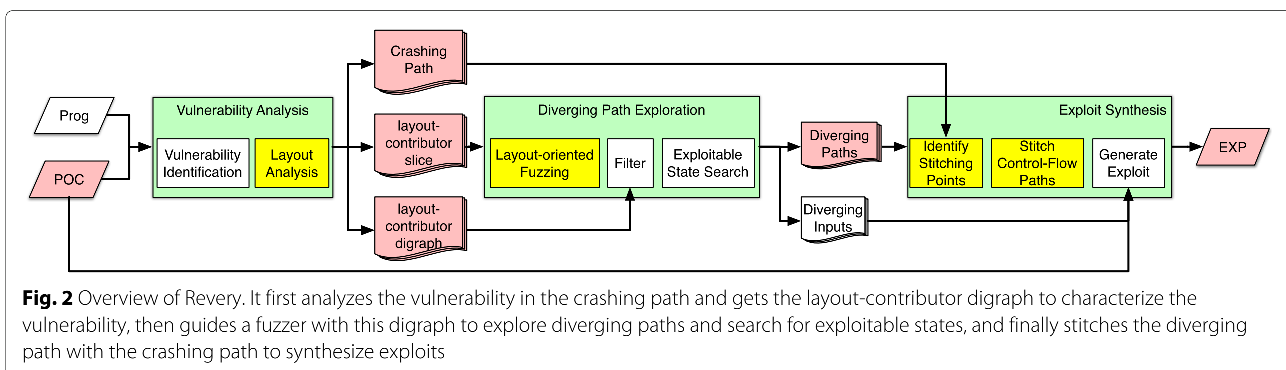

Fig. 2 Overview of Revery. It first analyzes the vulnerability in the crashing path and gets the layout-contributor digraph to characterize the vulnerability, then guides a fuzzer with this digraph to explore diverging paths and search for exploitable states, and finally stitches the diverging path with the crashing path to synthesize exploits

More importantly, it identifies *exceptional objects* corrupted by the vulnerability, e.g., `obj2` in the example. `Revery` also identifies the exceptional object's indexing objects, which could be used to locate the exceptional object, e.g., the global variable `gvar` in the example. Moreover, it retrieves *layout-contributor* instructions from the execution trace, which create the exceptional and indexing objects and set up their point-to relationships, e.g., line 7 in the example. These objects and contributor instructions are used to construct a layout-contributor digraph.

Diverging path exploration

`Revery` searches for exploitable states in diverging paths, to solve the *exploit derivability* issue. Rather than using symbolic execution, it employs fuzzing.

First, it employs a novel *layout-oriented fuzzing* solution to explore diverging paths. To facilitate exploit generation, only diverging paths with memory layouts similar as the PoC input's will be explored. So, it drives a fuzzer to explore paths close to the crashing path, in a similar way as directed fuzzing solutions (Böhme et al. 2017). But instead of using the full crashing path, it uses the aforementioned *layout-contributor* instructions as the fuzzer's guidance. The fuzzer could thus produce diverging inputs to exercise the diverging paths (e.g., `9->11->13->14` in the figure) with proper memory layouts.

Then, `Revery` searches for exploitable states in the diverging paths. Several heuristics are used to identify exploitable states. For example, if a memory store operation's destination is controlled by the corrupted object, e.g., line 14, it is an exploitable state.

Furthermore, `Revery` also searches for hijacking points in these diverging paths. Hijacking points sometimes are not obvious. So `Revery` uses some heuristics to infer hijacking points. For example, line 15 in the figure is a second-order hijacking point, which could be enabled if line 14 overwrites the global function pointer.

PoC stitching

Once an exploitable state (together with a diverging input) in a diverging path is found, `Revery` will try to synthesize a new input to trigger both the vulnerability and the exploitable state. In general, it first finds the stitching points in the crashing path (e.g., line 11) and in the diverging path (e.g., line 13), with some specific data flow analysis.

Then it utilizes a lightweight symbolic execution to explore potential sub-paths between these two stitching points (e.g., `11->13`), and stitch the crashing path with the diverging path to synthesize an exploitation path (e.g., `9->10->11->13->14->15`), and finally generate inputs to exercise the exploitation paths. Several optimizations are deployed to make the symbolic execution lightweight.

Therefore, `Revery` could produce *EXP inputs* able to trigger both vulnerabilities and exploitable states. It could help experts to quickly generate working exploits. In certain cases, `Revery` is able to directly generate exploits. For example, `Revery` could generate an exploit input to hijack the control flow, by utilizing the exploitable state at line 14 to overwrite the global function pointer `handler`.

Our solution: `FUZE`

We also proposed a novel solution `FUZE`, to solve the *exploit derivability* issue in OS kernel and evaluate the exploitability of kernel Use-After-Free vulnerabilities. We design `FUZE` to first run a PoC program and perform analysis using off-the-shelf address sanitizer. Along with the facilitation of a dynamic tracing approach, `FUZE` could identify the critical information pertaining to the vulnerable objects as well as the time window needed for consecutive exploitation.

Using the information identified, we then design `FUZE` to automatically vary the contexts of that PoC for the purpose of easing the process of synthesizing new PoC programs. We alter the context of a PoC program by inserting a new system call that dereferences the vulnerable object in between the occurrence of the dangling pointer and its dereference. Technically speaking, we therefore design and develop an under-context fuzzing approach, which automatically explores the kernel code space in the time window identified and thus pinpoints the system calls (and corresponding arguments) that can drive the kernel panic in a new context.

Similar to the context represented by that original PoC, a new context (.i.e new kernel panic) does not necessarily assist an analyst to craft a working exploit. Moreover, a security analyst generally has difficulty in determining, following which contexts he could craft a working exploit. Therefore, we further design `FUZE` to automatically evaluate each of the new contexts. Intuition suggests that we could summarize a set of exploitable machine states based on the exploitation approaches commonly adopted. For each context, we could then examine whether the corresponding terminated kernel state matches one of these exploitable machine states.

`FUZE` sets each byte of the freed object as a symbolic value and then perform symbolic execution under each context. This allows `FUZE` to explore the exploitable machine states in a more complete fashion and thus thoroughly pinpoint the set of contexts useful for exploitation.It should be noted that, symbolic execution under the context does not mean that symbolically executing kernel code at the site of kernel panic. Rather, it means that we perform symbolic execution right after the site of dangling pointer dereference. As we will

demonstrate and discuss in the following section, such a design could prevent incurring path explosion without reaching to any sites useful for exploitation. In addition, it enables FUZE to use off-the-shelf constraint solvers to accurately compute the content that needs to spray in between the occurrence of a dangling pointer and its dereference.

Vulnerability analysis

To exploit a vulnerability, it is necessary to locate the vulnerability point and the program state at that point. Furthermore, to solve the exploit derivability issue, exploitable states around the vulnerability state should be searched for. Therefore, Revery performs *vulnerability identification* to locate the vulnerability, and performs *layout analysis* to characterize the vulnerability state. And FUZE extracts information needed for consecutive exploitation by using an off-the-shelf kernel address sanitizer KASAN (KASAN 2017) along with a dynamic tracing mechanism.

Vulnerability identification

Given a PoC input, Revery first needs to identify its corresponding vulnerability point. Dozens of solutions have been proposed to detect memory errors, e.g., AddressSanitizer (Serebryany et al. 2012) and Valgrind (2018). However, AddressSanitizer and Valgrind will slightly change the memory layout of target applications, and thus are not suitable for exploit generation.

Revery utilizes a different technique, named memory tagging (MT, also known as memory coloring, memory tainting, lock and key) to locate vulnerabilities. A recent work (Serebryany et al. 2018) has implemented memory tagging in hardware. However, it encodes tags in memory pointers and thus affects the program states. Moreover, it only detects spatial memory violations, but not temporal violations.

Revery uses a shadow memory to non-intrusively track the tags of pointers and heap objects. It also tracks the status of heap objects, enabling detection of not only spatial vulnerabilities (e.g., heap overflow) but also temporal vulnerabilities (e.g., use-after-free).

In principle, each pointer is expected to access a specific memory object of valid status. If it is used at runtime to access an object of different tags or invalid status, then a security violation is caught. Figure 3 shows an example of vulnerability identification.

Memory tags

Each heap object and pointer is attached with a memory tag, indicating its lineage. This tag will be uniquely generated when an object is created, and propagate to the object's pointers and other related pointers as a taint label (taint analysis). Moreover, each heap object is associated with a status, i.e., uninitialized, busy, or free, standing for three status in its life-cycle, i.e., allocated but not initialized, initialized and being used, or freed. It is worth noting that, a freed memory region could be allocated to new objects, and its memory status and tag will change accordingly.

In some corner cases, developers could use one object's pointer to get another object's pointer, with an arithmetic operation. It will wrongly propagate the first object's tag to the second pointer. Fortunately, this is rare for heap objects, since the offsets between heap objects are not fixed. The only exception is heap management functions, which could inspect adjacent objects in this way, no matter what semantics these objects would have. So Revery will

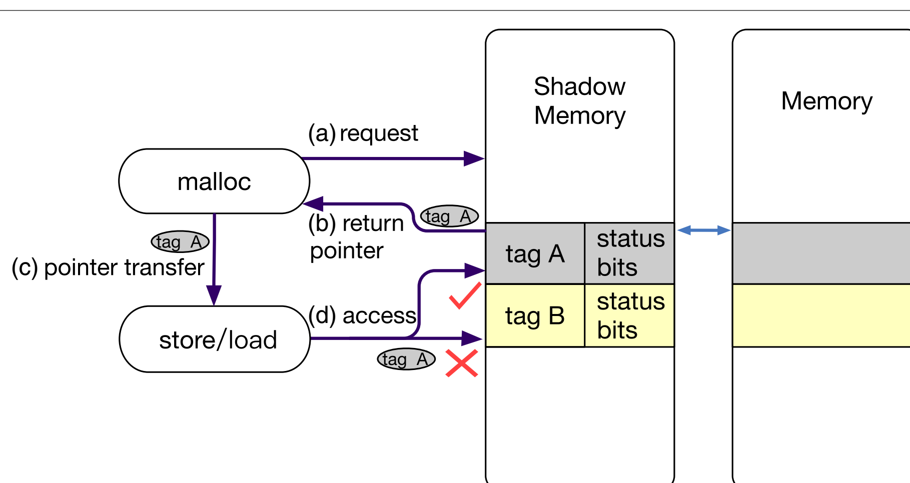

Fig. 3 Illustration of heap-based vulnerability identification. Each heap object and pointer is associated with a memory tag. An extra status is attached to each memory object

disable tag propagation and validation for these special functions. It is worth noting that, this optimization is only for cross-object pointer deriving. Revery supports in-object pointer driving as normal.

Security rules

For each heap memory access instruction (i.e., load and store), we could get the pointer's tag `tag_ptr` and target memory region's tag `tag_obj` and status `status_obj`. The memory access must not violate the following security rules:

- **V1: access intended objects**: Instructions should only access intended objects, i.e., `tag_obj` and `tag_ptr` must match.
- **V2: read busy objects**: Load instructions should not access freed or uninitialized memory, i.e., `status_obj` must be `busy`.
- **V3: write alive objects**: Store instruction should not access freed memory, i.e., `status_obj` must be `busy` or `uninitialized`.

Any violation of these rules will cause a vulnerability. For example, a buffer overflow memory access will violate the rule V1. An uninitialized variable vulnerability will violate the rule V2. A use-after-free (UAF) vulnerability could violate either rule V1, V2 or V3. If the freed object's memory has not been taken by other objects, then read access to it will violate V2, and write access to it will violate V3. If the freed object's memory is taken, then its tag will change, and any access to it via the original dangling pointer will violate the rule V1.

Layout analysis

Revery further analyzes object layouts to characterize the vulnerability state and retrieve instructions contributing to the state.

Vulnerability-related object layout

Each heap-based vulnerability (including heap overflow and UAF) is related to one *exceptional object*, whose content is (or will be) corrupted by the vulnerability. Further operations on these objects could lead the *weird machine* to exploitable states.

Assume the vulnerability point uses a pointer with tag `tag_ptr` to access a target object with tag `tag_obj`. If it is a write access, the object with tag `tag_obj` is the exceptional object, which will be corrupted by this write access. If it is a read access and this vulnerability is a UAF, the object with tag `tag_ptr` is the exceptional object, which will be corrupted by new object allocations that take the same memory. Revery currently does not support other types of read access violation well.

Further, Revery also tracks all indexing objects that can be used to locate exceptional objects. These exceptional objects and indexing objects are connected with the point-to relationship. As a result, Revery could get a digraph of objects, denoted as *layout digraph*. This layout digraph characterizes the vulnerability state to some extent. Figure 4b shows an example layout digraph.

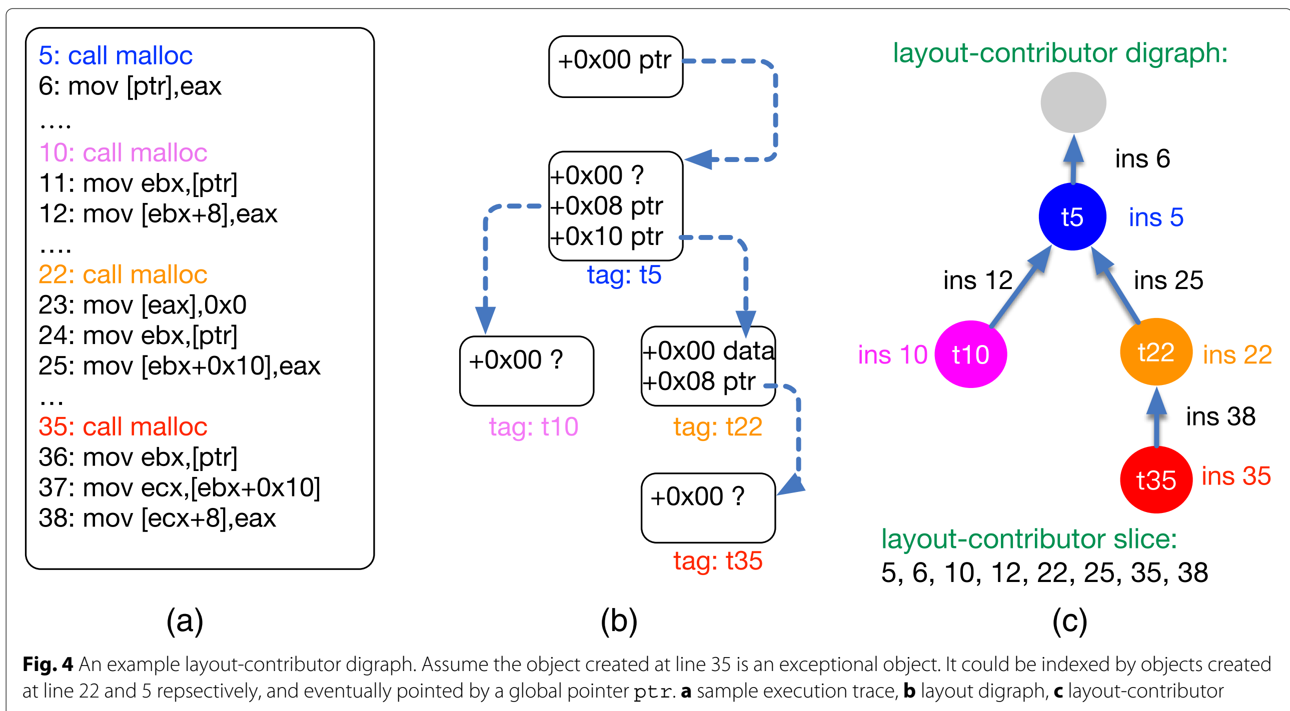

Fig. 4 An example layout-contributor digraph. Assume the object created at line 35 is an exceptional object. It could be indexed by objects created at line 22 and 5 repsectively, and eventually pointed by a global pointer `ptr`. **a** sample execution trace, **b** layout digraph, **c** layout-contributor

Vulnerability-related code

As aforementioned, the *weird machine* has to enter specific initial states, including the vulnerability state. So, it is necessary to prepare a similar object layout as the vulnerability's, both in diverging paths and exploitation paths. Thus, instructions contributing to the layouts are important.

We can see that, the following two types of operations could contribute the object layout: (1) memory allocations that creates new objects, and (2) store operations that assign an object's field with a pointer to another object. As a result, `Revery` could retrieve all such contributor operations, which operate on objects in the layout digraph, and generate a *layout-contributor digraph*.

More specifically, each node in this digraph is an exceptional object or an indexing object, with an attribute of the object's creator instruction and memory tag. Each edge in the digraph represents a point-to relationship between two objects, with an attribute of the pointer assignment instruction. Given a target exceptional object, we could use backward slicing to construct this digraph. Figure 4c shows an example layout-contributor digraph. This digraph has a simpler form, called *layout-contributor slice*, which is a sequence of contributor instructions in execution order.

Critical information extraction in OS kernel

Here, we describe the information extracted through kernel address sanitizer as well as the design of the dynamic tracing mechanism, followed by how we leverage them both to identify other critical information for exploitation.

Information from Kernel Address Sanitizer. `KASAN` is a kernel address sanitizer, which provides us with the ability to obtain information pertaining to the vulnerability. To be specific, these include (1) the base address and size of a vulnerable object, (2) the program statement pertaining to the free site left behind a dangling pointer and (3) the program statement corresponding to the site of dangling pointer dereference.

Design of Dynamic Tracing. In addition to the information extracted through `KASAN`, consecutive exploitation needs information pertaining to the execution of system calls that trigger vulnerabilities. As a result, we design a dynamic tracing mechanism to facilitate the ability of extracting such information. To be specific, we first trace the addresses of the memory allocated and freed in Linux kernel as well as the process identifiers (`PID`) attached to these memory management operations. In this way, we could enable memory management tracing and associate memory management operations to our target PoC program. Second, we instrument the target PoC program with the Linux kernel internal tracer (`ftrace`). This could allow us to obtain the information pertaining to the system calls invoked by the PoC program.

Other Critical Information Extraction. With the facilitation of dynamic tracing along with `KASAN` log, we can extract other critical information needed for exploitation. For example, Using the information obtained through `KASAN`, we can easily identify the address of the vulnerable object and tie it to the free operation indicated by kfree(). With `PID` associated with each memory management operation, we can then pinpoint the life cycle of system calls on the trace and thus identify **close()**, the system call tied to the free operation.

Since system call socket() manifests as an incomplete trace, we can easily pinpoint that it serves as the system call that dereferences the dangling pointer. Associating this information with debugging information and source code, we can easily understand how the dangling pointer was dereferenced and further track down which variable this dangling pointer belongs to.

Diverging path exploration in `Revery`

To solve the exploit derivability issue, it is necessary to explore diverging paths and search exploitable states in them. In this section, we will introduce how `Revery` explores diverging paths.

Alternative choices

Existing automated exploit generation solutions, e.g., AEG (Avgerinos et al. 2011) and Mayhem (Cha et al. 2012), heavily rely on symbolic execution to explore the crashing path or reachable paths from the vulnerability point, in order to search exploitable states along the path exploration. However, symbolic execution has several severe challenges, and is not suitable for path exploration or exploitable state searching.

First, it is not scalable in path exploring. It suffers from the path explosion issue caused by branches and loops in programs. Even when analyzing one path, it costs too many resources. Moreover, the symbolic constraints are often too complicated to solve.

Second, symbolic execution may get blind to certain exploitable states. It has to concretize some symbolic values along the exploration, by adding extra constraints of concretized value assignments. It is impossible to try all candidate concretized values, thus misses certain values and causes blindness to certain exploitable states.

For example, it will concretize the symbolic arguments of *memory allocation* in a path, in order to model the memory states and explore following sub-paths. It is likely that only a small number of allocations could cause exploitable states. So the concretized memory allocation may lead to a non-exploitable state.

Moreover, it will also concretize *symbolic indexes* in memory access operations, because otherwise the operations' results are unknown. Similarly, it could also lead to non-exploitable states.

Layout-oriented fuzzing

Revery utilizes fuzzing solely to explore diverging paths and search for exploitable states. As shown in vulnerability discovery, fuzzing is more effective than symbolic execution in exploring paths and program states. So, it is likely that fuzzing could also help find diverging paths and exploitable states faster.

Revery employs a novel layout-oriented fuzzing solution guided by the layout-contributor digraph, to explore diverging paths that build similar memory layouts as the vulnerability.

Design

Revery extends the popular coverage-guided fuzzer AFL to perform fuzzing. Instead of relying solely on code coverage to guide path exploration, Revery uses layout-contributor digraph as a guidance to tune the direction of exploration and mutation.

Similar to directed fuzzing (Böhme et al. 2017), Revery drives the fuzzer to explore paths close to the crashing path. It only aims at matching instructions in the layout-contributor slice, and ignores other instructions in the crashing path. The design choices are made from the following three intuitions.

For simplicity, we introduce several terminologies. Given an input I_a, it could hit several layout-contributor instructions (maybe not in the same order as the guiding slice). The full list of such instructions is denoted as L_a, and its longest common subsequence (LCS) with the target guiding slice is denoted as P_a.

- **Intuition 1:** An input that hits all layout-contributor instructions, in the same order as the guiding slice, could construct a similar memory layout as the vulnerability. Layout-contributor instructions are responsible for creating the exceptional object of a vulnerability and its indexing objects, as well as setting the point-to relationships among them. So, an input hitting the full layout-contributor slice could probably construct similar memory layouts.
- **Intuition 2:** An input that hits a longer subsequence of the guiding slice is more likely to derive inputs hitting the full slice.
  In other words, if input I_a's LCS P_a is longer than I_b's LCS P_b, then the input I_a is better than I_b. As shown in Fig. 5, assuming the target slice is in path a=>c=>e=>f, then an input exercising the path a=>c=>d is better than other inputs exercising a=>b. Further mutations on this input could derive inputs hitting the full guiding slice faster.
- **Intuition 3:** Inputs hitting fewer layout-contributor instructions are more likely to introduce fewer troubles for further exploit generation.
  In other words, for two inputs I_a and I_b, if their LCS P_a and P_b have a same length, but the layout-contributor instruction list L_a is longer than L_b, then

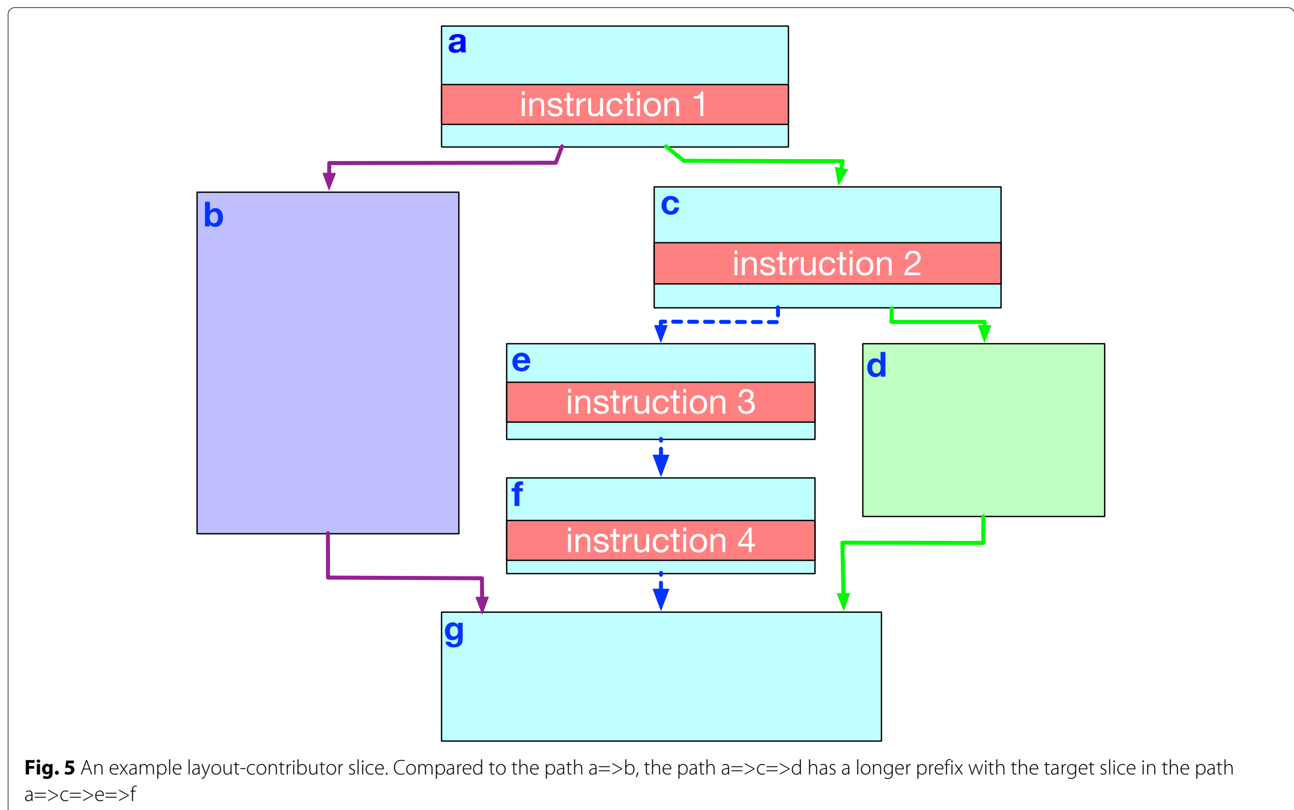

Fig. 5 An example layout-contributor slice. Compared to the path a=>b, the path a=>c=>d has a longer prefix with the target slice in the path a=>c=>e=>f

the input I_b is better than I_a. In this case, the input I_a has more duplicated or out-of-order contributor instructions than I_b, which could cause redundant object creation or layout construction, making the memory layout too complicated to exploit.

Implementation details

`Revery` extends the popular fuzzer AFL (Zalewski 2018). As shown in Fig. 6, AFL applies a continuous loop to explore paths. It (1) keeps a queue of good testcases, i.e., seeds; and (2) selects a seed from the queue; and then (3) mutates the seed to get a bunch of new testcases, and then (4) run the target binary program with the generated testcases in QEMU, and track the coverage, and then (5) identify seeds based on coverage information. `Revery` modifies AFL in the following two aspects.

Tracking Slice Hit Count `Revery` adds an extra buffer `HIT` in the shared memory between QEMU and the fuzzer driver, in addition to the existing bitmap used for code coverage tracking. `HIT[0]` is used to track the count of slice hit, while `HIT[i]` is used to track whether the i-th instruction in the guiding slice has been hit or not.

More specifically, each time a layout-contributor instruction is executed, QEMU will increase the slice hit count `HIT[0]`. If this instruction is the n-th (n>=1) instruction in the guiding slice, then QEMU will set `HIT[n]` if and only if `HIT[n-1]` has been set. In this way, the fuzzer driver could get the slice hit count in `HIT[0]`, and the LCS of guiding slice in `HIT[1:N]`.

Tuning Fuzzing Directions `Revery` modifies the fuzzer driver to make use of the collected slice hit information. Basically, it slightly changes the algorithms of seed selection. When picking up a seed from the queue to mutate, it first prioritizes seeds that have longer LCS, as discussed in Intuition 2. Then among seeds with LCS of same length, it prioritizes seeds with fewer slice hit count, as suggested in Intuition 3. Finally, it prioritizes seeds with smaller size and faster execution time, same as AFL's default policy.

Diverging inputs filtering

With layout-oriented fuzzing, `Revery` could find diverging inputs able to trigger the same layout-contributor slice as the PoC input. However, unlike layout-contributor digraph, the data flow constraints are missing in the layout-contributor slice. So the diverging inputs sometimes do not match the target layout-contributor digraph built from the crashing path. `Revery` thus takes an extra step to isolate diverging inputs that could match the target layout-contributor digraph.

In general, it first aligns the diverging path with the crashing path, and locates the instructions responsible for creating the exceptional object. Then, it constructs a new layout-contributor digraph of the exceptional object from the diverging path by backward slicing, in a same way as the crashing path. Finally, it matches this new digraph against the target digraph, by comparing each node's memory tag and its creator instruction's address in two digraphs. Figure 7 shows an example of how the match works.

If these two digraphs do not match, then this diverging input will be discarded. Otherwise, the diverging input is kept. Moreover, these two digraphs' nodes (i.e., heap objects) will be aligned accordingly, as well as the memory tags of all nodes. So, we could infer each object's counterpart between the diverging path and crashing path, enabling further common analysis on these two paths.

Exploitable states searching

Even if the diverging paths have similar layouts as the vulnerability, not all of them are exploitable. `Revery` further removes diverging paths that do not have exploitable states.

Exploitable state

The exceptional object could affect other objects, and sometimes will be directly or indirectly used in some sensitive operations. The program states resulting from these sensitive operations are denoted as exploitable states.

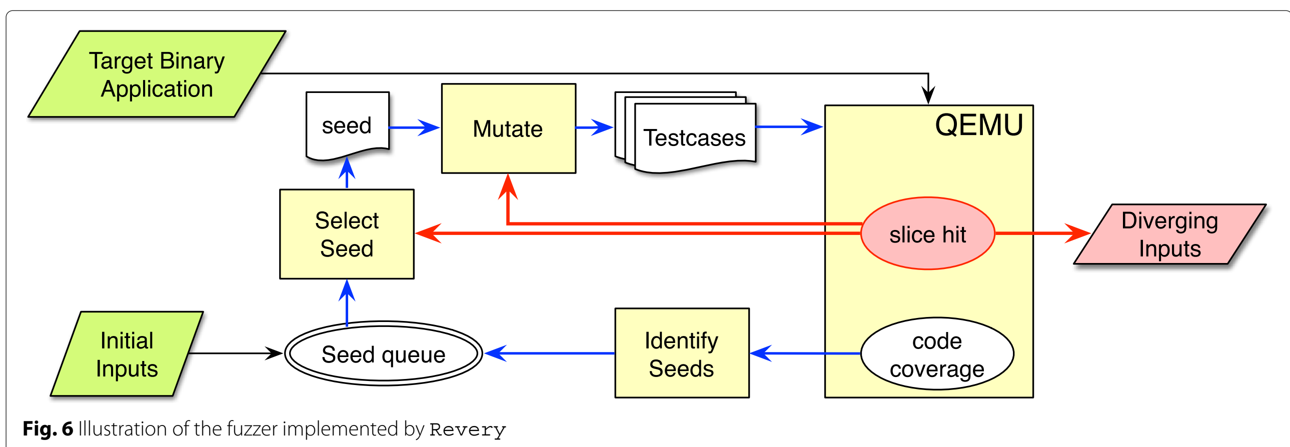

Fig. 6 Illustration of the fuzzer implemented by `Revery`

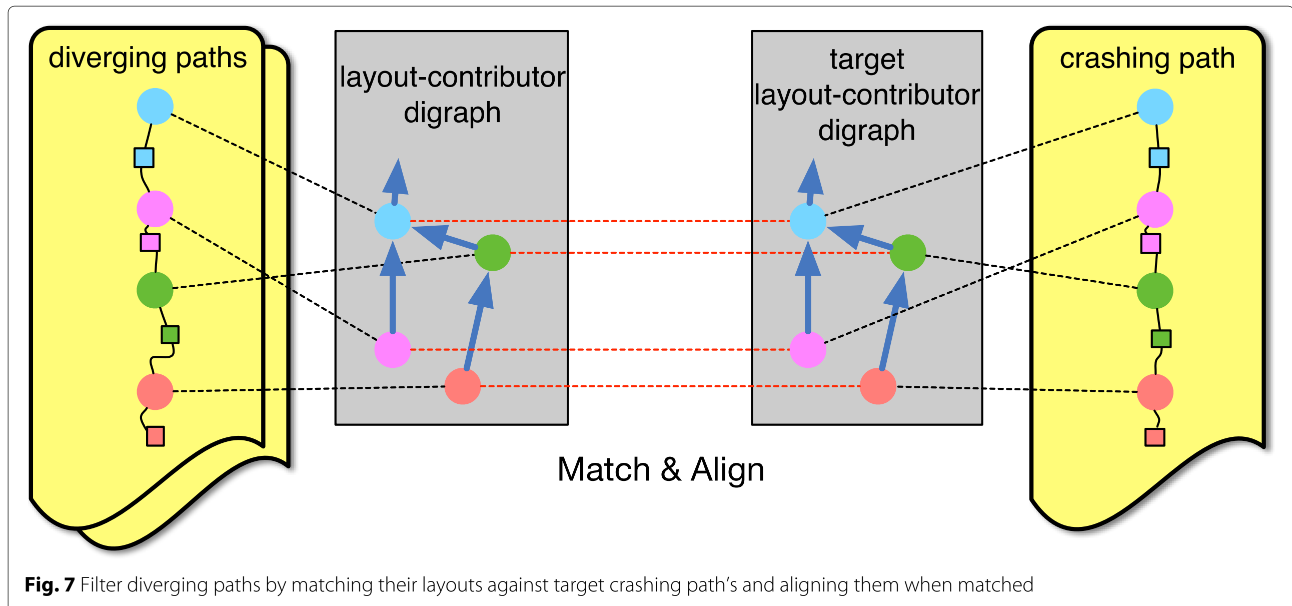

Fig. 7 Filter diverging paths by matching their layouts against target crashing path's and aligning them when matched

In this paper, we mainly consider two types of sensitive (exploitable) operations, i.e., *memory write* and *indirect call*. For example, if the target address of a memory write is affected by the exceptional object, then attackers may control where to write and cause AAW (arbitrary address write), i.e., a commonly used exploitable state in practice. If attackers could affect the target of indirect calls, including virtual function calls and indirect *jmp* instructions etc., then they could hijack the control flow. In addition, `Revery` offers a template for experts to extend the definition of exploitable points, e.g., operations launching the unlinking attack.

Exploitable states searching

This problem thus becomes identifying sensitive instructions whose operands are affected by the exceptional objects. Taint analysis is a straightforward solution.

`Revery` marks each object creation operation as a taint source, and attaches a unique taint label to it. Each operation propagates all source operands' taint labels to the destination. At each sensitive instruction (i.e., memory write or function call), the target address' taint labels will be checked if they contain the exceptional object's taint label. If yes, then this sensitive instruction is exploitable.

Diverging path exploration in `FUZE`

To solve the exploit derivability issue in OS kernel, `FUZE` utilizes kernel fuzzing to explore other system calls and thus diversifies running contexts for exploitation facilitation. In the following, we describe the detail of our kernel fuzzing. To be specific, we first discuss how to initialize a context for fuzz testing. Then, we describe how to set up kernel fuzzing for system call exploration.

Fuzzing context initialization

`FUZE` utilize kernel fuzzing to identify system calls that dereference a dangling pointer. To do this, we must start kernel fuzzing after the occurrence of a dangling pointer and, at the same time, ensure the fuzz testing is not intervened by the pointer dereference specified in the original PoC. As a result, we need to first accurately pinpoint the site where a dangling pointer occurs as well as the site where the pointer is dereferenced by the system call defined in the PoC program. As is demonstrated above, this can be easily achieved by using the information extracted through `KASAN` and dynamic tracing.

With the two critical sites identified, our next step is to eliminate the intervention of the system call that is specified in the original PoC and also capable of dereferencing the dangling pointer. To do this, we wrap a PoC program as a standalone function, and then instrument the function so that it could be augmented with the ability to trigger a free operation but refrain reaching to the site of dangling pointer dereference. With this design, we could encapsulate initial context construction for kernel fuzzing without jeopardizing the integrity of kernel execution.

Based on the practices of free operation and dangling pointer dereference defined in a PoC program, we design different strategies to instrument a PoC program (i.e. the wrapping function). For a single thread PoC program with a free operation and consecutive dereference occurring in two separated system calls, we instrument the PoC program by inserting a `return` statement in between the

system calls because this could prevent the PoC itself entering the dangling pointer dereference site defined in the PoC program. For a multiple-thread PoC program, the dangling pointer could occur in the kernel at any iteration. Therefore, our instrumentation for such PoC programs inserts system call `ioctl` at the end of the iteration. Along with a customized kernel module, the system call examines the occurrence of the dangling pointer and performs PoC redirection accordingly

`KASAN` checks the occurrence of a dangling pointer at the time of its dereference, and we need to terminate the execution of a PoC before the dereference of a dangling pointer. As a result, we cannot simply use `KASAN` to facilitate the ability of the kernel module to identify dangling pointers.

To address this issue, we follow the procedure below. From the information obtained from `KASAN` log, we first retrieve the code statement pertaining to the dereference of the dangling pointer. Second, we perform an analysis on the kernel source code to track down the variable corresponding to the object freed but leaving behind a dangling pointer. Since such a variable typically presents as a global entity, we can easily obtain its memory address from the binary image of the kernel code[1]. By providing the memory address to our kernel module, which monitors the allocation and free operations in kernel memory, we can augment the kernel module with the ability to pinpoint the occurrence of the target object as well as alert system call `ioctl` to redirect the execution of the wrapping function to the consecutive kernel fuzzing.

Listing 1 The pseudo-code indicating the way of performing concurrent kernel fuzz testing

```
1 pid = fork();
2 if (pid == 0)
3 PoC_wrapper(); // PoC wrapper function running
      inside namespaces
4 else
5 fuzz(); // kernel fuzzing
```

Under-context kernel fuzzing

To perform kernel fuzzing under the context initialized above, we borrow a state-of-the-art kernel fuzzing framework, which performs kernel fuzzing by using sequences of system calls and mutating their arguments based on branch coverage feedbacks. Considering an initial context could represent different environment for triggering an UAF vulnerability, we set up this kernel fuzzing framework in two different approaches.

In our first approach, we start our kernel fuzzing right after the fuzzing context initialization. Since we wrap an instrumented PoC program as a standalone function, this can be easily achieved by simply invoking the wrapping function prior to the kernel fuzzing. In our second approach, we set up the fuzzing framework to perform concurrent fuzz testing. In Linux system, namespaces are a kernel feature that not only isolates system resources of a collection of processes but also restricts the system calls that processes can run. For some kernel UAF vulnerabilities, we observed that the free operation occurs only if we invoke a system call in the Linux namespaces. In practice, this naturally restricts the system call candidates that we can select for kernel fuzzing. To address this issue, we fork the PoC program prior to its execution and perform kernel fuzzing only in the child process. To illustrate this, we show a pseudo code sample in Fig. 1. As we can observe, the program creates two processes. One is running inside namespaces responsible for triggering a free operation, while the other executes without the restriction of system resources attempting to dereference the data in the freed object.

In addition to setting up kernel fuzzing for different initial contexts, we design two mechanisms to improve the efficiency of the kernel fuzzing framework. First, we escalate fuzzing efficiency by enabling parameter sharing between the initial context and the fuzzing framework. For kernel UAF vulnerabilities, their vulnerable objects are typically associated with a file descriptor, an abstract indicator used for accessing resources such as files, sockets and devices. To expedite kernel fuzzing for hitting these vulnerable objects, we set up the parameters of system calls by using the file descriptor specified in the initial fuzzing context.

Second, we expedite kernel fuzzing by reducing the amount of system calls that the fuzzing framework has to examine. In Linux system 4.10, for example, there are about 291 system calls. They correspond to different services provided by the kernel of the Linux system. To identify the ones that can dereference a dangling pointer, a straightforward approach is to perform fuzz testing against all the system calls. It is obvious that this would significantly downgrade the efficiency in finding the system calls that are truly useful for exploitation facilitation.

To address this problem, we track down a vulnerable object using the information obtained through the aforementioned vulnerability analysis. Then, we search this object in all the kernel modules. For the modules that contain the usage of the object, we retrieve the system calls involved in the modules by looking up the SYSCALL_DEFINEx() macros under the directory pertaining to the modules. In addition, we include the system calls that belong to the subclass same as the ones already retrieved but not present in the modules. It should be noticed that this approach might result in the missing of the system calls capable of dereferencing dangling pointers. As we will show in "Evaluation" section, this approach however does not jeopardize our capability in finding system calls useful for exploitation.

Exploitable states searching

FUZE perform symbolic execution under the context with the goal of determining whether a diverging context could direct kernel execution to an exploitable machine state. In the following, we first describe how to set up symbolic execution based on the context obtained through the aforementioned kernel fuzzing. Then, we discuss how to identify the machine states truly useful for exploitation by using symbolic execution.

Symbolic execution setup

The random input fed into kernel fuzzing could potentially crash kernel execution without providing useful primitives for exploitation (e.g. writing arbitrary data to an arbitrary address). As a result, we start our symbolic execution right before the site where kernel fuzzing dereferences a dangling pointer. To do this, we need to pinpoint the site of dangling pointer dereference, pause kernel execution and pass the running context to symbolic execution.

Different from kernel fuzzing, symbolic execution cannot leverage kernel instrumentation to facilitate this process. This is simply because we use symbolic execution for exploit generation and the exploit derived from instrumented kernel cannot be effective in a plain Linux system.

To address this issue, we utilize the information obtained through KASAN and dynamic tracing. As is mentioned in "Critical information extraction in OS kernel" section, the information obtained carries the code statement pertaining to the dereference of a dangling pointer. Since this information represents in the source code level, we can easily map it to the plain Linux system, and set a breakpoint at that site.

This approach could guarantee to catch the occurrence of a dangling pointer. However, the setup of the breakpoint could intervene kernel execution even at the time when the dangling pointer does not occur. This is because the statement could also involve in regular kernel execution. To reduce unnecessary intervention, we design FUZE to automatically retrieve the log obtained from the aforementioned dynamic tracing, and then examine if the pointer pertaining to the statement refers to an object that has already been freed at time the execution reaches to the breakpoint. We force the kernel to continue its execution if the freed object is not observed. Otherwise, we pause kernel execution and use it as the initial setting for consecutive symbolic execution.

Exploitable machine state identification

Starting from the initial setting, we create symbolic values for each byte of the freed object. Then, we symbolically resume kernel execution and explore machine states potentially useful for vulnerability exploration. To identify machine states exploitable, we define a set of primitives indicating the operations needed for exploitation. Then, we look up these primitives and take them as candidate exploitable states while performing symbolic execution.

Since primitives represent only the operations generally necessary for exploitation, but not reflect their capability in facilitating exploitation, we further evaluate the primitives guided by exploitation approaches commonly adopted, and deem those passing the evaluation as our exploitable states. In the following, we specify the primitives that FUZE looks up and detail the way of performing primitive evaluation.

Primitives Specification. We define two types of primitives – *control flow hijacking* and *invalid write*. They are commonly necessary for performing exploitation under a certain assumption.

A *control flow hijacking* primitive describes a capability that allows one to gain a control over a target destination. To capture this primitive during symbolic execution, we examine all indirect branching instructions and determine whether a target address carries symbolic bytes (e.g. call rax where rax carries a symbolic value). This is because the symbolic value indicates the data we could control and its occurrence in an indirect target implies our control over the kernel execution.

An *invalid write* primitive represents an ability to manipulate a memory region. In practice, there are many exploitation practices dependent upon this ability. To identify this primitive during symbolic execution, we pay attention to all the write instructions and check whether the destination address or the source register or both carry symbolic bytes (e.g. mov qword ptr [rdi], rsi where both rdi and rsi contain symbolic values). The insight of this primitive is that the symbolic value indicates the data we could control and its occurrence in a source register or a destination address or simultaneously both implies a certain level of control over an memory area.

Primitive Evaluation. As is described above, it is still unclear whether one could utilize the aforementioned primitives to facilitate his exploitation. Given a control flow hijacking primitive, for example, it may be still challenging for one to exploit an UAF vulnerability because of the mitigation integrated in modern OSes (e.g. SMEP and SMAP). To select primitives truly valuable for exploitation (i.e. exploitable machine states), we evaluate primitives as follows.

As is specified in Nikolenko (2016), with SMEP enabled, an attacker can use the following approach to bypass SMEP and thus perform control flow hijacking. First, he needs to redirect control flow to kernel gadget xchg eax, esp; ret. Then, he needs to pivot the stack to user space by setting the value of eax to an address in user space. Since the attacker has the full control to the pivot stack, he could prepare an ROP chain using the stack along with the instructions in Linux kernel. In this way, the attacker does

not execute instructions residing in user space directly. Therefore, he could fulfill a successful control flow hijack attack without triggering SMEP.

In this work, we use this approach to guide the evaluation of primitives. At the site of the occurrence of a control flow hijacking primitive, we retrieve the target address pertaining to the primitive as well as the value in register eax. Since the target address carries a symbolic value, we check the constraint tied to the symbolic value and examine whether the target could point to the address of the aforementioned gadget. Then, we further examine if the value of eax is within range (0x10000, τ). Here, (0x10000, τ) denotes the valid memory region. 0x10000 represent the end of an unmapped memory region, and τ indicates the upper bound of the memory region in user space.

Given SMEP enabled, another common approach (Argyroudis 2012) for bypassing SMEP and performing control flow hijacking is to leverage an invalid write to manipulate the metadata of the freed object. In this approach, one could leverage this invalid manipulation to mislead memory management to allocate a new object to the user space. Since one could have the full control to the user space, he could modify the data in the new object (e.g. a function pointer) and thus hijack the consecutive execution of Linux kernel.

To leverage this alternative approach to guide our evaluation, we retrieve the source and destination pertaining to each invalid write primitive. Then, we check the value held in the destination. If that points to the metadata of the freed object, we further inspect the constraint tied to the source. We deem a primitive matches this alternative exploitation approach only if the source indicates a valid user-space address or provides one with the ability to change the metadata to an address in user space.

In addition to the approaches for bypassing SMEP, there is a common approach (Konovalov 2017) to bypass SMAP and perform control flow hijacking. First, an attacker needs to set register rdi to a pre-defined number (e.g. 0x6f0 in our experiment). Then, he needs to redirect the control flow to function native_write_cr4(). Since the function is responsible for setting register CR4 – the 21st bit of which controls the state of SMAP – and rdi is the argument of this function specifying the new value of CR4, he could disable SMAP and thus perform a control flow hijack attack.

To use this approach to guide our primitive evaluation, we examine each control flow hijacking primitive and at the same time check the value in register rdi. To be specific, we check the constraints tied to register rdi as well as the target of the indirect branching instruction. Then, we use a theorem solver to perform a computation which could determine whether the target could point to the address of native_write_cr4() and at the same time rdi could equal to the pre-defined number.

It should be noticed that this work does not involve leveraging information leak for bypassing KASLR and acquiring the base address of kernel code segment. This is because there have been already a rich collection of works that could easily facilitate the acquirement of the base address of kernel code segment (e.g. Gruss et al. (2016); Jang et al. (2010)) and the facilitation of information leak provided by FUZE is neither a necessary nor a sufficient condition for successful exploitation. In addition, it should be noted that the symbolic execution applied above naturally provides FUZE with the ability to compute the data that needs to be sprayed to the freed object. In this work, we therefore utilize off-the-shelf constraint solver (i.e. SMT) to compute values for all the symbolic variables while the symbolic exploration reaches to the machine states exploitable.

Exploit synthesis

In this section, we will introduce how to synthesize new exploits from PoC inputs and diverging inputs.

Once an exploitable state is found in a path, existing AEG solutions usually generate exploits by solving the path, vulnerability and exploit constraints. However, as discussed in "Alternative choices" section, symbolic execution solely is not effective in exploit generation.

Therefore, Revery uses symbolic execution as few as possible. It uses a lightweight symbolic execution as a bond to stitch the crashing path and diverging path together, and reuses the PoC input and diverging inputs to further reduce complicated constraints, making symbolic execution more practical.

Figure 8 shows the general workflow of exploit synthesis. In practical, it first identifies stitching points, and then explores sub-paths between stitching points and synthesize exploitation path, and finally solve related constraints to generate working exploits.

Identify stitching points

We first introduce how Revery identifies stitching points in both the crashing path and diverging path.

Stitching points in the crashing path

In order to successfully exploit the victim program, its vulnerability must be first triggered, and some exceptional objects are corrupted. Revery thus chooses locations where exceptional objects are corrupted in the crashing path as stitching points.

As mentioned in "Vulnerability-related object layout" section, in the crashing path, each write access violation corrupts an exceptional object, and thus it is a candidate stitching point. For each read access violation in a UAF vulnerability, the exceptional object is the one that

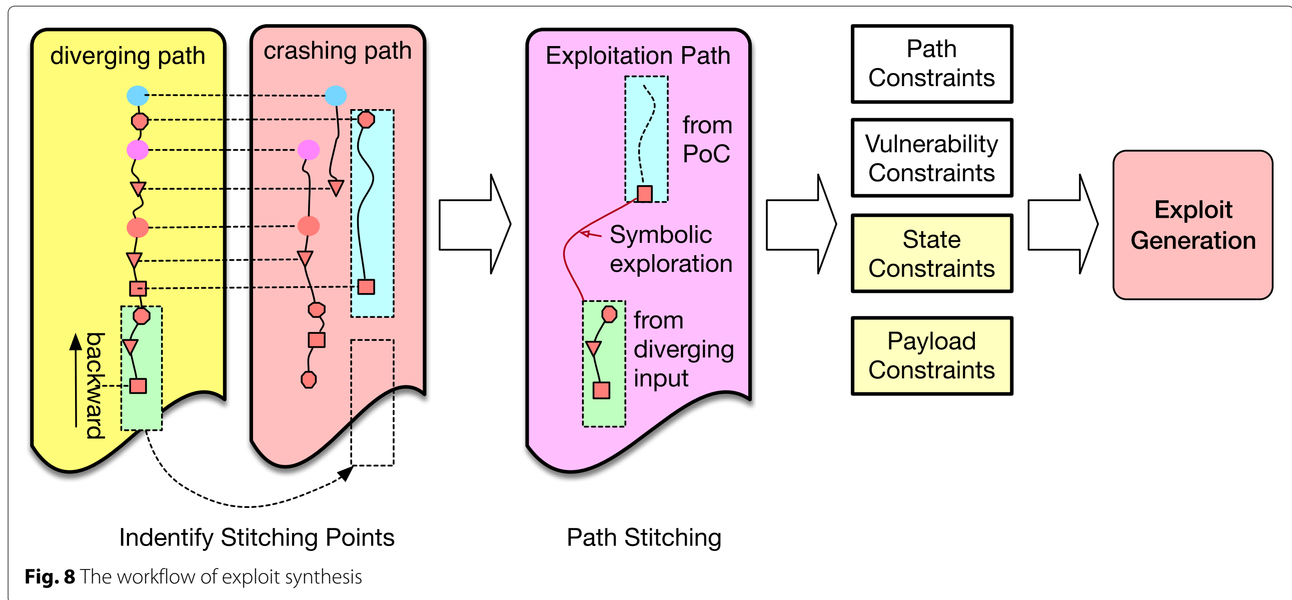

Fig. 8 The workflow of exploit synthesis

has been freed but still pointed by the dangling pointer. This exceptional object's memory region will be occupied by another memory allocation. Revery takes the new memory allocation operation as a candidate stitching point.

Since there could be multiple violations in one crashing path, there could also be multiple stitching points. Revery will try to stitch each of them with the diverging path.

Stitching points in diverging paths

In order to successfully exploit the victim program, exploitable operations must be performed on corrupted exceptional objects or collateral objects.

What are good stitching points? Every instruction could be used as stitching points. But not all of them are good ones. A proper stitching point should satisfy several criterions:

- *Not too close to entry points.* Otherwise, many duplicated operations as the crashing path will be performed. Since duplicate operations (e.g., object initializations) will not happen in a legitimate control flow, it is infeasible to find a path to connect this stitching point with its counterpart in the crashing path.
- *Not too close to exploitable points.* Otherwise, a longer path is required to connect this stitching point with its counterpart, requiring more efforts of symbolic execution. The stitching point can be set before certain operations, e.g., initialization of exploitable points' operands, to save symbolic execution efforts.
- *Minimum data dependency.* The data flow after the stitching point in the diverging path should have few intersections with the data flow before the stitching point in the crashing path.

How to find stitch points? At a high level, Revery matches the diverging path's data dependency against the crashing path's, and locates the differences. Then it uses the instruction which causes the differences in the diverging path as stitching point.

First, Revery builds the layout-contributor digraph of the exploitable operation's operand in the diverging path. Then it matches this digraph against the digraph of the exceptional object in the crashing path. If the former is a sub-graph of the latter, it means the crashing path has already set up all data dependencies for the exploitable operation. Then, the instruction in the diverging path, which is right after the last write access to the exploitable operations' operands, is chosen as the stitching point.

Otherwise, there are different nodes or edges in the diverging path's digraph, i.e., the diverging path has alternated the dependency of the exploitable operations. In this case, Revery chooses the earliest instruction (object creation or write) in the diverging path, which causes differences in the digraph, as the stitching point.

Control-flow path stitching

In order to stitch the crashing path and the diverging path together, Revery explores potential sub-paths connecting the stitching points in these paths. In general, it

relies on symbolic execution to explore paths. However, Revery utilizes several heuristics to efficiently guide symbolic execution.

First of all, Revery uses the function call stack to guide the path exploration. It inspects the call stacks at the two stitching points respectively, and finds the differences. Figure 9 shows two example call stacks. These differences in call stacks indicate the direction of path exploration. Function invocations in the crashing path (e.g., g1, g2, ..., gM in the figure) should be returned one by one first, while function invocations in the diverging path (e.g., h1, h2, ..., hK in the figure) should be called one by one later.

In other words, when exploring potential paths, Revery will add the return instruction of function gM, ..., g2, g1 as target instructions one by one, and then add the entry point of function h1, h2, ..., hK as target instructions one by one. These target instructions are dominator points between the two stitching points. Then Revery will explore potential sub-paths between these intermediate target instructions.

Revery further mitigates the sub-path exploration by reusing existing paths. For example, if there is already a sub-path connecting two intermediate destinations in either the diverging path or the crashing path, Revery will reuse this sub-path. Revery also performs a simple loop identification algorithm, and finds a sub-path to escape the loop as soon as possible, in order to reduce the burden of symbolic execution. Sometimes, the reused sub-path would cause the overall path unsolvable, Revery will try to remove these sub-paths and search for alternative sub-paths.

In this way, Revery greatly reduces the burden of symbolic execution when exploring sub-paths to connect the stitching points.

Exploit generation

Once a sub-path connecting two stitching points is found, a candidate exploitation path is constructed. Revery could also solve the vulnerability constraints, path constraints and exploit constraints to generate final exploit samples. However, it is inadequate.

Exploitable state constraints

Simply solving constraints of the exploitation path may not trigger the same exploitable state as the diverging path. Revery thus adds several extra data constraints to the exploitation path, ensuring the program state is still exploitable.

First, the memory allocation sizes in the exploitation path should be the same as the diverging path, in order to trigger the exploitable states as in the diverging path. Revery records the concrete sizes of all memory allocations when analyzing the diverging path. In the exploitation path, if a memory allocation which was in the diverging path has a symbolic size, then Revery will add a constraint to ensure this size equals to the concrete value in the diverging path.

Second, Revery will align the digraph of the crashing path with the diverging path's. Certain symbolic addresses in the diverging path are logically the same as their counterparts in the crashing path. So, in the stitched exploitation path, extra constraints must be introduced to claim the equality between these symbolic addresses.

Payload constraints

With the aforementioned exploitable state constraints, together with the vulnerability and path constraints, Revery is able to generate *EXP inputs* to trigger both exploitable states and vulnerabilities. These inputs could help security experts to construct a full exploit.

In certain cases, Revery is able to directly generate working exploits. At the exploitable point, Revery could construct payload constraints which could lead to control flow hijacking. If the exploitable point is a function call (e.g., indirect *call* or jmp instruction) and its target is a symbolic value, Revery adds an extra constraint to set the target to attacker controlled value. If the exploitable

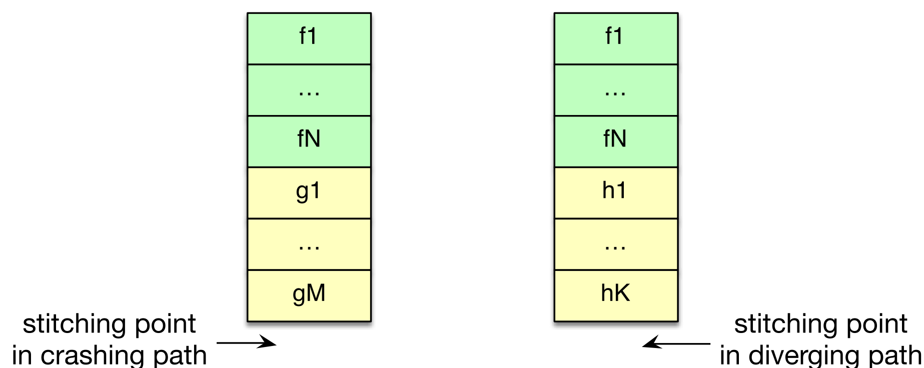

Fig. 9 Example call stacks of stitching points

state is a write access, and both the destination address and content to write are symbolic, then Revery adds an extra constraint to overwrite a known address (e.g., Global Offset Table entries or global function pointers) with attacker controlled value.

In this way, Revery could generate exploits to hijack control-flow for certain cases. However, it is not always guaranteed to succeed.

Evaluation

We implemented a prototype of Revery based on the binary analysis engine angr (Shoshitaishvili et al. 2016) and the popular fuzzer AFL (Zalewski 2018). It consists of 1334 lines of code to analyze vulnerabilities, 190 lines of code to explore diverging paths with fuzzing, and 1249 lines of code to stitch paths and generate exploits.

In this section, we present the evaluation results of this system. The experiments are conducted in a Ubuntu 17.04 system running on a server with 115G RAM and Intel Xeon (R) CPU E5-2620 @ 2.40GHz*24. We evaluated Revery against 19 vulnerable programs collected from 15 CTF (capture the flag) competition, 14 of them can be found in CTFTIME (CTF TIME 2018)[2].

To thoroughly evaluate the effectiveness of Revery, we selected the target programs from CTF events based on the following rules: (1) no source code or debug symbols exist for these programs; (2) each program must have at least one heap-based vulnerability; (3) the diversity of vulnerability types must be large; and (4) the quality of the source CTF events is well acknowledged.

All programs are tested in a regular modern Linux operating system (Ubuntu 17.04), with the defense DEP (Andersen and Abella 2004) enabled. Unlike traditional environments, we disabled ASLR (PaX-Team 2003) in the evaluation. In practice, an information disclosure vulnerability or exploit is required to bypass ASLR. The current prototype of Revery could not generate information disclosure exploits yet.

We also demonstrate the utility of FUZE using 15 real-world kernel UAF vulnerabilities. Regarding the configuration of FUZE, we performed kernel fuzzing and symbolic execution using a machine with Intel(R) Xeon(R) CPU E5-2630 v3 2.40GHz CPU and 256GB of memory. As is mentioned in "Exploitable states searching" section, the address space layout randomization is out of the scope of this work. Last but not least, we therefore disabled CONFIG_RANDOMIZE_BASE option in all Linux kernels that we experiment.

To showcase FUZE can truly benefit the exploitation, we performed end-to-end exploitation using the exploitable machine states we identified. To be specific, we computed the data that needs to be sprayed based on the constraints tied to the exploitable states. Then, we performed the heap spray with three different system calls – add_key(), msgsnd(), sendmsg() – by following the techniques introduced in Xu et al. (2015). To fulfill exploitation using the exploitable states identified, we eventually redirect the execution to an ROP chain (Nikolenko 2016) commonly used for exploitation. To illustrate the exploits generated through the facilitation of FUZE, we have released some example exploits along with the virtual machine at (Anonymous 2018). In addition, we discuss those kernel UAF vulnerabilities, the exploitation of which FUZE fails to provide with facilitation.

Exploits by Revery

Table 1 shows the list of programs we evaluated. Out of 19 programs, Revery successfully exploited 9 of them, i.e., able to hijack their control flow. Revery could trigger the exploitable states for 5 more programs, i.e., providing exploit primitives for experts to launch successful exploits. It failed to analyze the rest 5 programs. More details will be discussed later.

This table also shows in detail the name and CTF event of each program. It shows the type of the known vulnerability in each program, including heap overflow, off-by-one, UAF and double free. Further, it shows the crash type of each vulnerability, i.e., results of applying PoC inputs to the vulnerable programs. Some of them are caught by the memory manager's sanity checks (denoted as heap error in the table), some others crash at invalid memory read instructions. Most of them do not even crash.

In addition, it shows the violation type of each vulnerability detected by Revery, the final exploitable state triggered by Revery, and whether Revery could generate exploits or not. Revery could detect security violations in 16 out of 19 programs. It could trigger exploitable states of EIP hijacking, arbitrary memory write, and unlink attack for 3, 6 and 5 programs respectively. Revery could generate working exploits for first two types of exploitable states.

As a comparison, we also evaluated the open-source AEG solution Rex (2018) provided by the Shellphish team on these programs. As shown in the last column of the table, Rex could not solve any of these programs.

Case studies

In this section, we investigated these programs in detail, and analyzed why our solution Revery succeeded or failed.

Control-Flow hijacking exploits

Revery successfully generated control-flow hijacking exploits for 9 programs. With the given PoC inputs, 2 programs corrupt the heap metadata and are caught by the sanity checks deployed in glibc memory allocator. Three other programs crash at invalid memory read

Table 1 List of CTF pwn programs evaluated with `Revery`

	Name	CTF	Vul type	Crash type	Vio.	Final state	EXP	Rex
Control Flow Hijack	woO2	TU CTF 2016	UAF	heap err	V1	EIP hijack	Y	N
	woO2_fixed	TU CTF 2016	UAF	heap err	V1	EIP hijack	Y	N
	shop 2	ASIS Final 2015	UAF	mem read	V1	EIP hijack	Y	N
	main	RHme3 CTF 2017	UAF	mem read	V1	mem write	Y	N
	babyheap	SECUINSIDE 2017	UAF	mem read	V1	mem write	Y	N
	b00ks	ASIS Quals 2016	Off-by-one	no crash	V1	mem write	Y	N
	marimo	Codegate 2018	Heap BOF	no crash	V1	mem write	Y	N
	ezhp	Plaid CTF 2014	Heap BOF	no crash	V1	mem write	Y	N
	note1	ZCTF 2016	Heap BOF	no crash	V1	mem write	Y	N
Exploit-able State	note2	ZCTF 2016	Heap BOF	no crash	V1	unlink init	N	N
	note3	ZCTF 2016	Heap BOF	no crash	V1	unlink init	N	N
	fb	AliCTF 2016	Heap BOF	no crash	V1	unlink init	N	N
	stkof	HITCON 2014	Heap BOF	no crash	V1	unlink init	N	N
	simple note	Tokyo Westerns 2017	Off-by-one	no crash	V1	unlink init	N	N
Failed	childheap	SECUINSIDE 2017	Double Free	heap err	V1	-	N	N
	CarMarket	ASIS Finals 2016	Off-by-one	no crash	V1	-	N	N
	SimpleMemoPad	CODEBLUE 2017	Heap BOF	no crash	-	-	N	N
	LFA	34c3 2017	Heap BOF	no crash	-	-	N	N
	Recurse	33c3 2016	UAF	no crash	-	-	N	N

Out of 19 applications, `Revery` could generate exploits for 9 of them, and generate EXP inputs to trigger exploitable state for another 5 of them, and failed for the rest 5

instructions, whose results are only dumped by functions like `printf`, which could not cause control-flow hijacking. The rest 4 programs do not even crash with the provided PoC.

Limit of State-of-the-art AEG Solutions. Such vulnerabilities are usually considered as non-exploitable by exploitability assessment tools. To successfully exploit these vulnerabilities, we have to avoid the metadata corruption being caught by sanity checks, and accurately model the memory allocator if using symbolic execution.

So state-of-the-art AEG solutions could not generate exploit automatically for them. We have tested all these programs with Rex (2018), an automated exploit generation tool that developed by the Shellphish team, which won the first in offense in CGC. But it failed to generate exploits for any of them.

Performance of Revery. By exploring exploitable states in diverging paths, `Revery` can generate exploits for all 9 programs. For example, `WoO2` and `WoO_fixed` crash because one object is freed twice. To exploit this kind of vulnerabilities, heap Fengshui (Sotirov 2007) is needed, which is too complicated for automated solutions. Instead, `Revery` goes back to the vulnerability point, and finds a diverging path which could lead to EIP hijack.

Three of the exploitable states could hijack the program counter, and the other six could cause arbitrary address write (AAW). AAW is a well-known exploit primitive, could enable many exploits. For example, it could be used to modify the global offset table (GOT) and hijack the control flow.

Exploitable states

Sometimes `Revery` is not able to generate working exploits, even if it has found the exploitable states and stitched an exploitation path. As shown in the table, `Revery` could trigger exploitable states but fail to generate working exploits for 5 programs.

For these programs, there is no critical data fields (e.g., function pointer, VTable pointer etc.) in the exceptional object, and it is extremely challenging to automatically generate exploits against them. Instead, we have to utilize the corrupted metadata in the exceptional objects to exploit the specific heap allocators.

`Revery` utilizes layout-oriented fuzzing to find a diverging path that will free the exceptional object, and trigger an exploitable state. Given that the `glibc` library uses a double-linked list to maintain objects, unlinking a node from this list (due to certain memory operations) will

update forward and backward nodes' pointers, causing an unintended memory write operation. This is known as `unlink attack` (Unlink Exploit 2018).

However, to successfully exploit such states, we have to arrange the heap layout, with heap Fengshui and other techniques, which is out of the scope of this paper. However, with the inputs generated by `Revery`, experts could manually massage the heap layouts and write an exploit much quicker.

Failed cases

As aforementioned, `Revery` cannot guarantee to generate working exploits or trigger exploitable states. In our experiments, `Revery` failed for 5 programs.

Limitations of Vulnerability Detection For some of the programs, `Revery` fails to detect the security violations. For example, the challenge `SimpleMemoPad` has a buffer overflow inside objects, i.e., it will corrupt the neighbor data fields rather than neighbor objects. `Revery` currently only supports object level corruption detection. We leave it as a future work to support detection of in-object buffer overflow.

Limitations of Angr Our solution `Revery` relies on angr (Shoshitaishvili et al. 2016) to perform symbolic execution. Angr emulates all syscalls by itself, which has not fully implemented yet. Alternatively, angr rewrites library functions in Python, and hooks the original functions. However, this is far from finished too. As a result, angr cannot support most real world programs. This is also the major reason why we only evaluate `Revery` on CTF programs.

Efficiency of layout-oriented fuzzing

We further evaluated the efficiency of `Revery` in terms of diverging path exploration and exploitable states searching. We compared our layout-oriented fuzzing with the original fuzzer AFL. To evaluate the efficiency of layout-oriented fuzzing fairly, we run `Revery` and AFL at the same time, and use a same exploitable state searching module in `Revery` to evaluate the test cases generated by both fuzzers.

Figure 10 shows the time interval used by `Revery` and AFL to find the first input that hits all instructions in layout-contributor slice. On average, `Revery` is 122% faster than AFL.

`Revery` also spends less time than AFL to find an exploitable state in diverging paths. As shown in Fig. 11, AFL failed to find exploitable states for 3 programs in 8 hours. By contrast, `Revery` has found exploitable states for all the programs. For programs that both AFL and `Revery` succeed, `Revery` is 247% faster than AFL on average.

In short, with layout-oriented fuzzing, `Revery` could find diverging paths and exploitable states much faster than AFL.

We also compared layout-oriented fuzzing with Driller(Stephens et al. 2016). The result shows that Driller didn't find any exploitable state for all the programs in 4 h.

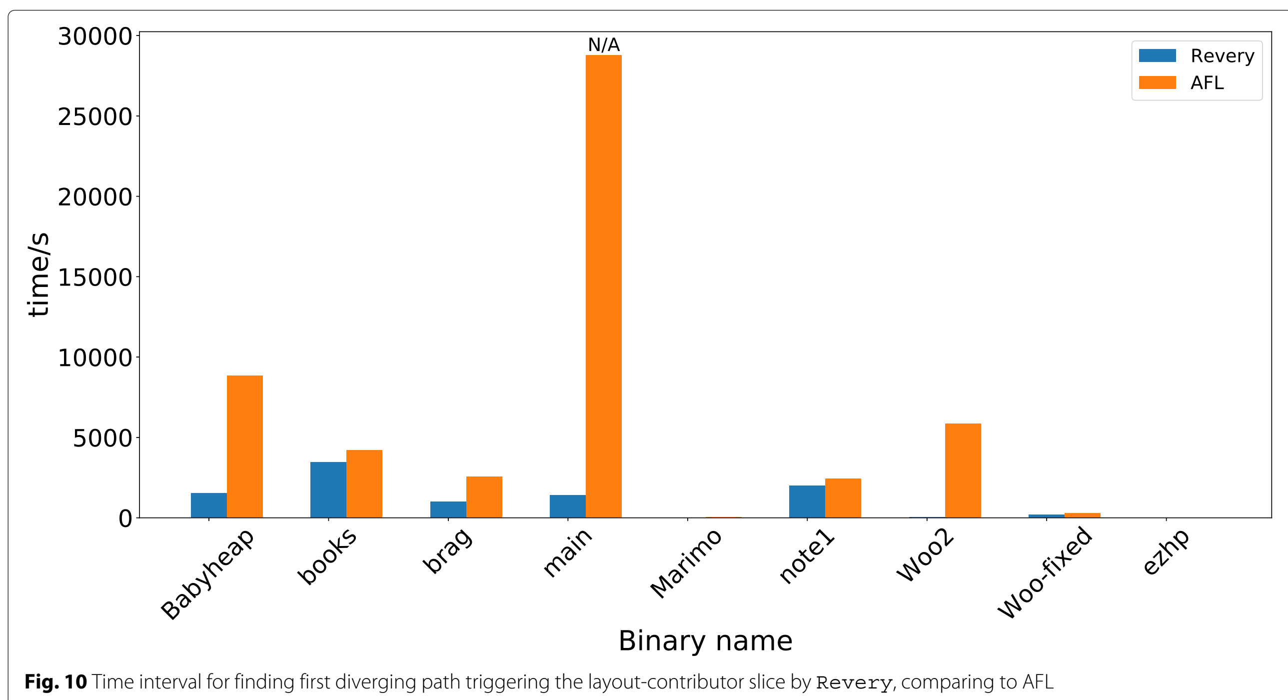

Fig. 10 Time interval for finding first diverging path triggering the layout-contributor slice by `Revery`, comparing to AFL

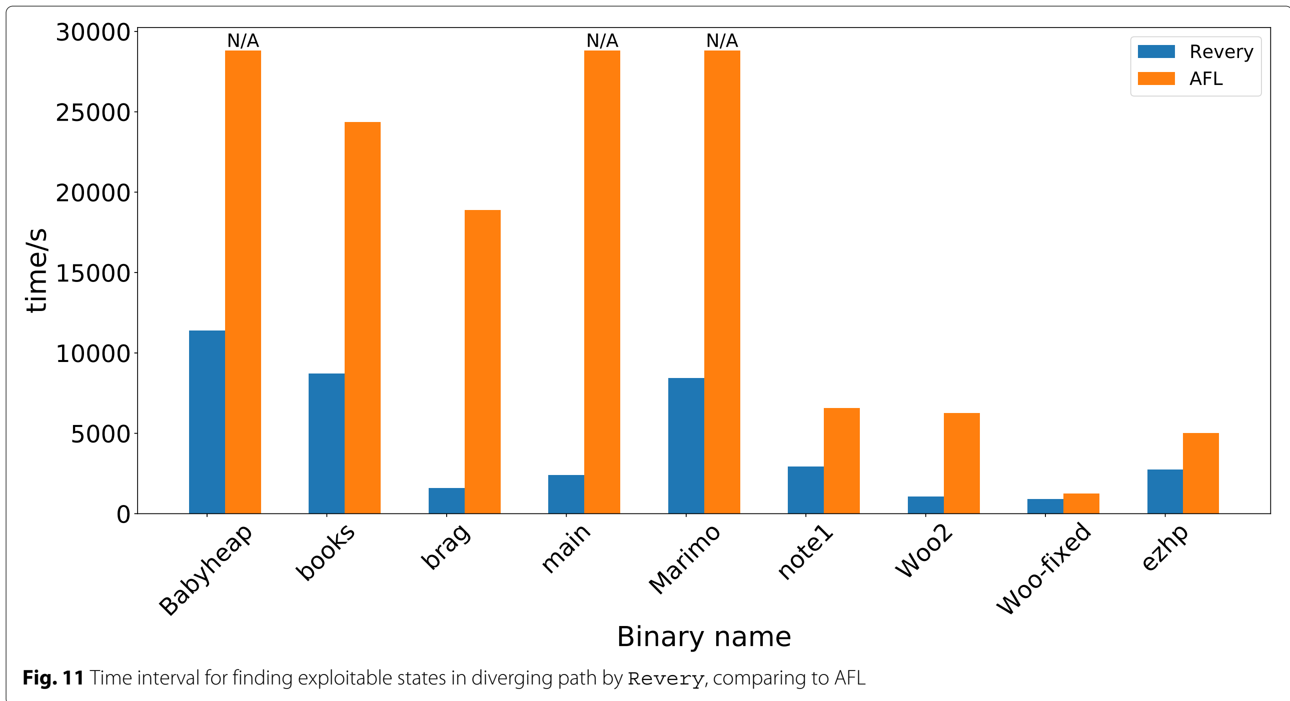

Fig. 11 Time interval for finding exploitable states in diverging path by `Revery`, comparing to AFL

And only in one program (i.e., `note1`) the symbolic execution engine of Driller is invoked. This is probably because driller is designed for CGC programs and has some bugs for ELF binaries.

Efficiency of control-flow stitching

Given the candidate exploitable states, `Revery` utilizes a novel control-flow stitching solution to generate inputs to trigger both the vulnerability and exploitable states. In theory, symbolic execution could be used solely to explore paths from the vulnerability point to the exploitable states. To compare the efficiency between them, we thus evaluated `Revery` and a strawman symbolic exuection tool SYMBEX based on angr.

Overall results

Table 2 shows the evaluation results on 14 programs which angr is able to handle. `Revery` could generate EXP inputs to trigger exploitable states for all 14 programs in minutes. But SYMBEX could only solve 4 of them. The exploitable points of these 4 programs are right after the vulnerability points and before the crashing points, and thus require no efforts to explore paths. SYMBEX failed to solve the program `main` in four hours, and failed for the rest 9 programs.

Path Reusing Rate We use path reusing rate to assess the quality of stitching points that `Revery` found. This rate is computed based on the count of basic blocks reused from the diverging path, compared to the count of basic blocks in the exploitation path. A higher reusing rate indicates that the stitching point is better for exploit generation. As shown in the table, more than half of the programs has a path reusing rate higher than 60%.

Table 2 Comparison with symbolic execution

Name	Vul type	Revery Gen. Time (s)	Path reuse rate	Revery EXP. work	SYM. Gen. Time (s)	SYM. EXP. work
shop 2	UAF	238	100%	YES	Failed	NO
note2	BOF	70	100%	YES	Failed	NO
ezhp	BOF	56	98.0%	YES	Failed	NO
fb	BOF	60	85.1%	YES	Failed	NO
note3	BOF	83	84.1%	YES	Failed	NO
main	UAF	146	71.1%	YES	>4h	-
stkof	BOF	208	65.5%	YES	Failed	NO
marimo	BOF	264	62.2%	YES	Failed	NO
simplenote	BOF	263	41.9%	YES	Failed	NO
babyheap	UAF	442	27.8%	YES	Failed	NO
note1	BOF	161	84.0%	YES	412	YES
b00ks	BOF	81	83.3%	YES	91	YES
woO2	UAF	38	22.7%	YES	39	YES
woO2_fixed	UAF	38	22.7%	YES	38	YES

Failure Analysis SYMBEX failed for 9 programs. We pointed out that, traditional symbolic execution is unable to infer some exploitable state constraints and thus fails to generate exploits. As discussed in "Exploit synthesis" section, `Revery` could infer these constraints from the diverging path.

Effectiveness of FUZE

Table 3 specifies the amount of distinct exploits publicly available for each kernel UAF vulnerability as well as their capability of bypassing mitigation mechanisms commonly adopted (i.e. `SMEP` and `SMAP`). We use this as our baseline to compare with exploits generated under the facilitation of `FUZE`. We show this comparison side-by-side in Table 3.

With regard to the ability to perform exploitation and bypass `SMEP` illustrated in Table 3, we first observe that there are only 5 publicly available exploits capable of bypassing `SMEP` whereas `FUZE` enables exploitation and `SMEP`-bypassing for 5 additional vulnerabilities. This indicates the facilitation of `FUZE` could not only significantly improve possibility of generating exploits but, more importantly, escalate the capability of a security analyst (or an attacker) in bypassing security mitigation.

For all the vulnerabilities that an attacker could exploit and bypass `SMEP`, we also observe a significant increase in the amount of unique exploits capable of bypassing `SMEP`. This indicates that our kernel fuzzing could diversify the running contexts and thus facilitate our symbolic execution to identify machine states useful for exploitation. It should be noticed that we count the amount of distinct exploits shown in Table 3 based on the number of contexts capable of facilitating exploitation but not the exploitable states we pinpointed. This means that, the exploits crafted for the same UAF vulnerability all utilizes different system calls to perform control flow hijacking and mitigation bypassing.

Regarding the capability of disabling `SMAP` shown in Table 3, we discovered only 2 exploits publicly available and capable of bypassing `SMAP`. They attach to 2 different vulnerabilities – `CVE-2016-8655` and `CVE-2016-4557`. Using `FUZE` to facilitate exploit generation, we observe that `FUZE` could enable and diversify exploitation as well as `SMAP`-bypassing for 2 additional vulnerabilities (see `CVE-2017-8824` and `CVE-2017-15649` in Table 3). In addition, we notice that `FUZE` fails to facilitate `SMAP`-bypassing for `CVE-2016-4557` even though a public exploit has already demonstrated its ability to perform exploitation and bypass `SMAP`. This is for the following reason. As is described in "Exploitable states searching" section, `FUZE` explores exploitability through control flow hijacking. For some exploitation such as privilege escalation, control flow hijacking is not a necessary condition. In this case, the exploit publicly available performs privilege escalation which bypasses `SMAP` without leveraging control flow hijacking.

In addition to the ability of bypassing mitigation and diversifying exploits, Table 3 reveals the capability of `FUZE` in facilitating exploitability. As we will discuss in the following session, there are 4 kernel UAF vulnerabilities for which `FUZE` cannot perform fuzzing because the PoC programs obtained all perform free and dereference operations in the same system call. However, we observe that `FUZE` can still facilitate exploit generation particularly for the vulnerabilities tied to `CVE-2017-17053` and `CVE-2016-10150`. This is for the following reason. Kernel fuzzing is used for diversifying running contexts. Without its facilitation, `FUZE` only performs symbolic execution and explores machine states exploitable under the context tied to the PoC program. For the two vulnerabilities above, their running contexts attached to the PoC programs have already carried valuable primitives, which symbolic execution could track down and expose for exploit generation.

Last but not least, Table 3 also specifies some cases which `FUZE` fails to facilitate exploitation. However, this does not imply the ineffectiveness of `FUZE`. For the case tied to `CVE-2015-3636`, the vulnerability can be triggered only in the 32-bit Linux system, in which the Linux kernel has to access a fixed address defined by marco `LIST_POISON` prior to an invalid free. In a 64-bit Linux system on an x86 machine, this address is unmappable and

Table 3 Exploitability comparison with and without FUZE

CVE-ID	# of public exploits		# of generated exploits	
	SMEP	SMAP	SMEP	SMAP
2017-17053	0	0	1	0
2017-15649	0	0	3	2
2017-15265	0	0	0	0
2017-10661	0	0	2	0
2017-8890	1	0	1	0
2017-8824	0	0	2	2
2017-7374	0	0	0	0
2016-10150	0	0	1	0
2016-8655	1	1	1	1
2016-7117	0	0	0	0
2016-4557	1	1	4	0
2016-0728	1	0	3	0
2015-3636	0	0	0	0
2014-2851	1	0	1	0
2013-7446	0	0	0	0
Overall	5	2	19	5

thus this vulnerability cannot be triggered. For the case tied to CVE-2017-7374, the NVD website (Database 2017) categorizes it into a kernel UAF vulnerability. After carefully investigating the PoC program and analyzing the root cause of this vulnerability, we discovered that the root cause behind this vulnerability is actually a null pointer dereference. In other words, the vulnerability could make kernel panic only at the time when a system call dereferences a null pointer. Up until the submission of this work, for the cases tied to CVE-2013-7446, CVE-2017-15265 and CVE-2016-7117, both exhaustive search and FUZE have not yet discovered any exploits indicating their ability to perform exploitation. This is presumably because these vulnerabilities could result in only a Denial-of-Service to the target system or they could be exploitable only in support of other vulnerabilities.

Efficiency of FUZE

Table 4 specifies the time spent on identifying the first context capable of facilitating exploitation or, in other words, the context from which the consecutive symbolic execution could successfully track down an exploitable machine state. We observe that FUZE could perform fuzz testing against 9 vulnerabilities. For all of them, FUZE could pinpoint a valuable context within about 200 min, which indicates a relatively high efficiency in supporting exploit generation. For the rest cases, there are mainly two reasons behind the failure of our fuzz testing. First, our kernel fuzzing has to start after the occurrence of a dangling pointer. However, for the case tied to CVE-2015-3636, the invalid free operation cannot be triggered in 64-bit Linux kernel. Second, for the other 4 cases, the free and dereference are entangled in the same system call. This practice leaves a short time frame for kernel fuzzing, and FUZE performs only symbolic execution.

To perform kernel fuzzing in a more efficient manner, syzkaller customizes these system calls and extends their amount to 1,203. As is mentioned in "Diverging path exploration in FUZE" section, we trim the set of system calls that FUZE has to explore for the purpose of improving the efficiency of FUZE. In Table 4, we show the amount of system calls that FUZE has to explore during 12-hour kernel fuzzing. For all the cases except for that tied to CVE-2014-2851, we can easily observe that FUZE cut more than 60% of system calls. Among them, there are approximately half of the cases, for which kernel fuzzing needs to explore only about 100 system calls. This implies the contribution to the efficiency in exploitation facilitation.

In addition to the efficiency of kernel fuzzing, Table 4 demonstrates the performance of symbolic execution. More specifically, the table shows the minimum, maximum and average length of the path from a dangling pointer dereference site to a control flow hijacking or an invalid write primitive. Across all cases except for CVE-2015-3636 – which we cannot trigger a UAF vulnerability in a 64-bit Linux system – we observe that the maximum number of basic blocks on a path is 86. This indicates primitives usually occur at the

Table 4 The Efficiency of fuzzing and symbolic execution

CVE-ID	Fuzzing		Symbolic execution		
	Time	# of syscalls	Min # of BBL	Max # of BBL	Ave # of BBL
2017-17053	NA	NA	6	18	13
2017-15649	26 m	433	4	39	21
2017-15265	NA	NA	4	5	5
2017-10661	2 m	26	7	14	11
2017-8890	139 m	448	13	86	48
2017-8824	99 m	63	2	33	23
2017-7374	NA	NA	NA	NA	NA
2016-10150	NA	NA	1	1	1
2016-8655	1m	448	4	27	14
2016-7117	NA	NA	1	1	1
2016-4557	1 m	133	3	48	29
2016-0728	1 m	7	21	31	26
2015-3636	NA	NA	NA	NA	NA
2014-2851	146 m	1203	1	5	3
2013-7446	209 m	448	1	2	1

site close to dangling pointer dereference. By setting symbolic execution to explore exploitable machine states within a maximum depth of 200 basic blocks, we could not only ensure the identification of exploitable states but also reduce the risk of experiencing path explosion.

Related work

Automatic exploit generation

Revery aims at automatic exploit generation, which is still an open challenge. A few number of solutions have been proposed.

AEG based on symbolic execution

APEG (Brumley et al. 2008) is the first automated exploitation solution based on patch analysis. AEG (Avgerinos et al. 2014) develops a novel preconditioned symbolic execution and path prioritization techniques to generate exploits at the source code level. Mayhem (Cha et al. 2012), which is built based on the hybrid symbolic execution and memory index modeling techniques, can automatically generate exploits at the binary level.

These solutions symbolically execute the whole program and are not scalable in path exploration. Unlike Revery, they are unaware of exploitable state constraints. Previous AEG solutions (such as (Avgerinos et al. 2014)) could not solve the derivability problem. They usually only support stack overflow (where exploitable point is the crashing point) and format string (where exploitable point is the vulnerability point) vulnerabilities, rather than heap-based vulnerabilities. Since the exploitable points are in the crashing paths, no diverging paths are explored by these solutions.

Moreover, they will concretize symbolic values in order to make symbolic execution practical. For example, Mayhem proposes a prioritized concretization solution. As aforementioned, concretization could lead to non-exploitable states.

AEG based on crash analysis

Sean Heelan (Heelan 2009) makes use of dynamic taint analysis and program verification to generate control-flow-hijack exploits based on the crashing PoC input. Similarly, starting from the crashing point, CRAX (Huang et al. 2012) symbolically executes the program to find exploitable states and automatically generates working exploits at the binary level.

These solutions only search the crashing paths for exploitable states. As aforementioned, exploitable states do not always exist in crashing paths. So they will be hindered by the exploit derivability issue. By contrast, Revery explores exploitable states not only in crashing paths but also in diverging paths.

Data-oriented AEG

FLOWSTITCH (Hu et al. 2015) automatically generates data-oriented exploits, able to reach information disclosure and privilege escalation, by stitching multiple data flows without breaking the control flow.

Although it also uses stitching, it is quite different from Revery. First, it targets data-flow stitching, while the control-flow is intact, making symbolic execution easier. Second, it only produces exploits of data-only attacks, instead of control-flow hijacking attacks.

Other AEG solutions

Ardilla (Kieżun et al. 2009) and Chainsaw (Alhuzali et al. 2016) are AEG solutions for web applications. Ardilla can create SQL injection and cross-site scripting (XSS) attacks automatically. Chainsaw is a system that reasons systematically over the navigation structure and uses the database state of web applications to automatically generate working exploits. They are quite different from AEG for binary applications, including Revery.

Directed fuzzing

Revery utilizes fuzzing to explore diverging paths. There are many advances in this field in recent years.

Coverage-guide fuzzing

There are many works which aim to increase code coverage of fuzz testing, called coverage-guide fuzzing. AFL (Zalewski 2018), libFuzzer (Serebryany 2016), honggfuzz (Swiecki 2016), AFLFast (Böhme et al. 2016), VUzzer (Rawat et al. 2017) and CollAFL (Gan et al.) are some state-of-the-art coverage-guided fuzzers. In general, they prioritize the seeds with higher code-coverage for further mutation. However, they do not target specific code or memory states, and thus are not efficient in exploring diverging paths which must satisfy some requirements.

Target-directed fuzzing

The most similar work to our focus is AFLGo (Böhme et al. 2017), a greybox fuzzing tool. AFLGo (Böhme et al. 2017) prioritizes seeds that are closer to a piece of predetermined target code, enabling efficient directed fuzzing. In our solution, we are interested in seeds that can trigger multiple layout-contributor instructions, rather than one instruction. AFLGo is not effective in exploring diverging paths which have multiple target points. Revery guides a fuzzer with layout-contributor slice to explore diverging paths and search for exploitable states efficiently.

Discussion

AEG is an open challenge. Revery only moves one step towards this goal. It has many challenges, including but not limited to:

- **Advanced Defenses.** More and more defenses are proposed and deployed in practice, in order to stop popular attacks. These defenses not only raise the bar for human attackers, but also hinder automated solutions. For example, Revery could not bypass ASLR because it lacks the ability of information disclosure. It could trigger exploitable states for 5 of 19 programs, but not able to generate working defenses, because of the sanity checks deployed in heap allocators.
- **Heap Layout Massaging.** A large number of heap-based vulnerabilities could only be exploited in specific memory layouts. Due to the complexity of memory allocators and the program behavior, it is very challenging to generate inputs to build memory layouts as expected.
- **Combination of Multiple Vulnerabilities.** In practice, a successful exploit usually require multiple vulnerabilities. We have to assemble different vulnerabilities and utilize their corruption effects to craft a final exploit.
- **Program Comprehension and Analysis.** To successfully exploit a program, it is necessary to understand the program behavior, e.g., what input will cause what output, and make dynamic decisions at runtime. In addition, few program analysis solutions could extract such information. As aforementioned, the widely used symbolic execution has many limitations too.

Conclusion

Existing AEG solutions are facing the challenges from exploit derivability issue, symbolic execution bottleneck, heap-based and kernel UAF vulnerabilities. We proposed two solutions able to search exploitable states in diverging paths rather than crashing path, with a novel oriented fuzzing and a control-flow stitching solution. They could trigger both vulnerabilities and exploitable states for a big portion of vulnerable applications. They could also successfully generate working exploits for certain vulnerabilities. For some programs which have higher complexity and scalability, such as OS kernel, we can facilitate exploitation of kernel UAF vulnerabilities.We show that our framework could explore OS kernel and identify various system calls essential for exploiting an UAF vulnerability and bypassing security mitigation. We has moved one step towards practical AEG. But there is a long way to go.

Endnotes

[1] At the fuzzing stage, our objective is to identify system calls for diversifying running contexts but not directly for generating exploitation. Therefore, we disable kernel address randomization for reducing the complexity of tracking down dangling pointers.

[2] We did not evaluate Revery on CGC programs which have heap-based vulnerabilities or real world programs, because the binary analysis engine angr (Shoshitaishvili et al. 2016)'s constraints solving ability is not enough for complex programs. And we are still working on it.

Acknowledgements
Not applicable.

Funding
We would like to thank the anonymous reviewers for their constructive comments. This work is supported by the Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences and Beijing Key Laboratory of Network Security and Protection Technology, as well as Beijing Municipal Science and Technology Project (No.Z181100002718002), National Natural Science Foundation of China (No. 61572481 and 61602470, 61772308, 61472209, 61502536, and U1736209), and Young Elite Scientists Sponsorship Program by CAST (No. 2016QNRC001).

Availability of data and materials
All public dataset sources are as described in the paper.

Authors' contributions
YW and WW designed the study. YW and WW performed the experiments. CZ, YW, WW and XX wrote the paper. XG and WZ reviewed and edited the manuscript. All authors read and approved the manuscript.

Authors' information
Xiaorui Gong is a senior engineer at School of Cyber Security, University of Chinese Academy of Sciences. His research focuses on software and system security.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note
Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Author details
[1]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China. [2]Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China. [3]College of Information Sciences and Technology, Pennsylvania State University, University Park, United States. [4]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. [5]Key Laboratory of Network Assessment Technology, CAS, Beijing, China. [6]Beijing Key Laboratory of Network Security and Protection Technology, Beijing, China.

Received: 5 November 2018 Accepted: 20 February 2019
Published online: 29 March 2019

References
!exploitable Crash Analyzer (2018). http://msecdbg.codeplex.com/. Accessed 1 May 2018
Alhuzali A, Eshete B, Gjomemo R, Venkatakrishnan V (2016) Chainsaw: Chained automated workflow-based exploit generation. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM. pp 641–652
Andersen S, Abella V (2004) Data Execution Prevention: Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies. http://technet.microsoft.com/en-us/library/bb457155.aspx
Anonymous (2018) Demo exploit. https://www.dropbox.com/s/xk7bijxd66650ee/demo.tar.gz?dl=0

Argyroudis P (2012) The Linux kernel memory allocators from an exploitation perspective. https://argp.github.io/2012/01/03/linux-kernel-heap-exploitation/
Avgerinos T, Cha SK, Lim B, Hao T, Brumley D (2011) Aeg: Automatic exploit generation. In: Network and Distributed System Security Symposium
Avgerinos T, Cha SK, Rebert A, Schwartz EJ, Woo M, Brumley D (2014) Automatic exploit generation. Communications of the ACM 57(2):74–84
Azad B (2016) Mac OS X Privilege Escalation via Use-After-Free: CVE-2016-1828. https://bazad.github.io/2016/05/mac-os-x-use-after-free/#use-after-free
Böhme M, Pham V-T, Nguyen M-D, Roychoudhury A (2017) Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM. pp 2329–2344
Böhme M, Pham V-T, Roychoudhury A (2016) Coverage-based greybox fuzzing as markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM. pp 1032–1043
Brumley D, Poosankam P, Song D, Zheng J (2008) Automatic patch-based exploit generation is possible: Techniques and implications. In: Proceedings of the 29th IEEE Symposium on Security & Privacy, Oakland
Cha SK, Avgerinos T, Rebert A, Brumley D (2012) Unleashing mayhem on binary code. In: Security and Privacy (SP), 2012 IEEE Symposium On. IEEE. pp 380–394
CTF TIME (2018). https://ctftime.org. Online: accessed 01-May-2018
Database NV (2017) CVE-2017-7374 Detail. https://nvd.nist.gov/vuln/detail/CVE-2017-7374
Dullien T, Flake H (2011) Exploitation and state machines. Proc Infiltrate
Gan S, Zhang C, Qin X, Tu X, Li K, Pei Z, Chen Z Collafl: Path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy (SP), vol. 00. pp 660–677. https://doi.org/10.1109/SP.2018.00040. https://doi.ieeecomputersociety.org/10.1109/SP.2018.00040
Gruss D, Maurice C, Fogh A, Lipp M, Mangard S (2016) Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS). ACM
He L, Cai Y, Hu H, Su P, Liang Z, Yang Y, Huang H, Yan J, Jia X, Feng D (2017) Automatically assessing crashes from heap overflows. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE Press. pp 274–279
Heelan S (2009) Automatic generation of control flow hijacking exploits for software vulnerabilities. PhD thesis, University of Oxford
Hu H, Chua ZL, Adrian S, Saxena P, Liang Z (2015) Automatic generation of data-oriented exploits. In: USENIX Security Symposium. USENIX Association, Washington, D.C. pp 177–192. http://blogs.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu
Huang S-K, Huang M-H, Huang P-Y, Lai C-W, Lu H-L, Leong W-M (2012) Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In: Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference On. IEEE. pp 78–87
Jang Y, Lee S, Kim T (2010) Breaking kernel address space layout randomization with intel tsx. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)
jndok (2016) Analysis and exploitation of Pegasus Kernel vulnerabilities. https://jndok.github.io/2016/10/04/pegasus-writeup/
KASAN (2017) The Kernel Address Sanitizer(KASAN). https://github.com/google/kasan/wiki
Kieżun A, Guo PJ, Jayaraman K, Ernst MD (2009) Automatic creation of SQL injection and cross-site scripting attacks. In: ICSE 2009, Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, Vancouver. pp 199–209
Konovalov A (2017) Exploiting the Linux kernel via packet sockets. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html. Accessed 18 Jan 2018
Nikolenko V (2016) Linux Kernel ROP - Ropping your way to # (Part 1). https://www.trustwave.com/Resources/SpiderLabs-Blog/Linux-Kernel-ROP---Ropping-your-way-to---(Part-1)/
PaX-Team (2003) PaX ASLR (Address Space Layout Randomization). http://pax.grsecurity.net/docs/aslr.txt
Rawat S, Jain V, Kumar A, Bos H (2017) VUzzer: Application-aware Evolutionary Fuzzing. In: Network and Distributed System Security Symposium
Rex (2018) Shellphish's automated exploitation engine. https://github.com/shellphish/rex. Online: accessed 01-May-2018
Schwartz EJ, Avgerinos T, Brumley D (2011) Q: Exploit hardening made easy. In: USENIX Security Symposium. Usenix. pp 25–41
Serebryany K (2016) Continuous fuzzing with libfuzzer and addresssanitizer. In: Cybersecurity Development (SecDev), IEEE. IEEE. pp 157–157
Serebryany K, Bruening D, Potapenko A, Vyukov D (2012) Addresssanitizer: A fast address sanity checker. In: the 2012 USENIX Annual Technical Conference. USENIX Association,. pp 309–318
Serebryany K, Stepanov E, Shlyapnikov A, Tsyrklevich V, Vyukov D (2018) Memory tagging and how it improves C/C++ memory safety. CoRR abs/1802.09517. 1802.09517
Shoshitaishvili Y, Wang R, Salls C, Stephens N, Polino M, Dutcher A, Grosen J, Feng S, Hauser C, Kruegel C, et al (2016) Sok:(state of) the art of war: Offensive techniques in binary analysis. In: Security and Privacy (SP), 2016 IEEE Symposium On. IEEE. pp 138–157
Sotirov A (2007) Heap feng shui in javascript. Black Hat Eur
Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna G (2016) Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS Vol. 16. pp 1–16
Swiecki R (2016) Honggfuzz. Available online a t: http://code.google.com/p/honggfuzz
Unlink Exploit (2018). https://heap-exploitation.dhavalkapil.com/attacks/unlink_exploit.html. Online: accessed 01-May-2018
Valgrind (2018). http://valgrind.org. Accessed 1 May 2018
Vanegue J (2013) The automated exploitation grand challenge. In: Presented at H2HC Conference
Xu W, Li J, Shu J, Yang W, Xie T, Zhang Y, Gu D (2015) From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In: Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security (CCS). ACM
Zalewski M (2018) American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. Online: accessed 01-May-2018

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com